LEGION: PROGRAMMING DISTRIBUTED HETEROGENEOUS ARCHITECTURES WITH LOGICAL REGIONS

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

Michael Edward Bauer
December 2014

Abstract

This thesis covers the design and implementation of Legion, a new programming model and runtime system for targeting distributed heterogeneous machine architectures. Legion introduces *logical regions* as a new abstraction for describing the structure and usage of program data. We describe how logical regions provide a mechanism for applications to express important properties of program data, such as locality and independence, that are often ignored by current programming systems. We also show how logical regions allow programmers to scope the usage of program data by different computations. The explicit nature of logical regions makes these properties of programs manifest, allowing many of the challenging burdens of parallel programming, including dependence analysis and data movement, to be off-loaded from the programmer to the programming system.

Logical regions also improve the programmability and portability of applications by decoupling the specification of a program from how it is mapped onto a target architecture. Logical regions abstractly describe sets of program data without requiring any specification regarding the placement or layout of data. To control decisions about the placement of computations and data, we introduce a novel *mapping interface* that gives an application programmatic control over mapping decisions at runtime. Different implementations of the mapper interface can be used to port applications to new architectures and to explore alternative mapping choices. Legion guarantees that the decisions made through the mapping interface are independent of the correctness of the program, thus facilitating easy porting and tuning of applications to new architectures with different performance characteristics.

Using the information provided by logical regions, an implementation of Legion

can automatically extract parallelism, manage data movement, and infer synchronization. We describe the algorithms and data structures necessary for efficiently performing these operations. We further show how the Legion runtime can be generalized to operate as a distributed system, making it possible for Legion applications to scale well. As both applications and machines continue to become more complex, the ability of Legion to relieve application developers of many of the tedious responsibilities they currently face will become increasingly important.

To demonstrate the performance of Legion, we port a production combustion simulation, called S3D, to Legion. We describe how S3D is implemented within the Legion programming model as well as the different mapping strategies that are employed to tune S3D for runs on different architectures. Our performance results show that a version of S3D running on Legion is nearly three times as fast as comparable state-of-the-art versions of S3D when run at 8192 nodes on the number two supercomputer in the world.

Acknowledgement

This thesis, and more importantly, the Legion runtime, would not have been possible without the help and support of many people. Foremost on this list is my advisor, Alex Aiken. Alex sought me out my first year at Stanford and encouraged me to pursue work on programming systems despite knowing that my primary training was in computer architecture. Through many setbacks and obstacles along the way, Alex remained patient as I developed the skills necessary to become a true systems programmer. Alex fostered an environment that emphasized solving important problems regardless of the risks, instead of working on incremental improvements. Time and again, this atmosphere has proven crucial for enabling projects like Legion to flourish, and I feel privileged to have been a part of it.

I would also like to thank the other professors who have made my time at Stanford so successful. Bill Dally gave me my start at Stanford working on the Sequoia project, inspiring many of the ideas in Legion. Pat Hanrahan articulated the domain specific language story which has driven much of the design of Legion. By running the Pervasive Parallelism Lab, Kunle Olukotun facilitated many of the personal connections necessary for advertising Legion. Both Juan Alonso and Eric Darve have been early adopters and contributed significantly to the development of Legion.

A significant portion of this thesis is a result of direct collaboration with scientist from the national labs as well as departments beyond computer science here at Stanford. Pat McCormick from Los Alamos National Lab was the earliest champion of Legion and recognized its potential from the start. To this day he has remained our most vocal supporter and has guided us through the sometimes turbulent waters of

the supercomputing community. Also from Los Alamos, Sam Gutierrez, Charles Ferenbaugh, and Kei Davis were some of the earliest Legion users and provided valuable feedback on the initial designs. Hemanth Kolla and Ankit Bhagatwala from Sandia National Lab were very generous with their time in explaining the science and implementation of S3D. Ramanan Sankaran from Oak Ridge National Lab provided both code and advice for running experiments on Titan. Shima Alizadeh and Chao Chen from the Stanford mechanical engineering department have both been very supportive early users of Legion. Special thanks go to Jackie Chen, from Sandia National Lab, who trusted us to use her production S3D code as the primary vehicle for showcasing Legion when no others were willing to take the risk.

Surviving graduate school would not have been possible without the support of my fellow graduate students. Conversations, both technical and recreational, over many years with Tom Dillig, Isil Dillig, Rahul Sharma, John Clark, Manolis Papadakis, Wonchan Lee, and Zhihao Jia have made me a better computer scientist and a better person. Adam Oliner and Peter Hawkins were outstanding mentors throughout my early years of graduate school. James Balfour, Curt Harting, and Ted Jiang made for the best company and kept me grounded in my hardware roots. Kshipra Bhawalkar has been my constant voice of wisdom for ten years through both undergraduate and graduate school. Katherine Breeden voluntarily proofread this entire thesis and instigated many early-morning runs which often proved vital to the maintenance of my sanity. Elliott Slaughter was the first person brave enough to join the Legion project and patiently suffered through many of my bugs while giving gentle feedback, for which I will always be grateful. My academic brothers, Eric Schkufza and Zach DeVito, both journeyed with me through the many trials of graduate school and there is no way I would have made it through without them.

From the beginning, the creation and development of Legion has been a joint partnership with Sean Treichler. I know for certain that Legion would not exist in the form that it does today without our combined effort. Working with Sean has resulted in one of the most productive and creative periods of my life, due in a large part to the vast knowledge and experience he was willing to share with me. I'm confident saying that I will never engage in another collaboration like the one we

have had.

Since this is the final stage of my formal education, I want to thank my family for their enduring love and support throughout my journey. My parents Steven and Kristine Bauer have been there for me from my first day of school, twenty three years ago, to today. It has been a long and arduous endeavor at times, but their light has always guided me along the path. I would also like to especially thank my aunt and uncle, Tom and Colleen Kistler, for all their support over the years; it has truly been like having a second set of parents. Lastly, my brother Rick Bauer has always been there for me, and has been instrumental in reminding me that there exists a natural world beyond the thought spaces in which I so frequently find myself lost.

To all my family and friends: thank you for everything! Thank you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The rate of progress of science and engineering is tied to the ability of scientists and engineers to conduct experiments. In the past several decades, the advent of computers and the increasing cost and complexity of actual experiments has motivated many scientists to conduct research via computational methods. The large and growing number of computational scientists has been a core driver behind the consistent improvement of the world's supercomputers. Figure 1.1 shows the progression of performance capabilities of the world's top 500 supercomputers over time [7]. The steady march of performance improvements is indicative of the increasing importance that is placed on computational science and engineering.

Despite the growing importance of supercomputing to the progress of science and engineering, the programming tools available for conducting computational science are relatively primitive. The same programming systems that were developed at the dawn of supercomputing are still in use today even though hardware performance has increased by many orders of magnitude (note the logarithmic scale in Figure 1.1). Furthermore, the performance improvements of modern hardware have only been made possible through the deployment of increasingly complex and exotic architectures. Consequently, the burden of fully leveraging the computational power of current supercomputers is shouldered directly by computational scientists who are ill-equipped to handle the mounting complexity.

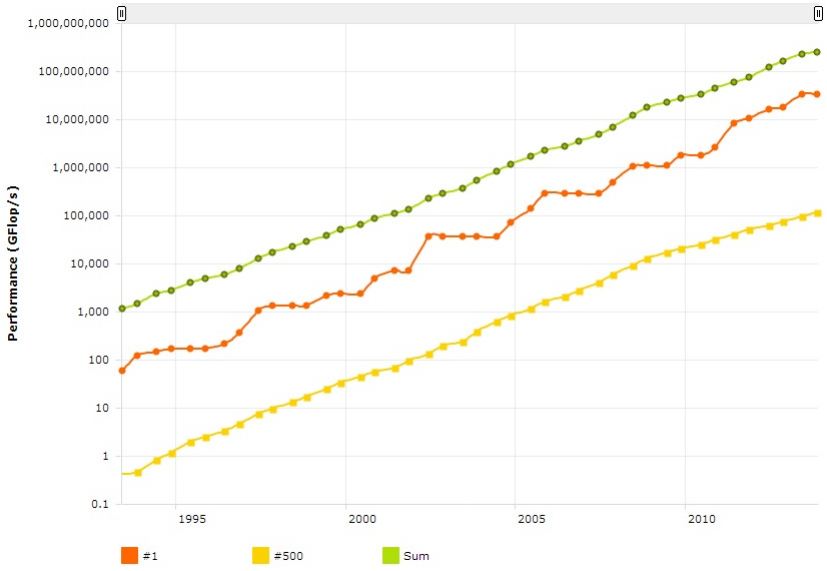


Figure 1.1: Peak Performance of the Top 500 Supercomputers[7]

The obvious solution to this problem, as with most complex problems in computer science, is abstraction. Extracting the full potential from modern and future supercomputers requires the development of new programming models with the necessary abstractions to simplify programming while still achieving high performance and easing the costs of porting and tuning important scientific codes. In this thesis we present Legion, a new programming model and runtime system based on the abstraction of logical regions as one solution for programming current and future distributed heterogeneous supercomputers.

1.1 Dissertation Overview

The design and implementation of the Legion programming system has many different layers of abstraction to address the multitude of problems associated with modern supercomputing. The remainder of this chapter is dedicated to providing a detailed description of our motivation for Legion and the design principles used to guide our decision making when developing the necessary abstractions.

The technical discussion of Legion begins in Chapter 2 by introducing logical regions as the fundamental abstraction and describes the resulting programming model built with logical regions as the foundation. Chapter 3 initiates our description of the implementation of the Legion runtime system by covering the architecture of the different layers of the system. Chapters 4-7 cover the details of the Legion runtime implementation including traversals of the logical and physical region trees, the distribution of meta-data structures, and garbage collection. Chapter 8 introduces Legion's novel mapper interface and illustrates how mappers can be used to intelligently tune applications for different machine architectures. In Chapters 9 and 10 we show two important extensions to Legion that add both expressivity and resiliency to the programming system. In Chapter 11 we demonstrate that Legion is a real system, capable of handling S3D, a production combustion simulation. Finally, we give a detailed accounting of related work in Chapter 12 and draw conclusions in Chapter 13.

1.2 The State of Programming Supercomputers

Supercomputers have a long history of being some of the most challenging machines to program. Before delving into the details of Legion, we first investigate the state of programming modern supercomputers to discover the origins of many of the problems faced by today's computational scientists. While there are many approaches to programming supercomputers, we cover only the most common ones here. A more complete description of related work and other programming systems for supercomputers can be found in Chapter 12.

By far, the most ubiquitous programming system in use for programming supercomputers today is the Message Passing Interface (MPI) standard [45]. MPI is a Singe-Program Multiple-Data (SPMD) programming model where many copies of the application are run in parallel to operate on different sets of data. MPI provides a uniform interface for performing communication between multiple processes running on different nodes of a supercomputer by sending messages. Additionally, it also

provides support for performing low-latency synchronization such as barriers and collective operations. The MPI interface is designed to be general enough that it can be implemented on top of many different interconnects including Ethernet, Infiniband, Cray Gemini and Aires, and IBM Blue Gene. This approach makes it possible to migrate MPI codes to new machine architectures.

While MPI provides a universal interface for programming supercomputers, it suffers from a fundamental flaw that both impedes its performance portability and compromises its ability to support modular code, thus resulting in applications that are inherently difficult to maintain. Consider the following example MPI pseudo code which illustrates the posting of an asynchronous receive followed by some anonymous work `Y` before using the received results:

```
receive(x, ...);
Y;
sync;
f(x);
```

This pseudo code represents a common MPI idiom. In order to hide the latency of communication, MPI applications must use asynchronous sends and receives to overlap communication with additional work (e.g. `Y`). This leads to two problems.

1. It is the responsibility of the programmer to find the useful computation `Y` to overlap with the `receive`. There are several constraints on this choice of `Y`. The execution of `Y` must not be too short or the latency of the `receive` will not be hidden, and it must not be too long or the continuation `f(x)` will be unnecessarily delayed. Thus, for each new target machine the choice of `Y` will have to be re-tuned for the communication latency of the underlying interconnect. Furthermore, on machines with dynamically routed interconnect networks [1], it will be impossible for programmers to accurately determine a choice for `Y` as it will vary dynamically both in space and time with the load on the network.

2. Since `Y` is not permitted to use `x`, `Y` must be an unrelated computation, resulting in code which is inherently non-modular. Being able to construct modular code

is a fundamental pillar of software engineering and a crucial tool in creating abstractions that manage software complexity. MPI requires programmers to destroy modularity for the sake of performance, ultimately resulting in code that is difficult to maintain and refactor.

These deficiencies permeate many of the applications that use MPI today, resulting in codes which are inherently difficult to maintain and not performance portable. In Chapter 2 we will demonstrate how Legion's programming model is capable of automatically overlapping computation with communication in a way that still permits and encourages modular code.

While MPI is the primary programming system used for communicating between nodes, programmers use additional tools for extracting performance within a node. Most frequently, programmers use OpenMP [24] to take advantage of loop-level data parallelism. The OpenMP runtime maintains a set of background threads which are used for executing parallel loops. Programmers are responsible for annotating parallel loops with pragma statements that detail how loops should be parallelized. While this approach benefits from its simplicity, it too is fundamentally flawed. OpenMP is ignorant of cache hierarchies and non-uniform memory access (NUMA) domains, resulting in code with sub-optimal performance. Furthermore, OpenMP is only capable of distributing work across the homogeneous CPU processors in a blocking fashion, making it impossible to offload work from other loops in parallel onto accelerators. In Chapter 8 we will show how the Legion mapping interface allows tasks to run in parallel across both CPUs and GPUs simultaneously while also allowing mappings that are both cache- and NUMA-aware.

To target accelerators most programmers use either CUDA [4] or OpenCL [36]. Both CUDA and OpenCL provide the necessary primitives for copying data between the various memories (e.g. framebuffer and system memory) as well as the ability to execute kernels on the accelerator cores. While these languages are sufficient for programming accelerators, they place a significant burden on the programmer. Programmers are responsible for issuing the explicit (asynchronous) data movement commands and then properly synchronizing these data movements with computation using either streams in CUDA or events in OpenCL. Much like MPI, this again places

the burden of overlapping computation with data movement on the programmer and also makes it easy to introduce complex synchronization bugs. Furthermore, the data movement operations for CUDA and OpenCL do not compose with the operations in MPI, commonly requiring the data first be moved to a node with an MPI communication primitive and then later moved again with a CUDA or OpenCL call. Synchronizing between all of these different operations because of interface incompatibility adds latency and increases code complexity.

Due to the programming challenges inherent in using CUDA and OpenCL, recently OpenACC [3] has been proposed as a way to use accelerators with a model similar to OpenMP. To use OpenACC programmers also annotate loops with pragmas much like OpenMP. In addition to specifying how to parallelize the loop, programmers must also specify the data allocations that will be accessed. In some respects this is a positive step as it is the first instance of a programming system being aware of the structure of program data. By leveraging this knowledge, OpenACC can automatically move data between system memory and the GPU framebuffer memory. However, OpenACC fails to fully understand the data dependences between consecutive loops and by default always copies data both down and back from the GPU for each loop. Programmers can address this performance problem by adding additional annotations stating which data can be left on the GPU and which data can be released without copying it back. However, incorrect annotations or code refactoring can easily introduce bugs which the programming system can neither detect nor fix.

As a whole, these programming systems constitute an eclectic collection of tools with little common infrastructure and vastly different semantics. Scientific programmers are forced to choose a subset of these tools that match their target machine and then provide the considerable glue logic for mediating between the various APIs and programming models. Writing this glue code is both tedious and error-prone, ultimately adding unnecessary complexity to applications which are already extremely sophisticated. It is obvious that this ad hoc approach to programming supercomputers is neither ideal nor scalable and therefore new approaches must be found.

1.3 Motivation

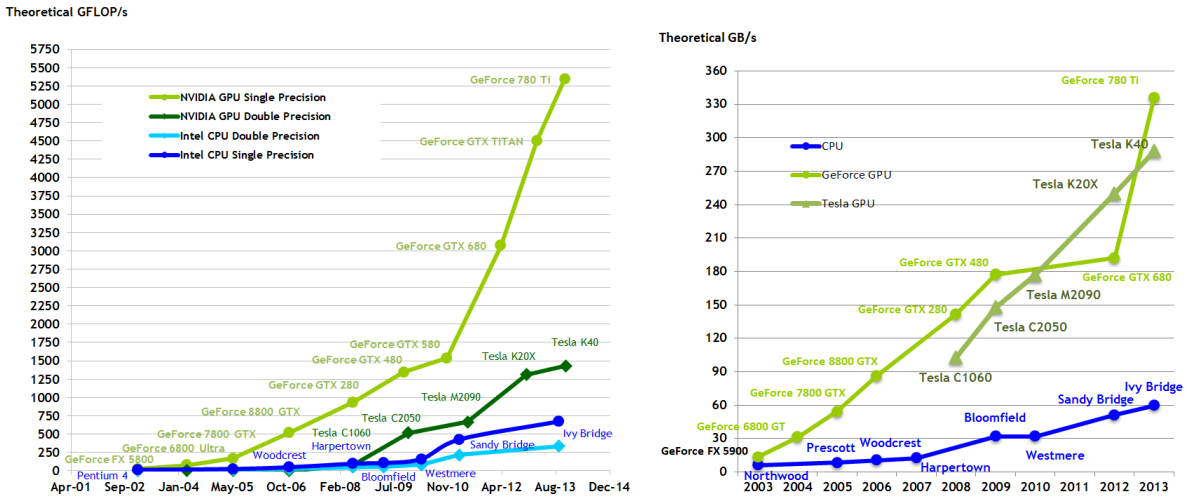
Our motivation in designing Legion is not simply to fix the problems of current programming systems outlined in the previous section, but rather to construct a system that meets the needs of programmers for both modern and future supercomputing architectures. To meet this constraint, the design of Legion is based on three important and noticeable trends in supercomputing that forecast the future of the field: the increasing cost of data movement relative to the cost of computation, the increasing amount of dynamism in both hardware and software, and the growing prevalence of heterogeneity of both processors and memories. We now discuss each of these trends in turn.

1.3.1 Cost of Data Movement

While the peak floating point throughput of modern supercomputers has continued to scale with Moore's law, the maximum theoretical bandwidth provided by both memory and communication interfaces has scaled linearly at best. Figure 1.2a and Figure 1.2b [4] show the theoretical peak floating point and bandwidth throughput trends respectively for both CPUs and GPUs over time. The growing disparity between compute throughput and bandwidth is driving more applications to be limited by the cost of data movement (either in terms of performance or power). In the future only the most computationally expensive applications (e.g. matrix multiply) will remain limited by compute throughput.

In addition to the disparity between compute throughput and bandwidth, the latency of communication is continuing to grow. As machines increase in size, the cost of communicating between nodes also increases. This trend is irreversible as the latency of communication will always be fundamentally limited by the speed of light.

Consequently, the cost of data movement is quickly coming to dominate the performance of most scientific computing applications. Also, since data movement is one of the more expensive operations in terms of power for hardware, these costs extend beyond raw execution time, and further into the power budgets of modern hardware.



(a) Peak Theoretical Floating Point Throughput (b) Peak Theoretical Memory Bandwidth

Figure 1.2: Comparing Peak Floating Point and Bandwidth Performance[4]

1.3.2 Hardware and Software Dynamism

Another important trend is the burgeoning amount of dynamism present in both software and hardware. Initially most scientific computations operated on structured Cartesian grids. However, recent scientific applications are coming to operate on more irregular data structures such as graphs [29] and unstructured meshes [25]. These applications construct data structures at runtime to match the input data on which they are going to operate. Being able to support these kinds of applications requires programming systems capable of handling dynamic partitioning and mapping of both data and tasks.

Even more dynamic are applications that perform operations such as Adaptive Mesh Refinement (AMR) [39, 8]. These codes react to changes in the simulation to create new simulation spaces and computations. Handling these applications requires the ability to dynamically re-balance both data and task distributions at runtime throughout the execution of the application. These kinds of applications are becoming more prevalent and require programming systems capable of reacting to their dynamism.

In addition to the dynamism of emerging software, hardware is also becoming more dynamic. The most obvious evidence of dynamism in hardware is the presence

of dynamically routed interconnect networks that leverage on-line routing algorithms based on traffic flow [1] which adds variability to the latencies of communication. More subtly, the strict power budgets of future architectures will introduce hardware with low-power cores [6] and support for dark silicon [27]. Finally, as circuit feature sizes continue to shrink, the prevalence of hardware faults, both soft and hard, will make hardware less reliable, requiring dynamic recovery mechanisms.

1.3.3 Heterogeneous Architectures

The third, and most prominent trend, is the emergence of heterogeneous architectures. The introduction of accelerators into the previously homogeneous world of supercomputing fundamentally disrupted the nature of how supercomputers were both used and programmed. Accelerators such as NVIDIA and AMD GPUs as well as Intel Xeon Phis have greatly complicated programming models. Accelerators introduce dissonance in processor speeds by providing additional cores optimized for throughput instead of latency. Furthermore, accelerators introduce heterogeneity not just in kinds of processors, but in kinds of memory as well. For example, most GPUs require programmers to explicitly manage the placement and movement of data between framebuffer memories, zero-copy memories, and on-chip shared memories. More recently, new kinds of memory are being proposed for use by both CPUs and GPUs. The incorporation of flash memory, non-volatile memory (NVRAM), and stacked DRAM will further add to the heterogeneity of memory hierarchies.

The expanding presence of heterogeneity in system architectures has presented programmers with the daunting challenge of determining the best *mapping* of their applications to the target hardware. A mapping is simply a determination of how to assign computations to processors, and how to place data within the memory hierarchy. In homogeneous systems mappings are generally simple: work is evenly divided amongst cores with identical performance and data is evenly divided between nodes. Heterogeneity opens up many new degrees of freedom that make determining the best mapping significantly more challenging. To further complicate the problem, mapping decisions today are usually baked into the source code as we described

in Section 1.2. Changing a mapping routinely requires significant code refactoring, including modifications to both data movement and synchronization code that can easily introduce correctness bugs. For this reason, programmers usually only explore a few mappings before settling for sub-optimal performance.

1.4 Legion Design Principles

Our design of Legion is based on both the problems experienced by current programmers, described in Section 1.2, as well as the motivating trends noted in Section 1.3. The consequence of all these factors is that current scientific programmers face immense complexity when implementing their applications. It is therefore crucial that Legion provide abstractions for managing this complexity.

To guide our design of Legion abstractions, we conceptualized the construction of a scientific application as being composed of four primary components divided along two axes as shown in Figure 1.3. On the horizontal axis we distinguish between the code written for specifying an application in a machine-independent way (e.g. a generic algorithm) versus the code used for mapping the application onto a specific architecture. On the vertical axis we differentiate between code used for specifying how an application should be executed, which we refer to as the *policy* of an application, from the code used to carry out the policy, which we call the *mechanism* of the application.

Based on this classification of the various components of an application, we developed the two design principles on which Legion is based. First, mechanism should be decoupled from policy, and second, applications should be decoupled from their mapping. These principles suggest where abstractions should be placed and serve as the foundation on which the rest of the Legion architecture is built. Specifically, the design goal of Legion is two-fold:

- Decouple mechanism from policy - provide abstractions for applications to specify algorithms and data partitioning policy, while Legion provides the implementation of these abstractions (mechanism).

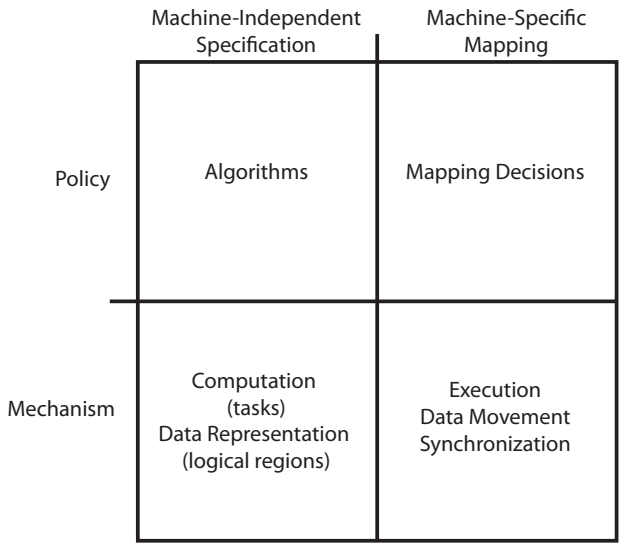


Figure 1.3: Components of Supercomputing Applications

- Decouple specification from mapping - provide two sets of abstractions: one for describing applications independent of particular machines (specification) and another for describing how an application is targeted to a specific architecture (mapping).

The scope of Legion and its various components can be seen in Figure 1.4. We now detail each of these design principles in more detail with regards to how they relate to the trends described in Section 1.3.

1.4.1 Decoupling Policy from Mechanism

Legion provides an implementation of the necessary mechanisms for performing computations, orchestrating data movement, and engaging in synchronization. It remains the application developer's responsibility to specify the policies for an application including the algorithms to be used and the mapping of computations and partitioned data onto the target architecture.

To be more specific, it is important that Legion applications be able to select partitioning algorithms for data because the partitioning strategy is closely tied to the algorithmic choices that cannot be made correctly for every application. On the

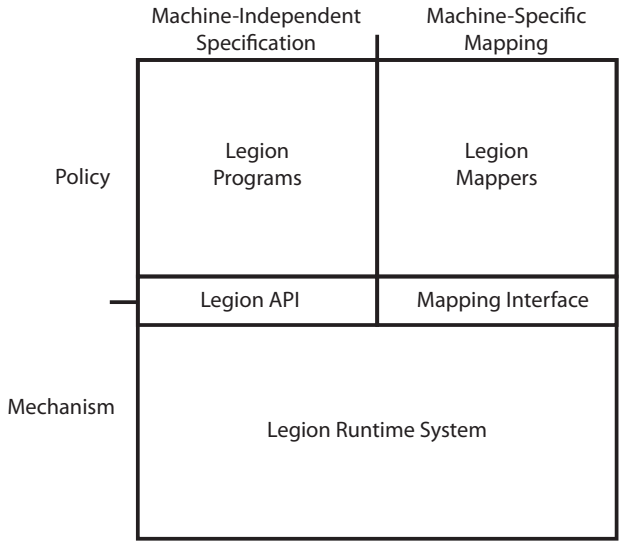


Figure 1.4: Legion Design Overview

other hand, it is also important that Legion carry out the partitioning operation, both because it relieves the programmer of a tedious task and also because Legion needs to know the partitioning of data in order to carry out other functions. As another example , Legion applications should be able to map tasks onto any processors and data into any memories, but Legion should be responsible for implementing the necessary mechanisms (e.g. copies and synchronization) for doing this in a way that is consistent with the application specification.

It is our assertion that this decoupling is essential to the future of any programming system for supercomputers because of the increasing dynamism and growing heterogeneity discussed in Section 1.3. Application codes need to be capable of being developed without needing to re-implement data movement, synchronization, and scheduling routines for each change in algorithm or target architecture that appears on the horizon. By decoupling policy from mechanism we can reduce the amount of code that programmers need to write when changing the structure of applications for new architectures.

1.4.2 Decoupling Specification from Mapping

Our second design principle for Legion is that applications should be described as machine-independent specifications and that any decisions required for targeting a particular architecture should be explicitly decoupled and specified as part of an application's mapping. All three of the trends identified in Section 1.3 are drivers of this design goal. The increasing complexity of applications and architectures coupled with increasing cost of data movement is causing an explosion in the space of potential mappings of an application. Coupling the specification of an application with the mapping fundamentally impedes the exploration of this space and results in poor performance (by any metric, including both performance and power efficiency). Unless these two aspects are explicitly decoupled so that the search for optimal mappings can be abstracted and automated, applications will never be capable of achieving peak performance.

Our goal in Legion is therefore straightforward: provide a programming system capable of supporting machine-independent specifications of applications and a separate mechanism for mapping (and re-mapping) specifications onto both current and future architectures. If successful, Legion applications will be both easy to tune and port to new machines. To achieve this goal, we will need machine-independent abstractions for describing both computation (tasks) and data (logical regions), as we discuss in Chapter 2.

1.4.3 Legion Non-Goals

While we have chosen to tackle several of the significant problems facing supercomputing, we are not claiming to solve all such problems. Specifically, there are two problems which we are not attempting to solve: the determination of the best mapping and programmability.

We have avoided having Legion implement any policy decisions and have instead insured that all policy decisions are explicitly exposed either to the application or the mapper. The reason for this is simple: we do not believe that it is possible for any programming system to always make the ideal policy decisions for the cross-product

of all application codes with all target architectures. Instead, Legion lets applications and mappers decide what the important policy decisions are for each of their specific domains without any interference from Legion. More importantly, we have exposed these decisions in a disciplined way that organizes policy decisions into application policies and mapping policies, allowing developers to address them independently.

A difficulty with exposing all policy decisions is precisely that it places responsibility for determining the best policies on the user. While there is no question that a more automatic, and hence less detailed, way of determining mappings is desirable, we believe that exposing all important decisions is a necessary first step to solving the larger problem of determining the best way to program large heterogeneous machines. By separating the complexity of implementing all the mechanisms from the added complexity of determining the best policies, we have teased apart many of the entangled issues facing programmers today. As with many problems in computer science, divide-and-conquer is an effective strategy. Our implementation of Legion demonstrates that a solution exists to the problem of handling the complexity of implementing all of the various mechanisms for supercomputers. Whether a companion solution exists to the general problem of determining the best mapping remains to be seen, but we believe that Legion provides the first real platform for addressing this problem in a tractable manner.

The second Legion non-goal is programmability. Our goal in Legion is to provide the bare minimum set of abstractions for finding solutions to the design goals presented previously. It is unlikely that these minimal abstractions will be sufficient for most end user scientists. Instead we think that Legion is best used as an intermediate target for providing a machine-independent basis for the construction of higher-level abstractions. We discuss this issue in more detail when describing our target users in Section 1.4.4.

There is an additional non-goal for this thesis as well. Legion represents a fundamentally new point in the design space of parallel runtimes. Many of the unique aspects of the design of Legion do not make sense outside of the context of a full Legion implementation. Evaluating ideas in isolation is therefore extremely difficult and this thesis makes no attempt to provide justification for individual features. Instead,

we provide a more comprehensive, end-to-end evaluation of Legion. Chapter 11 will describe an implementation of a production-quality version of the combustion simulation S3D. The challenge of running a full-scale application such as S3D on thousands of nodes stresses all of the features that we describe in this thesis. Achieving high performance for these runs mandates that all the features operate in efficiently in concert. The resulting performance improvements achieved by our Legion version of S3D will justify the many features described in this thesis.

1.4.4 Target Users

Based on the principles outlined above, the goal of Legion is to provide a general purpose programming model and runtime system for programming large distributed heterogeneous machines. However, while our ultimate aim is to make it easier to develop codes for computational science, our target end users are not the computational scientists that will ultimately use the machines for conducting experiments. Instead it is our belief that Domain Specific Languages and Libraries (DSLs), such as the Liszt language for mesh-based PDE solvers [25], are the correct level of abstraction for computational scientists to use when developing codes. Legion is designed primarily to be a target for DSL developers as well as advanced system programmers.

There are two reasons that we feel that Legion is a good target for DSLs. First, DSL developers need an intermediate target for their implementations. The alternative is that DSL developers need to directly support all potential target machines now, as well as in the future. This task is non-trivial and consumes significant programmer resources. By targeting Legion, DSLs can express a machine-independent version of their applications and then independently map them onto different architectures using Legion's mapping interface, thereby eliminating the need for maintaining separate hardware backends for each machine type.

The second reason that we believe Legion is aptly suited as a target for DSLs is that it provides a common framework through which to compose DSLs. Composing multiple DSLs for developing applications has been an important goal of the DSL community. Doing so, however, is a challenge because each DSL develops its own

data structures and computational frameworks. However, composing DSLs that target Legion is significantly easier. DSLs that all use logical regions as a data model and launch tasks based on region usage can easily cooperate and allow Legion to automatically infer the necessary data dependences between tasks. While demonstrating the feasibility of this approach is still a work in progress, it has long been our vision for Legion usage.

1.5 Collaborators and Publications

The work done for this thesis was in collaboration with Sean Treichler, Elliott Slaughter, and Alex Aiken. Some of the ideas discussed in this thesis were also the subject of several conference papers [11, 12, 13, 48, 49].

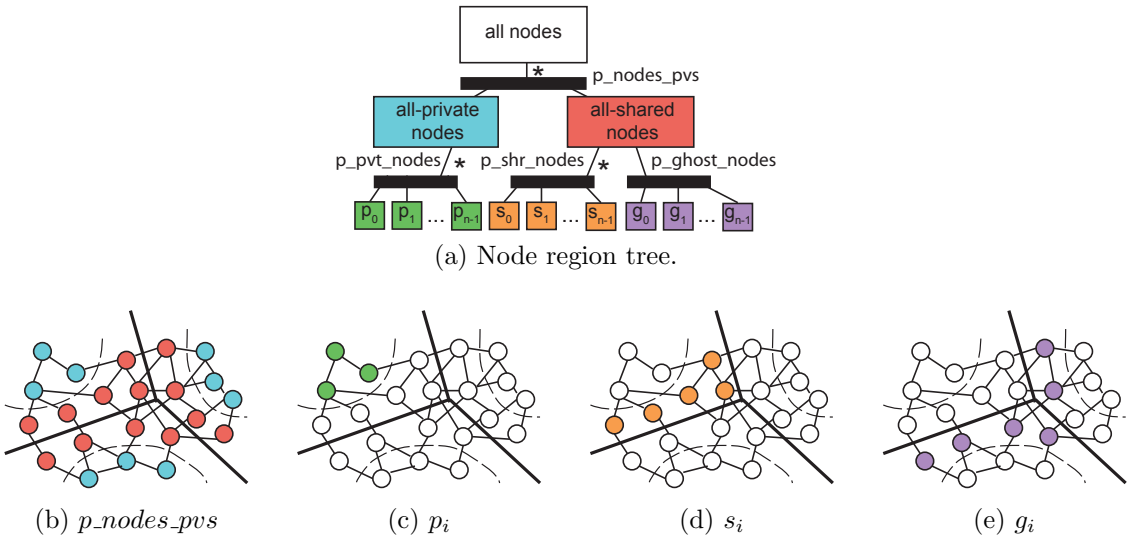
Chapter 2

The Legion Programming Model

In this chapter we outline the design of the Legion programming model. We describe the important features of Legion as well as their interactions to create a composable programming infrastructure. We emphasize that the Legion programming model is general enough to be implemented in most modern programming languages and is not tied to any particular language. In this chapter we focus solely on the features of the programming model and do not describe how they are implemented. Our discussion of the implementation of the Legion runtime begins in Chapter 3. We start with a motivating example to illustrate some of the features of Legion, and how they address many of the problems encountered when writing code for distributed heterogeneous architectures.

2.1 Motivating Example: Circuit Simulation

To make our introduction of the features of Legion concrete, we use a circuit simulation as a running example. Listing 2.1 shows pseudo-code for an electrical circuit simulation written in a C-like language. The circuit simulation represents the circuit as a graph consisting of nodes with different voltage and edges representing various circuit components. At each time step the simulation calculates currents, distributes charge between different nodes, and updates the voltage at each node. We provide a brief overview of the code here. The details of the pertinent Legion abstractions are



(a) Node region tree.

(b) *p_nodes_pvs* (c) p_i (d) s_i (e) g_i

Figure 2.1: Partitions of *r_all_nodes*

covered in the following sections.

A `Circuit` structure contains handles to the names of two logical regions: a collection of nodes and a collection of wires (lines 3-4 of Listing 2.1).[1] It is important to note that logical regions simply describe collections of data and do not imply any placement or layout in the memory hierarchy. We discuss how logical regions are created and used starting in Section 2.2.

2.1.1 Circuit Regions and Partitions

In order to run the circuit simulation in parallel, the circuit graph must be partitioned into *pieces* that can be distributed throughout the memory hierarchy. The number of pieces to create is ultimately chosen by the mapper based on application and architecture specific considerations; the mapper's decision is communicated by a *tunable* variable[28]. Tunable values are necessary for abiding by our second design principle of decoupling specification from mapping. The choice of how many pieces to create is machine specific and therefore must be determined as part of the mapping process.

[1]Note that all pointers declare the region to which they point. For example, the definition of `Wire` (line 2) is parametrized on the region `rn` to which the `Node` pointers in fields `in_nodes` and `out_nodes` point.

```
1  struct Node { float voltage, new_charge, capacitance; };
2  struct Wire⟨rn⟩ { Node@rn in_node, out_node; float current, ... ; };
3  struct Circuit { region  r_all_nodes; /* contains all nodes for the circuit */
4                   region  r_all_wires; /* contains all circuit wires */ };
5  struct CircuitPiece {
6    region  rn_pvt, rn_shr, rn_ghost; /* private, shared, ghost node regions */
7    region  rw_pvt;                   /* private wires region */ };
8
9  void simulate_circuit(Circuit c, float dt) : RWE(c.r_all_nodes, c.r_all_wires)
10 {
11   // The construction of the colorings is not shown.  The colorings wire_owner_map,
12   tunable int num_pieces;
13   // node_owner_map, and node_neighbor_map have num_pieces _PIECES colors
14   // 0..num_pieces − 1. The coloring node_sharing map has two colors 0 and 1.
15   //
16   // Partition of wires into num_pieces pieces
17   partition⟨disjoint⟩ p_wires = c.r_all_wires.partition(wire_owner_map);
18   // Partition nodes into two parts for all−private vs. all−shared
19   partition⟨disjoint⟩ p_nodes_pvs = c.r_all_nodes.partition(node_sharing map);
20
21   // Partition all−private into num_pieces disjoint circuit pieces
22   partition⟨disjoint⟩ p_pvt_nodes = p_nodes_pvs[0].partition(node_owner_map);
23   // Partition all−shared into num_pieces disjoint circuit pieces
24   partition⟨disjoint⟩ p_shr_nodes = p_nodes_pvs[1].partition(node_owner_map);
25   // Partition all−shared into num_pieces ghost regions, which may be aliased
26   partition⟨aliased⟩ p_ghost_nodes = p_nodes_pvs[1].partition(node_neighbor_map);
27
28   CircuitPiece pieces[num_pieces];
29   for(i = 0; i < num_pieces; i++)
30     pieces[i] = { rn_pvt: p_pvt_nodes[i], rn_shr: p_shr_nodes[i],
31                   rn_ghost: p_ghost_nodes[i], rw_pvt: p_wires[i] };
32   for (t = 0; t < TIME_STEPS; t++) {
33     for (i = 0; i < num_pieces; i++) calc_new_currents(pieces[i]);
34     for (i = 0; i < num_pieces; i++) distribute_charge(pieces[i], dt);
35     for (i = 0; i < num_pieces; i++) update_voltages(pieces[i]);
36   }
37 }
38                          // ROE = Read−Only−Exclusive
39 void calc_new_currents(CircuitPiece piece):
40         RWE(piece.rw_pvt), ROE(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
41   foreach(w : piece.rw_pvt)
42     w→current = (w→in_node→voltage − w→out_node→voltage) / w→resistance;
43 }
44                        // RdA = Reduce−Atomic
45 void distribute_charge(CircuitPiece piece, float dt):
46         ROE(piece.rw_pvt), RdA(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
47   foreach(w : piece.rw_pvt) {
48     w→in_node→new_charge += −dt ∗ w→current;
49     w→out_node→new_charge +=  dt ∗ w→current;
50   }
51 }
52
53 void update_voltages(CircuitPiece piece): RWE(piece.rn_pvt, piece.rn_shr) {
54   foreach(n : piece.rn_pvt, piece.rn_shr) {
55     n→voltage += n→new_charge / n→capacitance;
56     n→new_charge = 0;
57   }
58 }
```

Listing 2.1: Circuit simulation

Traditionally, a graph, such as the one for our circuit simulation, is only partitioned a single time into different pieces based on the number of nodes for the target machine. This single partitioning scheme, however, fails to capture all of the different relationships between nodes and wires within the circuit graph. Instead, our Legion circuit simulation employs a two-level partitioning scheme to more precisely capture information about the different sets of the circuit graph necessary for implementing the circuit simulation.

Figure 2.1a depicts a *logical region tree* data structure that demonstrates how the nodes for the circuit graph can be better represented using a multi-level partitioning into logical sub-regions. Intuitively, the logical region tree data structure shows how the nodes within the graph our broken down into subsets represented by logical sub-regions. Our multi-level partitioning is still based on the idea of creating circuit pieces, but we describe the pieces as a union of different subsets. We first partition all the nodes into two distinct subsets, the *all-private* and *all-shared* nodes. These different sets are depicted in Figure 2.1b. The all-private nodes are colored in blue (light-grey) and have no edges which cross a piece boundary. The all-shared nodes are colored in red (dark-grey), and have at least one edge that crosses a piece boundary. This *coloring* of the nodes is reflected in the code by the `node_sharing_map` (line 19) that is passed to Legion to create a partition of the logical region containing all nodes. The resulting partition `p_nodes_pvs` has exactly two logical sub-regions (one for each color), with each node belonging to the logical sub-region based on how it was colored. We discuss the details of partitioning further in Section 2.3.

After our first level of partitioning, we again partition the two logical sub-regions for the all-private and all-shared sub-regions again to describe the subsets for different circuit pieces. Lines 22 and 24 show the calls to partition each of these two sub-regions into further sub-regions describing the subsets for different circuit pieces. The coloring for the upper-left circuit piece for the private and shared nodes is shown in Figures 2.1c and 2.1d respectively.

Another feature of the Legion programming model is that it allows multiple partitions of logical regions to be made, creating multiple views onto the same set of data. For example, in the circuit simulation, we also need to create an additional partition

for describing the *ghost nodes* necessary for each circuit piece. Figure 2.1e shows the coloring for the ghost nodes of the piece in the upper left of the circuit graph. The ghost node partition is a second partition of the all-shared logical region. Note that unlike prior partitions, the ghost node partitioning may color some nodes with multiple colors because some nodes share edges with multiple pieces. We therefore say that this is an *aliased* partition since not all of the sub-regions are disjoint (line 26). The ability of the Legion programming model to capture disjointness and aliasing properties of logical regions will be crucial for constructing a high-performance implementation of Legion.

Based on the logical sub-regions that we create, we can now create circuit pieces for different logical sub-regions. Lines 5-7 of Listing 2.1 declare the type of a `CircuitPiece` structure. Each circuit piece is defined by the private, shared, and ghost logical sub-regions in the logical region tree of all nodes. A circuit piece also names the set of wires that it uses. There is a single disjoint partitioning of the wires logical region into sub-regions based on pieces on line 17. Lines 29-31 then assemble the different circuit pieces for our simulation. It is important to note that the entire assembly of the circuit pieces is done dynamically, including the coloring and partitioning operations. The dynamic nature of this process allows Legion to adapt to both the circuit topology as well as the underlying machine structure at runtime.

A final important detail regarding the partitioning for the circuit simulation is that the actual choice of the partitioning into pieces was selected by the application and not Legion. This reflects our first design principle that we should decouple policy from mechanism. It is the responsibility of the application to choose the partitioning algorithm (policy) for creating the pieces. Legion then provides the mechanism for capturing the result: specifically the colorings for creating partitions and logical sub-regions. Regardless of the chosen partitioning algorithm, colorings are flexible enough to capture the resulting partitioning.

2.1.2 Circuit Tasks and Operations

Line 9 of Listing 2.1 declares the main function for the circuit simulation. The `simulate_circuit` function is an example of a Legion task. All Legion tasks are required to name the logical regions they access along with the *privileges* and *coherence modes* with which they access the logical regions. The privilege and coherence annotation `RWE` on line 9 specifies that the `simulate_circuit` task will access the logical regions `c.r_all_nodes` and `c.r_all_wires` with *read-write* privileges and *exclusive* coherence. Privileges specify the kind of side-effects the task can perform on the logical region. Coherence specifies what other tasks can do with regions at the same time as the task using the regions (if anything). We discuss tasks and privileges in more detail in Section 2.5. Coherence modes are part of an extension to the Legion programming model that we cover in Chapter 9.

Lines 32-58 perform the actual circuit simulation by making three passes over the circuit for each time step. Each pass loops over the array of pieces (constructed on lines 29-31 from the partitions) and asynchronously launches a sub-task computation to be performed for each piece. There are no explicit requests for parallel execution in Legion nor is there explicit synchronization between the passes. Which tasks can be run in parallel within a pass and the required inter-pass dependencies are determined automatically by the Legion runtime based on the region access annotations on the task declarations. The details of how these dependences are covered can be found in our description of the Legion execution model in Section 2.5.3.

The tasks launched on lines 33-35 are *sub-tasks* of the main `simulate_circuit` task. A sub-task can only access regions (or sub-regions) that its parent task could access; furthermore, the sub-task can only have permissions on a region compatible with the parent's permissions. For example, the `calc_new_currents` task reads and writes the wires sub-region and reads the private, shared, and ghost node sub-regions for its piece (line 40). The requested regions for the `calc_new_currents` task therefore meet the requirements of the Legion programming model. We discuss the additional types of privileges available for tasks to use in Section 2.5.1.

The implementation of each of the different *leaf* tasks for our simulation are shown on lines 39-58. Leaf tasks are tasks that perform only computation and do not perform

any additional Legion operations. In general, tasks are permitted to recursively launch additional sub-tasks which we discuss in Section 2.5.2. Tasks are also permitted to perform other kinds of operations such as inline mapping, explicit copies, and execution fences that are discussed in Section 2.7.

2.2 Logical Regions

The primary abstraction for describing data in Legion is *logical regions*. The motivation for logical regions stems directly from our second design principle regarding decoupling application specification from mechanism. While many programming systems have abstractions for computation, few systems have data models for abstracting the description of program data. Logical regions decouple the description of data from any mapping related data decisions:

- Logical regions have no implied location or placement in the memory hierarchy.
- Logical regions have no implied layout of data in memory such as struct-of-arrays (SOA), array-of-structs (AOS), hybrid, etc.

As an example, recall from the circuit simulation in Section 2.1 how logical regions capture the structure of the circuit graph and the various important subsets without committing to the placement or layout of any data. Logical regions, therefore, provide the crucial abstraction for describing the structure of program data without constraining data to any particular target architecture, thus taking the first step in decoupling the specification of an application from its mapping. We now detail how logical regions are created and used.

2.2.1 Relational Data Model

To describe the structure of program data, we require that logical regions be general enough to support arbitrary data structures. To meet this need, we drew inspiration from databases and the work in [33] that demonstrated that *relations* are a general

abstraction capable of encoding most complex data structures in use in real applications. For example, in the circuit simulation, the graph of circuit elements is easy to encode as two separate relations: one for describing the sets of nodes, and another for describing the sets of wires. Logical regions are therefore based on relations and have notions of *index spaces* (e.g. sets of rows) and *field spaces* (e.g. columns). We describe each of these properties in detail in Sections 2.2.2 and 2.2.3 respectively.

Before proceeding, it is important to note that logical regions in many ways are not full relations in the traditional database sense. Relations in databases normally support a full (or nearly full) implementation of relational algebra including operations such as inner and outer joins. In Legion, our goal is not to re-implement a database system. Instead, we are adapting the relational data model provided by relations to serve as the basis for describing data in Legion programs, and supporting a minimal subset of relational algebra necessary for describing the operations used to achieve high performance on our target class of machines. We highlight the important Legion operations that have natural analogs to database operations where appropriate.

2.2.2 Index Spaces

To describe the rows of logical regions we use *index spaces*. Index spaces encapsulate the keys for logical regions. Currently, Legion permits two different kinds of index spaces to be created for describing the sets of keys: *unstructured* and *structured* index spaces. By specifying different kinds of index spaces, applications can indicate the best representation for index spaces. Unstructured index spaces suggest that there is little structure in the keys, while structured index spaces are used for representing index spaces with one or more collections of dense keys (e.g. N-dimensional Cartesian grids). We now describe each of these kinds of index spaces in turn.

Unstructured index spaces capture arbitrary collections of keys and allow for the dynamic allocation and freeing of opaque keys that we call *pointers*. Note that these pointers are significantly different from traditional pointers in a language such as C. Pointers in C imply a specific location in memory where data has been allocated. In Legion, pointers are simply keys for accessing logical regions associated with a

given index space. Thus pointers in Legion are portable and can be used regardless of where a logical region has been mapped. One similarity with C pointers is that Legion permits dynamic allocation and the freeing of pointers in an unstructured index space, much like how C permits dynamic memory allocation and freeing. The logical regions in our circuit simulation use unstructured index spaces due to the irregular nature of the graph. Both the logical region containing node data and the logical region containing wire data are based on unstructured index spaces and pointers are allocated in these index spaces as necessary based on the topology of the graph.

In contrast to unstructured index spaces, structured index spaces are represented by one or more Cartesian grids of points. If more than one grid is specified, they must all be of the same dimension. We use these kinds of index spaces for describing applications with more regular structures such as linear algebra and applications that work on regular grids like S3D (see Chapter 11).

The Legion programming model is also open to extension in the variety of kinds of index spaces. Presently, only two kinds of index spaces are supported; however, we envision having many different kinds of index spaces for representing different sparsity patterns. For example, in the case of sparse matrix applications, it is useful to represent different sparse matrices in different ways based on the local structures they contain within them. For these cases we want Legion to be able to support index spaces that capture different ways of efficiently representing keys for different entries in a logical region.

It is important to note that our design of index spaces strictly adheres to our first design principle of separating policy from mechanism. Legion applications can completely control the kind of index spaces when creating logical regions, thereby specifying the policy to use for different kinds of data. Independently, the mechanisms for implementing each of the various kinds of index space are completely contained within the Legion runtime and opaque to the programmer, thereby allowing Legion to manage the complexity of the implementation and to incorporate optimizations for specific target architectures.

2.2.3 Field Spaces

To represent the columns (also called fields) of logical regions, Legion uses *field spaces*. Field spaces encapsulate the set of all fields available in a logical region. Each field is defined by a pair of values: a unique name for the field (usually an integer), and the type of the field. The type of a field can be either a plain old data (POD) type (in the traditional C sense), or a compound type. Legion users are free to decide the whether fields should consist of POD base types or compound types with the understanding that the finest granularity at which Legion will be able to reason about data is at the granularity of individual fields.

Field spaces are another example of the decoupling of policy and mechanism in Legion. Field spaces allow users to control the policy of what constitutes a field in a field space, while Legion provides the mechanisms for how field spaces are implemented and used. For our circuit simulation each POD type associated with either a node or a wire (e.g. charge, voltage, capacitance, etc.) is made into an individual field in a field space, thus giving Legion total visibility of all the fields within both the node and wire logical regions.

One important restriction of field spaces in Legion is that we require that the maximum number of fields in a field space be statically upper bounded. This restriction is necessary for an efficient Legion implementation, described in more detail in Section 4.3.3. It is important to note that this restriction does not limit the expressivity of Legion as applications are free to create an unlimited number of field spaces (and corresponding logical regions) and therefore the total number of fields in any application is unbounded. The restriction only places a cap on the maximum number of fields in a single field space.

2.2.4 Region Trees

Logical regions in Legion are created by taking the cross product of a field space with an index space. Each invocation of this cross product generates a new logical region. Logical regions created in this way can also be partitioned into logical sub-regions

that we discuss in Section 2.3. We refer to a logical region and all of its logical sub-regions as a *region tree*. Since each cross-product of a field space with an index space results in a new logical region, we assign to each region tree a *tree ID*. Therefore, each logical region can be uniquely identified by a 3-tuple consisting of an index space, a field space, and a tree ID. An alternative design would have been to define each cross-product of a field space with an index space as a unique logical region. Ultimately, we decided that it is beneficial in many cases to define multiple logical regions based on the same field space and index space. We will give an example of this pattern in Chapter 11.

The dynamic nature of logical regions is important. Field spaces, index spaces, and logical regions can all be created and destroyed at runtime. This gives applications the freedom to dynamically determine both the number and properties of logical regions. By making all operations associated with index spaces, field spaces, and logical regions dynamic, Legion is well positioned to react to the dynamism in both software and hardware described in Section 1.3.2.

2.3 Partitioning

When writing code for distributed memory architectures, one of the most important components of an application is determining how data is partitioned. Traditionally, partitioning of data is done to assign different subsets of data to different nodes. In Legion, partitioning is designed to be a more general operation, not tied to an actual distribution of data to specific nodes. Instead, partitioning in Legion is an operation that is used to name different subsets of data that will be used by different computations. Drawing an analogy to relational database systems, logical sub-regions are similar to views onto a subset of a relation. Furthermore, Legion supports hierarchical partitioning, allowing subsets to be further refined, in order to describe tiered sets of data. By creating hierarchical logical sub-regions through partitioning, applications can precisely scope the set of data used by different computations.

In keeping with our first design principle, Legion does not provide any automated partitioning schemes. Instead it is the responsibility of the application to determine

the best way to partition data for the computation it intends to perform, thereby specifying the policy for how to partition data. In conjunction, Legion provides the mechanism for capturing the result of the partition in terms of logical regions. For example, in the circuit simulation, the partitioning of the circuit graph into pieces could be done by a third party library such as METIS. The results of the computed partitioning are then captured by Legion in the form of partition objects with logical sub-regions.

Creating partitions with logical sub-regions is done with *colorings*. Colorings are objects that describe an intended partition of an index space. Technically, a coloring is a map from *colors* to sets of points in an index space. For unstructured index spaces, colorings are maps from colors to sets of individual pointers, while for structured index spaces colorings are maps from colors to one or more Cartesian groups of points. For each color in a coloring, Legion will create one logical index sub-space of the parent index space. For every logical region based on the parent index space, a corresponding logical sub-region will exist for the newly created index sub-space. Figure 2.1 illustrates this process for the circuit simulation.

Partitions in Legion are normally used to create multiple logical sub-regions simultaneously. Legion records two important properties about the set of logical sub-regions created in a partition. First, Legion records whether the index sub-spaces that are created by a partition are *disjoint*. A partitioning is defined to be disjoint if each entry in the parent index space of the partition is assigned to at most one of the index sub-spaces. Legion does permit colorings in which a single point in the parent index space is assigned to multiple index sub-spaces. Partitions containing points colored more than once are called *aliased*, since some index sub-spaces contain the same point[2].

The second property of partitions tracked by Legion is *completeness*. If every point in a parent index space is mapped to at least one of the index sub-spaces, then the partition is said to be complete, otherwise it is incomplete[3]. We will see how

[2]More mathematically inclined readers may prefer to think of colorings as a relation (in the mathematical sense), with the relation being considered disjoint if it is a function, and aliased otherwise.

[3]For mathematically inclined readers, this property is surjectivity.

Legion benefits from tracking these properties of partitions in Chapters 4 and 5.

2.3.1 Tunable Variables

Another example of separating specification from mapping in the design of Legion is *tunable variables*. In many cases, partitioning is done based on the size of the machine and the number of processors available for different computations, all of which are aspects of the target machine. There obviously exists a natural tension with our second design goal that requires algorithm specification to be decoupled from the mapping. To handle this decoupling we borrow an idea from the Sequoia programming model [28], and provide tunable variables as a feature of Legion.

Tunable variables are assigned a value at runtime by a Legion mapper. While we defer our discussion of Legion mapper objects until Chapter 8, for now it is sufficient to know that Legion mappers can determine the value of tunables based on both machine and application specific information. By explicitly describing a variable as a tunable, applications can decouple the specification of an application from its mapping, ultimately allowing mappers to determine optimal values for tunables. For example, in the circuit simulation, a tunable variable is used to determine the number of pieces to direct METIS to compute for the circuit graph. The mapper that picks this value is likely to make the determination of the tunable value based on the number of nodes or number of processors in the target machine. For example, in our circuit simulation we use the tunable variable `num_pieces` (line 12 in Listing 2.1) to determine the number of pieces to create when partitioning the circuit graph. The mapper will likely determine the value for this tunable variable based on the number of nodes or processors in the machine.

There are two important details regarding tunable variables. First, tunable variables are not specific to partitioning, and can be used for any purpose in any task. This allows applications to explicitly declare any variable that should be explicitly chosen by a mapper as a tunable variable. Second, unlike the statically-specified tunable variables in Sequoia, tunable variables in Legion are dynamically determined. This allows mappers to change the values that it assigns to tunable variables across

different task instances and react to changes in the hardware or software as part of assigning specific values to tunable variables.

2.3.2 Multiple Partitions

One of the most important aspects of the Legion programming model is that it supports multiple partitions of an index space. Multiple partitions allow applications to describe many different views onto the same collection of data. Supporting multiple partitions is important because most data in applications is viewed in different ways depending on the phase of the computation. For example, in the circuit simulation, there are shared nodes that have at least one edge that crosses a piece boundary. In some cases, these shared nodes need to be accessed by computations on the piece that owns them, while in other cases, shared nodes are accessed as ghost nodes for an adjacent piece. Being able to capture these multiple views onto the same data is crucial to capturing the data usage patterns of real applications.

2.3.3 Recursive Partitions

Another important aspect of Legion is that it supports recursive partitions. Applications can recursively partition index sub-spaces into further index sub-spaces, thus creating arbitrarily deep index trees and corresponding logical region trees.

The ability to create recursive partitions is important for two reasons. First, recursive partitions allow applications to partition data in a way that aligns with deep memory hierarchies, which was one of the key insights of the Sequoia programming language [28]. Second, supporting recursive partitioning is necessary for capturing non-trivial subsets of data. For example, the circuit simulation uses two levels of partitioning to first break nodes into all-private and all-shared nodes, and then recursively partitions those sub-regions to express the subsets of nodes in each of the pieces. This allows the logical region tree to capture important disjointness information about the different subsets of nodes.

2.3.4 Interaction with Allocation

Partitioning in Legion is flexible enough to support two different approaches that we term *allocate-then-partition* and *partition-then-allocate*. The circuit simulation is an example of allocate-then-partition. The circuit graph is initially allocated in logical regions for describing the nodes and wires. We then partition these logical regions into logical sub-regions. The allocate-then-partition strategy is used for data structures that are constructed up front and then later need to be partitioned into sub-regions.

Alternatively, the partition-then-allocate strategy can be used for creating data structures in parallel. Index trees and corresponding logical region trees can be created in which no rows are allocated. This is done by using completely empty colorings. Rows can then be allocated in different index sub-spaces in parallel, allowing data structures to be created in logical regions in parallel. When a row is allocated in a logical sub-region, it is automatically allocated in all of the ancestor regions of the logical region (in keeping with the definition of logical regions). The partition-then-allocate approach is especially useful for constructing data structures in parallel when the size of the data structure is sufficiently large that the parent logical region cannot be manifested in any memory in the target machine.

It is important to note that partition-then-allocate and allocate-then-partition are not exclusive. Well-structured Legion applications employ a mixture of strategies to a create logical region trees. First, partition-then-allocate is used to load data structures into logical regions in parallel. Next, new partitions are created of the existing logical regions to describe alternative views onto the already allocated data.

2.3.5 Region Tree Properties

As a result of partitioning, Legion applications can create arbitrary index space trees and corresponding logical region trees for describing the structure of program data. Logical region tree data structures are the crux of the Legion programming model. By creating logical region trees through partitioning, Legion applications can encode important information about the structure of program data so that it can be communicated to an implementation of Legion. Specifically, logical region trees capture

the following properties:

- Locality - entries in logical regions have an implied locality. Partitioning rows in a logical region into the same logical sub-region further indicates that a task is likely to use the same rows, strengthening the implied locality.
- Disjointness - partitions capture information about which logical regions share data. Disjoint partitions aid Legion in inferring when two logical regions are independent.
- Sub-Regions - knowing sub-region relationships allows Legion to reason about subsets of data.
- Aliasing - as the complement to disjointness, logical regions also allow Legion to infer when two logical regions may potentially be aliased, either by aliased partitions or by sub-region relationships.

As we will see in the coming chapters, a Legion implementation can leverage all of the information encoded in logical region trees to both achieve high performance, and make developing Legion applications easier.

Another important aspect of logical region trees is that they capture these properties of program data in a way that is independent of the target machine. Logical region trees are not tied to the hardware in any way, thereby allowing applications to decouple the specification of an application from how it is mapped onto a target architecture.

2.4 Motivating Example: Conjugate Gradient

Before proceeding with our description of the rest of the programming model, we pause briefly to introduce another example that will be useful for understanding some of the features introduced in the coming sections. Listing 2.2 shows the basic outline of code for a simple conjugate gradient (CG) solver written in Legion. The code operates on a sparse matrix a (stored in a logical region) and a vector v (stored in another logical region). The solver iterates up to a maximum number of steps to find

```
1  struct SparseMatrix {
2    region lr;
3    partition⟨disjoint⟩ part;
4    int num_rows, elmts_per_row;
5  }
6  struct Vector {
7    region lr;
8    partition⟨disjoint⟩ part;
9    int num_elmts;
10 }
11 void conjugate_gradient(SparseMatrix a, Vector x) : RWE(a.lr, v.lr) {
12   tunable int num_pieces;
13   a.part = a.lr.partition(num_pieces);
14   x.part = x.lr.partition(num_pieces);
15   Vector r_old(x.n_elmts), p(x.n_elmts), A_p(x.n_elmts);
16   // ... create and partition logical regions with num_pieces
17
18   // Ap = A * x
19   spawn⟨num_pieces⟩ spmv(a.part, x.part, A_p.part);
20   // r_old = b - Ap
21   spawn⟨num_pieces⟩ subtract(b.part, A_p.part, r_old.part);
22   // initial norm
23   const double L2Norm0 = spawn⟨num_pieces⟩ L2norm(r_old.part);
24   double L2norm = L2Norm0;
25   // p = r_old
26   copy(r_old, p);
27
28   predicate loop_pred = true;
29   future r2_old, pAp, alpha, r2_new, beta;
30   for (int i = 0; i ⟨ MAX_ITERS; i++) {
31     // Ap = A * p
32     spawn⟨num_pieces⟩ @loop_pred spmv(A.part, p.part, A_p.part);
33     // r2 = r' * r
34     r2_old = spawn⟨num_pieces⟩⟨+⟩ @loop_pred dot(r_old.part, r_old.part, r2_old);
35     // pAp = p' * A * p
36     pAp = spawn⟨num_pieces⟩⟨+⟩ @loop_pred dot(p.part, A_p.part, alpha);
37     // alpha = r2 / pAp
38     alpha = spawn @loop_pred divide(r2_old, pAp);
39     // x = x + alpha * p
40     spawn⟨num_pieces⟩ @loop_pred daxpy(x.part, p.part, alpha);
41     // r_old = r_old - alpha * A_p
42     spawn⟨num_pieces⟩ @loop_pred daxpy(r_old.part, A_p.part, -alpha);
43     r2_new = spawn⟨num_pieces⟩⟨+⟩ @loop_pred dot(r_old.part, r_old.part, r2_new);
44     beta = spawn @loop_pred divide(r2_new, r2_old);
45     spawn⟨num_pieces⟩ @loop_pred daxpy(r_old.part, p.part, beta);
46     future norm = spawn⟨num_pieces⟩⟨+⟩ @loop_pred dot(r_old.part, r_old.part, L2norm);
47     loop_pred = spawn @loop_pred test_convergence(norm, L2norm) : false;
48   }
49 }
```

Listing 2.2: Conjugate Gradient Example

a vector x such that $ax = v$. Lines 18-26 show the set-up stages of the computation and lines 30-48 comprise the main iterative loop.

To solve the system in parallel, this example partitions the sparse matrix by rows and the dense vector into pieces (specified by the tunable variable on line 12) that correspond to the number of rows. Additional logical regions are created and partitioned isomorphically for storing intermediate values. For every operation of the CG solver a parallel task is launched to handle the different subsets of the computation. While the partitioning scheme for this example is straightforward, we will see shortly how index space task launches, futures, predicates, and explicit region-to-region copies are all necessary for implementing the CG solver efficiently.

2.5 Tasks

While logical regions provide the crucial abstraction in Legion for describing data in a machine independent way, tasks serve the same purpose for computations. Legion tasks describe computations on two kinds of arguments: parameters that are passed by value (much like traditional functions) and logical regions. By describing computations as tasks that operate on logical regions, Legion programs can specify an application in a way that is independent from any target architecture. In this section we cover the details of how Legion applications describe tasks.

2.5.1 Privileges

Unlike most programming models that maintain a heap abstraction that can be implicitly accessed by any computation, Legion requires each task be explicit about the data that it can touch by naming the logical regions the task will access. Legion tasks name the regions they access through *region requirements*. When a Legion task is launched, it provides a set of region requirements.

In addition to naming the logical regions that a task will access, region requirements also specify the *privileges* with which the task will access each field of each

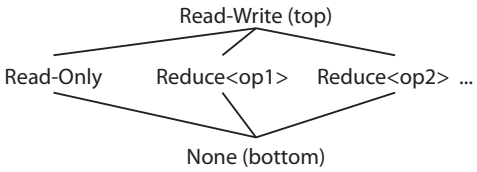


Figure 2.2: Legion Privileges Semi-Lattice

region. Privileges in Legion are one of `read-only`, `read-write`, or `reduce` (with reductions parametrized by a reduction operator). Privileges specify the kinds of side-effects that the task is permitted to have on the specific fields of the logical region in a region requirement. For example, in the circuit simulation, each `calc_currents` task requests `read-write` privileges on several fields of its wire region, but only requests `read-only` privileges on its node region. Privileges in Legion form a semi-lattice shown in Figure 2.2 [4]. As we will describe in Section 2.5.3, the use of privileges is important for the discovery of parallelism in a Legion program.

2.5.2 Sub-Tasks

The model of computation in Legion is a tree of tasks. A root task is initially executed on some processor in the system and this task can launch an arbitrary number of sub-tasks. Each sub-task can in turn launch its own sub-tasks. There is no limit to the depth of the task tree in a Legion program execution. While we note that this approach is different from the traditional SPMD programming model for targeting supercomputers, we claim that SPMD is a special-case of the Legion programming model and can be easily replicated using existing features in Legion, as we will describe in Chapter 9.

The Legion programming model requires that sub-tasks within a Legion application maintain the following property: sub-tasks are only permitted to access fields of logical regions with a subset of the privileges of their parent task. We call this property the *containment property*. To be more concrete, in order for the containment property to hold for a sub-task, then all of the following must be true for each logical region a sub-task requests privileges on:

[4]Note that each different kind of reduction forms its own entry in the semi-lattice of privileges.

- The parent task must also have privileges on the specified logical region.
- For each field, the parent task must also have privileges on the specified field.
- The parent task must have privileges that are a (non-strict) superset of the privileges requested by the child task as defined by the privilege semi-lattice in Figure 2.2.

In some ways, the containment property resembles functional programming models, in which sub-functions only have access to the data directly passed from their parent function. The only difference being that Legion permits tasks to explicitly side-effect logical regions in accordance with the declared privileges. The need for the functional nature of the containment property will become evident in Section 3.2.3 when we describe the hierarchical scheduling algorithm that it enables.

As a brief aside, we note that the containment property required of Legion programs makes Legion an interesting hybrid between a functional and an imperative programming model. At a coarse-granularity (between tasks), Legion appears like a functional model with tasks only accessing subsets of data that their parent task had access to. However, at a fine-granularity tasks are composed of statements that mutate the state of logical regions, much like imperative programs manipulating data in one or more heaps. In many ways, this approach makes porting code to Legion easier. Existing imperative code can be wrapped inside of a task, and only the dependences between tasks based on region usage need be considered at a functional level by a Legion implementation.

To this point, we have yet to describe how privileges are initially created in a Legion program. When a task creates a logical region it is granted full `read-write` privileges for all fields in the logical region. It is then free to pass these privileges through sub-task calls to child tasks. If the creating task also returns a region it allocated, the task returns the full `read-write` privileges to its enclosing parent task. It is important to note that this happens automatically and is tracked implicitly by a Legion implementation. The implicit nature of privilege tracking is important because it guarantees that privileges cannot be stored anywhere in memory or passed to other sub-tasks in any way except via sub-task calls and tasks completing. This aspect of

privileges will be a crucial element in enabling the implementation of the hierarchical scheduling algorithm described in Section 3.2.3 because it implies privileges are not first-class objects, and can therefore always be tracked by a Legion implementation.

In our circuit simulation, each `simulate_circuit` task initially holds read-write privileges on both of the logical regions that contain all of the node and wire data for the circuit graph. Therefore, each of the sub-tasks launched by `simulate_circuit` for the main time-step loop meet the functional property required by the Legion programming model because all tasks are asking for a sub-set of privileges that the parent task holds.

2.5.3 Execution Model

Before proceeding with our discussion of the remaining features associated with tasks, we briefly introduce the Legion execution model. In Legion, all sub-tasks (and other operations discussed in Section 2.7) are launched asynchronously. It is the responsibility of the Legion runtime to maintain sequential program order semantics for these sub-tasks and operations. To achieve high performance, Legion employs implicit parallelism to discover where sub-tasks and operations can be run in parallel based on the logical regions used by sub-tasks and other operations. Legion allows sub-tasks and operations to execute in parallel when they are *non-interfering* on their logical region arguments. We give a formal definition of what it means for tasks and other operations to be non-interfering in Section 4.1. One of the primary contributions of this thesis is the data structures and algorithms for implicitly discovering parallelism based on non-interference. We cover these techniques in more detail in Chapters 4 and 5.

Ensuring that all Legion sub-tasks and other operations are asynchronous is a key component of the Legion design. By guaranteeing that Legion is fully asynchronous, Legion applications can launch many outstanding sub-tasks and other operations, allowing a Legion implementation to automatically discover as much parallelism as possible. Discovering large amounts of independent work is crucial to hiding the large latencies present in modern supercomputers. Finding significant parallel work

also allows a Legion implementation to hide much of the latency associated with its dynamic analyses. To the best of our knowledge, this fully asynchronous execution model has no name in the academic literature. We refer to it as *deferred execution*.

The essence of a deferred execution model is the promise that all operations are asynchronous and it is the responsibility of the Legion implementation to maintain the sequential program order semantics of an application. In order to avoid exposing latencies, applications should refrain from performing blocking operations that might impede a Legion implementation from discovering additional parallelism. As we will see later in this section, there are several ways that tasks can block their execution. While it is encouraged for tasks to run arbitrarily far into the future to expose many sub-tasks and operations, there is naturally a limit to the number of operations that should be in flight from the execution of a given task. We discuss some of the features that Legion employs for limiting the number of sub-tasks and operations in flight in Sections 3.2.1 and 8.1.

Our circuit simulation exemplifies the benefits that can be obtained from deferred execution. Since all task launch operations are performed asynchronously, the execution of the the loops on lines 32-36 of Listing 2.1 can be unrolled arbitrarily far, allowing the Legion runtime to see a large window of potential tasks to run. As we will see in later chapters, the deferred execution model will allow a Legion implementation to pipeline many of the operations associated with executing a task and thereby hide the latency both of dynamic program analysis and communication between nodes.

2.5.4 Index Space Tasks

While the standard way of launching sub-tasks from a parent task is to launch a single task, Legion also provides a mechanism for launching many sub-tasks simultaneously. Legion permits applications to issue *index space task launches* where many tasks are launched with a single call. Index space task launches are useful because they allow for many tasks to be put in flight simultaneously, which can help in reducing runtime overheads when many child tasks need to be launched. Furthermore, index space task launches allow applications to express important properties about groups of tasks

including the requirement that they all be capable of synchronizing with each other throughout their execution (see Chapter 9 for more details). When an index space task launch is performed, the application specifies an index space of points with the understanding that Legion will create a single task for each point in the index space. There are numerous index space task launches in the CG solver shown in Listing 2.2 (lines 19, 21, 34, 36, 40, 42, 43, 45, 46, and 47). In each case we launch `num_pieces` different tasks for each of the different chunks of the vectors being computed.

In order to aid applications in describing index space task launches, Legion supports a special kind of region requirement called a *projection*. Projection region requirements are different from normal region requirements in that they do not need to explicitly name the region requirements for the task corresponding to each point in the index space launch. Instead projection region requirements allow index space task launches to name either a logical region or logical partition to serve as an upper bound on logical regions on which tasks within the index space task launch will request privileges. In the CG solver in Listing 2.2, all of the index space task launches use partition projection requirements with the partitions of the vectors and/or sparse matrix as the upper bound.

Projection region requirements also provide a function to be invoked by a Legion implementation which takes as arguments the upper bound logical region or partition and a point in the index space launch, and returns the specific logical region to be used for the particular task. The projection function must be pure, but it is provided information about the shape of the logical region tree that it is traversing in order to aid its computation. For the most common cases, Legion provides a built-in projection function that linearizes all the points in the index space and maps them onto the individual sub-regions of a partition modulo the total number of sub-regions. This built-in function is sufficient for handling all of the projection requirements in the CG solver example and is therefore not shown.

Having upper bounds on the logical regions that all tasks in an index space task launch will access enables a Legion implementation to optimize some aspects of dependence analysis that we will discuss in Chapter 4. While index space task launches allow many tasks to be launched in parallel, the Legion programming model does

restrict the regions that individual points access to be non-interfering with all other points in the same index space launch.

2.5.5 Futures

Much like traditional functions, Legion tasks are permitted to return values. When a task call is performed in Legion it does not block. This is an essential component of the Legion deferred execution model. The asynchronous nature of tasks in Legion necessitates that a place-holder object be used to represent the return values from a task. As is common in the literature, we refer to these values as *futures*.

In the case of index space task launches, a *future map* can be returned which contains a single future entry for each point in the index space of tasks launched. Applications can then extract these futures and use them as if they had been generated by a single task launch. Alternatively, in some cases, applications might want to reduce all the futures from an index space task launch down to a single value. For example, consider the dot-product computations done by the CG solver in Listing 2.2 (line 34). Each task in the dot-product index space task launch computes a sum for a different part of the vectors. All of these values need to be reduced down to a single value for the dot product, therefore a sum reduction (indicated by `<+>` on the index space launch) is passed as the reduction function to use for combining the results of an index space tasks' return values into a single future value.

Future values can be used in one of three different ways. First, a parent task that has a future value for one of its child tasks can explicitly wait on the future result. This blocks the parent task until the future is *complete* (when the child task completes and returns the value). Waiting on a future should in general be an uncommon operation in Legion as it prevents the parent task from continuing execution and exposing additional work to the Legion runtime. Waiting on a future result also has ramifications for resilience which we discuss in Chapter 10. Futures are not first-class objects and are not permitted to be stored in logical regions. Applications must explicitly wait on a future result to write the value into memory.

The second way that a future can be used is that it can be passed as a input

parameter to another child task in the same parent task. Passing a future as an argument to another sub-task launch does not cause the additional sub-task launch to block. As an example, notice the `divide` function in the CG solver on lines 38 and 44. This task takes two futures and uses their values to compute a new future that is the result of dividing the first future by the second. This way of using futures is much more consistent with Legion's deferred execution model because it avoids blocking the parent task.

The third way to use a future value is to create a predicate value from it. Boolean futures can automatically be converted into predicates, while other future types can be turned into predicates with an associated function that evaluates the value of the future and returns a boolean. Line 47 of the CG solver creates a predicate from a boolean future value returned by the `test_convergence` task. As we will see in the next section, these predicates can be used for predicating task execution.

2.5.6 Predicated Execution

One of the more important features of Legion is its ability to allow sub-task launches to be predicated. Predicated sub-tasks allow parent tasks to continue executing and launching sub-tasks without needing to block to resolve conditional values (e.g. for `if` statements or loops). For example, consider the CG solver in Listing 2.2. We start out with a `loop_pred` predicate that is used to predicate all task launches in the iterative loop which we indicate by the syntax `@loop_pred` in any task launch. The predicate starts out as `true` and then is converted from a future value that is returned by the `test_convergence` task (line 47) at the end of every iteration. In this way, the CG solver unrolls the loop and issues tasks even without knowing whether the computation has converged or not. This permits the runtime to begin the analysis for these tasks even before it knows if they will ultimately be executed. In an optimized version of the CG solver, we use a tunable variable to control how far ahead we want to unroll the loop. We further discuss the need for predicates in Section 2.5.6.

A separate issue involving predication is whether or not a Legion implementation

is permitted to speculate on the values of predicates in order to start executing predicated tasks before the predicate value is known. In keeping with our second design principle, the decision of when and where to speculate is independent of the specification of the algorithm for an application and it is therefore determined by Legion mapper objects. We discuss how speculation can be directed by a Legion mapper in Section 10.2.1.

2.5.7 Task Variants

Determining the correct algorithm to use for different applications is not always an obvious decision, especially in the context of different target architectures. Legion therefore allows applications to register multiple functionally equivalent *variants* of the same task. This idea is borrowed from Sequoia [28]. By specifying multiple functionally equivalent variants for the same task, Legion allows applications to explore different algorithms as part of the process of mapping an application to a target architecture. We discuss how Legion mappers can select which variant of a task to use in Section 8.1.3.

Another important property of Legion task variants is that they can be used to specialize both task implementations as well as algorithms for a particular target architecture. Applications can explicitly register different task variants for a target processor kind. Alternatively, the Legion programming model also permits the registration of task *generators*. Task generators are meta-functions capable of generating specialized variants given constraints on the kind of processor and the layout of the physical instances for the task. We cover task generators in greater detail in Section 8.1.5.

2.5.8 Task Qualifiers

The final aspect of Legion tasks is the ability for applications to add qualifiers on task variants. Task qualifiers provide additional information to the Legion runtime regarding operations that task will (or will not) do at runtime that can aid in performance optimizations. There are three different kinds of qualifiers that can be associated

with a task: *leaf*, *inner*, and *idempotent*. We cover each qualifier in turn. Note that tasks are not required to specify any of these qualifiers, as the qualifiers only provide opportunities for performance optimizations for a Legion implementation.

The first qualifier that can be associated with a task variant is whether the task variant is a *leaf* task. A leaf task is a task variant that is only permitted to operate on the region data that it requested, and not to perform any Legion operations. Since it performs no Legion operations, it is not permitted to launch any sub-tasks and is therefore always a leaf in the dynamic task tree. Leaf tasks allow a Legion implementation to reduce the meta-data associated with a task.

The second qualifier that can be associated with a task variant is whether the task variant is an *inner* task. An inner task is the opposite of a leaf task. Inner tasks are not permitted to actually access the logical regions they requested privileges on, but are only allowed to launch sub-tasks and pass privileges through to child tasks. Inner tasks can perform any arbitrary Legion operations that are necessary as long as they do not access logical region data. Inner tasks allow a Legion implementation to begin executing a task even before its logical region data is ready in order to discover additional task parallelism generated by the inner task.

The final qualifier that can be specified for a task variant is whether the task variant is *idempotent*. A task is idempotent if the only side-effects that it performs are modifications to logical regions. If the task performs alternative side effects (e.g. writing files, printing output, etc.) then it cannot be considered idempotent. In general the majority of tasks in a Legion application should be idempotent with only a few tasks accessing external resources. Idempotent tasks enable resiliency optimizations that are discussed in more detail in Chapter 10.3.

In our circuit simulation from Section 2.1, all of the sub-tasks launched for calculating currents, distributing charge, and updating voltages could be annotated as leaf tasks because they do not launch any sub-tasks. These tasks could also be labeled as idempotent because they only mutate the logical regions for which they have requested privileges. The `simulate_circuit` task is a different; depending on whether it reads and/or writes information about the circuit to/from a file determines whether it is idempotent or not.

2.6 Physical Instances

While logical regions provide names for describing different sets of program data, the execution of tasks requires access to actual data in a specific place. *Physical regions* describe an actual instance of a logical region that resides in a specific memory in the memory hierarchy and represents a specific layout of the data contained within the logical region. The Legion programming model guarantees that when a task starts it will have physical regions for each of the region requirements that it requested. The choice of where these physical regions are located and how they are laid out in memory is determined by Legion mapper objects discussed in Chapter 8, but Legion mappers are only permitted to map physical instances into memories that are visible from the target processor for a task.

2.6.1 Logical to Physical Mapping

The distinction between logical and physical regions is unfamiliar to programmers new to Legion, but there is a simple analogy to conventional languages. In most modern programming languages, compilers automatically manage the mapping from variables to physical registers used in machine assembly code. Logical and physical regions are analogous to variables and registers. Currently, it is the responsibility of the Legion application developer to manage both logical and physical regions in the same application[5].

One important detail regarding physical regions is that they must be *unmapped* before launching sub-tasks that use the corresponding logical region that the physical region represents. Unmapping a physical region before launching a sub-task that uses the same logical region is necessary to avoid data races and consistency problems between a parent task and a child task accessing the same data. In practice, failure to unmap regions used by child tasks can result in a deadlock where a child task depends

[5]We note that as of the time of the publication of this thesis, a Legion language is in development that does not require a distinction between logical and physical regions. Instead the Legion language compiler can automatically manage the logical and physical regions of the Legion programming model automatically, much in the same way that modern compilers manage the mapping from variables to hardware registers.

on its parent task and the parent task cannot complete until the child task completes. To maintain the correctness of a Legion application, a Legion implementation should detect when a sub-task is accessing a logical region mapped by the parent and issue the corresponding unmap and re-map operations immediately before and after the sub-task launch. Applications can optimize performance by unmapping physical regions and then re-mapping them only when they need to be accessed. We discuss how mapping and unmapping operations are performed in Section 2.7.

2.6.2 Accessors

To maintain one of the Legion core design principles a necessary level of indirection is required to access data from physical regions. Physical regions are located in a specific memory in the memory hierarchy and have a given layout. Both of these properties of a physical region are determined by the mapping process. To maintain the invariant that a Legion application is machine-independent, we require data from physical regions to be accessed using *accessor* objects. Accessors abstract the details of physical region locations and data layouts when performing reads, writes, and reductions to a physical instance. In many cases, accessors can be specialized at runtime based on physical region properties, but Legion also provides general accessors capable of accessing any physical region.

In addition to accessors, Legion task variants can also be specified with generators that do not require accessors. Using meta-programming techniques generators can emit specialized code for specific physical regions at runtime without requiring general and specialized accessor objects for different physical regions. We discuss generators in more detail in Section 8.1.5.

2.7 Operations: Mapping, Copies, and Barriers

In addition to allowing parent tasks to launch sub-tasks, the Legion programming model also allows parent tasks to issue other kinds of operations. Some of these operations function primarily as productivity features, while others are also useful for

achieving high performance. For example, inline mapping operations are mainly a productivity operation because they allow applications to immediately gain access to a physical instance of a logical region after the region has been created. Alternatively, explicit region-to-region copies are valuable for performance as they allow applications to explicitly describe data movement between regions while allowing the runtime to optimize the implementation of these data transfers.

In keeping with Legion's deferred execution model all of these operations are issued asynchronously. It is the responsibility of a Legion implementation to properly order these operations with respect to other operations in the stream of sub-operations. A Legion runtime will automatically infer dependences between operations based on region requirements.

2.7.1 Inline Mappings

A natural operation that Legion supports (and one that was omitted from earlier Legion versions), is *inline mappings*. In older versions of Legion, the only way to map a physical instance of a logical region was to launch a sub-task that would map the logical region as part of the task being mapped. However, in many cases, it is useful for tasks to decide to map a logical region locally as part of their execution. For example, it would be natural for the `simulate_circuit` task from our circuit simulation to perform an inline mapping when loading a circuit topology from a file. To handle mapping and unmapping of logical regions within a task, inline mapping and unmapping operations are performed.

An inline mapping operation takes a region requirement similar to sub-task launches, but instead returns a handle to a physical region that will correspond to a mapped physical instance once the operation completes. In many ways the physical region handle acts like a future. Before an application can create an accessor to the physical instance it must explicitly wait for the region to be ready. We note that needing to explicitly wait on the physical region to be ready runs counter to deferred execution, but there is no alternative when the data in a logical region must be explicitly accessed by a task. To encourage the discovery of as much parallelism as possible, we make

inline mapping launches asynchronous, thereby allowing applications to perform as much additional work as possible before it becomes necessary to wait for a physical region to be ready.

In addition to inline mappings, Legion also allows applications to launch unmap operations for physical regions. Unmapping operations enable tasks to indicate that they are no longer accessing a physical region, allowing other operations such as subtasks to avoid dependences on the parent task. Unmap operations can be used to unmap physical regions that were mapped either directly by the parent task or by inline mapping operations. Once a physical region has been unmapped, it can always be remapped using an inline mapping operation.

2.7.2 Explicit Region Copies

Another operation that has been useful for developing Legion applications is the ability to issue explicit copies between fields in different logical regions. In general, most Legion applications allow the Legion runtime to implicitly issue data movement operations based on logical region uses. However, some applications explicitly manage different logical regions for the same data. As a simple example, the CG solver in Listing 2.2 issues an explicit region-to-region copy operation on line 26 to initialize the data in a new logical region. A more substantial example of the use of explicit copy operations will be the S3D application that we introduce in Chapter 11 that maintains explicit ghost cell logical regions that are updated using explicit region-to-region copies.

Like other operations, explicit region-to-region copies are issued using region requirements to describe the source and destination logical regions as well as the fields to be copied between them. Legion requires that the destination logical region have an index space that is *dominated* by the index space of the source logical region. An index space i_0 dominates and index space i_1, if i_0 contains all of the points of i_1. The two index spaces do not need to come from the same index space tree, but they do need to be the same dimension to check for dominance. Checking for dominance guarantees that a full copy of the data will be manifested in the destination logical region

and ensures that no values in the destination region are left undefined by the copy. Similar to both sub-tasks and inline mapping operations, the region requirements for copy operations are mapped by a mapper object.

2.7.3 Execution Fences

The final kind of operation that Legion supports are *execution fences*. Execution fences provide a means for applications to delineate sets of operations and sub-tasks into epochs, either for the purpose of mapping or execution. It is important to note that execution fences are not barriers. Execution fences do not block execution of a parent task, but instead are issued asynchronously like all other operations into the stream of operations and sub-tasks.

The Legion programming model supports two kinds of execution fences: *mapping* fences and *execute* fences. Mapping fences require that all operations prior to the fence have completed mapping before any operations after the mapping fence are permitted to begin mapping. In Chapter 4 we will see two examples of how mapping fences can be used for optimizations. Mapping fences are also useful for providing context information to mapper objects by creating different epochs of a parent task's execution (e.g. operations and sub-tasks corresponding to specific time-steps).

Alternatively, execute fences ensure that all sub-operations in a parent task have finished executing before any operations after the fence are permitted to execute. Execute fences provide a mechanism for adding additional constraints on the execution order of operations (in addition to logical region data dependences and future dependences). Execute fences are a very blunt tool and should only be used at a coarse granularity for controlling execution.

2.8 Machine Independent Specification

Throughout this chapter we have introduced many features of the Legion programming model. In the process we have highlighted important decisions made in the design that reflect our initial design principles laid out in Section 1.4. The absence

of certain features is equally important. Our entire discussion of the Legion programming model has avoided describing where tasks are scheduled or where logical regions representing program data are placed in the memory hierarchy. In cases where variables depended on the target machine, we allow users to directly declare these variables as tunables and defer assigning values until the mapper chooses a value at runtime. Consequently, all aspects of a Legion program that may depend on the target architecture are explicitly decoupled from the Legion program. It is important to note that the only reason that this is possible is because Legion provides abstractions for both computation (tasks) and data (logical regions). While many other programming systems have abstractions for parallel computation, the novel aspect of Legion is the introduction of logical regions as the basis for an abstract data model. With logical regions, all aspects of a program specification can be completely decoupled from its mapping.

Chapter 3

The Legion Runtime Architecture

Having completed our discussion of the Legion programming model in Chapter 2, we begin in earnest describing our implementation of Legion. Legion is currently implemented entirely as a runtime system. We decided on a runtime implementation for two reasons. First, a runtime system is fully dynamic and requires no static analysis of programs, thereby allowing it to react to the dynamic nature of both modern supercomputing applications and hardware. Second, implementing Legion as a runtime system in an existing language (in this case C++) makes it easier for application developers and DSL compiler writers to port their code as they are already familiar with the target language and simply need to adapt to the new programming model.

Before delving into the details of our implementation we begin with a brief overview of the various components of the current Legion ecosystem in Section 3.1. We then introduce a useful analogy in Section 3.2 for describing how the Legion runtime system operates that will be carried through many of the following chapters.

3.1 Runtime Organization

Due to the complex nature of both applications and the underlying hardware, the Legion runtime is composed of several different components at different levels of abstraction. Figure 3.1 illustrates the different levels of the Legion software stack,

including several levels (such as the Legion language and compiler) that are still under development. The many different layers are designed to encapsulate the complexity associated with the various aspects of our Legion implementation. We now discuss each of the different levels of the software stack in further detail.

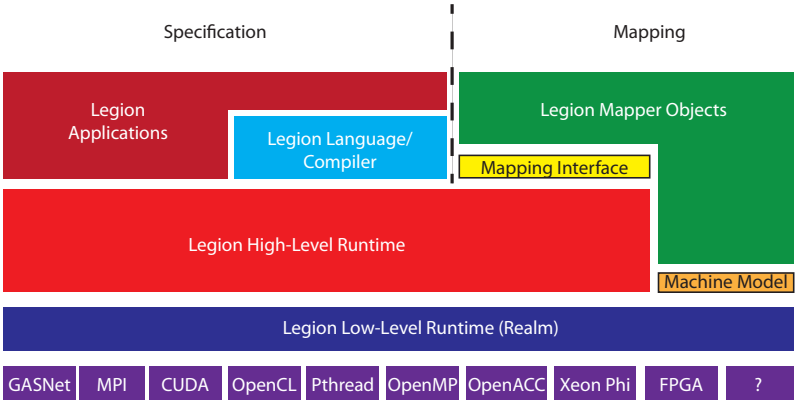


Figure 3.1: Legion Software Stack

3.1.1 Legion Applications

At the top of Figure 3.1 are Legion applications. Legion applications can be either applications directly written using Legion or Domain Specific Languages and Libraries (DSLs) that use Legion to provide additional portability and performance. Legion applications can be targeted either at the Legion runtime API or at the higher-level Legion language developed in [49]. The design and implementation of the Legion language, type system, and compiler are beyond the scope of this thesis and are not discussed further. From this point forward we will assume all Legion applications make use of the Legion runtime API.

Included in Figure 3.1 is a dividing line that delineates which components of the Legion software stack apply to writing machine-independent specifications and which apply to determining the best ways of mapping applications onto a target architecture. The features discussed in Chapter 2 are used entirely for writing machine independent specifications of applications. Alternatively, the features of the Legion

mapping interface, that are covered in detail in Chapter 8, are entirely contained on the right side of this divide. The line dividing the machine independent specification of an application from its mapping does not extend beyond the application/mapper level because it is the responsibility of the Legion runtime to mediate the interaction between machine-independent application specifications and the mapping decisions made by mapper objects.

3.1.2 High-Level Runtime

The Legion high-level runtime is an implementation of the Legion programming model described in Chapter 2. Our current implementation of Legion supports a C++ interface for writing Legion programs. While applications are written in C++, they are required to abide by the Legion programming model. Therefore, tasks are permitted to only access data in regions that they request, all sub-tasks must be launched through the runtime interface, and global variables along with dynamic memory allocation are not permitted (logical regions are used instead).

It is the responsibility of the Legion high-level runtime to take applications written in the Legion programming model and convert them into the primitive operations supplied by the low-level runtime interface discussed in Section 3.1.3. The mechanisms by which the high-level runtime performs this transformation are the primary technical contributions of this work. We begin our technical description of the high-level runtime in Section 3.2.

The other responsibility of the high-level runtime is to mediate calls to mapper objects through the mapper interface. As an application is executing, many decisions need to be made regarding where tasks should be assigned and where physical instances of logical regions should be placed. The high-level runtime directs these queries to the appropriate mapper object through the Legion mapper interface[1]. Using the results of the mapping calls, the high-level runtime then carries out the specified mapping of an application onto the target architecture.

[1]In this way the mapper interface is actually a call-back interface where methods of the mapper objects are invoked by the runtime.

3.1.3 Low-Level Runtime

The design and implementation of the Legion low-level runtime is covered in detail in [48] and is mostly beyond the scope of this thesis. However, we briefly cover the important details of the low-level runtime interface necessary for understanding the interaction with the high-level runtime.

The low-level runtime interface provides a useful set of primitives for orchestrating computation, data movement, and synchronization in a distributed memory system. Specifically, it provides the following primitive operations:

- Tasks - launch tasks on specific processors either on local or remote nodes
- Instances - allocate and free physical instances in memories
- Copies - issue copies or reductions between physical instances
- Synchronization - provide *reservations* as an atomicity primitive

The most important aspect of the low-level runtime is that all of these primitives are non-blocking. All operations are asynchronous and return immediately with an *event* that names a time in the future when the operation will be complete. Furthermore all operations in the low-level runtime can take events as preconditions that are required to trigger before an operation can be executed. By using events to compose these primitives, the high-level runtime can construct large directed acyclic graphs (DAGs) of operations.

Figure 3.2 shows an example low-level runtime DAG generated by the high-level runtime from a real application. In this image, boxes correspond to operations, while edges correspond to event dependences. Yellow boxes are copies, red boxes are reductions, blue boxes are individual tasks, purple boxes are point tasks from an index space task launch, and green boxes are inline mappings. This graph shows all of the event dependences computed by the high-level runtime. In practice many of these event dependences are pruned by the high-level runtime through optimizations that are discussed in Chapter 5, but are left in place for the purposes of debugging and visualization.

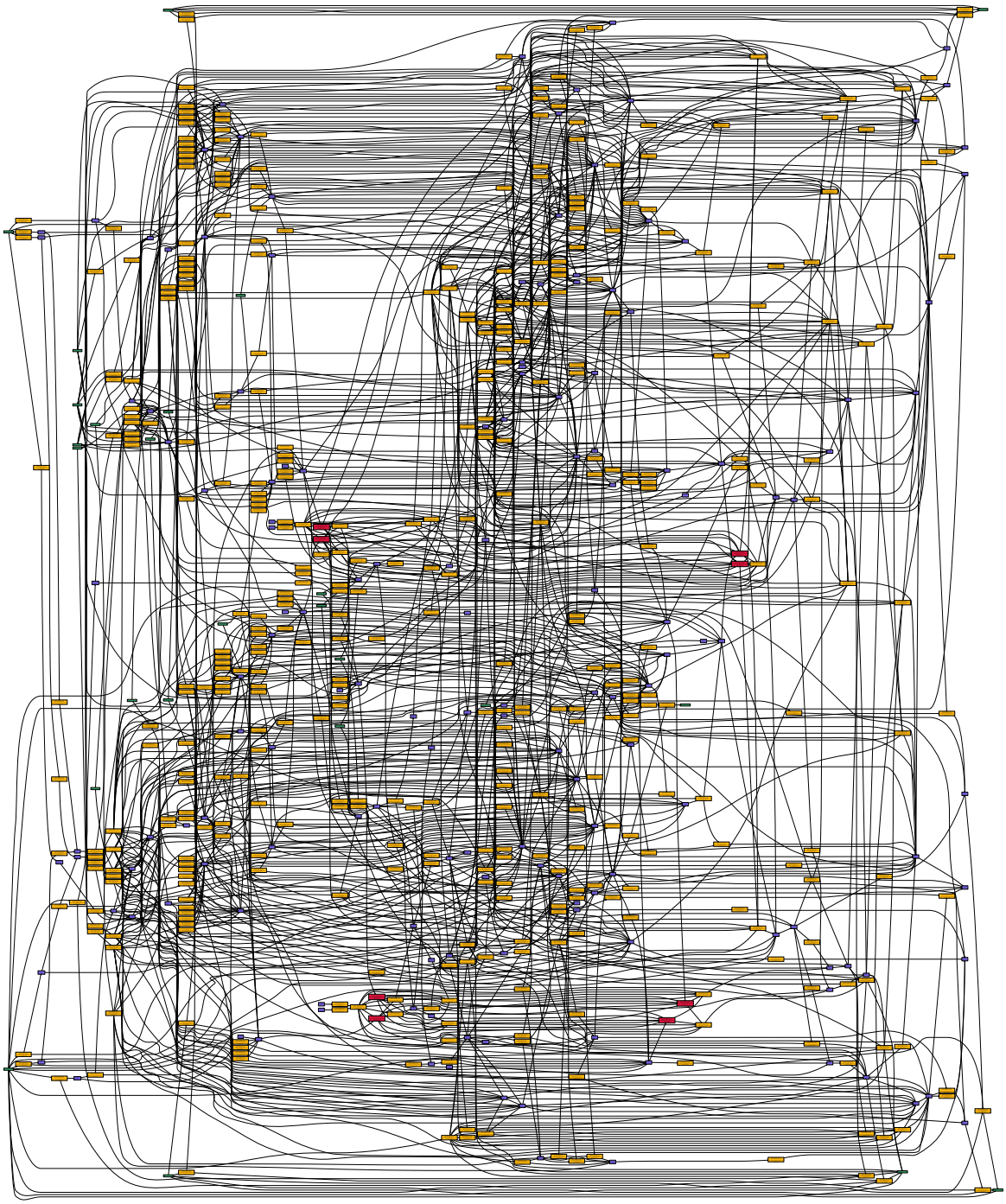


Figure 3.2: Example Low-Level Runtime Event Graph from the Pennant Application

While the high-level runtime is responsible for the construction of the DAG of operations, the low-level runtime is responsible for scheduling the operations onto the target hardware while abiding by the constraints specified by event dependences. It is important to note that many of the DAGs computed for real applications contain significant amounts of parallelism. For example, in Figure 3.2, the graph is very tall, indicating that there are many operations that can be done in parallel as event dependences primarily point from left to right. By exposing all of this parallelism in the DAG, the low-level runtime can discover significant amounts of work that can be used to overlap long latency events such as data movement and synchronization. Most importantly all of this parallelism is extracted automatically by the Legion high-level runtime without requiring any assistance from the programmer. This is only possible because Legion knows which logical regions different tasks will be accessing and can therefore safely infer when tasks and other operations can be executed in parallel. In contrast, the burden of hiding long latency operations in MPI by overlapping computation and communication is placed directly on the programmer (see Chapter 1).

The low-level runtime also serves another useful purpose in the Legion software stack: portability. The low-level runtime interface acts as an abstraction layer for various hardware interfaces. The primitives provided by the low-level runtime interface are sufficiently simple that they can be implemented on all current hardware architectures. Furthermore, their simplicity also suggests that they should be relatively straightforward to implement on future architectures as well. Consequently, Legion can easily be ported to both current and future architectures without needing to re-implement the entire software stack, but simply by providing a new low-level runtime implementation that incorporates new hardware. Since the majority of Legion code resides in the high-level runtime, the portability enabled by the low-level runtime is crucial for making Legion a portable runtime system.

3.1.4 Mapper Interface and Machine Model

The Legion mapping interface is the mechanism by which Legion applications can directly control how they are mapped onto target hardware. As was mentioned in

Section 3.1.2, the mapping interface is actually a reverse interface operated with a call-back model. Whenever mapping decisions need to be made, the Legion runtime performs a call to the corresponding mapper object to make the decision.

While mapper functions must always respond to queries, by associating them with mapper objects, the mapper functions are actually methods on mapper objects, permitting mapper objects to memoize state that can be used for answering future mapper queries. We discuss details of the various mapping queries that can be made and potential implementations of them in Chapter 8.

In order to aid in making mapping decisions that depend on the target architecture, the mapper objects have direct access to a singleton object called the *machine* object. The machine object is an interface for introspecting the underlying hardware on which the application is executing. As shown in Figure 3.1, the machine object is provided by the low-level runtime interface directly to the mapper. Using the machine object provided by the low-level runtime, mapper objects can glean information about all of the processors and memories in the system as well as many of their characteristics such as the latencies and bandwidths for accessing or copying data between different memories. Currently, the machine object is static and does not change during the duration of an application, but in the future it may dynamically be modified to reflect the changing state of hardware (e.g. if a node crashes). The need to react to such events further underscores the need for mappers to be dynamic objects capable of reacting to events at runtime.

3.1.5 Runtime and Mapper Instances

Since Legion is designed to operate in a distributed memory environment, it is useful to understand how instances of the Legion runtime and mapper objects are deployed throughout the nodes of a machine. Figure 3.3 shows where both Legion runtime instances and mapper instances are instantiated in a distributed machine with respect to both the nodes and processors. An instance of the Legion high-level runtime is created on every node in the machine. This instance of the high-level runtime is responsible for managing all of the tasks currently executing on any of the processors

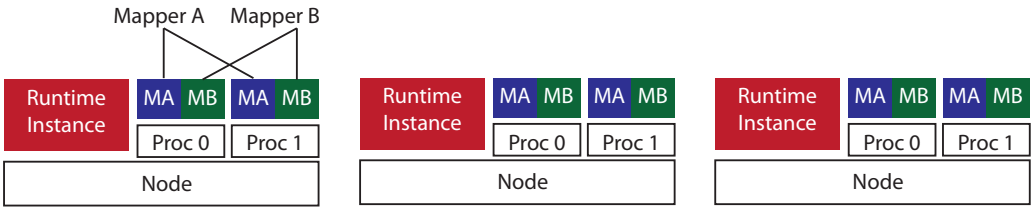


Figure 3.3: Instantiation of Runtime and Mapper Instances

on the node. It is important to note that this approach mandates that the Legion high-level runtime interface be thread safe since tasks running on different processors can invoke runtime calls simultaneously.

In a slightly different manner, mapper objects are instantiated such that there exists a one-to-one mapping between mappers and processors on a node. Thus, there exists a different mapper object for each processor made available by the low-level runtime. The reason for this is two-fold. First, by having a unique mapper for each processor, it simplifies the process of implementing a Legion mapper, as mapper objects can be immediately customized for the kind of processor that they are managing. Second, by having separate mappers for each processor, we immediately avoid any potential sequential bottlenecks because tasks mapping on different processors will never conflict with each other over a common mapper object.

3.2 Program Execution Framework

Having discussed the details of the runtime architecture we now begin our discussion of the Legion high-level runtime and how it executes Legion programs. In Chapter 2 we described how the Legion programming model requires all programs to be described as a tree of tasks that operate on logical regions. One useful way to conceptualize the execution of this tree of tasks is as a hierarchical collection of streams of tasks with the execution of each task generating a new stream of sub-tasks. To execute each stream of tasks we rely on a task pipeline that we introduce in Section 3.2.1. Section 3.2.2 provides an overview of the different stages in the Legion task pipeline. In Section 3.2.3 we describe why the hierarchical stream of task abstraction is safe,

allowing streams of tasks with a common parent task to be considered independently without needing to reason about interactions with other streams.

3.2.1 Out-of-Order Pipeline Analogy

The dynamic nature of the Legion runtime requires efficiency in the analysis of a stream of tasks because this analysis is overhead that detracts from the performance of an application. In order to hide runtime overhead and extract parallelism from a stream of tasks, Legion borrows the idea of an out-of-order pipeline from hardware architectures and adapts it for use within a distributed software environment. For several decades, hardware designers have relied on out-of-order instruction pipelines to both extract instruction level parallelism from programs and hide the overheads associated with dynamic instruction analysis. Legion applies this abstraction at the coarser granularity of tasks. The stream of sub-tasks with region requirements is analogous to the stream of instructions with register names that hardware processors execute.

Our implementation of the Legion runtime exploits this analogy extensively to both implicitly extract parallelism from a stream of tasks and to hide the overheads associated with dynamic runtime analysis. Many of the stages of the Legion task execution pipeline described in Section 3.2.2 share obvious analogs to pipeline stages in traditional out-of-order hardware processors. The crucial insight that enables this analogy is that logical regions provide the necessary information for describing the data accessed by tasks. As we discuss in Chapter 12, logical regions differentiate the Legion runtime from many of the other parallel task pipeline systems.

3.2.2 Legion Pipeline Stages

Figure 3.4 shows the different stages of the Legion runtime task pipeline[2]. Each task progresses through the stages of the pipeline in order. However, the Legion runtime is free to re-order tasks with respect to each other if tasks are determined to be

[2]Some pipeline stages are interchangeable depending on mapper decisions. We cover the details of this relationship shortly.

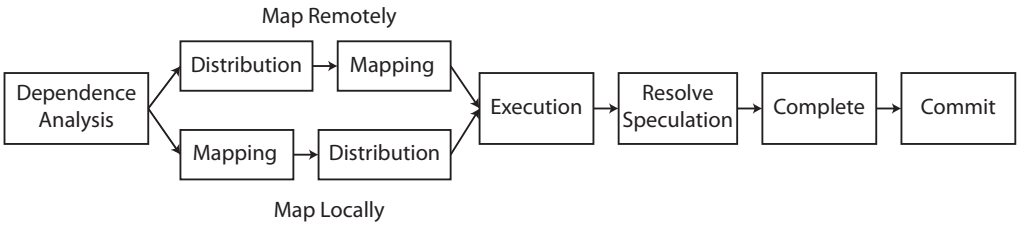


Figure 3.4: Legion Runtime Task Pipeline Stages

non-interfering. We briefly introduce each of the different pipeline stages and where possible describe analogies to similar stages in hardware out-of-order processors. The details of many of these stages will be the subject of later chapters of this thesis.

The first stage in the Legion task pipeline is dependence analysis. This stage determines which tasks are safe to run in parallel and which tasks have dependences based on logical region usage. Dependence analysis is computed with respect to the sequential order in which tasks are issued to the runtime. We refer to this initial order of tasks as *program order*. Dependences always proceed from tasks coming earlier in program order to tasks coming later in program order. The natural partial order on dependences ensures that deadlock cannot result from the dependence analysis stage. The dependence analysis stage is similar to the wake-up select stage of out-of-order processors that decides which instructions have had all of their dependences satisfied and are therefore safe to execute. The implementation of the dependence analysis pipeline stage is covered in detail in Chapter 4.

Once a task has all of its dependences satisfied (because all of its dependences have mapped), a task progresses to the mapping phase of the pipeline. The mapping phase is composed of two tightly coupled stages that are interchangeable depending on decisions made by the mapper for a task. The ordering of the mapping and distribution stages is determined by whether the mapper wants to map the task on the node where the task will run, which we call *remote mapping*, or map on the node where the task originated, which we call *local mapping*. When a task is remotely mapped, it is first sent to the target node during the distribution stage along with its meta-data and then it is mapped during the mapping stage onto the target processor. In the case where a task is locally mapped, it is mapped on the origin node and then

sent to the target node to be executed. There is an obvious trade-off to be made by mappers here: mapping remotely permits more tasks to be mapped in parallel on different nodes, but requires movement of meta-data, while locally mapping a task does not require meta-data movement, but serializes some task mappings. We discuss this trade-off in more detail in Chapter 8.

While the distribution stage has no obvious analog in hardware out-of-order processors, the mapping stage is closely related to the register renaming stage in hardware processors. In order to permit hardware processors to extract more instruction level parallelism, logical register names are mapped onto a separate set of physical registers. Register renaming provides an extra level of indirection that allows the hardware to remove conflicts such as write-after-read (WAR) anti-dependences. In Legion, the mapping stage yields an even more powerful level of indirection. The mapping stage provides the level of indirection necessary for decoupling the specification of applications from how they are mapped onto the target hardware. During the mapping stage, mapper objects are queried about how tasks should be assigned to processors and where physical instances of logical regions should be placed in the memory hierarchy. Mapping logical regions onto physical instances not only allows Legion to remove conflicts such as WAR anti-dependences on logical regions, but also allows the mapper to control where instances are placed in the memory hierarchy for performance. The memory hierarchy therefore acts as a many-tiered register file for storing different physical instances of a logical region. We discuss the implementation of the mapping stage in Chapter 5 and the implementation of the distribution stage in Chapter 6.

The execution stage of the Legion pipeline is nearly identical to the execution stage for instructions in hardware processors. In hardware, an instruction is assigned to an execution unit where it is performed. In Legion, tasks are assigned to a processor which then executes the task. The primary difference is that while instructions in hardware are not capable of generating new instructions, in Legion, the execution of a task can generate a new stream of sub-tasks that then must also be executed by the runtime. We discuss the implications of the hierarchical nature of streams of tasks in Section 3.2.3.

The fifth stage of the pipeline is used for checking when speculation has been resolved. For tasks that are not predicated, this stage does not require any work. However, for tasks that were predicated and for which the mapper of the task chose to speculate on the predicate value (see Section 10.2.1), this is the stage where the results of the speculation are checked. If a task was correctly speculated, then this stage performs no work. However, if the speculation was incorrect, then work must be performed by the runtime to either re-execute the task if the speculative value of the predicate was incorrectly guessed to be false, or the results of the task must be undone if the value of the predicate was incorrectly guessed to be true. Furthermore, any tasks with dependences upon the mis-speculated task must also be rolled back. We discuss the details of this process in greater detail in Chapter 10.

The sixth stage of the pipeline is the completion stage that determines when a task has finished executing. Due to Legion's deferred execution model, it is possible for a task to finish its execution stage of the pipeline and continue to progress through the pipeline while the sub-tasks that it launched have yet to execute. The completion stage is the point where a task waits for all of its child tasks to reach the completion stage. It is important to note that this definition of completion is inductive: in order for a task to complete, all of its descendant tasks must already have completed. Once a task is complete, it has correctly executed and may propagate information such as the result of its future value. Tasks that depend on the completion of this task are also free to begin executing. However, for resiliency purposes this is not the final stage of the pipeline.

The final stage of the pipeline is the commit stage. The commit stage of the pipeline is used for knowing when it is safe to reclaim the meta-data associated with a task. While a task may have successfully executed, it is possible that the data generated by the task is corrupted post execution (e.g. by a bit-flip in memory). Under these circumstances a task may need to be rolled back and re-executed. In order to perform a roll-back it is necessary to retain the meta-data (task ID, region requirements, etc.) for re-executing the task. A task can only be committed once we are certain that no roll-back of the task will ever occur. We discuss the conditions for determining when it is safe for a task to be committed in Chapter 10 and describe

how mappers can choose to preemptively commit tasks for performance reasons in Section 10.3.1.

3.2.3 Hierarchical Task Execution

As we mentioned in Section 3.2.2, the Legion pipeline analogy does not directly align with hardware out-of-order pipelines due to the hierarchical nature of the Legion programming model. In a Legion program the execution of task within the pipeline can result in an entirely new stream of sub-tasks with region requirements that need to be executed. The obvious question then is whether these tasks from different streams need to be considered when performing operations such as dependence analysis and mapping. Importantly, due to a crucial design decision introduced in the Legion programming model, tasks need only consider other tasks within the same stream (i.e. originating from the same parent task) when progressing through the stages of the task pipeline.

Recall from Section 2.5.2 that a task is only permitted to request privileges on a subset of the regions and fields for which its parent task held privileges. This important property is sufficient to prove a useful theorem: if two tasks t_1 and t_2 are non-interfering, then all sub-tasks of t_1 will be non-interfering with all sub-tasks of t_2. The proof of this theorem is based on a formal operational semantics of the Legion programming model and is beyond the scope of this thesis. A full proof of the theorem can be found in [49].

The consequences of this theorem are profoundly important to the implementation of the Legion runtime. This theorem permits the Legion runtime to implement a hierarchical scheduling algorithm. Tasks only need to consider other tasks in the same stream (which all come from the same parent task) when traversing the task pipeline. Since all tasks in the same stream originate on the same node, no communication is necessary for performing dependence analysis. The hierarchical nature of the Legion task pipeline also allows many tasks to be traversing the task pipeline in parallel on different nodes. This property is essential to enabling Legion to scale well on thousands of nodes.

Chapter 4

Logical Dependence Analysis

The first stage in the Legion task pipeline is logical dependence analysis. Given a stream of sub-task launches from a common parent task, the dependence analysis stage is responsible for computing which tasks are *interfering* and therefore must have dependences (we give a formal definition of non-interference in Section 4.1). Unlike other programming systems that compute dependences between tasks based on inferred data usage, either statically (e.g. by doing pointer analysis), or dynamically (e.g. transactional memory), Legion has specific names for the sets of data being accessed by tasks in the form of logical regions. The concrete names for different sets of data provided by logical regions will considerably simplify the dependence analysis.

A naïve implementation of the Legion dependence analysis stage would perform a pairwise test for non-interference between a sub-task and all of the tasks launched before it in the same parent task. For a stream of N tasks, dependence analysis is known to require $O(N^2)$ non-interference tests be performed[1]. In practice, the size of N is bounded by a sliding window reflecting tasks that have yet to complete. A task only needs to perform non-interference tests against tasks within this window. Figure 4.1 shows a representative example. Task t_8 only needs to perform dependence tests against tasks in the stream S that remain within the window and therefore not complete. However, the task window is usually on the order of a few hundred to

[1]In Legion, non-interference tests are actually performed between the regions used by tasks. If tasks on average have R region requirements, then dependence analysis is actually $O(N^2R^2)$.

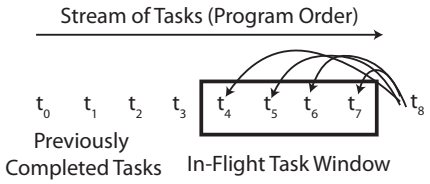


Figure 4.1: Example Task Window for Dependence Analysis

a thousand tasks in many applications. While finding an asymptotically superior algorithm for dependence analysis is unlikely, we can introduce data structures that can significantly improve the constant factors associated with the dependence analysis algorithm.

In this chapter, we begin by giving a precise definition of when two tasks are non-interfering (Section 4.1). Based on the properties of the non-interference test, we design an algorithm that leverages the logical region tree data structures maintained by the runtime to accelerate non-interference tests (Section 4.2). We then elaborate on how storage for the meta-data associated with dependence analysis is efficiently maintained (Section 4.3). In Section 4.4, we describe the mapping dependence graph produced by the logical dependence analysis. Finally, we describe an optimization for memoizing the results of the dependence analysis in Section 4.5.

4.1 Task Non-Interference

Before proceeding with our discussion of how we implement the dependence analysis stage, we first need to give a precise definition of what it means for two tasks to be non-interfering. Two tasks are non-interfering if all pairs of region requirements between the two tasks are non-interfering. A pair of region requirements are non-interfering if any one of the following three disjunction non-interference conditions are met:

- **Region Disjointness** - the logical regions in the two region requirements are disjoint (e.g. there are no shared rows).

- **Field Disjointness** - the sets of fields requested by each of the two region requirements are independent (i.e. there are no shared columns).
- **Privilege Non-Interference** - either both region requirements are requesting read-only privileges, or both region requirements are requesting reduction privileges with the same reduction operator.

Figure 4.2 gives a visual depiction of the three different non-interference criteria from the node's logical region in the circuit simulation from Chapter 2. The red and blue rectangles illustrate the data requested from two different region requirements. Non-interference can be proven in any of the three dimensions: if the two logical regions access disjoint sets of rows, if the sets of fields requested are disjoint, or if the privileges are non-interfering.

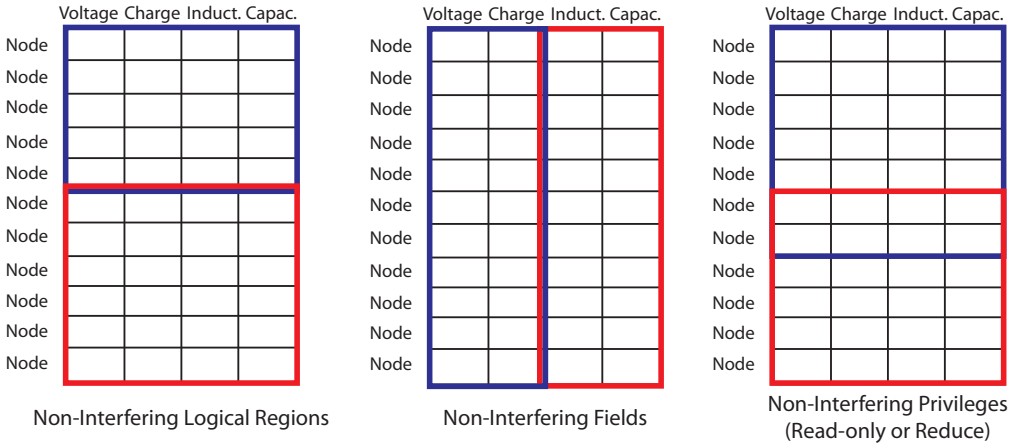


Figure 4.2: Example Non-Interference Criteria from the Circuit Simulation

Due to the disjunctive nature of the three conditions, they can be applied in any order. If any of the three non-interference criteria are met, then the pair of region requirements are non-interfering and testing the remaining conditions can be skipped. Consequently, the order in which these criteria are tested can have a significant performance impact. It is therefore important that we pick an order for testing these conditions that minimizes the number of non-interference criteria tested.

The ordering that we select is region disjointness, field disjointness, and finally privilege non-interference. This ordering stems from the observation that Legion

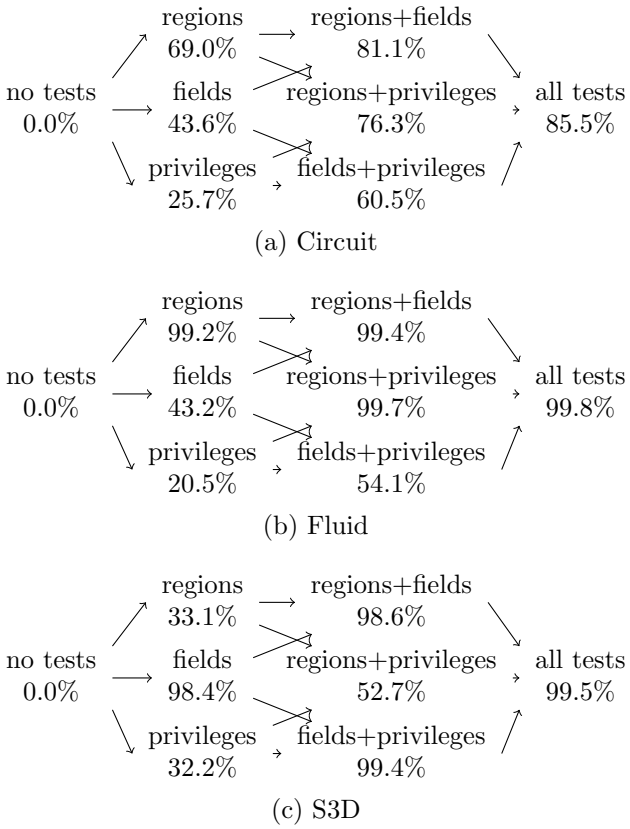


Figure 4.3: Non-Interference Test Success Rates by Application

programs commonly express data parallelism at several different granularities both across nodes and within nodes. It is therefore most likely that two tasks will be proven to be non-interfering using the disjointness of their logical regions. After this, field set disjointness is most likely. Finally, privilege non-interference is a more complicated test and is therefore placed last where it is least likely to be performed. While it is possible to write Legion applications that perform better with a different test ordering, it has been our experience that the performance of this ordering is sufficient for achieving high non-interference test throughput. As we show in Section 4.2, this ordering also lends itself to a natural implementation based on logical region trees.

To provide empirical evidence for our chosen ordering of these tests, Figure 4.3 shows decision diagrams for potential orderings of non-interference tests for three real world applications: a circuit simulation, a fluid flow simulation, and the combustion

simulation S3D discussed in Chapter 11. At each node of the decision diagrams, we show the percentage of non-interference tests that would succeed with that subset of tests. The percentage at the end shows the overall percentage of non-interference tests that succeed for a given application.

Ideally, we want to minimize the overall cost of the non-interference tests, which favors the early use of cheaper and/or more efficient tests. Although there is considerable variability between applications, region disjointness is the most effective test overall. The use of region tree data structures is essential to making this test inexpensive, and it is the clear choice for the first test. Field disjointness is the next obvious test as it finds significant parallelism, especially in S3D. By using *field masks* (discussed in Section 4.3.3) this test can also be made inexpensive which justifies performing it second. Finally, the more expensive privilege non-interference test is placed last to minimize the number of invocations.

4.2 Logical Region Tree Traversal Algorithm

The goal of the logical region tree traversal algorithm is to improve the efficiency of the dependence analysis for a stream of sub-tasks S within a parent task P. To perform the dependence analysis for any sub-task T in S, we need to find all other tasks that come before T in S that interfere with at least one of the region requirements requested by T. We therefore need to perform a separate analysis for each region requirement of T.

To detect interference on region requirements, our analysis operates over the logical region tree forest. As part of our analysis, we ensure that after each task in S has performed its non-interference tests, it registers itself as a *user* of the region tree node for each logical region on which it requests privileges. A user record stores information about the task including its requested fields, privileges, and coherence. User records allow later tasks to determine whether dependences should be registered from later tasks in S. By registering users at the region tree nodes on which they requested privileges, we will be able to easily elide non-interference tests based on logical region disjointness (the first non-interference condition from Section 4.1). In

practice, we do not need to store all the users on each logical region tree node from previous tasks in the stream S. Instead, we can aggressively prune tasks that have finished executing or for which we can prove there exists transitive dependences. We discuss these optimizations further in Section 4.3.

To perform the dependence analysis for a sub-task T, we need to traverse the region tree forest for each region requirement in T to find any potential tasks from S that are not disjoint on region usage. We first compute the path from the logical region requested to the corresponding logical region on which the parent task P owns privileges. This path is guaranteed to exist because of the restriction enforced by the Legion programming model that all sub-tasks can only request privileges for a subset of the privileges held by the parent task (see Section 2.5.2). Using this path we can immediately determine the nodes in the logical region tree that must be visited because they are interfering on region disjointness. This set of nodes includes all nodes in the region tree along the path, as well as any sub-trees of nodes along that path that may contain interfering nodes. By computing this path and any potential interfering sub-trees, we immediately elide any region requirements of previous tasks that are disjoint on region usage, including region requirements in different region trees within the region tree forest.

As an example of the region tree path, consider a sub-task launched inside of the `simulate_circuit` task from the example in Chapter 2. Figure 4.4 illustrates both the computed path and other logical regions that would need to be analyzed for interferences for a region requirement requesting privileges on the `r_all_shared` logical region. The interference path would run from the root node of the logical region tree where the `simulate_circuit` task has privileges to the `r_all_shared` logical region where the task is requesting privileges. In addition, the task would also need to visit all the sub-trees of the `r_all_shared` logical region. All of the nodes that would need to be analyzed for interference are colored in red. Nodes that can be skipped due to being non-interfering are shown in blue. It is important to note that the two-level partitioning scheme that we chose in Chapter 2 is what allows this analysis to directly omit all logical regions in `r_all_private` nodes from consideration when analyzing region requirements that request privileges on logical regions in the

`r_all_shared` logical region sub-tree.

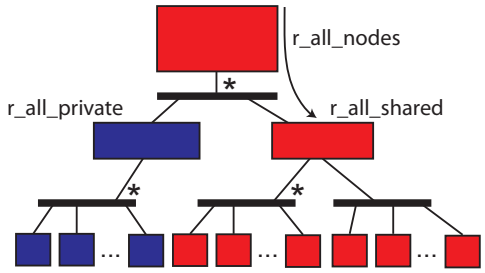


Figure 4.4: Example Non-Interference Path for Dependence Analysis

Having determined the set of region tree nodes that must be examined because they are interfering on logical region usage, we can now perform the second dimension of the non-interference test based on field usage. For each node that interferes based on region usage, we iterate through the list of users registered at the node. We then check whether the set of fields requested by each user is disjoint from the set of fields in the region requirement being tested from T. If they are disjoint, then the two region requirements are non-interfering, otherwise we perform the non-interference test on privileges. If both the field sets are interfering and the privileges are interfering, then we record a dependence between the two tasks, otherwise that pair of region requirements is non-interfering. Finally, after finishing the analysis for a region requirement, we register the region requirement at the destination node of the path, ensuring that any future sub-tasks in the stream S will be able to determine the necessary dependences.

While this discussion has primarily described sub-tasks as the elements within the stream S, because the region tree analysis is performed based on region requirements, the analysis is easily generalized to all other operations within the Legion programming model, including inline mappings and explicit copy operations that also use region requirements to describe their data usage.

4.3 Logical Region Tree Data Structures

While the correctness of the traversal algorithm described in the previous section is easy to determine, achieving high-performance requires optimization of both the algorithm as well as the data structures used for the traversal. We first give a brief description of how region tree data structures are stored in the Legion runtime in Section 4.3.1 and then we describe each of the optimizations in subsequent sections.

4.3.1 Region Tree Shape

The shapes of region tree data structures in the region tree forest are determined by the runtime calls made by the application for creating index spaces and partitioning them. Figure 4.5 gives one example of a region tree forest that might be instantiated for an application. There are two index space trees, rooted by I_0 and I_1, and two field spaces A and B. Every region tree in the region tree forest is associated with an index space tree and a field space. For example, the region tree rooted by logical region R_0 is associated with index space I_0 and field space A. Note that the creation of region trees is done dynamically and therefore multiple region trees can be created with the same index space and field space (e.g. both region trees rooted by R_1 and R_2 are associated with the same index space and field space, but are distinct logical region trees). Region trees are not automatically created for each pair of a field space with an index space tree as is the case with index space I_1 and field space B.

In both index space trees and region trees, the node type alternates between levels. Even numbered levels in the tree (starting with zero at the root) consist of index space nodes for index space trees and logical region nodes for logical region trees. Odd levels in the trees contain partition nodes. In the internal representation of these data structures, Every node in both index space trees and region trees maintain pointers both to their parent node and to all of their child nodes. Every region tree node also maintains a pointer back to its corresponding index space tree node.

Index space trees are eagerly instantiated when runtime calls are made by the application. However, to save space, logical region trees are lazily instantiated from index space trees. When a new top-level logical region is created, only a node for

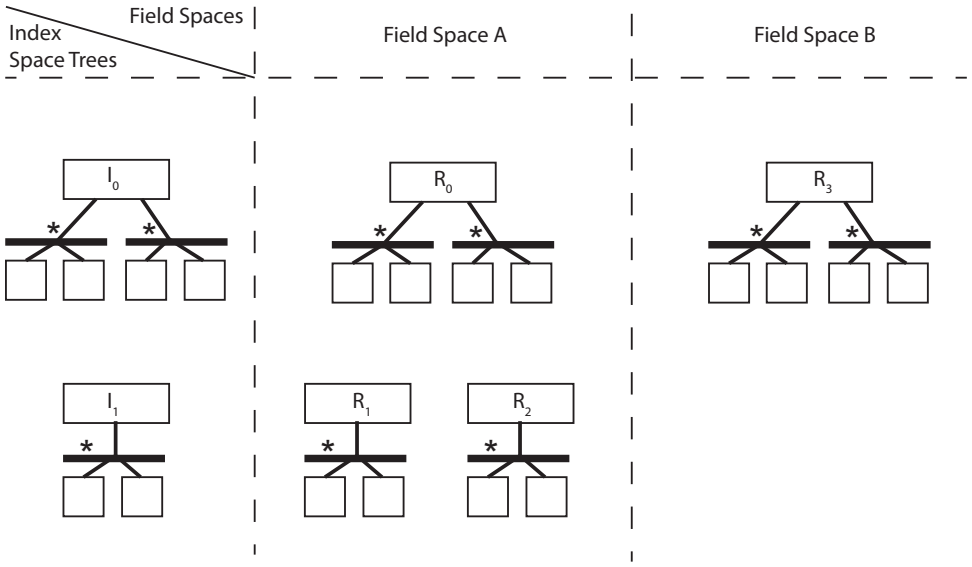


Figure 4.5: Example Relationships Between Index Space Trees, Field Spaces, and Logical Region Trees

that root region is created. The other nodes in the region tree are only created when they are requested as part of a dependence analysis traversal. Consequently, on many nodes in the machine, only a subset of the full region tree forest is ever instantiated for a real Legion application, thereby saving considerable space[2].

Index space tree nodes (and therefore by proxy region tree nodes) also keep track of two important properties of their child nodes: disjointness and completeness. Disjointness records which children of a node are entirely independent of each other. Disjointness information is used by the dependence analysis to determine which nodes in the region tree must be visited. All logical regions in sub-trees rooted by a region tree node that is potentially aliased with another region tree node along the interference path must be analyzed for interference. In some cases, disjointness information is provided by the application, such as when disjoint partitions are created, indicating that all pairs of children within a partition are disjoint. Alternatively, if enabled, the runtime can dynamically test whether pairs of children are disjoint (e.g. testing whether two specific logical regions in an aliased partition are independent). In the

[2]To further save memory, The migration of region trees to other nodes is also done lazily as we describe in Chapter 6.

case of the dynamic disjointness tests, the results can be memoized to help amortize the cost of performing the (sometimes expensive) tests.

The second property recorded by all index space tree nodes is completeness. A child is considered to be complete if every row in the index space node is contained in at least one child index space. Completeness will be used to perform several important copy reduction optimizations described in Chapter 5.

The disjointness and completeness properties of child nodes behave differently under dynamic allocation of rows in index spaces permitted by the Legion programming model (see Section 2.2.2). The disjointness of two children is invariant under dynamic allocation. If a new row is allocated in one child node, then it is impossible for it to be allocated in the other child node. However, completeness of a child node is impacted by dynamic allocation. Consider an index space I with two initially complete index partitions P_1 and P_2. If a new entry is allocated in one of the index sub-spaces of P_1 it is therefore also allocated in index space I. While P_1 is still complete, P_2 can no longer be considered complete. Therefore, while disjointness information can be safely memoized under dynamic allocation, completeness can only be cached and must always be invalidated whenever dynamic allocation is performed.

4.3.2 Epoch Lists

The first optimization that we perform is designed to reduce the number of users that must be considered on each node that we traverse. The critical insight is that we don't need to record all dependences that exist between a task T and earlier tasks in the stream S if T has transitive dependences through other tasks in S. While there are many ways that we could detect transitive dependences, we focus on a subset of transitive dependences that are easy to detect: those dependences that exist through the same field with the same logical region. The crucial insight is that tasks within the same node using the same fields will form *epochs* of tasks with the same privileges. For example, there maybe an epoch of tasks performing reductions to a field, followed by an epoch of tasks that read the field (note there can also be epochs containing multiple tasks with read-write privileges by using relaxed coherence modes that are

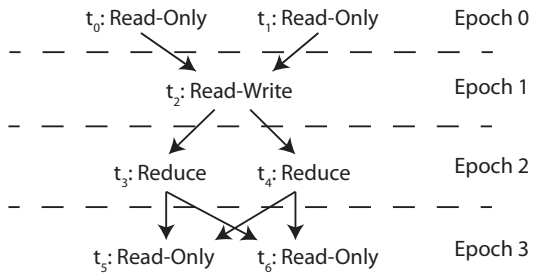


Figure 4.6: Epoch Lists Example

discussed in Chapter 9). By only storing the most recent epochs of tasks for each field, we can reduce the number of tasks that must be considered when traversing a region tree node.

Figure 4.6 shows example epochs that might be generated from a stream of tasks (accessing only a single field). Epoch zero consists of tasks that are all accessing the field with read-only privileges. The next epoch contains a single read-write task that records mapping dependences on all tasks from epoch zero. Epoch two is a reduction epoch, while epoch three is again a read-only epoch. The important observation is that each operation in an epoch needs to record a dependence on all operations from the previous epoch. Therefore at most, two epochs need to be tracked at a time.

To track epochs, we create two *epoch lists* that are used to store all the tasks necessary for performing dependence analysis in the given node. The first epoch list stores a set of *current* epoch users while the second list stores the set of *previous* epoch users. The important observation is that when performing dependence analysis, only one of two things will occur: either a user will register dependences on all the users in the current epoch (for a field), in which case it will start a new current epoch, or it will join the current epoch and must therefore record dependences on all the users in the previous epoch list (for a field). These two epoch lists eliminate the need for storing all previous users from the stream of tasks not in the two most recent epochs, thereby reducing the number of tasks for which we need to perform non-interference tests when registering a user within a region tree node.

When traversing a given region tree node for the dependence analysis we need to detect which of the two scenarios apply to the user being analyzed. To detect the

first scenario, we test the region requirement from T being analyzed against all of the region requirements in the current epoch list to see if it interferes with all the users for a specific set of fields. If the region requirement does interfere with all users for some set of fields, we say that the task has *dominated* those fields, and should start a new epoch for those fields. For all the dominated fields, the users of those fields in the previous epoch list are filtered out, and the users from the current epoch list are moved from the current epoch list back to the previous epoch list. If the region requirement also has non-dominated fields, then it must also traverse the previous epoch list to search for interfering region requirements. It is important to note that because we only add region requirements at the destination node of the traversal algorithm, it is possible not to observe any users of specific fields in the current epoch list. In these cases, the unobserved fields must be considered non-dominating fields. The alternative of considering them dominated would result in premature filtering of region requirements from the previous epoch list.

One important design decision associated with epoch lists is whether to store previous users in different lists for every field or to maintain a single list of tasks and filter based on field set disjointness as part of the traversal. We opted to implement the latter, as there can often be hundreds or thousands of fields that could result in large list overheads for only a few users. Furthermore, most users request multiple fields that would result in duplicated meta-data for the same user over many lists. Instead we maintain a single current epoch list and a single previous epoch list and rely on a fast mechanism for testing for field set disjointness that we describe in Section 4.3.3.

4.3.3 Field Masks

For many Legion applications, including the combustion simulation S3D discussed in Chapter 11, field spaces can contain on the order of hundreds to thousands of fields. In order to quickly evaluate the field set disjointness condition for non-interference, we need an efficient representation of a set of fields. We use *field masks* that store fields as bit masks.

Field masks present a very compact way of representing the potentially large space of fields that can be used by region requirements. It is important to note that we can only implement field masks efficiently because of the static upper bound on the number of fields in a field space, which was described as part of the Legion programming model in Section 2.2.3. Having unbounded field masks would require dynamic memory allocation that invalidates many of the important compiler optimizations that make operations on field masks fast.

Field masks support all of the common operations associated with bit masks including conjunction and disjunction as well as set subtraction. The three most important operations that are supported by field masks are conjunction, testing for an empty set, and testing for disjointness with another field mask. These operations are useful for detecting the field set disjointness condition of non-interference and serve as the basis of region tree traversal algorithms for both logical state in this chapter as well as physical state that we discuss in Chapter 5. To accelerate these important operations on field masks we employ a simple optimization technique: two-level field masks. Two-level field masks contain a 64-bit summary representation of the field mask. A bit set at index i in the summary mask indicates the presence of at least one set bit at an index j in the field mask, where $mod(j, 64) == i$. Before actually testing for conjunction, emptiness, or disjointness the summary masks for the field masks are tested first. Since testing the summary masks only involves executing a single instruction (on 64-bit architectures), it can easily reduce the work associated with important field mask operations, especially for field spaces with large upper bounds on the number of fields.

In the general case, field masks are implemented using 64-bit unsigned integers. However, where possible, our implementation of field masks also takes advantage of the underlying hardware by using SSE and AVX vector intrinsics for performing field mask operations. It is important to note that these vectorized bit operations do not conflict with any of the underlying paths for vectorized floating point hardware in most processors. Therefore our field masks do not cause any performance degradation when floating point units are shared between multiple cores within a chip (as is the case on same target architectures such as the AMD Interlagos architecture).

4.3.4 Region Tree States

The next optimization that we perform for the traversal algorithm adds some additional state to each node in the region tree to reduce the number of sub-trees in the region tree that must be traversed when checking for region disjointness in the non-interference test. In our original version of the algorithm described in Section 4.2, we computed a path from where the parent task P had privileges to where the region requirement from sub-task T was requesting privileges. In addition to traversing all of the nodes along this path, we also described that we needed to check all sub-trees that contain regions that might not be disjoint from the logical region in the target region requirement. To avoid the traversal of many sub-trees, we add state to every region tree node that records which sub-trees are *open*. A sub-tree is open if there exists at least one region requirement from another sub-task from the stream S that has been registered in the sub-tree.

To further add to our efficiency, we also track the fields that are open for different sub-trees, as well as the privileges of the open fields. For example, a sub-tree might only be open for one field or a small number of fields. If those fields are disjoint from the fields in the region requirement of T, then we do not need to traverse the sub-tree. Similarly, a sub-tree might be open for a set of fields with read-only privileges which indicates that all sub-tasks registered in the sub-tree are requesting read-only privileges. If the region requirement from T being analyzed is also requesting read-only privileges then dependence analysis for the sub-tree can be skipped because we know all users are non-interfering on privileges.

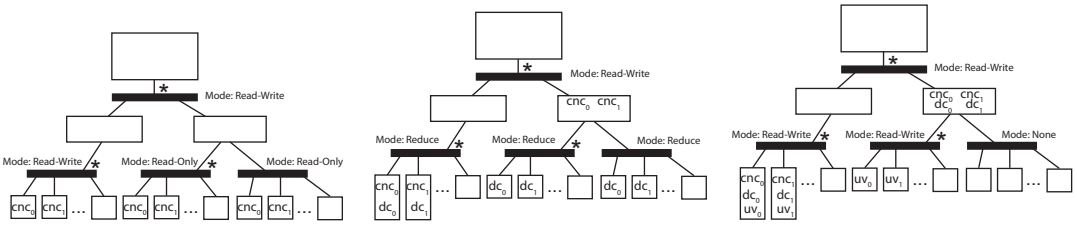
Care must be taken to maintain the proper state at each node of the region tree. Therefore, we perform the traversal along the path from where the parent task P has privileges to where the region requirement of T is requesting privileges, we *open* the appropriate child nodes and mark which fields are open with specific privileges. In some cases the correct sub-tree might already be open with the same or a super-set of the privileges necessary (see the semi-lattice from Figure 2.2). However, when privileges are different, there are two possible solutions. First, we could elevate privileges in the semi-lattice to soundly represent the set of region requirements in a sub-tree; this approach is easy to implement, but results in a lack of precision.

Alternatively, we could *close* all potentially conflicting sub-trees and then open the sub-tree we intend to traverse with the precise privileges. In order to maintain the fidelity of the dependence analysis, we chose to implement the later option.

A close operation on a sub-tree mutates the state of the region tree in two ways. First, a close operation on a field is the equivalent of dominating all the users in the current epoch of the node at the root of the close operation We therefore siphon all the users for the field(s) being closed from the previous epoch list and filter any users from the current epoch list into the previous epoch list. We then hoist all of the tasks in the current epoch lists in the sub-tree, for the given field(s), up to the previous epoch list of the node at the root of the sub-tree being closed. While this reduces the precision of the tasks for region analysis, it is always sound as they now appear to be using a region node that is a superset of the region node they were originally using[3]. Furthermore, this loss of precision is minimal since the primary reason for performing close operations is to open a different sub-tree whose regions all conflict the sub-tree being closed (e.g. closing one partition sub-tree to open another). After the close operation is complete, the traversal can continue by opening up the sub-tree of the next node in the appropriate mode.

Since close operations effectively mutate the state of the epoch lists for the fields that are closed, we record the close operations that need to be performed between the previous epoch and the current epoch in a *close list*. It is important that each user added to the current epoch list of a set of fields also record dependences on all close operations that must be performed for those fields. The reason for this is that these close operations must be performed prior to mapping any of the region requirements for a task. Since multiple tasks in the same epoch are able to map in parallel, then it is the responsibility of the first task that maps in an epoch to perform the close operations in the physical state of the tree. We discuss the details of how close operations are performed in the physical tree in Chapter 5. The close list is also filtered along with the previous epoch list when one or more fields begins a new epoch.

[3]Ultimately the loss in precision is inconsequential because extra mapping dependences need to be computed between users accessing logical regions in different partitions in order to correctly perform the pre-mapping traversals described in Section 5.1.1.



(a) State After CNC (b) State After DC (c) State After UV

Figure 4.7: Example Circuit Region Tree States

Figure 4.7 shows the state of the logical partitions in the node logical region tree after different sets of task perform their dependence analysis. The first set of tasks to map are the `calculate_new_currents` (CNC) tasks. In Figure 4.7a, all of the CNC tasks have completed their analysis and registered themselves at the appropriate logical region nodes based upon their region requirements. In the process of traversing from the root node, each task has opened up partitions with the appropriate privileges[4]. In this example, we show the steady-state behavior in the middle of the loop, therefore intermediate partitions are already open with read-write privileges indicating the existence of modified state from earlier loop iterations.

Figure 4.7b shows the state of the region tree after the `distribute_charge` (DC) tasks have finished their dependence analysis. Note that all of the CNC tasks in the shared sub-tree have been moved back to the `all_shared` logical region as the result of a close operation necessary to transition from the partition being open with read-only privileges to reduction privileges. The `all_private` sub-region did not require such a transition as it was already open with read-write privileges that subsume reduction privileges. Figure 4.7b also demonstrates the benefits of reduction privileges: both the `p_shr_nodes` and `p_ghost_nodes` partitions can be open at the same time with reduction privileges.

Lastly, Figure 4.7c shows the state of the region tree after the `update_voltages` (UV) tasks have completed their analysis. Another close operation was necessary to close up the `p_shr_nodes` and `p_ghost_nodes` partitions to transition from reduction

[4]In practice all nodes (both privileges and logical regions) must be opened, but we only show the partitions for simplicity.

privileges to the read-write privileges needed by the UV tasks on the `p_shr_nodes` partition. Each of the UV tasks records a dependence on this close operation as part of their dependence analysis.

4.3.5 Privilege State Transitions

Figure 4.8 shows the state transition diagram for child nodes based on privileges. For a given field in a specific state, arcs show the transition that must occur based on the privileges being requested by the region requirement of T. For cases where the state of the privileges move down or across the privilege semi-lattice, close operations must first be performed before the next sub-tree can be opened. One interesting observation about this state diagram is that it is very similar in many ways to the state diagram of directory-based cache coherence protocols, with the only differences being that the privilege state diagram has more potential states (involving reductions), and that the granularity of logical regions is much larger than individual cache lines. By operating at a coarser granularity, our Legion implementation is able to amortize the cost of having a more complex state graph and maintain state for each field of a logical region in software instead of in hardware.

Figure 4.8 has an interesting property when it comes to reductions. Note that there exists an intermediate state called *single-reduce*. In this state, only a single sub-tree is permitted to be open for a specific kind of reduction. This state acts as an intermediate state for deferring the decision of whether a sub-tree from a node should be opened in read-write or reduction mode. The decision is ambiguous when the first reduction region requirement is registered in a sub-tree: the runtime cannot be sure whether more reductions of the same kind will open other sub-trees, or if other operations such as reads or write will be performed in the same sub-tree. If a different kind of reduction or a read or a write task proceeds down the same sub-tree, then the state is changed to be in read-write mode. However, if a later sub-task attempts to open a different sub-tree with the same reduction mode, then the state is changed to be in multiple-reduce mode. Deferring the decision about whether to be in multiple-reduce or read-write mode for reduction is important because it avoids

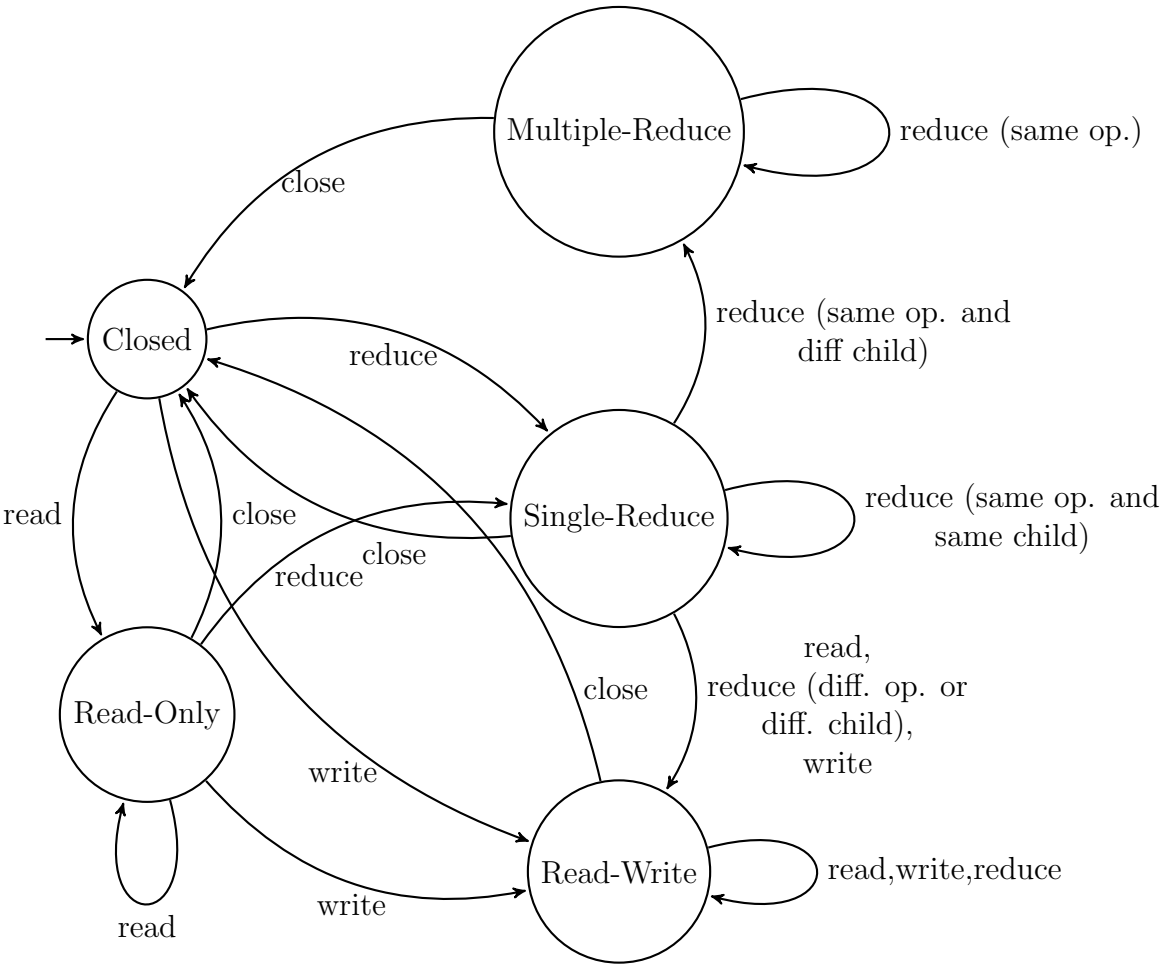


Figure 4.8: Open Child Privilege State Diagram

guessing by the runtime about the proper way to open the sub-tree, that could result in unnecessary close operations being performed.

4.3.6 Logical Contexts

The final optimization that we perform on region trees is designed to reduce the memory usage required for storing state associated with region trees. Recall from Section 4.3.1 that region tree data structures are de-duplicated across runtime instances. Since there may be many tasks executing simultaneously within a runtime

instance, all of these tasks will be generating different streams of sub-tasks. We therefore need a mechanism for differentiating the state that must be stored in the region tree forest from different streams of tasks.

This is achieved by differentiating different streams of sub-tasks as *logical contexts*. On every node in the region tree forest there exists an array of *logical state* objects that store the necessary meta-data for supporting the logical region tree traversal algorithms described in this chapter (e.g. epoch lists, close lists, open children). The array of logical state objects is indexed by a logical context ID. Before a parent task P begins to execute, it requests a logical context ID from the runtime. When P executes, it generates a stream of sub-tasks all of which are analyzed within the logical context allocated to the parent, meaning they use the parent's logical context ID to index into the logical state arrays on the region tree nodes when performing their dependence analysis.

In order to ensure correctness of Legion programs, before executing, each task is allocated a logical context by the runtime instance. This is necessary since any task can launch arbitrary sub-tasks. However, logical contexts can be an expensive resource to allocate because of the implied memory usage required on all instantiated nodes of a region tree. To reduce context usage, Legion programmers can indicate that certain task variants are actually leaf task variants (see Section 2.5.8). Leaf task variants are guaranteed not to perform any sub-task or other operation launches. The knowledge that no sub-tasks will be launched by an executing task is sufficient to allow the runtime to elide the allocation of a context to the executing task, thereby reducing both context and memory usage.

4.4 Dynamic Dependence Graph

The result of the dependence analysis stage is a *dynamic dependence graph* for a given stream of operations generated by a parent task. The dynamic dependence graph is a directed acyclic graph where nodes represent operations (e.g. sub-tasks, inline mappings, explicit region copies) and edges represent dependences that result from interfering region requirements. It is impossible for there to exist cycles in this graph

as the dependence analysis stage is performed serially for all sub-tasks within the stream S generated by parent task P. While there are no cycles within the dynamic dependence graph, there can be multiple edges between nodes as sub-tasks may have multiple interfering region requirements. Exactly one dynamic dependence graph is computed for the execution of each parent task. While the entire graph needs to be computed, we describe how the entire dynamic dependence graph does not need to persist throughout the lifetime of the parent task P.

Figure 4.9 shows an example dynamic dependence graph from the S3D application described in detail in Chapter 11. Boxes represent operations that are performed as part of the computation while edges represent computed dependences between operations as a result of the logical dependence analysis. Edges primarily point from left to right. The vertical span of the graph is therefore indicative of the amount of task-level parallelism available in S3D. It is important to realize that this graph was generated from a fraction of a time step in S3D for the smallest chemical mechanism possible. Production S3D runs generate graphs that would consume many pages if depicted here.

The directed edges indicating interference between sub-tasks serve a dual purpose. First, dependences between sub-tasks are used to determine when it is safe for a sub-task to progress to the mapping stage of the pipeline. A task is only permitted to progress to the mapping stage once all of the sub-tasks on which it has a dependence have finished the mapping stage of the pipeline. Under these circumstances, we refer to the edges as *mapping dependences*. By requiring all mapping dependences be satisfied for a sub-task before it can map, we ensure that it will properly compute any dependences (either true or anti-dependences) as part of its physical analysis (see Chapter 5). To know when mapping dependences have been satisfied, sub-tasks register themselves with the source sub-tasks of all their dependence edges. If the source sub-task has yet to map, it will accept the registration, otherwise it will indicate that it has already finished the mapping stage of the pipeline and indicate that no mapping dependence is necessary. Each sub-task records how many successful registrations it makes; this is the number of other sub-tasks that it must wait to complete the mapping stage before it can progress to the mapping stage. When

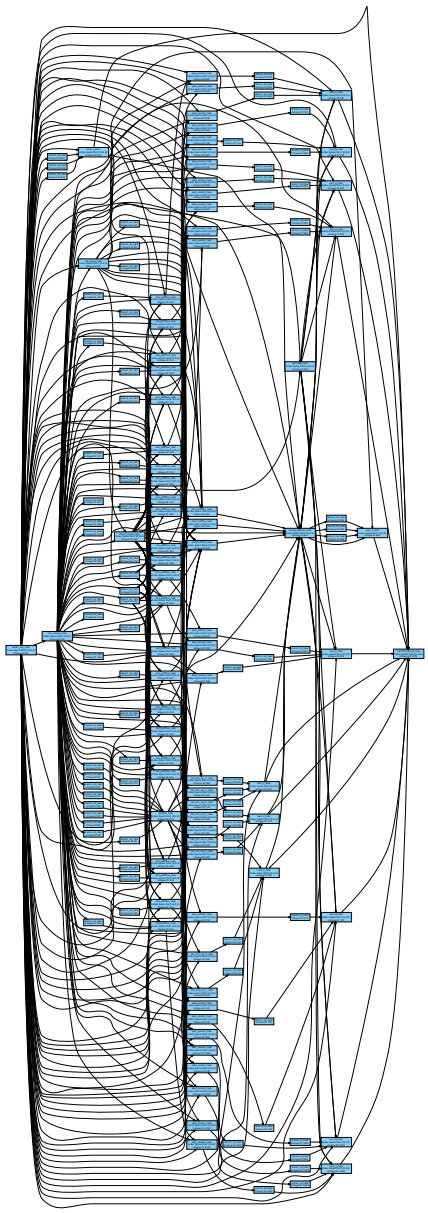


Figure 4.9: Example Dynamic Dependence Graph from S3D

a task finishes the mapping stage, it notifies all its registered waiters to indicate that it has finished mapping. If any of the waiters no longer have any outstanding registrations, then they are placed on the *ready queue* of tasks that are ready to map, and the mapper is queried to choose which tasks in the ready queue should be mapped

next (see Section 8.1 for more details on mapper queries and the ready queue).

The second purpose of the edges in the dynamic dependence graph are to act as *commit edges*. Edges which represent true data dependences (those on which the privilege of the source is a reduction or a write, and the privilege of the destination is either read-only or read-write) and for which the destination logical region is at least as large as the source logical region are considered commit edges. Commit edges help to govern the reclamation of the dynamic dependence graph so that the entire graph need not persist throughout the duration of the execution of the parent task. Long running parent tasks can generate hundreds or even thousands of sub-tasks and operations, resulting in very large dynamic dependence graphs. As tasks finish, commit edges govern the parts of the dynamic dependence graph that can be reclaimed.

The direction of commit edges is the opposite of the direction for dependences in the dynamic dependence graph. Every node N in the dynamic dependence graph registers itself with all of the other nodes that are sources on commit edges that point to N. As tasks commit, they notify all tasks that have registered commit dependences on them. If at any point all of the commit edges pointing at a node have been satisfied then the node can be safely reclaimed. The intuition is that once all the commit edges have been satisfied, then there are no later tasks in the stream of sub-tasks S that can trigger a roll-back that might require a re-execution of the task at node N. There are two other ways in which tasks can pass the commit stage that allows them to be reclaimed, both of which are discussed in Chapters 8 and 10 respectively.

Due to the number of places in both the region tree forest and the dynamic dependence graph that contain references to objects that represent sub-task and other operations, it is expensive to go through all these data structures and prune out references. Instead, we borrow an idea from [48] and recycle the objects that represent sub-tasks and other operations. We assign each use of these objects a *generation* that identifies a specific operation that the task is using. We then update all references to these objects to include the generation. All methods on the object (e.g. for performing registration) require that the generation be passed as well. If the generation is older than the current generation, then the method will return that the version of the

operation being referenced has already committed. Once a task has committed, it increments its generation, and then adds itself back to the pool of available objects managed by the runtime to be recycled.

4.5 Memoizing Logical Dependence Traces

One very important optimization that the dependence analysis stage supports is the ability to capture *traces* of task executions. Capturing traces of sub-tasks is inspired by and has a direct analogy to a common feature in hardware out-of-order processors: trace caches. The idea behind capturing traces of execution is that the dependence analysis stage is the only inherently sequential computation in the Legion task pipeline. All other stages permit multiple tasks or operations to be processed in parallel. The sequential nature of this stage means that it can be very expensive to do for large numbers of tasks. By capturing traces of a subset of operations in a stream, we can memoize the dependence analysis results and replay them later when the trace is encountered again.

The motivation for incorporating trace capture as an important optimization is that many Legion applications employ large loops of repetitive task execution. For example, most scientific simulations execute for a large number of time steps, executing either the same or similar sets of tasks for each time step. Similarly, iterative solvers execute the same set of tasks until they converge. Owing to the multitude of Legion applications that share this structure, we deemed it prudent to include tracing as an optimization.

Unlike traditional hardware trace caches that are invisible to the application, the Legion programming model requires applications to explicitly indicate the beginning and end of a trace with a runtime call. Both the start and end runtime call take a trace ID parameter for identifying the trace. The first time a dynamic trace ID is encountered within a context, the runtime initializes an object for capturing the trace. Each sub-task or other operation that is launched during the trace is recorded. All dependences in the dynamic dependence graph between operations in the trace are also recorded. Dependences between an operation inside the trace and a second

operation outside of the trace are not recorded. As we will see, we rely on a second mechanism to handle these dependences when the trace is re-executed. It is important to note that we can eagerly prune dependences for tasks that have already finished executing when capturing a trace. Even through some operations in a trace may have already committed, which commonly occurs in larger traces, we still must record the dependences since there is no guarantee that the same scenario will occur when the trace is replayed. Once the trace capture is complete, the trace is frozen and can be re-used. Trace objects only exist within the context of a parent task and are automatically reclaimed when the parent task completes.

After a trace has been captured, the next time the trace is encountered, the Legion runtime can automatically replay the results of the dependence analysis stage instead of having to recompute the dependences. In order to avoid missing dependences between operations within the trace and operations that come before and after the trace, the runtime inserts a mapping fence (see Section 2.7.3) both before the trace begins and after the trace ends. These fences ensure that any missing dependences across the trace boundary are included. While these fences do add additional dependences to the dynamic dependence graph which may constrain execution, the assumption is that the size of the trace will be sufficiently large to amortize the additional cost of the mapping fences.

Another important requirement of the tracing feature is that the re-execution of the trace executes an identical stream of operations to the one that was captured. The onus is currently on the user to guarantee this property, and any failure to maintain this invariant will result in a runtime error. In the future, we hope that additional logic can be added to both the Legion compiler and runtime to automatically add support for tracing. It may be possible for the Legion runtime to recognize long streams of isomorphic tasks that are constantly replayed and should therefore be traced. Furthermore, it should be possible for the Legion compiler to recognize loops in Legion tasks that generate the same stream of sub-tasks with no control statements and insert the proper tracing calls.

Chapter 5

Physical Dependence Analysis

After tasks have completed the logical dependence analysis stage of the Legion task pipeline, they then proceed to the mapping stage of the pipeline (see Chapter 3). The process of mapping entails selecting a processor on which to execute each task and then determining memories in which to place physical instances for each of the logical regions on which the task has requested privileges. It is the responsibility of the runtime to compute all of the data movement operations and preconditions necessary for actually executing a task in accordance with the requested logical region privileges and coherence. We refer to this process as physical dependence analysis as it involves actually computing the operations necessary for mapping a Legion application onto real hardware.

To ensure that dependences and data movement operations are determined correctly, a task T is only allowed to map when all of the other tasks in the dynamic dependence graph on which T has a dependence have mapped. This invariant guarantees that when a task T maps, it will see the correct version of the meta-data stored in the physical state of the region tree forest and pick up any necessary dependences on other tasks and operations that came before T in program order. An important property of this invariant is that it permits Legion to map some tasks while logical dependence analysis is being performed on other tasks. It is impossible for task T to record a mapping dependence on any task that comes after it in program order. This property allows Legion to execute these stages of the pipeline in parallel for different

tasks, aiding in the latency hiding of the dynamic analysis.

In many ways the physical dependence analysis for the mapping stage is similar to the logical dependence analysis: both algorithms involve traversals over the region tree forest to determine dependences. However, there are two primary differences. First, tasks can begin physical dependence analysis once all their mapping dependences have been satisfied, permitting multiple tasks to be performing their physical dependence analysis in parallel. Second, the state maintained in the region tree forest will be different: the runtime must track the various physical instances of logical regions as well as the low-level events necessary for computing preconditions for low-level runtime operations.

In this chapter we describe the physical dependence analysis that implements the mapping stage of the task pipeline. We begin in Section 5.1 by giving a general overview of the traversal algorithm. In Section 5.2 we cover the details of the data structures and optimizations we apply to make the physical region tree traversal fast. Section 5.3 covers the implementation of several important features of the Legion programming model including virtual mappings, reduction instances, composite instances, and compound region requirements. Finally, in Section 5.4 we describe the necessary algorithms for enabling parallel traversal of the region tree forest for non-interfering tasks.

5.1 Physical Region Tree Traversal Algorithm

The physical region tree traversal algorithm is in some ways similar to the logical region tree traversal algorithm described in Chapter 4. However, the goal of the physical traversal algorithm is not to construct a dynamic dependence graph, but instead to issue the appropriate operations to the Legion low-level runtime (see Section 3.1.3) necessary for the execution of tasks and other operations. Since the issuing of low-level operations also corresponds to the binding of an application to the hardware, the physical traversal algorithm makes several mapper queries to determine how each task should be mapped onto the target hardware, ensuring that all decisions that impact performance can be controlled by the application developer. A partial semantics

of these mapper calls are given in this chapter, while a complete description of the queries that can be made to mapper objects can be found in Chapter 8.

In order to map a task, a target processor must be first be selected via a mapping query. After the processor is selected the runtime requests that a mapper determine a ranking of memories for each region requirement by an additional mapping call. Based on these rankings, the runtime attempts to map each of the region requirements to a physical instance in one of the memories in the target ranking[1]. In order to map each of the different region requirements, the runtime either locates or tries to create a physical instance in one of the target memories. After a physical instance is selected, the necessary copy operations are determined and event dependences computed. To perform these actions for each region requirement of a task, four different stages are required. Each stage requires a traversal of a different component of the physical state of the region tree forest. We discuss each of these different traversals in turn.

5.1.1 Premapping Traversal

The first traversal of the region tree is the *premapping traversal*. For each region requirement in the task, the premapping traversal walks the physical region tree from where the parent task had privileges to where region requirement is requesting privileges similar to logical region traversal. However, instead of searching for interfering tasks, this traversal *opens* all sub-trees along the path and performs any necessary close operations recorded by the logical traversal in Section 4.3.4 that have yet to be performed. The intuition is that the logical dependence analysis algorithm ensures that tasks map in the proper order by recording dependences, therefore, in order to guarantee that the physical region tree remains in the proper state, only the open and close operations need to be performed during this first stage. If any close operations along the path have already been performed by another task in the same epoch then they can be skipped.

Unlike close operations in the logical region tree traversal that only mutate the state of the meta-data in the region tree forest, close operations performed on the

[1]As part of this process, the runtime also filters memories that are not visible from the target processor to maintain the correctness of the eventual mapping.

physical state are translated to low-level runtime operations. To perform a close operation, a physical instance of the logical region at the root of the sub-tree to be closed, must first be either selected or created. (Note this physical instance must also have sufficient space for all the fields that are being closed.) Since the decisions about placement and layout of this physical instance can have performance ramifications, the `rank_copy_targets` mapper call is invoked to pick both the location and layout of the physical instance to be used as the target of the close operation. The mapper query is told about all the existing physical instances that meet the necessary qualifications in terms of space and needed fields, and the mapper can decide whether to re-use an existing instance or request the creation of a new instance in any number of memories. The mapper also has the ability to choose if multiple instances should be created in different memories for the close, which is a useful feature when there may be many tasks running in different parts of the machine that will require a copy of the data in the closed logical region. Like many mapper calls, it is possible for the `rank_copy_targets` mapper call to *fail-to-map*. If there exists no physical instance to re-use, and all the target memories selected by the mapper are full, then the mapping call will fail. Under such circumstances, the premapping traversal will also fail. When this occurs the mapper is notified by the `notify_failed_mapping` call and the task is placed back on the ready queue, where the mapper can choose when to next try mapping the task.

Once one or more target instances for the close operation have been successfully identified, the close operation can proceed. To close a sub-tree, the target physical instances are first brought up to date at the root of the tree by issuing copies from the existing physical instances containing valid data. (We discuss the implementation of these copies later in this section.) If the targets are already valid instances, then nothing is done. The close operation then proceeds down the sub-tree being closed and issues copies from any instances containing dirty data to the target instances. Since the most recent dirty data is always at the lowest levels of the open sub-tree, by traversing from top-to-bottom, we guarantee that dirty versions of data are applied in the correct order to the target instances. Once the close operation is complete, the target instances are all registered as valid versions of data at the node at the

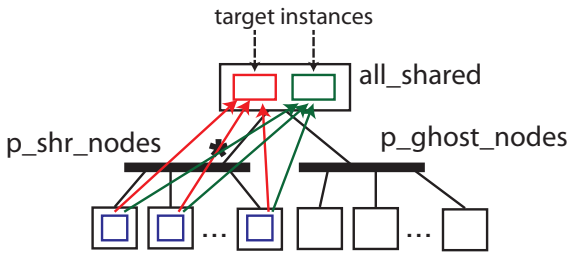


Figure 5.1: Example Close Operation from the Circuit Simulation

root of the close operation. If any dirty data was found, all previous valid physical instances at the root that were not targets are invalidated. After the close operation is complete, we record that it has been performed so that any other tasks in the same epoch will not need to repeat it.

Figure 5.1 shows an example close operation from the circuit simulation that occurs between the `update_voltages` and `calc_new_currents` tasks. The first instance of the `calc_new_currents` tasks to premap will be the one that triggers the close operation. In this particular close operation, all of the data in the `p_shr_nodes` partition must be copied back to at least one instance of the `all_shared` logical region. Performing this close operation is a necessary precondition for opening the `p_ghost_nodes` partition that is necessary to premap all of the `calc_new_currents` ghost logical regions. In this particular example, the mapper decided to create two target physical instances of the `all_shared` logical region (shown in red and green), so copies are issued from all physical instances in logical regions of the `p_shr_nodes` partition. After the copy operations are complete, both instances of the `all_shared` logical region will contain identically valid versions of the data.

One important detail regarding the premapping traversal is that it must always be performed on the same node as the parent task before any physical region tree meta-data is moved to remote nodes for mapping (see Chapter 6 for more details). The reason for this is that we need to be sure that close operations are performed exactly once, as the updates that they perform to the region tree meta-data must be serializable. Therefore, since all sub-tasks start on the same node as the parent task, by requiring them all to complete the premapping traversal on the parent task we can be sure that close operations are only performed one time. Requiring premapping

traversals to be done on the parent node also has the added benefit of reducing the amount of meta-data that must be sent to other nodes for remote mapping as only the sub-trees of logical regions requested by the region requirement are needed for the remaining stages.

5.1.2 Mapping Traversal

The second traversal that is performed is the *mapping traversal*. The purpose of this traversal is two-fold: first, in the case of projection region requirements it traverses from the region tree node where privileges were requested to the region tree node where mapping will take place, and second, it actually selects which physical instance to use (or re-use) for mapping the region requirement without making it valid. Unlike the premapping traversal that was required to be performed on the origin node of the task, the mapping traversal can be performed remotely. To map remotely, the necessary meta-data for the physical region tree must be moved to the target node. We discuss movement of meta-data for remote mapping further in Chapter 6. We now cover the two phases of the mapping traversal in more detail.

The first phase of the mapping traversal computes the path from where the original region requirement requested privileges to where the individual task's region requirement is requesting privileges. For individual tasks and other operations that use normal region requirements, this path consists of a single node and requires no work. However, for index space tasks that can use projection region requirements (see Section 2.5.4), this path can be arbitrarily long. Figure 5.2 shows the difference between the premapping and mapping paths for a projection region requirement of the `update_voltages` task from the circuit simulation. The premapping traversal to the `p_shr_nodes` partition is done exactly once for the index space task. However, the mapping traversal is done individually on a point-wise basis from the `p_shr_nodes` partition to the logical regions on which each point task is requesting privileges.

The one important invariant that is different in the mapping traversal from the premapping traversal is that no close operations will need to be performed as part of the mapping traversal. By using projection region requirements to scope an upper

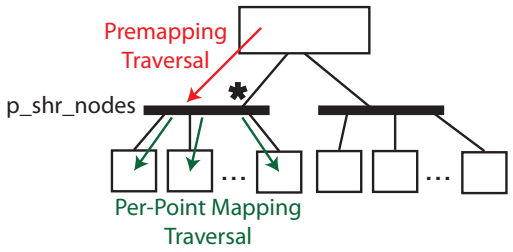


Figure 5.2: Premapping and Mapping Paths

bound on the privileges requested by all tasks in an index space task launch, we guarantee that all the necessary close operations are computed and performed as part of the index space task's premapping, and therefore no close operations need to be performed for point tasks within the index space task. As part of the mapping traversal, the necessary sub-trees within the region tree physical state are opened to reflect that there are potential users in the sub-tress.

After arriving at the destination node, the traversal then attempts to either find or create a physical instances of the target logical region to hold the data necessary for satisfying the region requirement. Determining the physical instance to select requires input from the mapper since both the placement of the physical instance as well as its layout can impact the performance of the application. We therefore ask the mapper to assign a ranking of memories and set of properties (e.g. layout information) for all region requirements prior to performing any of the mapping traversal by invoking the `map_task` mapper call. The mapper decorates the region requirements for the task with its chosen ranking of memories and required set of constraints on the layout of the physical instance. Based on these requirements, the runtime then sequentially walks through the list of memories, and for each memory attempts to either find or create a physical instance that meets the constraints. When looking for a physical instance that meets the constraints, the runtime can select any instance from the current node, or from any of the nodes above it. In the process of selecting nodes from further up in the tree, we must be careful to invalidate fields from higher nodes when those same fields for instances lower in the tree contain dirty data. It is safe to re-use the instances, but it will require additional data movement to bring the fields up-to-date. If no instance can be found or created for any of the region requirements then the

mapping stage fails, and the mapper is notified with the `notify_failed_mapping` mapper call.

It is important to note that after the mapping traversal is complete, we do not actually issue any data movement operations for updating physical instances. Instead this is done in the third traversal when we register the physical instances (see Section 5.1.3). The reason for this discrepancy is that in some cases, we may only discover that a mapping has failed after we have completed the mapping traversal for several other region requirements. In these circumstance, we need to free any physical instances that we have created as part of the mapping traversal in order to avoid resource deadlock. Rolling back these allocations would be complicated if we had already issued copy operations to bring the instances up-to-date. Therefore, we defer issuing any of these operations until we know for sure that the mapping will be successful for all region requirements that a task requested.

In some cases, the mapper for a task may determine that it does not require the creation of a physical instance for a region requirement, and instead only needs to pass privileges for the task to further sub-tasks. Under these circumstances we allow mappers to request *virtual mappings* of a region requirement, which will not create a physical instance. Virtual mappings trivially succeed in the mapping traversal phase. We discuss the implications of virtual mappings in more detail in Section 5.3.3.

5.1.3 Region Traversal

Once we have completed the mapping traversal for all region requirements for a task, we know that the mapping stage of the pipeline will be successful. At this point it is safe to issue the necessary copy operations for ensuring that the physical instance selected for each region requirement has the correct version of the data for the requested fields of the logical region. To issue the necessary copies required for making the physical instances valid copies of the data, we first determine the set of fields that are invalid. It is possible that the instance that we have selected is already valid for all the fields at the target logical region, in which case no copies need to be issued. Its also possible that one or more fields are invalid, and therefore must be

updated.

To perform the updates we first compute the set of *valid instances* for each field. The valid instances are the instances that contain valid data for the specific fields that need to be updated from the perspective of the target logical region. When computing the set of valid instances, it is legal in some cases to use physical instances from logical regions higher in the region tree. Determining when instances from logical regions higher in the region tree can be used is non-trivial due to the existence of buffered dirty data within the physical instances. To compute the valid instances from the perspective of a logical region R, we start at R and traverse back up the tree. As we traverse up, we maintain a field mask that describes the *search fields* for which we are looking for a valid instance. Before traversing up the next node, we remove any fields that contain dirty data at the current node from the search field mask, thereby preventing us from considering instances higher in the region tree as valid. At each node we look for physical instances that contain valid data for the target field. We record all physical instances that have valid data for any of the fields for which we are searching. We maintain a field mask for each of these physical instances; the field mask records the subset of fields for which they contain valid data. When the search field mask is empty, or we reach the region at the top of the parent task's privilege context, we can return the set of valid instances.

Figure 5.3 depicts an example search for valid instances in a region tree for two fields A and B. Starting from the query region, the traversal proceeds up the tree. Initially both the A and B fields are in the search field mask. Instance I_4 contains a valid copy of A and is added to the result set. We then progress to the parent logical region where we discover instance I_3 that contains a valid instance for both A and B which is added to our result set. Before traversing the grandparent region, we first must remove field A from the search field mask because it is a dirty field in the parent logical region. This indicates that any instances in ancestor regions will contain stale data for field A. Therefore, when we arrive at the grandparent region, we can safely add instance I_2 to the result set for field B. However, we cannot add either I_1 or I_2 for field A because A is no longer in our search field mask. Ultimately, our result set indicates that instances I_3 and I_4 contain valid data for field A and instances I_2 and

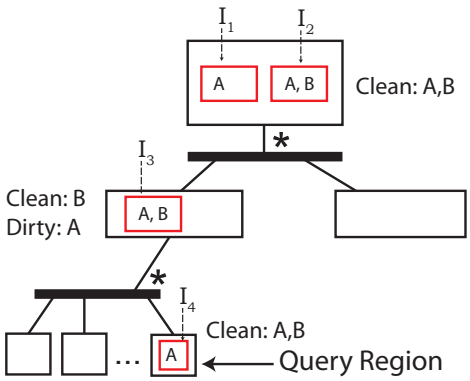


Figure 5.3: Example Valid Instance Views Computation

I_3 contain valid data for field B.

Once we have the valid set of instances for all the fields we need to update, we can issue the copy operations to update our target instance. In many cases, there may be multiple physical instances that can serve as source instances for the copies. Since the choice of sources may impact performance, we consult the mapper when necessary about selecting the sources for copy operations. If there are multiple valid instances, we query the mapper to rank the memories that should serve as sources for the copies. We invoke the mapper by calling the `rank_copy_sources` mapper call and provide the mapper with a list of the memories that currently contain valid physical instances and the memory of the destination physical instance. The mapper returns an ordered ranking of the memories or a subset of the memories in which case the other memories are appended to the ranking in a random order to maintain correctness[2]. Using this ranking, the runtime then issues the necessary update copies from physical instances based on the ranking of memories. Only one copy needs to be issued for each field. Where possible, copies are issued jointly as gathers from multiple physical instances over multiple fields to afford the low-level runtime maximum visibility in optimizing copy routines and fusing together data movement to ensure bulk data transfers.

Now that we have an up-to-date physical instance at the target logical region, we lastly need to flush any dirty data from below in the tree up to the current logical

[2]Even if the mapper does not rank all memories, the runtime still needs to guarantee that copies get issued to update all fields with valid data, hence the necessary appending to the list.

region. We examine all of the open children to determine if they have any dirty data. If they do we issue a close operation with our selected physical instance as the target. This ensures that any dirty data below in the tree is properly added to our physical instance. The privileges requested by the region requirement determine if the sub-trees can be left open: read-only privileges permit sub-trees to remain open while read-write and reduce privileges do not since these privileges invalidate the instances in the sub-trees. After any close operations are performed we can register our new physical instance as a valid physical instance at the current node in the region tree. If any dirty data was flushed up from sub-tress, then we update the set of dirty fields at the current node and invalidate any other physical instances at the current node (since they are no longer valid). We then add our physical instance to the list of valid physical instances for the fields requested by our region requirement.

5.1.4 Physical Instance Traversal

After we update all the field data for each physical instance that is going to be used for a task, we now need to determine the set of dependences that must be satisfied before we can safely use each physical instance. These dependences take the form of low-level runtime events corresponding to other tasks and operations that are already using the physical instances. Once we have computed the set of event preconditions, we can merge them together into a single low-level runtime event and launch the corresponding low-level runtime task with the proper event precondition.

To determine the event preconditions for a new user, we perform a traversal over the *instance view tree* for the target physical instance. An instance view tree is a data structure that mirrors a region tree and tracks users of a physical instance from the perspective of different logical regions. We cover the details of instance view trees in further detail in Section 5.2.3.

Figure 5.4 shows the instance views in the instance view tree that need to be checked when performing a physical instance traversal for a new user of a physical instance. Views highlighted in red must be visited to check for dependences, while those in blue can be skipped. Specifically, the user must visit the instance view at

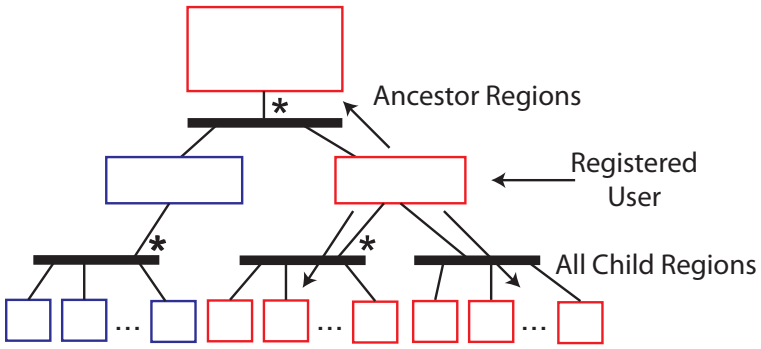


Figure 5.4: Instance View Tree Traversal

which it will be registered as well as all ancestor views and all sub-tree views. While it is possible for there to exist interfering users from the blue instance views, they will be detected in the ancestor parent views because all users are registered in all their ancestor instance views as well (see Section 5.2.3 for more details). At each instance view that is traversed, the user being registered examines all the currently registered users for interference based on the same criteria as was presented in Section 4.1. To aid in discovering disjointness on logical regions of users registered in ancestor instance views, we also annotate users with which sub-tree they came from, allowing region disjointness to be determined without traversing any sub-trees. The output of the instance view traversal is a set of low-level runtime events that describe all the interfering users in the instance view tree. The runtime can then use this set of events to construct a single low-level runtime event that serves as the precondition for an operation using the physical instance. Once the analysis is complete, the user is itself registered in the tree with its termination event. The user is registered in all instance views from the view in which it is requesting privileges up through all ancestor instance views to the root of the instance view tree.

After we have completed the physical instance traversal for each of the region requirements of a task, we have the set of all low-level runtime events that must trigger before the task can run. Using the low-level runtime interference, we merge all these events together and launch the task with the computed event as a precondition. The event that is returned is used as the event that marks when the task will be done executing and we use it for registering the user in the instance view tree.

One important detail is that the same instance view tree traversal algorithm described here is also used in Section 5.1.3 to compute the preconditions for issuing copy operations to and from physical instances. Both the source and destination region trees are analyzed. Copies take the form of a special kind of user with exclusive coherence and either read-only, read-write, or reduce privileges depending on the kind of copy being performed. The low-level runtime also takes an event precondition on copies and returns an event for describing when the copy will be finished. Using this low-level event, copies also register themselves as users in the instance view trees to ensure correct dependence analysis when computing the set of precondition events for any future user.

5.2 Physical Region Tree Data Structures

Much like the state required for the logical dependence analysis described in Chapter 4, our physical region tree traversal for performing the mapping analysis also requires the storage of meta-data on nodes in the region forest. While some of the the meta-data stored on individual nodes is the same as the logical meta-data, some aspects are different. We describe the specific state required for each node in Section 5.2.1. In addition to storing state on each node, the physical region tree traversal will also require two new kinds of data structures: *instance managers* for tracking physical instances and *instance views* for tracking the operations that are using the physical instances. We provide descriptions of these data structures in Section 5.2.2 and 5.2.3 respectively.

Similar to the logical dependence analysis data, physical region tree traversal meta-data is decorated on nodes of the region tree forest. To differentiate the state for different task executions, we again use the notion of contexts, initially introduced in Section 4.3.6. The same context ID that is allocated for non-leaf parent tasks for performing dependence analysis is used for performing the physical region tree traversal for sub-tasks launched within a parent task (with the exception for virtual mappings discussed in Section 5.3.3). Each region tree node maintains a separate array of physical meta-data objects that can be directly indexed by the context ID,

allowing traversals from different contexts to occur independently.

5.2.1 Region Tree State

On each node of the region tree, several different meta-data structures are maintained. First, similar to the logical state data, each physical state for a node tracks which of the sub-trees are *open*. There is a slight difference in the definition of open for the physical state of the region trees: a sub-tree is said to be open for a field f if it contains at least one valid physical instance for f at some node in the sub-tree. Note that this is different from the logical state where a sub-tree was considered open if there were operations registered in the sub-tree. Unlike the logical region tree state, where both field and the privilege modes for open sub-trees were tracked, the physical region tree traversal only needs to track which field of sub-trees are open. The reason for this is that the logical region traversal memoized the results of the privilege analysis when the close operations were recorded for each operation. By applying the close operations as part of the premapping traversal discussed in Section 5.1.1, sub-trees will be implicitly open with the proper privileges (e.g. multiple sub-trees will only be open if multiple read-only or reduction tasks are active in those sub-trees). Open information for individual fields are still required to know whether a sub-tree is actually open or whether it was closed by another task in the same epoch during its premapping traversal.

The second piece of meta-data state stored on each region tree node is a field mask that tracks dirty information. The dirty field mask records which fields at the current region tree node contain dirty information (e.g. have been written to). When a region tree is going to be closed, dirty information is necessary for computing which data movement operations are required to propagate information back to higher levels in the region tree.

The final component of the physical state at each region tree node is a list of *valid instances*. This list contains a list of instance view objects (see Section 5.2.3) that represent physical instances that contain a valid version of data from the local region tree node. Each valid instance is accompanied by a field mask that describes the

specific fields for which the physical instance contains a valid instance of the data. It is often possible that a physical instance only contains valid data for a subset of the fields for which it has allocated space.

5.2.2 Physical Instance Managers

When physical instances are allocated, the Legion runtime creates a *physical instance manager* to keep track of the properties of the physical instance. These properties include the memory in which the instance is allocated, the location in that memory, the layout of the fields, and the linearization of the index space. When copies are issued either to or from a physical instance, the physical instance manager is responsible for generating the necessary instructions for directing the low-level runtime about how to perform the copy. For instances with hundreds or thousands of fields, this information can be expensive to compute. We therefore cache the meta-data required for issuing copies so that it can be re-used when similar copies are requested.

Since instance managers are often large objects that contain no context specific information, we deduplicate them across physical contexts and ensure that at most one physical manager exists for each physical instance on a single node. We describe how physical managers are lazily migrated between nodes in Chapter 6. The per-node uniqueness property guaranteed by physical instance managers also serves a useful purpose in our garbage collection mechanism for physical instances that we describe in Chapter 7.

5.2.3 Physical Instance Views

While instance managers are used to describe the properties of a physical instance, we use a separate collection of objects to track the users of physical instances. An *instance view* is an object that records all operations (e.g. tasks, copies, inline mappings) that are using a specific physical instance from the perspective of a specific logical region. A physical instance initially created for a logical region R can serve as a valid physical instance for any logical region in the sub-tree of R. In some cases there might be multiple tasks using the same physical instance from the perspective of two disjoint

logical sub-regions. In cases where these tasks might be interfering on everything except region disjointness, it is important to be able to detect these cases to avoid false dependences. By providing different instance view objects for the same physical instance, we can maintain precise information about how different users are using the same physical instance and accurately detect when dependences need to exist between users.

An instance view tracks the set of users of a specific physical instance from the perspective of a single node in the region tree. The collection of instance view objects for a specific physical instance are therefore organized as a tree that mirrors the shape of the region tree from which the physical instance originates. For physical instances that contain valid data in multiple contexts there is a separate set of instance views for each physical context within the region tree to ensure that users from different contexts do not need to perform non-interference tests against users from other contexts[3]. All users are registered with instance views from the region in which they requested privileges to the root of the instance view tree. (Note the root of the instance view tree is always the logical region that initially created the physical instance.) Registering users in this way will enable the simple algorithm for finding interfering users described in Section 5.1.4. When a user is registered we record the privileges, coherence, and a field mask for describing which data is being accessed and how it is being accessed. Each user also records a low-level runtime event that will be triggered upon completion, allowing later tasks to know when the user is done using the physical instance.

As an important optimization for tracking the users from the perspective of an instance view, we also make use of *epoch lists*, similar to the ones introduced in Section 4.3.2. Epoch lists provide a mechanism for reducing the number of users that must be tracked by each instance view object. Identical to the epoch lists of logical state objects, each instance view object maintains two epoch lists: a current epoch list and a previous epoch list. Whenever an added user dominates all of the previous users for a specific set of fields, we can remove the users of those fields in the previous

[3]This is sound because of the hierarchical privilege property: if two tasks are non-interfering then so are any children they generate.

epoch list and filter the users from the current epoch list into the previous epoch list. If the user does not dominate the users in the current epoch list, then it must also check for dependences in the previous epoch list and then add itself to the current epoch list. The same observable condition on domination applies: a user performing interference tests must observe at least one other user of a field before being able to safely conclude that it dominates that field.

Due to the large number of users that can accumulate in the current and previous epoch lists, the runtime supports an important optimization for reducing the number of users. By default, lists are only modified by the standard instance view traversal protocol described in Section 5.1.4. However, users can also set an upper bound for the number of entries in the epoch lists. Whenever the upper bound is reached, the runtime performs three operations to compact the epoch lists: user deduplication, privilege merging, and field merging. The first compaction pass looks for users from the same operation that have been fragmented on fields, due to epochs changing for different fields at different times, resulting in multiple versions of a user ending up in the previous epoch list. The second compaction pass looks for users from the same epoch on the same fields, and merges these users together into a single user entry. Merging users also requires generating a new event that represents all the previous user events which is then stored in the entry for the user. Both of the first two compaction passes still maintain the precision of the analysis. The third operation reduces the precision of the analysis and therefore may be optionally disabled with the understanding that it will cause the upper bound on the epoch lists sizes to be a soft upper bound that may be exceeded. In the third compaction pass, users of different fields with the same privileges are merged, again resulting in a single merged event to represent multiple users. The resulting users must contain a union of all the fields that were merged, potentially resulting in false dependences being created between operations, but at the benefit of a hard upper bound on epoch list sizes.

5.3 Special Cases

Unlike the dependence analysis stage of the pipeline that maintains a relatively homogeneous implementation independent of the privileges and coherence of region requirements, many of the special features of the Legion runtime must be handled differently in the mapping stage of the operation pipeline. The reason for this is simple: the mapping stage is where the logical description of Legion programs is targeted at a specific target architecture. It is therefore this stage that must handle the cross-product of interacting Legion features and how they are mapped onto the hardware. We discuss the implementation of several of the more important features in this section.

5.3.1 Supporting Reduction Instances

One of the important features of the Legion programming model is that reduction privileges permit tasks to be non-interfering. In order to support this feature, Legion takes advantage of specialized *reduction instances* provided by the low-level runtime [48]. Reduction instances are a special kind of physical instance that only support reduction operations; arbitrary reads and writes are not permitted. The low-level runtime provides two kinds of reduction instances: reduction-list instances and reduction-fold instances. Reduction-list instances buffer reductions as a list, while reduction-fold instances maintain separate entries for each row in an index space, but at the significantly smaller cost of only needing to store the *ride-hand-side* (RHS) value of the reduction operator, which is commonly smaller than the target field of the reduction. If multiple reductions are applied to the same row, a reduction-fold instance can perform a *fold operation* to reduce the two intermediate values together, thereby reducing the amount of information that must be stored in the reduction instance[4]. Reduction-list instances work better for tasks that are performing sparse reductions, while reduction-fold instances work best for tasks performing dense reductions (and when a specialized fold operator is not defined for the reduction). Since the choice

[4]A fold operation has the type $RHS \rightarrow RHS \rightarrow RHS$, reducing two elements of the RHS type to one. Fold operations are required to be associative and commutative.

of reduction instance can have an impact on performance, the decision of the type of reduction instance to use is made by the mapper as part of the `map_task` mapper call.

For all region requirements that request reduction privileges, the runtime currently requires that a reduction instance be made. This restriction is what permits tasks requesting reduction-only privileges on otherwise interfering regions to both map and execute in parallel. To track reduction instances, each of the physical state objects for tracking physical traversal meta-data (described in Section 5.2.1), is updated to track all the fields that currently have outstanding reductions, as well as a collection of reduction instances available for the current region tree node. When close operations are performed, reductions are always flushed back up to the target physical instance after all other dirty data has been copied back. This implementation stems from a very important invariant maintained by the logical region tree dependence analysis: whenever sub-trees are open in reduction mode, there can never be any dirty data below the reduction operations. If dirty data does exist in a sub-tree before a reduction is mapped, a close operation is always performed before the tree is opened for reduction operations. This invariant significantly simplifies the reasoning about reductions and allows all reduction operations to be performed last as part of close operations. We will leverage this invariant again in Section 5.3.5 when we discuss composite instances.

5.3.2 Post Task Execution

One important aspect of task execution in Legion is that after a task has finished executing, the data associated with the logical regions on which the task held privileges may actually be strewn across the machine in many physical instances created by sub-tasks and other operations launched by the task. However, the sibling tasks of the parent task have made mapping decisions contingent upon the output of the task being stored in the physical instances that the task initially mapped. Therefore, after a task has finished launching sub-tasks, the Legion runtime issues close operations on the fields of any region trees on which a task held either read-write or

reduction privileges. The result of applying these close operations is that any dirty data residing in other physical instances throughout the memory hierarchy is flushed back to the physical instance that the task initially mapped. Since the purpose of these close operations is purely to aggregate dirty data, there is no need to issue close operations for region requirements with read-only privileges, as it is impossible for any dirty data to reside in these region trees. Issuing of close operations for region trees with potentially dirty data is necessary for correctness of Legion applications. However, as we will see in Section 5.3.3, there are some mapping decisions that can be made that eliminate the need for the close operations.

5.3.3 Handling Virtual Mappings

The hierarchical nature of the Legion programming model encourages tasks that recursively launch sub-tasks. The Legion programming model also requires that privileges for regions be passed from parent tasks down to sub-tasks. For many intermediate tasks in the task tree, it is unnecessary to instantiate a physical instance for the region requirements requested by the task. In some other cases the logical regions may be too large to fit in any memory in the machine. Under these circumstances, it is useful for a mapper to be able to request that a region requirement be *virtually mapped*. In a virtual mapping, no physical instance is actually created for the region requirement, but the task is still granted the requested privileges and can pass them down to any sub-tasks that it launches.

When a virtual mapping is performed for a task, there no longer exists a canonical physical instance of the data when the task starts. Any sub-tasks or other operations that request privileges on the regions therefore need to be mapped differently to ensure that they observe the correct versions of the data. Instead of mapping within the context of the parent task (which is meaningless because of the virtual mapping), we instead map these operations within the context of the closest enclosing ancestor task that did not virtually map the region, or the ancestor task that created the logical region. It is important to note that this definition is inductive, and allows for multiple nested tasks to all perform virtual mappings, which is important for writing

scalable Legion applications with many layers of nested tasks.

Since all sub-tasks and other operations that acquire their privileges from a virtually mapped region are not mapped directly in the parent task context, but instead are mapped in an ancestor task context, there is no need to explicitly issue close operations on the region trees after the task has completed. Instead the information about where data is located has already explicitly flowed back into the ancestor task context. To be sound, this process requires a slight modification to the handling of mapping dependences discussed in Section 4.4. Tasks that request virtual mappings on at least one region requirement cannot be considered mapped until they have finished executing and any sub-tasks or other operations that they launched have also finished mapping. This guarantees that all the necessary mapping information has flowed back into the ancestor task's context before any sibling tasks of the parent task with mapping dependences on the parent task can begin to map. Virtual mappings therefore present an interesting trade-off opportunity for mappers: using non-virtual mappings can enable more parallel mapping opportunities, at the cost of creating intermediate physical instances, while virtual mappings negate the need for creating intermediate physical instances while serializing the mapping process. Determining whether to use virtual mappings is therefore a performance decision that is always left to be determined by the mapper for a task.

Importantly, the same machinery that is used to handle virtual mappings is also used for handling logical regions that are created and returned from a task. Effectively, a newly created logical region is one that was anonymous and virtually mapped before the task was executed. If the newly created region is not deleted before a task completes execution, the privileges associated with that region flow back to the enclosing parent task. To be correct, the corresponding state of the newly created region must also flow back to the enclosing parent task context. The Legion runtime handles this by always performing physical region tree analyses for newly created regions in the context of the earliest ancestor of the creating task on the local node. By using this context, the necessary meta-data does not have to be copied back to enclosing task contexts as privilege information propagates back up the task tree, and instead only needs to be copied when moving across nodes, under which circumstances

it would need to be moved anyway. We discuss the movement of physical state data in detail in Chapter 6.

5.3.4 Optimizing Virtual Mappings with Inner Tasks

Virtual mappings have the detrimental effect of increasing the length of critical paths in the mapping process because sibling tasks of a parent task P with virtual mapped regions must wait for all child operations of P to map. In the general case this requires waiting for P to execute in order to discover all the children. However, because of deferred execution, it may be a significant amount of time before P actually begins, thereby delaying the mapping of any siblings of P. To reduce this latency, Legion permits an important optimization called the *inner task* optimization to reduce the latency of the mapping process when virtual mappings are used.

An inner task is a Legion task that only requests privileges on logical regions and never actually inspects the data contained in the logical regions[5]. Since the parent task P guarantees that it will not be accessing any data within its logical regions as part of its execution, the Legion runtime can actually start the execution of P immediately. Any child tasks of P will automatically pick up the correct dependences by traversing in the context of P's parent task. The inner task optimization also works if P had any non-virtually mapped regions: non-inner task children of P simply add the original precondition events for each physical region of P that they depend on to their preconditions. Ultimately, the inner task optimization reduces the latency of the mapping process by permitting the runtime to execute P early, which leads to early discovery and mapping of children of P.

[5]The term *inner task* originated in the Sequoia programming language where inner tasks were only permitted to launch sub-tasks with `mappar`, `mapseq`, or `mapreduce` operations and could have no control flow dependent on array data. The Legion definition of inner tasks is more flexible and permits arbitrary code execution, but with the same restriction that logical region data cannot be accessed.

5.3.5 Deferring Close Operations with Composite Instances

In early versions of the Legion runtime, all close operations on a physical region tree required the mapper object to select one or more physical instances from the region tree node at the root of the sub-tree being closed to serve as the target of the close operation. This requirement, while unobtrusive in most cases, still results in some cases for which it is impractical. For example, if the region node at the root of the sub-tree being closed constitutes a very large logical region that cannot fit in any memories in the machine, then the close operation cannot be performed. Furthermore, in some cases, building an instance as the target of a close, and then opening a new sub-tree results in a many-to-one followed by one-to-many communication pattern that serializes data movement through a single memory in the machine. Under some circumstances it is better to encourage many-to-many communication patterns. To address these issues, Legion maintains support for *composite instances*.

A composite instance is a snapshot of the physical state of a specific sub-tree in the region forest for one or more fields. Every composite instance supports multiple views onto it from other nodes in the region tree forest. These *composite views* share the same interface as instance view objects (see Section 5.2.3), but instead of being backed by a specific physical instance, are instead backed by a collection of physical instances from the captured region tree state.

Composite instances can never be used directly by tasks or as the target of copies. Instead, they can only serve as the source for copy operations. When a copy is issued from a particular composite view of a composite instance, all of the potentially overlapping parts of the captured tree state must be explored to determine the actual copies that must be performed. In essence, this analysis is an intersection test: for some target region R in one sub-tree, find all of the regions in another sub-tree that potentially intersect with R and issue copies from physical instances in those regions to the target physical instance to ensure that the physical instance of R contains valid data.

Issuing copies from composite instances can be expensive, therefore our implementation attempts to minimize any overhead. First, for each field, every composite view maintains a pointer to the node in the captured sub-tree that dominates the logical

region being represented by the composite view. Since these nodes are often well below the root of the composite instance, this reduces the number of nodes that must be considered for intersections when issuing copies. To further reduce overhead, the runtime also memoizes the results of domination and intersection tests between different index spaces and index partitions, allowing the cost of these tests to be amortized across the creation of many composite instances (with the exception that domination tests must be invalidated under dynamic allocation similar to completeness tests, see Section 4.3.1).

The impact of composite instances is profound. Instead of having to rebuild a physical instance at the root of a sub-tree in a memory, instead a composite instance is built. When other sub-trees are opened, the necessary copies, and only the necessary copies, are issued to bring physical instances in the new sub-tree up to date. Most importantly, multiple tasks that open a new sub-tree can be mapped in parallel, possibly on different nodes, and composite instances permit the necessary copies for updating each of the new physical instances in parallel, increasing the amount of data movement parallelism that is exposed to the low-level runtime. Ultimately, composite instances allow applications to fully leverage the multiple views onto logical regions enabled by the Legion programming model's multiple partitioning feature (discussed in Section 2.3.2). Supporting multiple partitions is essential for modern supercomputing applications that go through many phases of computation; composite instances make the Legion implementation of multiple partitions efficient.

5.4 Parallelizing Physical Tree Traversal

Unlike the dependence analysis stage discussed in Chapter 4, it is possible for multiple non-interfering tasks within the same context to map in parallel on the same region tree. We therefore need an approach to synchronize access to the physical state data structures in the region tree forest. Before beginning our discussion of our approach, we note that it is also possible for multiple non-interfering sub-tasks within the same context to be mapping in parallel on distinct nodes; we discuss the management of distributed physical state across multiple nodes in detail in Chapter 6. In this section

we focus on the synchronization necessary for performing simultaneous mapping of non-interfering tasks in the same context on a single node.

5.4.1 Exploiting Field Parallelism on Premapping Traversals

The first synchronization guarantee that needs to be upheld by the runtime involves the serializability of mapping operations for individual fields. As we discussed in Section 5.1.1, premapping traversal of a field need to be serialized to ensure that close operations are not duplicated. To maintain this property the context of the parent task tracks which of the region trees have active tasks performing premapping traversals. The parent task only permits one premapping traversal to be active in a sub-tree for a given field at a time. Note that this requirement only applies to premapping traversal, allowing any arbitrary number of mapping traversals to be occurring in parallel with any premappings on a field; if they could not be performed in parallel, they would have been determined to be interfering in the dependence analysis stage. Additionally, because the parent context tracks premapping traversals at the granularity of fields, multiple premapping operations on disjoint sets of fields may be proceeding in parallel. This approach to premapping maintains the necessary serializability guarantee while still permitting many parallel traversal operations.

5.4.2 Region Tree Locking Scheme

The second invariant that must be maintained is that access to the physical state on every region tree node is serialized. The reason this is necessary is that often many premapping and mapping traversals might traverse the same region tree node in parallel. These traversal are clearly non-interfering, but they still need to serialize their access to common data structures in the physical state such as the list of valid physical instances, which they might both be mutating (in a non-interfering way) at the same time.

To serialize access to the physical state object in each region tree node, we create a low-level runtime reservation object for the physical state in each context. Low-level reservations provide an atomicity primitive necessary for serializing access to the

physical state[6]. One important property of the premapping and mapping traversals is that if they are occurring in parallel we already know that they are non-interfering. This allows us to perform an important optimization when accessing the physical state object. Unlike traditional synchronization schemes (usually based on locks), that require an entire mutation of the state of an object to be done atomically, traversals in Legion can take a reservation to access the physical state, make a thread-local read-only copy, and then release the reservation. This local copy can then be safely used for performing operations without holding the lock. The reason that this is sound is that the dependence analysis already guarantees that no interfering traversal can be occurring in parallel, so any updates made to the physical state while a traversal is not holding a reservation are not important for correctness. By not requiring traversals to hold state locks throughout the duration of a traversal (either premapping or mapping), a much finer-grained serialization scheme is achieved which enables more parallelism in the dependence analysis stage of the pipeline.

5.4.3 Instance View Locking Scheme

The final serialization guarantee that we must make involves parallel traversal of the instance view objects. Similar to the need to serialize access to the physical state of region tree nodes in a context, we also must serialize access to the data structures inside of physical instance view objects. We again employ low-level runtime reservations as our atomicity primitives. The non-interfering property of traversals guaranteed by dependence analysis also applies to instance view traversals (see Section 5.1.4). Therefore we can employ the same optimization introduced in Section 5.4.2 of not needing to hold a reservation for an entire mutation of an instance view. Instead it is common for traversals to acquire the reservations in read-only mode to search for interfering users, release the reservation, and then re-acquire it in exclusive mode to add a new user to the epoch lists. Ultimately the two-phased locking (non-exclusive followed by exclusive) allows more traversals of instance views to occur in parallel.

[6]Reservations also allow us to encode that some operations are read-only and are therefore not exclusive, while others operations require exclusive access for mutating the physical state object.

Chapter 6

Distributed Execution

The third stage of the Legion task pipeline is the distribution stage. The distribution stage entails moving a task and any necessary meta-data to the target node on which the task is assigned to run by the mapper that owns the task. The collection of mappers that own a task also have control over whether the task is mapped *locally* (on the node on which the task originated) or *remotely* (on the target node). This decision determines the ordering of the mapping and distribution stages of the pipeline. If a task is assigned to a remote node, the decision also determines what meta-data must be moved in conjunction with the task.

In order to facilitate the distributed execution of Legion tasks, the runtime must operate similarly to a distributed system that is capable of reasoning about partitioned meta-data structures. In this chapter, we describe the algorithms necessary for allowing the Legion runtime to support distributed execution of Legion applications (with the caveat that we omit discussion of resiliency to Chapter 10). We begin in Section 6.1 by describing the way messages are communicated between nodes and the invariants supported by the communication model. Section 6.2 describes how both individual and index space tasks perform the distribution stage of the task pipeline. In Section 6.3 we detail how region tree data structures are migrated through a distributed address space. Finally, we cover the details of how Legion supports distributed physical contexts for parallel mapping of non-interfering tasks in Section 6.4.

6.1 Messaging Interface

The Legion high-level runtime operates on top of the low-level runtime, therefore there is no explicit mechanism for the high-level runtime to send messages between two nodes. Instead data can be moved in one of two ways: explicit copies or task launches on remote processors. Explicit copies are done between physical instances and are optimized for moving data structures in bulk. The low-level runtime implements these copies efficiently using asynchronous one-sided RDMA transfers. Alternatively, task launches on remote processors can be accompanied with an un-typed buffer of data that is copied along with the task to the remote node. In general, the un-typed buffer is only used for capturing task arguments and should be relatively small. The low-level runtime implements remote task launches using asynchronous active messages supported by the GASNet API[16]. The size of the buffer determines the size of the active message required: either small (less than 128B), medium (less than about 8KB depending on hardware), or large (anything greater than the medium size). Pre-allocated buffers of pinned memory is maintained by GASNet for small and medium active messages, making them considerably faster and lighter-weight than long active messages.

The Legion programming model encourages bulk transfers of data and therefore all application data is always moved by copies between physical instances. However, determining the best approach for moving meta-data is not as obvious. Region tree data structures are large and could therefore benefit from being moved in bulk. However, in practice, our implementation of distributed region trees (described in Sections 6.3 and 6.4) rely on an eventual consistency model that maintains persistent state on all nodes and relies on sending fewer small updates instead of re-sending large bulk data structures each time they are needed on a remote node. The Legion high-level runtime therefore relies on sending tasks to remote processors as the primary communication mechanism for internal meta-data.

6.1.1 Ordered Channels

While the Legion runtime uses remote task launch as its primary communication mechanism, it does so in a disciplined way. Specifically, we create the abstraction of an *ordered channel* for all messages between a pair of nodes. Ordering all messages between a pair of nodes provides a useful invariant for reasoning about the coherence of distributed data structures. In the case of the Legion runtime, by knowing that all messages from one node to another are ordered, we will be able to develop a novel algorithm for distributed garbage collection of physical instances in Chapter 7.

To implement our ordered channels, we create *message manager* objects on each node for sending messages to destination nodes. Message managers provide a thread safe interface that serializes all message requests from any thread running on a node. In order to serialize messages to the remote node, each message manager remembers the termination event of the most recent message. This event serves as the precondition for the next message to be run. Message managers also handle all incoming messages from the remote node that it is responsible for managing.

In order for Legion to remain scalable, message managers are lazily instantiated. In the worst case, every node in the system will attempt to talk to every other node in the system. If there are N nodes in the system, this will require each node to instantiate N message managers, ultimately requiring N^2 message managers across all nodes. Since message managers maintain some state and a buffer for sending messages, this can result in a significant cost of memory. In practice, most nodes only talk to a subset of other nodes in the machine. Therefore, the Legion runtime only instantiates a message manager for a destination node when either it is requested as a target, or it receives a message from the remote node.

6.1.2 Deferred Sends

While many of the messages sent by Legion are small update messages, it is possible to defer many messages for a short period of time and to aggregate many small messages into larger messages that gain back some of the efficiency of bulk data movement. The message manager interface allows other components of the Legion runtime to send

messages and to indicate whether these messages must be *flushed* (sent immediately) or whether they can be deferred for a short period of time. The message manager maintains a small buffer for aggregating messages. This buffer is intentionally sized to target the maximum size of GASNet medium active messages (usually between 8KB and 16KB). The message manager attempts to aggregate as many message as possible into this buffer until either the buffer fills up or a message that needs to be flushed is requested, at which point the aggregate message is sent. On the remote node, the message manager that receives the aggregate message then unpacks the smaller messages and invokes the proper handler.

The message manager API is also intelligent enough to handle message requests that are bigger than its buffer size, breaking them up into smaller messages that fit into GASNet medium active messages. By breaking up larger active messages into slightly smaller ones, we can get data moving through the network earlier, both reducing latency and improving communication parallelism. Note that even though messages are serialized by low-level event dependences, those dependences are handled by the remote node, meaning that the active messages for those tasks can be in-flight in the network simultaneously. Ultimately, the message manager interface allows the Legion runtime to strike a good balance between optimizing for message throughput and latency by targeting GASNet medium sized active messages. The message manager interface also simplifies the implementation of other runtime components, allowing them the illusion of sending many small messages, that are ultimately aggregated into larger messages.

6.2 Task Distribution

In practice, the mapping stage of the task pipeline can actually be thought of as two separate stages: the premapping stage and the actual mapping stage (see Chapter 5). The premapping stage must always be done on the origin node where a task was created. However, the actual mapping can be performed either on the node where the task was created (locally) or on the node where the task is going to be run (remotely). In this section we discuss the implications of the how both individual

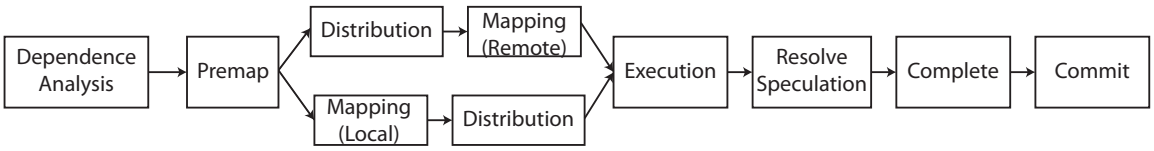


Figure 6.1: Individual Task Pipeline

(Section 6.2.1) and index space (Section 6.2.2) tasks are distributed and actually mapped. In Section 6.2.3, we discuss how task stealing is incorporated into task distribution.

6.2.1 Single Task Distribution

For single tasks, the distribution stage is fairly straightforward. Figure 6.1 shows the distribution pipeline for an individual task launch including the extra detail of physical traversals from Chapter 5. After the task is finished premapping all of its regions, it first checks to see the mapper has selected for it to be locally or remotely mapped. If the task is to be locally mapped, then the mapping is performed, the mapped regions are recorded, and the task is sent to the target node where it is launched onto the specified processor. If the task is to be remotely mapped, then the task is sent to the target node. If the task is not eligible for stealing (see Section 6.2.3), then the physical region tree state necessary for mapping the task is eagerly sent along with the task. If the task is eligible for stealing we defer sending the region tree meta-data until we know for sure that a task will attempt to map on a specific node. (We discuss the movement of physical region tree data in further detail in Section 6.4.) When the task arrives on the target node, it is placed in the ready queue for its intended processor.

Once a task is placed in the ready queue for the target processor, then the mapper associated with that processor can decide to either map the task or to send it to another processor to be mapped. If the mapper decides to map the task, then the runtime first checks to see if the necessary meta-data is resident. If the region tree meta-data is not local, then a request is sent to the owner node of the task for the necessary state. The owner node handles these requests and replies with the necessary

region tree meta-data. Once the meta-data arrives the task can be mapped. If the meta-data is already resident, the task is immediately added to the list of tasks to map.

Alternatively, a mapper may also decide to continue forwarding a task on to an additional processor. A mapper might choose to make this decision if the processor that it is managing is overloaded and it knows of other processors with less work to perform. Under these circumstances, even if the task has been marked not eligible for stealing, the remote state information is not automatically forwarded. While it is useful to forward state eagerly the first time to reduce latency, if a task is moved multiple times before being mapped, then the cost of moving the state outweighs the latency savings. When the task ultimately does map, it will first need to request the necessary region tree meta-data from the owner node. While this does add latency, it does reduce the overhead of moving the task multiple times as the meta-data only needs to be moved once from the owner node to the node where the task is ultimately mapped.

6.2.2 Index Space Task Distribution

Figure 6.2 shows the more detailed pipeline for the stages of an index space task launch. Index task distribution is more complicated than for single tasks. The reason for this is that index space task launches create many sub-tasks with a single call. To handle all of these tasks, the distribution stage for index space tasks contains a sub-stage called *slicing*. Slicing allows the mapper to break apart an index space task launch into *slices* that describe a subset of tasks in the original index space task launch. Slices are created by the `slice_domain` mapper call, which queries the mapper object to break apart an index space task into subsets of tasks (although the mapper is not required to break apart an index space, and can create a single slice that describes the entire set of tasks). Slices act as the unit of reasoning about distribution for index space task launches, with all tasks within a slice being moved together. By grouping tasks into slices, the runtime can deduplicate much of the meta-data necessary for moving sets of tasks between nodes.

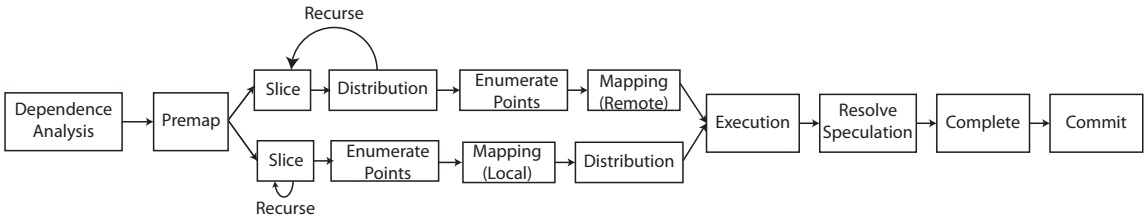


Figure 6.2: Index Task Pipeline

In some cases, index space task launches can consist of very large numbers of tasks that need to be distributed throughout an entire machine that may contain thousands of nodes. To make this process efficient, the slicing stage supports hierarchical division. When a slice is created, the mapper chooses two values for the slice object: a target processor and a boolean value indicating whether the slice should be recursively sliced. If the mapper asks for the slice to be recursively divided, then when the slice arrives on the target node, the mapper for the target processor is invoked with the `slice_domain` call to perform the recursive slice operation. Ultimately it is up the mapper objects of a task to determine how many times and how finely an index space task is divided.

After slicing is complete, slice objects are added to the ready queue, similar to individual tasks. Slices are the operations that traverse the pipeline, with the original index task operation waiting to complete the final three stages of the pipeline until all of its constituent slices have completed the equivalent pipeline stages. Slices behave like individual tasks until they are ready to be mapped. The mapper that owns a slice can choose to map it onto its processor or send it to another process to be mapped. Slices can also be marked eligible for stealing, as we will describe in Section 6.2.3. Once a slice is ready to be mapped, the runtime enumerates all the points in the slice, and generates a *point task* for each point in the slice. Each of these point tasks are then mapped individually using the same routines as individual tasks.

Similar to individual tasks, an index space of tasks can also be mapped both locally and remotely. In the case of local mapping, slicing is performed first, but none of the slices are sent remotely until each of the points within each slice have been mapped. For remote mappings, slicing is performed first, allowing slices to be moved

throughout the machine. Once a slice is ready to be mapped, it sends a request to the owner node for the region tree meta-data necessary to perform the mapping. The owner node responds with this information, allowing the slice to be mapped. Note that, unlike individual tasks, region tree meta-data is not eagerly sent to any nodes as part of the slicing process. The reason for this is that it is common for slices to be both recursively sliced and moved; therefore, we avoid the cost of moving the meta-data by lazily sending it only when it is explicitly requested.

6.2.3 Distributed Stealing

While the distribution phase of the task execution pipeline allows to tasks to be pushed to their target processors (as we described in the previous two sections), the distribution phase also allows tasks to be pulled to processors, with little or no work, through stealing. By supporting both push and pull models of task migration, Legion allows mappers sufficient flexibility for expressing optimal ways of mapping a particular application onto a target architecture.

To support stealing, Legion allows mappers to designate tasks as eligible to be stolen by setting the *spawn* flag in the `select_task_options` mapper call[1]. While stealing in many other programming models is done implicitly by the runtime, stealing in Legion is directed by mappers because it impacts the performance of an application. Mappers can send steal requests to any other mappers with the same ID on different processors. Note by restricting steal requests to other mappers with the same ID we ensure that a task will always be mapped by the same kind of mapper, albeit on different processors. Unlike many stealing systems, a steal request does not guarantee that a steal will succeed. Instead of having the runtime automatically decide which tasks are stolen by a steal request, the obvious potential performance impacts mandate that the decision be made by the mapper object. Each steal request is therefore converted by the runtime into a mapper call asking the target mapper which (if any) of the eligible tasks that it owns are permitted to be stolen. The mapper is free to specify none, all, or a subset of its eligible tasks. In order to make such a decision the

[1]The choice of *spawn* as the flag was inspired by the semantics of the Cilk language[30].

mapper can introspect information such as the location of physical instances available for different tasks that reveal locality information. We discuss these considerations further when we describe the full mapping interface in Chapter 8.

One important detail regarding stealing is that only eligible tasks still in the ready queue for a specific mapper can be stolen. Once a task has been mapped, then it is guaranteed to execute on the target processor and it cannot be stolen. For mappers that choose to employ stealing as a means for performing task migration, there exists a natural tension between mapping tasks early to get any precondition operations in flight, and deferring mapping to achieve better load balance through stealing. To aid in these decisions, mappers are given detailed information regarding the number of queued tasks on each processor allowing them to make informed decisions about when to map tasks. Importantly, all such decisions regarding when to map tasks and when to permit stealing, are under mapper control, ensuring that there is no implicit decisions made by the runtime that may impact performance.

Since stealing in Legion is primarily designed to migrate tasks between nodes, Legion does provides some additional support for avoiding unnecessary steal requests. When a steal fails, Legion blacklists the target processor as one in which a steal has failed from the source processor. No steal requests from the mapper of the source processor will be allowed to be sent to the mapper of the target processor until the mapper on the target processor receives additional tasks that are eligible for stealing in its ready queue. The runtime also records all the failed steal attempts at a given processor. When new tasks that are eligible for stealing are added to the ready queue, the runtime sends advertisements to all the previously failed stealing processors. These advertisements remove the processor from the blacklists of the attempted thieves.

Finally, while stealing in Legion will work between any pair of processors, it works best between different nodes, where it is more difficult to achieve load balance. Task migration between processors within a node is implemented in a fast way that requires no copying of data. However, it is often more efficient to perform load-balancing within a node using light-weight processor groups that we discuss in Chapter 8.

6.3 Distributed Region Trees

While a Legion application is executing, it is free to dynamically create index spaces, field spaces, logical regions, and partitions. Each of these operations are initially invoked by a task on a single node, meaning that the shape of the index space trees, field spaces, and region trees are initially unknown to other nodes immediately after creation. When a task is to be mapped remotely, it is the responsibility of the runtime to guarantee that the necessary index tree and region tree shape information exist on the remote node where the mapping is to be performed. In this section we describe how the runtime maintains this guarantee efficiently by lazily moving region tree shape information.

6.3.1 Leveraging Lazy Tree Instantiation

In order to map a task remotely, the state for performing the mapping must be sent to the target node. Before the state can be sent, the runtime first checks to see which region tree shape information must be sent to instantiate the necessary region tree data structures on the remote node for storing the state information. For each of the region requirements requested by a task, the runtime checks that necessary region tree shapes can be created on the remote node. The only traversals that are performed remotely are the mapping traversal and the instance view traversal. Therefore for each region requirement of the task, the runtime must ensure that the node corresponding to the requested logical region and all sub-trees are sent to the remote node. Additionally, because some instance view trees for physical instances may contain views from parent logical regions, all logical regions that are ancestors of the requested logical region must also be sent to the remote node.

While this may initially sound like a considerable number of region tree nodes to send, we cut the number of nodes that must be sent in half by leveraging the lazy region tree instantiation supported by the runtime (see Section 4.3.1). Instead of having to send both the region tree nodes as well as their corresponding index space tree nodes, we only need to send the necessary nodes from the index space tree and the root node of the logical region tree. This is sufficient from the perspective of

the remote node to instantiate the remaining parts of the region tree when needed, thereby reducing the amount of data that must be sent when moving a region tree to a remote node by half.

6.3.2 Creation and Deletion Sets

To further reduce the amount of data that must be sent when moving tasks between nodes, all index space tree and logical region tree nodes memoize the set of other nodes in the machine to which they have been sent. We call this set the *creation set*, and it records which nodes in the machine know about the creation of the index space tree node or region tree node from the perspective of the local node. It is possible that this set is stale and does not accurately represent all the nodes in the machine that are aware of the existence of a node. However, this lack of information can only result in duplicate sends of region tree and index tree data. In practice these duplicate sends are rare, and when they do occur, the sending node is immediately added to the creation set of all region tree and index space nodes on the destination node, to avoid further duplication.

In addition to creation sets, both region tree nodes and index space tree nodes also maintain deletion sets for tracking which nodes have been notified of the deletion of a node. Since node deletions are deferred (there may be sub-tasks still using the node), deletions of nodes in both the index space trees and region trees do not immediately delete a node, but instead simply mark it as deleted. As the deletion propagates and removes privileges working its way back up the task tree, the corresponding deletion information is only sent to the proper nodes and only when the destination node is unaware of the deletion.

To store both creation and deletion sets the runtime uses a *node mask* data structure that relies on the same base bit mask implementation as field masks (see Section 4.3.3). However, the upper bound on the number of entries in a node masks is determined by the number of nodes in the machine instead of the maximum number of fields in a field space. While these node masks require $O(N)$ memory storage (in bits), they make checking for elements in the set $O(1)$, which is important since the

common operation is traversing a tree to see which region tree and index space tree nodes still need to be sent to a destination node.

6.3.3 Distributed Field Allocation

The final aspect of maintaining distributed region tree data structures is handling dynamic field allocation[2]. Each runtime instance statically partitions a space of field names for each field instance, allowing them to be able to dynamically allocate field IDs without needing to communicate with other runtime instances. When a field is allocated on a specific node it is given a unique ID, but it is not sent to any other nodes. Instead when a field space is sent to a node, it sends all the field IDs that it knows about to the target node. If the field space already exists on the destination node, only the newly allocated fields are sent. Upon receipt of an update of newly allocated fields, the runtime always checks that the upper bound on the maximum number of fields in a field space has not been exceeded (see Section 2.2.3 for a description of the upper bound).

The difficult part of this distributed allocation scheme is maintaining a mapping of field IDs in a field space to indexes in the field masks. Each node maintains its own mapping. All nodes coordinate on a scheme in which they attempt to allocate fields at indexes that are equal to their node number modulo the number of nodes in the entire machine. When these locations are exhausted they progress to their neighbors locations. Ultimately the goal is to maintain the same mapping across all nodes so field masks can be directly unpacked and not require a transformation when they are sent from one node to another.

In some cases though, different fields will be allocated to the same index in different nodes. To handle this case, each field space maintains a permutation vector for every other node in the machine that describes the necessary transformation that must be applied to unpacked field masks received from every other node. Permutations are also stored as field masks and are implemented efficiently using the sheep and goats algorithm from [51]. In practice most of these permutations are the identity function.

[2]Dynamic allocation of rows in unstructured index spaces is handled by the low-level runtime.

The runtime tracks whether a permutation is the identity function or not so that the identity case can be easily skipped.

6.4 Distributed Contexts

One of the options afforded to the mapper of a task is to map the task remotely. To map remotely, the necessary region tree state for performing the mapping, registration, and instance view traversals from Chapter 5 must be available on the remote node. To satisfy this requirement the Legion runtime must provide a distributed implementation of the region tree meta-data for a physical context. In a distributed environment where many tasks can be mutating the state of a physical context in parallel on different nodes, maintaining the consistency of the physical state is challenging. In this section we detail the approach that enables Legion to support distributed physical contexts on multiple nodes in a consistent manner.

6.4.1 Eventual Consistency

In order to support distributed mappings of non-interfering tasks within the same context on different nodes, Legion permits the physical context from the owner node to be replicated on remote nodes. This allows non-interfering tasks to map in parallel on remote nodes. The challenging aspect to this approach is that tasks mapping in parallel may mutate the state of a physical context in different ways. To be safe, we need a mechanism to guarantee that these independent mutations can be reconciled at a later point in time. This deferred reconciliation of data structures at a later point in time is called *eventual consistency* in the distributed systems community literature.

The primary requirement for supporting eventual consistency is that the types of data being stored are instances of convergent replicated datatypes (CRDTs) [37]. A CRDT is a type that is defined by a set of (possibly unbounded) instances and a meet operator. The meet operator is required to be associative, commutative, and idempotent. The crucial insight for supporting eventual consistency for distributed region tree contexts is that the set of possibles states for sub-trees open in either

read-only or reduce mode represent instances of a CRDT.

For example, consider a region tree (or sub-tree) rooted by a logical region R in a context. For a given field, the state of the region tree can take on many potential values defined by the valid instances and open parts of the sub-tree. These different states of the tree rooted at R define the different elements in the CRDT. The meet operator is then defined to be the union operator that unifies any two states for the sub-tree rooted at R. It is important to note that this union operator only works if the state of the tree rooted in R is open in read-only or reduce mode, as these privilege modes can only open parts of the sub-tree or add new valid instances. Read-write privileges do not have this property as they can close sub-trees and remove valid instances which invalidates the commutativity of the union operator. Defined in this way, Legion can permit multiple distributed nodes to be mapped in parallel on distinct nodes for trees open in read-only and reduce mode and later be able to reconcile the state of the physical context.

6.4.2 Sending Remote State

State of a physical region tree for a physical context is sent to a remote node in one of two cases. First, it can be sent eagerly when an individual task that is not eligible for stealing is sent to a remote node (see Section 6.2.1). The second, and more common case, is that a request is received from a remote node for the physical state meta-data necessary for mapping either an individual task or a slice of an index space task. When these requests occur, the runtime locates the meta-data for the requesting task (this data is always resident because requests are only handled by the owner node where the task was originally created). For each of the region requirements for these tasks, the runtime traverses the region tree from the region on which privileges were requested through all open sub-trees for the requested fields. This traversal ensures that all the meta-data necessary for performing the mapping, registration, and instance view traversals will be resident on the remote node.

During the traversal, the state of each node in the region tree is sent to the target node including the dirty fields, valid physical instances, and open sub-trees. Only

state for the fields being requested by the region requirement are sent. Each state is sent as a separate message that can be deferred allowing the message managers (described in Section 6.1.1) to aggregate the smaller messages into larger ones. The traversal also keeps track of the necessary instance views and instance managers that must be sent to the remote node for performing the traversal. After the entire traversal is finished, the instance view and instance managers are also sent to the remote node. Instance managers are always sent in full, while instance views only need to send the members of the epoch lists for the fields being requested by the region requirement.

On the remote node, the messages are handled and unpacked into another physical context serves as a clone of the physical context on the owner node (note this context does not need to remain a clone and can diverge from the owner node under the conditions of CRDTs). Each message contains the logical region node for the state that is being unpacked, allowing the state to be updated directly. It is important to note that unpacking state is merged into the existing state rather than overwriting it as the context may already contain valid data for other fields from another task that mapped on the same remote node (see Section 6.4.4 for a detailed description of persistent remote contexts). The remote node also handles messages for instance view and instance managers. Instance managers and instance views are either updated or created by the runtime to ensure that there exists a unique one on the node[3].

After the traversal of all the region requirements in the physical state has been completed, the owner node replies to the requesting remote node with a message indicating that all of the physical state data has been sent. This message is marked as being unable to be deferred, which causes the message manager to flush the ordered channel to the remote node. Since all messages between a pair of nodes are both sent and received in order, by the time the remote node observes the notification, all of the necessary physical state has already been unpacked in its region tree. The runtime on the remote node can then resume the mapping process for the task.

[3]Instance managers are unique on a node, while instance views are unique within a context on a node.

6.4.3 Receiving Remote State Updates

When a task has finished mapping, it returns the updated state of the region tree back to the origin node of the task[4]. While eagerly flushing meta-data back to the origin node is not strictly necessary (we could have opted to employ a distributed coherence mechanism for tracking which nodes had valid state for a specific fields of each region tree node), we decided on this approach in order to ensure a simpler implementation[5].

To return the meta-data state back to the origin node, the runtime traverses the sub-trees for all regions on which the task held privileges, and sends back the state of these nodes. We currently send back the entire state as it is relatively cheap to represent. Upon receiving the remote state on the origin node, one of two operations is performed. If the state is coming from a region requirement that requested read-only or reduce privileges, the state is unioned with the existing state on the owner node. Performing a union is possible because the state of the region tree under read-only or reduce privileges is a CRDT (see Section 6.4.1), allowing states from distributed nodes to be safely merged. Alternatively, if the state is returning from a sub-tree on which the remote task had read-write privileges, then the existing state on the owner node is first invalidated, and then the returning state is written as the new state. Read-write privileges guarantee that there is only ever at most one task mutating the state of that sub-tree at a time, and the canonical version of that state is therefore always the most recently written one.

In addition to returning state for region tree nodes, state for the epoch user lists of instance view objects is also returned. Similar to returning state for region tree nodes, whether or not the state is merged or overwritten depends on the privilege context under which it is being returned. If the added users are being returned within a read-only or reduce privilege context, then they are merged with the existing set of users. Otherwise, the current epoch lists are invalidated for the fields being returned,

[4]There is an important interaction with virtual mappings here: tasks with virtual mappings cannot return their state until they have finished mapping all their child operations.

[5]The analogy to modern processors is the difference between a write-back and a write-through cache. Eagerly flushing data is like writing through the cache back to main memory.

and the new users are added to the lists as the new version of the epoch lists for the returned fields.

6.4.4 Persistence of Remote Contexts

One important optimization implemented by the runtime is deduplication of remote physical contexts. In many cases, multiple sub-tasks from the same parent task are sent to the same remote node to be mapped. Often, these sub-tasks will request some of the same state in order to map their regions. Under such circumstances, it is inefficient for the runtime to send multiple copies of the same physical state to a remote node. Instead, the runtime deduplicates these requests by ensuring that at most a single copy of any physical context exists on any remote node.

On each node, the runtime tracks all of the physical contexts that have been allocated, as well as whether the allocated contexts are for a task running on the local node, or whether the contexts are remote versions of a context from a different node. If a context is a remote version, then the runtime also tracks the identity of the parent task on the owner node associated with the remote clone context. When a task is received to be executed and mapped remotely on a node, the runtime first checks to see if an existing remote context has been allocated for the parent task. If it has, the child task is told of the appropriate context so that it can use it as the target of any remote state requests that are sent to the owner node. If no such context exists, then a new remote clone context is allocated and registered. Any future tasks that are sent to the node with the same parent task are then guaranteed to re-use the same context.

The parent task on the owner node also tracks the set of remote clone contexts that exist on remote nodes. Individual region tree nodes in the context on the owner node track the validity of state information for different fields on remote nodes. We discuss the protocol for maintaining this validity in Section 6.4.5. When the parent task commits on the owner node, it is safe to infer that there is no longer a need for the remote clone contexts to be maintained. At this point, the owner node sends messages to all of the nodes that contain remote clone contexts, instructing them to

invalidate the remote contexts and recycle them so that can be re-used for future tasks.

6.4.5 Invalidate Protocol for Remote Context Coherence

Maintaining coherence of the distributed region tree state is a challenging problem. Not surprisingly, this problem has a natural analog in modern hardware processors: maintaining cache coherence. Modern hardware processors support replication of cache lines whenever possible for different processors in the same chip while still guaranteeing a total order on writes to the cache line. Legion relies on a similar approach for maintaining coherence of the region tree state within a physical context that have been replicated throughout the nodes of a machine.

We considered two potential adaptations of cache coherence protocols on which to base our Legion implementation. Our initial approach was based on an update protocol. In an update protocol, data initially resides on a single node. As other nodes request data, they become subscribers. Any updates to the data are eagerly flushed out to all of the subscribers. Our initial implementation based on an update protocol maintained a set of subscribers for each region tree node. While the update protocol performed well at small node counts, it failed to perform well at larger node counts. At larger node counts, there are many unnecessary update messages sent that ultimately are not used for mapping any future tasks in a specific part of the region tree.

Our second design is based on an invalidation cache coherence protocol. In an invalidation protocol, remote copies of the data are invalidated, and requests are generated by the remote node to pull data from the owner node (which always has a valid copy) to the requesting node. Invalidation protocols scale better because they move data lazily, which ensures that data is only moved when it is actually going to be used. In our Legion implementation, each region tree node on the owner node maintains a *directory* object (similar to the directory maintained by distributed memory cache coherence protocols). The directory stores two kinds of information for each field: the *mode* it is in and for each field, the set of nodes that contain

valid versions of the state. Each field within a state can be in one of three modes corresponding to the different region privileges: read-only, read-write, and reduce. The read-only and reduce modes permit multiple nodes to have valid copies of the data simultaneously, while the read-write mode permits at most a single other node to maintain a valid copy of the state.

Whenever a state transition takes place for a node within a field, all outstanding remote copies are invalidated. The invalidation is both recorded on the owner node and on the remote nodes by messages sent from the owner node. It is important to node that the owner node always maintains a valid copy of the state for all fields of all nodes within a context because all tasks eagerly flush changes to the region tree back to the owner node. In many ways this approach is similar to write-through caches that always flush data back to the backing memory so that the directory owner always has a valid copy of the data. Remote nodes can then request updated versions of the state for any set of region tree nodes when necessary.

6.4.6 Distributed Futures

The final piece of distributed state within a context that must be considered are futures. Futures are local to the context of the parent task in which they are created. However, sub-tasks can request future values as arguments. If these tasks are mapped to processors on remote nodes, it is the responsibility of the runtime to ensure that the value of the future is propagated to the remote node when the future completes.

To support this feature, the runtime implements *distributed futures*. Distributed futures operate in a similar manner to instance managers (see Section 5.2.2). There is an owner distributed future created on the owner node and clones can be created on remote nodes. Each distributed future remembers the nodes where versions already exist as well as the node that contains the original owner node. Distributed futures are registered with the runtime to ensure that they are deduplicated within a node (guaranteeing that at most one distributed future exists on each node). Each remote distributed future registers itself with the owner version on the owner node. When the owner distributed future completes, it broadcasts out the value associated for the

future with any registered waiters on remote nodes. If the owner distributed future receives any requests after it has already completed, it immediately replies with the value of the future. When remote nodes receive the values for the future, they also complete immediately and notify any tasks that may have been waiting on them that they are complete (usually done with low-level runtime events). Distributed futures are reclaimed with the same garbage collection algorithm as instance managers which is discussed in detail in Chapter 7.

Chapter 7

Garbage Collection

Many of the resources associated with a Legion application are explicitly managed by the user. For example, logical regions must be explicitly allocated and deallocated by an application. However, while logical regions are explicitly managed by the application, the Legion runtime is responsible for managing the allocation and freeing of physical instances of logical regions. The reason for placing this responsibility on the runtime is a matter of safety. If an application (or a mapper) had the ability to explicitly allocate and deallocate physical instances it could unsafely delete instances that are in use or will be used by future tasks. Therefore, neither application code nor mapper objects are given permissions to explicitly allocate or deallocate physical instances. It is the responsibility of the runtime to explicitly manage physical instances and know when it is safe to reclaim them.

To support a safe reclamation scheme of physical instances, the Legion runtime employs a garbage collection algorithm. While there are many well known garbage collection routines for languages and runtimes, the one used in the Legion runtime is unique. The novelty of our approach stems from having to deal with two difficult aspects of the Legion programming model. First, garbage collection in Legion must be capable of operating in a distributed environment because tasks may be mapping to the same physical instance on different nodes. Second, garbage collection in Legion must be capable of operating within a deferred execution environment. As we will show, our proposed garbage collection scheme allows physical instances to be

reclaimed and the memory recycled even before the tasks using the physical instances have started execution. Dealing with these two different constraints has led us to a garbage collection algorithm that differs in many ways from traditional approaches.

The rest of this chapter is organized as follows: in Section 7.1, we give an overview of the garbage collection algorithm and how it operates both within nodes as well as across nodes. Sections 7.2 and 7.3 describe the two different collection patterns that comprise the basis for our garbage collection algorithm. Finally, in Section 7.4, we show how our garbage collection allows physical instances to be safely recycled even before all the tasks using them have started running.

7.1 Reference Counting Physical Instances

To garbage collect physical instances, we rely on existing data structures used for tracking the usage of physical instances: instance managers (described in Section 5.2.2) and instance views (described in Section 5.2.3). Figure 7.1 provides an illustration for kinds of references kept between the different components of the Legion runtime. Both operations and physical states can maintain references to instance view objects. Instance view objects in turn maintain references to instance manager objects. By using these existing data structures, we set up a two-level reference counting scheme described in detail in Section 7.1.1. To further improve the efficiency of our garbage collection algorithm, we also track different kinds of references between these data structures, allowing us to know when a physical instance will eventually be garbage collected, even though it may still be some time before the instance is actually collected. Different kinds of references permit us to construct a two-tiered approach to reference counting for detecting these conditions in Section 7.1.2.

7.1.1 Two-Level Reference Counting

In order to know that it is safe to garbage collect a physical instance, the runtime must be able to prove two conditions. First, it must be the case that there are no existing users of a physical instance (either tasks or copies). Second, the runtime

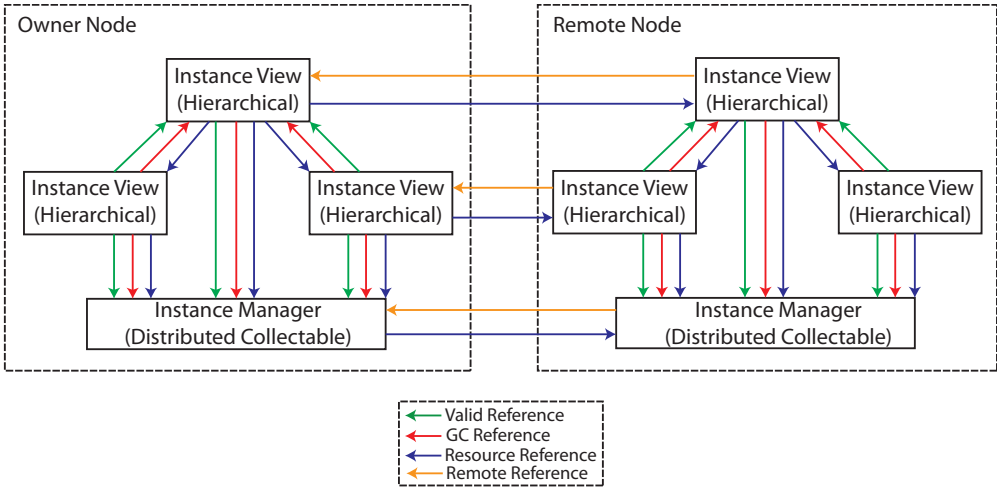


Figure 7.1: Legion Reference Counting Scheme

must be able to show that there will not be any future users of the physical instance. In both cases, the information for determining these conditions is available through the instance view objects. Instance view objects record both the current users of a physical instance from the perspective a given logical region in a context, as well as whether the instance view contains valid data for any of the fields in the physical instance. Whenever an instance view either has users of a physical instance or is a valid view, it adds a reference to the physical manager with which it is associated to prevent the physical instance owned by the manager from being collected.

Physical manager objects track all of the references that they see from instance view objects on the node in which they reside. Recall from Section 5.2.2 that there is at most one instance manager on a node for each physical instance. This is a useful property to maintain as it allows a single instance manager to be used for tracking references from all instance views in any physical context within a node.

While a single instance manager on a node can track references within the node, it cannot track references to a physical instance between nodes. Only when all of the nodes no longer have references is it safe to actually garbage collect the physical instance. To track when it is safe to actually perform garbage collection for a physical instance we use the *distributed collectible* algorithm described in Section 7.2. The distributed collectible algorithm both tracks references between nodes and also aids

in knowing when meta-data structures are safe to free.

7.1.2 Two-Tiered Reference Counting

While maintaining references for purposes of garbage collection is useful, it often represents a sub-optimal approach to freeing memory in Legion's deferred execution model. Garbage collection frees physical instances only after they are no longer needed. However, the Legion programming model is based on the premise that mapping will be occurring in advance of actual execution. When mapping ahead of actual execution, it is possible in Legion to know that a physical instance will eventually be collected well in advance of when the actual collection will occur. To detect these cases, we employ two different *tiers* of reference counting: *valid references* for detecting when it is safe to *recycle* a physical instance, and *garbage collection* references for detecting when the instance can actually be deleted. Based on the information provided by this two-tiered reference counting approach, we can free up the memory being consumed by the physical instance in advance of the actual garbage collection, thereby allowing additional tasks to map and further hiding the latency of the mapping process.

We know that a physical instance will eventually be garbage collected once none of its instance views represent valid versions of the data for any of the fields in the physical instance. At this point, it is impossible for any new users to be registered with any of the instance view objects and the instance will ultimately be collected once all of the current users are done using it. To detect this scenario, we differentiate the kinds of references that are kept between instance views and instance managers. If an instance view contains valid data, then it adds a valid reference to both its parent instance view (because that view is also valid) and to the instance manager. If an instance view has a user of the physical instance then it adds a garbage collection reference to both its parent view and the instance manager. In order to recycle a physical instance before it has been collected, an instance manager only needs to prove that there are no longer any valid references to the instance. In order to perform garbage collection, a physical manager must be able to show that there are both no

valid references and no garbage collection references. We discuss how the distributed collectible algorithm supports detecting these cases in a distributed memory setting in Section 7.2.

The two other kinds of references are *remote references* and *resource references*. Remote references represent aggregations of garbage collection references for the distributed collectable and hierarchical collectable objects discussed in Sections 7.2 and 7.3. Tracking only garbage collection references and not valid references between nodes is an important trade-off. Tracking valid references between nodes would allow us to recycle instances that were known in distributed contexts. In practice, we have found that the cost of performing tracking outweighs the performance benefits and therefore we only track garbage collection references between nodes with remote references for correctness.

Resource references allow us to track when the actual meta-data objects themselves can be deleted. Resource references that are held by an object are only removed when an object itself is deleted. Resource references are used to track when objects in higher tiers of the collection scheme no longer require access to an object. Similarly, resource references will also be used as part of the distributed and hierarchical collectable algorithms to determine when objects on remote nodes are safe to delete.

7.2 The Distributed Collectable Algorithm

Performing reference counting garbage collection in a distributed environment is a difficult proposition that requires a disciplined approach to managing references between nodes in order to avoid incurring large inter-node communication overheads. Our approach to solving this problem is based a pattern that we call a *distributed collectable*. All nodes that share access to a common resource maintain a distributed collectable object for tracking the usage of the resource. By using a series of different kinds of references, distributed collectable objects can significantly reduce the amount of reference counting communication between nodes and discern when it is safe to garbage collect a resource. The distributed collectable pattern serves as the basis for garbage collection of both physical instances and distributed futures in the

Legion runtime.

In this section we discuss the details of the distributed collectable algorithm. Section 7.2.1 describes how distributed collectable objects are created and organized on different nodes. In Section 7.2.2, we cover the details of the distributed collectable state machine and how it allows objects that inherit from it to make decisions regarding the resource being managed.

7.2.1 Distributed Collectable Objects

Distributed collectable objects are aggregators of references within a node. Each node maintains a unique distributed collectable object for a common resource (e.g. a physical instance). Clients of the distributed collectable algorithm can add two different kinds of references to a distributed collectable object: *valid* and *garbage collection*. These two kinds of resources are described in detail in Section 7.1.2. Importantly, by grouping together these references within a node, a distributed collectable object can summarize many references with a just a few inter-node references and thereby significantly reduce the amount of communication required for performing inter-node reference counting.

While there may be multiple instances of a distributed collectable object throughout many nodes on the system, one of these objects is appointed the *owner*. When a resource is allocated on a node, a distributed collectable object is made to manage the resource and immediately becomes the owner. All other instances of the distributed collectable object on other nodes are considered *remote*. In order to support communication between distributed collectable objects managing the same resource, each group of distributed collectable objects are assigned a *distributed ID* that is unique throughout the entire system. When a distributed collectable object is created on a node, it registers itself with the runtime using its assigned distributed ID. The distributed ID allows messages between distributed collectables on different nodes to be properly routed by the runtime.

Figure 7.2 illustrates an example set of distributed collectable objects on different nodes for representing a common resource. There are two kinds of references

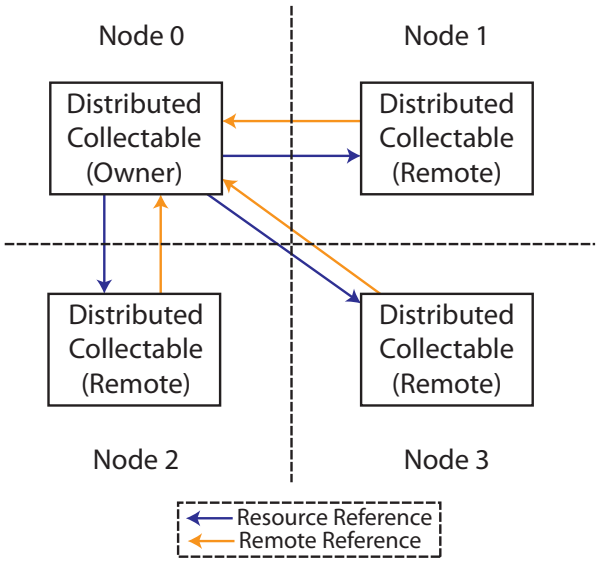


Figure 7.2: Distributed Collectable Example

maintained between distributed collectable objects on different nodes. First, all remote distributed collectable objects add *remote* references to the owner distributed collectable object when they have at least one valid reference or they have at least one garbage collection reference. Note that this is an existence property: only one remote valid reference is necessary to represent an arbitrarily larger number of valid references within a node. Adding a remote references requires sending a message from the remote distributed collectable object to the owner distributed collectable object. Ultimately, it is the owner object that makes the decision about when a resource can be recycled or garbage collected based on a combination of its local references as well as remote reference. We discuss this process in Section 7.2.2.

The other kind of remote reference maintained by the distributed collectable objects is a *resource* reference. A resource reference is not used for direct garbage collection, but instead is used for determining when it is safe to actually delete the distributed collectable objects themselves. Distributed collectable objects persist as long as the resource they are managing is still live. Once it has been reclaimed then it is safe to free the distributed collectable objects. The owner distributed collectable maintains resource references on all of the remote distributed collectable objects. This is done implicitly without any messages as the owner node can track which remote

instances it has heard from as part of tracking references. Once a resource is garbage collected, the owner distributed collectable will first remove all its resource references on the remote distributed collectable objects and then free itself. When the remote distributed collectable objects have their resource references removed, they know that it is safe to also free themselves.

One challenging part of the distributed collectable algorithm is the creation of a new distributed collectable object on a remote node. Each distributed collectable object maintains a set describing which nodes it already knows have instances of a distributed collectable objects with the same distributed ID. However, the algorithm permits these sets to be incomplete which may result in duplicate sends of a creation request to the same node. Since the runtime registers all distributed collectable objects, it can detect when duplicate creation requests have been received and ignore the additional requests.

The most difficult part of distributed collectable creation is preventing premature collections from happening while a creation request is in flight. There are two possible scenarios: the owner node receives a request for remote creation and a remote node receives a request for remote creation. We handle each of these two cases in turn.

First, in the case where the owner nodes receives the request to create a new remote distributed collectable, it immediately adds a remote reference to itself before sending off the message to the remote node to perform the creation. This additional reference guarantees that the owner node can not perform a collections while the request is in flight. When the remote distributed collectable is created on the remote node, it is initialized as already having a remote reference to the owner node.

The second case is more difficult and is illustrated in Figure 7.3. If a remote distributed collectable D receives a request to create a new distributed collectable on another remote node R it must ensure that the resource cannot be collected while the creation request is in flight. To achieve this, D first sends a message to the instance of the distributed collectable on the owner node adding a remote reference on behalf of the soon-to-be-created distributed collectable on R. After sending this message, D then sends a message to R, requesting the creation of a new distributed collectable with the same distributed ID. The reason this is sound is because communication

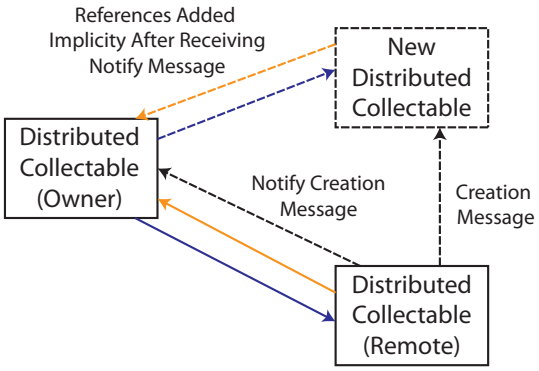


Figure 7.3: Distributed Collectable Creation

channels between nodes are ordered (see Section 6.1.1). We know that D has at least one remote reference on the owner. Any request to remove this remote reference from the owner will be ordered after the message sent to the owner node adding the new remote reference for the distributed collectable being created in R. Therefore, the total remote reference count on the owner node will remain greater than zero, and the resource will not be collected while the creation message is in flight.

7.2.2 Distributed Collectable State Machine

As part of its implementation, a distributed collectable object keeps track of its state for the various kinds of references that can be added and removed from it. Figure 7.4 shows an example of this state machine. Distributed collectable objects traverse this state diagram monotonically dependent upon when valid and garbage collection references are added or removed. Distributed collectable objects start off in the initialized stage and remain there until they receive their first valid or garbage collection reference. Once they receive their first valid reference, they move into the valid and referenced state. As soon as the number of valid references returns to zero, then we know that the distributed collectable object will eventually be collected. After the number of garbage collection references, or remote references representing garbage collection references, also drops to zero, then we know that it is safe to actually collect the resource. It is the responsibility of the client (in this case the rest of the Legion runtime) to ensure that valid and garbage collection references remain

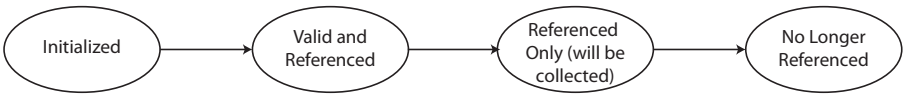


Figure 7.4: Distributed Collectable State Machine

positive until the resource is no longer needed. Failing to abide by this property can result in double recycling or double frees of an application resource.

7.3 The Hierarchical Collectable Algorithm

While the distributed collectable pattern is sufficient for handling a flat distribution of objects across all the nodes, it is insufficient for managing collections of objects with a more complex sharing pattern. Consider the case of instance view objects described in Section 5.2.3. Instance view objects are instantiated within a context and remote versions of them whenever a context is sent remotely. Initially this pattern of having an owner object and several remote copies may seem similar to the distributed collectable patter. However, instance view management is complicated by its interaction with virtual mappings (see Section 5.3.3). With virtual mappings it may be necessary for a remote instance view itself to be sent remotely depending on where tasks mapping within a virtual context are sent.

There are several different ways that we could handle this case. First, we could consider managing instance views using the distributed collectable algorithm. This would be sufficient for knowing when it is safe to add and remove valid and garbage collection references. However, it would significantly complicate the update pattern for adding users of instance views. Recall from Section 6.4.5 that the Legion runtime uses an eager write-back, but lazy update model for notifying instance views about new users. This means that users must be eagerly sent back to an enclosing physical context, but only lazily sent to other copies of an instance view within the same context. If we attempted to use the distributed collectable algorithm here, the notion of the *owner* instance view would take on different values depending on the physical context. This would both complicate the garbage collection scheme as well as the bookkeeping necessary to track which instance views needed to be sent updates in

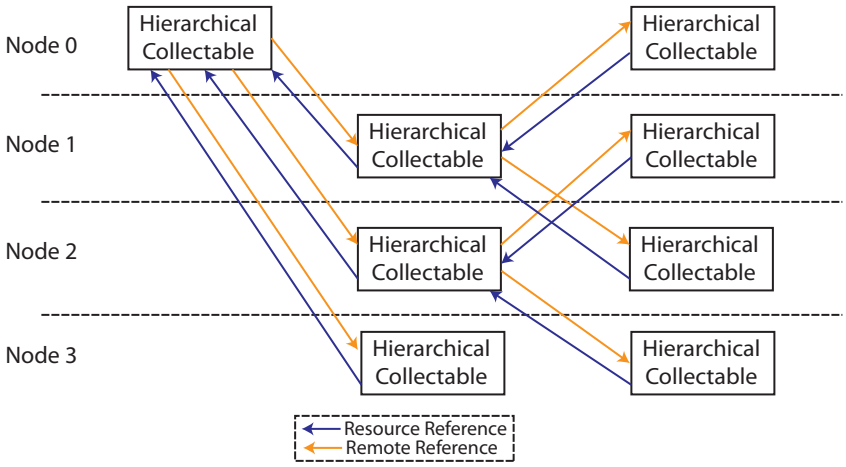


Figure 7.5: Hierarchical Collectable Example

different physical contexts. Instead we chose to implement a different solution.

We call our alternative approach the hierarchical collectable algorithm. The general principle behind this garbage collection approach is that each object in the scheme can act as both a remote object as well as an owner object for its own collection of remote objects. Unlike the distributed collectable algorithm, the hierarchical collectable algorithm makes no attempt to maintain uniqueness of objects within a node. In the hierarchical collectable algorithm there can be multiple copies of the same object within a node. While this may seem redundant, the tree of hierarchical collectable objects precisely captures the communication network necessary for sending updates when virtual mapped physical contexts are nested across several remote nodes. Figure 7.5 shows an example of a hierarchical collectable object with three different levels. Note that some instances of the hierarchical collectable from different levels are duplicated on the same node.

To implement the hierarchical collectable pattern, each object is assigned a unique distributed ID. (Note that this differs from the distributed collectable pattern where all objects within a group are assigned a common distributed ID.) Copies of a hierarchical collectable object can be requested on remote nodes. Each hierarchical collectable object maintains a record of which nodes already have remote copies of itself and the distributed IDs used to name those copies on each remote node. When

a new copy of a hierarchical collectable is needed on a remote node, a message is sent to the runtime on the target node requesting the creation. The owner node allocates the distributed ID and includes it in the creation message. The creation of the remote version of the hierarchical collectable object on the remote node then registers itself with the runtime on that node, permitting the same distributed ID based communication scheme used by the distributed collectable objects.

In addition to sending the creation message to the remote node, the owner also adds a remote reference to itself. When the creation of the remote object is complete, it already holds a remote reference to the owner node ensuring that there is never any premature garbage collection. This is simpler than the distributed collectable scenario because only the owner node will ever create remote copies of itself.

Any newly created hierarchical collectable object can then serve as an owner for its own set of remote copies. In the case of instance view objects, if the instance is in a physical context that is virtually mapped, the runtime may create further remote context copies that necessitate the creation of additional hierarchical collectable objects on remote nodes. Since each hierarchical collectable object keeps track of its own set of remote copies there is no deduplication.

In order to perform garbage collection of the hierarchical collectable objects themselves, the same resource reference approach is used as was employed by the distributed collectable algorithm. When the root of a hierarchical collectable tree is garbage collected, it deletes itself and removes all of its resource references on remote copies. This process then results in the deletion of the remote copies that recursively remove their resource references off of remote copies, ultimately triggering the eventual reclamation of the entire hierarchical collectable tree.

7.4 Recycling Physical Instances in a Deferred Execution Model

The most novel aspect of the Legion garbage collection scheme is that it permits safe recycling of physical instances before they are garbage collected. Specifically,

physical instances can be recycled as soon as there are no longer any valid references to them. As part of Legion's deferred execution model, it is possible that valid references to a physical instance are all removed even before the instance is used by an operation. By aggressively recycling physical instances, the Legion runtime can make more memory available for operations to map, permitting greater run-ahead of the deferred execution model and further hiding latency.

Recycling physical instances needs to be done in a disciplined way to avoid deadlock. When a physical instance is recycled it is placed in a pool for physical instances that are available for recycling. Each instance is also marked indicating the depth in the task tree at which the instance was allocated. To avoid deadlock, instances are only able to be recycled for use with operations that are at the same or a higher level of the task tree. This is an imprecise, but sufficient condition for guaranteeing deadlock freedom. Consider the case where a parent task P maps a physical instance I. As part of the execution of P it launches a child task C. While P was executing, a sibling task of P mapped causing the invalidation of all data in I, and thereby placing I in the pool of instances to be recycled. If C then attempts to recycle the instance it is required to wait on all previous users of I, including P. When C must wait on P we have a dependence cycle because by definition P must wait on C to complete. By restricting cycling to operations at the same or higher level of the task tree we guarantee deadlock freedom.

When a request to create a new physical instance is made as part of the physical dependence analysis (Chapter 5), the runtime first checks to see if an instance is available in the recycling pool that can be used. Currently, physical instances are only allowed to be recycled if they exactly match the memory usage of the requested physical instance. This requires that the physical instance reside in the requested target memory and the total memory usage be identical. We require strict memory usage matching in order to avoid fragmentation that may result in memory exhaustion and resource deadlock.

If an instance to recycle is found in the pool then it is returned to satisfy the creation request. Before the physical instance is returned, it is assigned a *use event*, that is a merged event of all the termination events for all the current users of the

physical instance in all of its instance views across all contexts. The use event becomes a precondition for any operation that uses the recycled physical instance to ensure that there are no data races for re-using the physical instance.

At some point the garbage collection for each physical instance in the recycling pool will be triggered. When this occurs, the runtime checks to see if the physical instance is still in the pool of physical instances eligible for recycling. If it is, then the physical instance is removed from the pool and then deleted, thereby returning the available memory to the low-level runtime and preventing any future recycling. If the instance has already been recycled and is in use by future operations then no action needs to occur.

Recycling of physical instances has some important implications for the performance and memory usage of Legion programs. Recycling often permits the Legion runtime to map farther into the future as more memory resources appear to be available for mapping. Recycling of physical instances can also reduce the total memory usage of Legion applications as the same memory locations can be re-used for multiple tasks. While these effects of recycling are often beneficial, they do come at the cost of sometimes reducing the total amount of parallelism exploited by the Legion runtime. Consider two non-interfering tasks t_1 and t_2. If both tasks end up using the same physical instance with t_2 recycling an instance created by t_1, then an event dependence is added between the two tasks to prevent a race, reducing the total amount of parallelism exploited. This trade-off between parallelism and memory usage exists in many parallel programs and is primarily a performance issue. As with all performance issues, we allow applications to make decisions for themselves via the mapping interface. By default recycling is enabled, but can be disabled on a per region requirement granularity by setting a flag on the region requirement as part of the `map_task` mapper call. As part of future work we plan to investigate ways of providing mappers a more global approach to mapping groups of tasks so that they may better manage the trade-off between memory usage and parallelism.

Chapter 8

The Legion Mapping Interface

The Legion mapping interface is the mechanism by which a machine independent Legion program is targeted at a specific hardware architecture. While we have mentioned some of the Legion mapping calls in earlier chapters of this thesis, we have yet to fully cover all the details of the mapping interface. In this chapter we give an overview of most of the features of the mapping interface except those specific to relaxed coherence modes and speculation/resilience that are covered in Chapters 9 and 10 respectively. We also provide a description of how the default mapper interface provides a useful initial implementation of this interface.

Listing 8.1 contains a C++ instantiation of the mapper interface as an abstract class. Recall from Chapter 3 that the interface is actually a reverse interface in the sense that the Legion runtime invokes the functions of the mapper interface whenever performance decisions need to be made. It is the responsibility of the mapper object to answer the query. In order to handle queries, mappers provide results by mutating fields on the objects passed to them. For brevity we do not show declarations for these objects here.

There are two important observations to make about the interface presented in Listing 8.1. First, because the interface is a class, each mapper will be created as a separate object. Having mappers be objects is important because it allows them to be stateful, and to use the results of previous mapping queries from the runtime as part of the process of answering later queries. Second, all of the methods of the

mapper interface are virtual. This makes creating specialized mappers easy as only modified methods need to be overridden to create a new mapper from an existing implementation. We now detail the semantics of each of the mapping calls in Listing 8.1 in the remaining sections.

8.1 Mapping Tasks and Regions

The primary purpose of the Legion mapper is to decide how tasks are assigned to processors and how each of the region requirements requested by those tasks are mapped onto specific memories in the memory hierarchy. We first begin by describing the process by which tasks are selected for execution and how mappers can choose which mapper instance is responsible for actually mapping the task (Section 8.1.2). In order to map selected tasks, all mapper instances require an interface for introspecting the underlying hardware to discover the system architecture; we discuss the interface for performing this introspection and how it is used to map individual tasks in Section 8.1.1. We detail how tasks are placed and mapped in Sections 8.1.2 and 8.1.3 respectively. Sections 8.1.4 and 8.1.5 describe how mappers can specify region instance layouts and the implications for selecting different task variants that can correctly use the chosen layouts. The use of priorities for addressing critical path issues is covered in Section 8.1.6.

8.1.1 Processors and Memories

In order to make decisions about how tasks and regions are mapped, each mapper implementation needs a way to query the runtime about the shape of the underlying hardware. To provide this functionality, the runtime supports a singleton *machine* object on every node. The machine object provides methods for finding all processors and memory locations throughout the entire system, including processors and memories on remote nodes. It is important that all processors and memories be known on each node so that a task can be sent to any processor or mapped onto any remote node.

```
1  class Mapper {
2  public:
3    Mapper(LegionRuntime *rt);
4    virtual ~Mapper(void);
5  public:
6    virtual void select_task_options(Task *task) = 0;
7    virtual void select_tasks_to_schedule(
8                const std::list<Task*> &ready_tasks) = 0;
9  public:
10   virtual void target_task_steal(const std::set<Processor> &blacklist,
11                                  std::set<Processor> &target) = 0;
12   virtual void permit_task_steal(Processor thief,
13                                  const std::vector<const Task*> &tasks,
14                                  std::set<const Task*> &to_steal) = 0;
15 public:
16   virtual void slice_domain(const Task *task, const Domain &domain,
17                             std::vector<DomainSplit> &slices) = 0;
18 public:
19   virtual bool premap_task(Task *task) = 0;
20   virtual void select_task_variant(Task *task) = 0;
21   virtual bool map_task(Task *task) = 0;
22   virtual void postmap_task(Task *task) = 0;
23 public:
24   virtual bool map_inline(Inline *inline_op) = 0;
25   virtual bool map_copy(Copy *copy_op) = 0;
26   virtual bool map_must_epoch(const std::vector<Task*> &tasks,
27                               const std::vector<MappingConstraint> &constraints,
28                               MappingTagID tag) = 0;
29 public:
30   virtual void notify_mapping_result(const Mappable *mappable) = 0;
31   virtual void notify_mapping_failed(const Mappable *mappable) = 0;
32   virtual void notify_profiling_info(const Task *task) = 0;
33 public:
34   virtual bool rank_close_targets(Close *close_op) = 0;
35   virtual void rank_copy_sources(const Mappable *mappable,
36                                  const std::set<Memory> &current_instances,
37                                  Memory dst_mem,
38                                  std::vector<Memory> &chosen_order) = 0;
39 public:
40   virtual bool speculate_on_predicate(const Mappable *mappable,
41                                       bool &speculative_value) = 0;
42 public:
43   virtual int get_tunable_value(const Task *task, TunableID tid,
44                                 MappingTagID tag) = 0;
45 public:
46   virtual void handle_message(Processor source,
47                               const void *message, size_t length) = 0;
48 };
```

Listing 8.1: Abstract C++ class declaration of the mapper interface.

In addition to providing a means for discovering the names of all processors and memories in the system, the machine object also provides a means for querying important information about the various resources. For example, the machine object allows mappers to query for the kind of processor and memory. There are three kinds of processors: latency-optimized cores (CPUs), throughput-optimized cores (GPUs), and utility processors (CPU cores for performing runtime meta-work). There are many more types of memories than can be listed here, but several examples include system memory (DRAM), framebuffer memory (GPU GDDR), and registered memory (pinned memory that can be directly accessed by NIC hardware). By providing a means for classifying memories, mappers can be better informed about how to best approach mapping for a specific architecture.

To further aid in the mapper's decision making process, the machine object provides an interface for querying the affinity between various processors and memories. There are two kinds of affinity relationships: processor-memory affinity and memory-memory affinity. Processor-memory affinity exists when a processor can directly perform reads and writes to a memory using load and store instructions. Memory-memory affinity exists whenever there is a direct data movement path between the two memories. Affinity relationships are also annotated with the expected latency and bandwidth for performing operations between the different components, giving mappers some indication of the expected performance of different components.

In addition to providing mappers with a way to inspect the system through the machine object, mappers can also organize some components into collections. For example, instead of mapping a task onto a specific low-level processor, a mapper might wish to map it onto a collection of processors so that the first processor with additional cycles in the collection can perform the task. To makes this possible, the low-level runtime supports the creation of *processor groups*. Processor groups must contain processors that all originate from the same node and all have the same kind. The creation of a processor group also entails the creation of a common work-queue from which all the processors in the group will pull. For example, an application might wish to create a processor groups for all the processors that share a NUMA domain since it is likely a task mapped onto this processor group will run equally

fast on any of the processors in the group. Tasks mapped onto processor groups must always use memories that are visible from all processors. The runtime can detect violations of this requirement and will cause the task mapping to fail if they occur.

8.1.2 Selecting Tasks and Mapping Locations

Recall from Section 3.1.4 that one instance of every kind of mapper is instantiated for each application processor in the machine (CPUs and GPUs, but not utility processors). Once the dependence analysis and premapping stages of a task's execution are complete, the task is placed in the ready queue for the mapper that manages the processor on which the parent task is executing[1]. Whenever tasks are in the ready queue, the mapper is queried to select tasks that it would like to choose to map by the `select_tasks_to_schedule` mapper call. The set of tasks that are ready to map are given in an ordered list with tasks appearing earlier in the list having been in the list the longest. The mapper marks which tasks it wants to map next. The mapper can also select not to map any tasks. This is a useful approach in two cases. First, the mapper might determine that it has mapped sufficiently far into the future, and wants to keep more tasks on the ready queue and therefore available for stealing. Second, in the case of a debugging mapper, it may be useful to pause execution under some circumstances. If all mappers stop mapping tasks, then the application is temporarily halted[2].

The default mapper supports two modes for picking tasks to execute. These two modes correspond to different approaches for traversing the tree of tasks. In depth-first mode, the default mapper traverses the task tree in a depth-first manner, performing to select tasks that have the largest depths even if they have been in the queue a shorter time. In breadth-first mode, the default mapper always selects the tasks that have been in the queue the longest to map. In general, depth-first mode optimizes for latency while breadth-first mode optimizes for throughput. Our default

[1]If it is an index space the task will also be sliced first with slices placed in the target ready queues, see Section 6.2.2.

[2]Interestingly, this can result in a certain class of mapper bugs that make the application to appear to hang. The default mapper automatically guards against this case, but custom mappers must be careful to avoid this case themselves.

mapper currently optimizes for throughput by performing breadth-first traversal of the tree of tasks, but can be easily switched to a depth-first traversal mode with a command line flag that will be observed by all default mapper instances.

As part of selecting which tasks to map, the default mapper can also make two final decisions for each task regarding which processor will perform the mapping and whether the task will be mapped *locally* or *remotely*. There are three scenarios. First, a task can be aimed at the current processor being managed by the mapper, in which case it is guaranteed to be mapped locally. Second, the task can be aimed at a remote processor and be marked to map remotely. In this case the task is sent to the remote processor's mapper and placed in its ready queue where it can either be mapped or sent to another processor by the mapper of the remote processor. Third, the task can be sent to a remote processor and mapped locally. The current mapper will then map the task on the local node and then send the result to the remote node where the task is immediately launched onto the processor foregoing being placed in the remote processor's mapper's ready queue. The tradeoffs between mapping locally and mapping remotely are discussed in detail in Section 6.2.

8.1.3 Mapping a Task

Once a task is ready to be mapped (either locally or remotely), the `map_task` mapping function is invoked. At this point the target processor of the task has already been selected, and it is the responsibility of the mapper to create a ranking of memories in which to attempt to place physical instances for each region requirement. A separate ranking of memories can be specified for each of the region requirements, giving the mapper the flexibility to specify which data should be placed close to the processor and which data can be left further away. To aid the mapper in the decision making process, the runtime also provides information for each region requirement about available physical instances as well as which fields are already valid for these physical instances.

Our implementation of the `map_task` method in the default mapper relies on several very simple heuristics for generating mappings for real applications. First, in

order to select a processor for a given task, the mapper checks the variants available for the task to see which kinds of processors are supported. If a variant is available for GPU processors, then the default mapper will attempt to map the task onto a GPU processor[3]. If only CPU variants of the task are available, then the default mapper will target a CPU processor. If the task is an individual task, the default mapper will elect to keep the task on the origin processor (or a nearby GPU on the same node if a task has a GPU variant) in order to encourage locality of data with the parent task. When stealing is enabled (Section 8.2) these individual tasks can be pulled by other processors, however our initial preference for locality is important for reducing data movement. If the task is an index space of points, the default mapper round-robins tasks across all of the processors in the machine of the chosen kind (GPUs if a variant exists, otherwise CPUs). This approach is designed to foster load balancing for index space tasks that routinely launch at least as many tasks as there are processors in the machine.

Once the default mapper has selected a specific processor for a task, then it proceeds to assign rankings for each of the region requirements requested by the task. The default mapper initially discovers the set of all the memories visible from the target processor. The default mapper then ranks the memories based on the bandwidth available from the target processor with those having higher bandwidth being ranked better. The default mapper bubbles memories that already have valid instances for the data to the top, allowing for re-use of existing instances wherever possible and improving locality. This process is applied for each of the region requirements requested by the task, allowing the default mapper to create a customized mapping decision for each region based on both the performance of the existing hardware as well as the existing state of the data in the memory hierarchy.

8.1.4 Physical Instance Layout

In addition to specifying a ranking of target memories for each region requirement, the `map_task` mapping call also gives the mapper the ability to place constraints on

[3]Our default mapper implementation skews towards optimizing for throughput instead of latency since most Legion applications are limited by throughput.

the layout of data for each physical instance. These constraints take several forms.

- Field Ordering - constraints on the ordering of fields within the physical instance.
- Index Space Linearization - constraints on how the index space is linearized into memory including ordering of dimensions as well the function that must be used for linearizing domains (e.g. to support z-order curves, etc.).
- Interleaving - control how fields and index spaces are interleaved for data layout to support patterns such as array-of-structs (AOS), struct-of-arrays (SOA), and hybrid layouts as well as interleaving of dimensions with fields (e.g. 2-D slices of a 3-D index space with each slice containing two fields in AOS format).

These different kinds of constraints give mappers full control over the layout of data for physical instances and provide no limitations of the kinds of data layouts that are supported. Constraints are specified as sets and the relatively simple nature of the constraints (no arithmetic), allow for easy testing of set constraint satisfaction and entailment. To help the mapper in picking constraints, the runtime also supplies the mapper with the constraints that dictated the layout of the existing physical instances that can be reused for each region requirement.

The default mapper uses a simple heuristic for determining how to layout data. Data for GPU processors is always laid out in SOA format in order to guarantee memory coalescing of global loads[4]. For CPU processors, the mapper checks to see whether there exists variants capable of using vectorized layouts or there exists a variant generator (see Section 8.1.5) capable of generating vectorized CPU code. If either condition is met then the default mapper specifies a hybrid layout that interleaves all fields with a blocking factor equal to the width of the vector units on the target CPU. If vectors are not supported and no variant generator exists, all physical instances are assigned variants for generating AOS physical instances that optimize for linear striding through memory for CPU prefetch engines.

[4]Note SOA format also requires no ordering constraints on fields.

8.1.5 Task Variants and Generators

One important property guaranteed by the mapping interface is that no mapping decision will ever impact the correctness of a Legion application. To guarantee this property, the runtime is responsible for validating the selection of a task variant to execute based on the chosen processor and the selected physical instances along with their layout constraints. When variants are registered with the runtime, they also register constraints on their execution (e.g. processor kind and physical instance layout constraints). After all of the mapping decisions have been made, the runtime iterates over the set of variants for the given task to find the set of variants whose constraints can be satisfied by the chosen processor and physical instances. There are three possibilities for this set. First, the set can be empty in which case no variant can be selected. Under these circumstances the runtime triggers a mapping failure and asks the mapper to retry the mapping (see Section 8.3 for more information on mapper feedback calls). Second, the set can contain a single variant, in which case, the runtime immediately selects this variant as the one to use and runs the task. Finally, in the third case, there can be multiple valid variants that can be used. In this circumstance, the runtime invokes the `select_task_variant` mapper call to ask the mapper which variant it would prefer to use. Whenever the default mapper encounters multiple valid variants it picks the one with the largest number of constraints based on the assumption that a more highly constrained variant has been further specialized and is therefore likely to achieve higher performance.

Due to the large potential design space of variants for a task (often hundreds to thousands based on the cross-product of processor kinds and various instance layout options), it is unreasonable to expect authors of Legion programs (or even higher level programming systems such as DSL compilers) to emit all combinations of these task variants up front. Instead, Legion allows applications to register task *generators*. Task generators are Lua functions, written within the Lua-Terra meta-programming framework [26], that can emit Terra versions of leaf tasks based on an input set of constraints. Once generated, the Terra function is then JIT compiled and can be used as a Legion task. If a task generator function is registered with the runtime, it will be invoked any time an existing variant does not exist for a given set of constraints. The

generator is responsible for emitting a Terra task based on the set of processor and data layout constraints that have been passed to it by the runtime. The generator can also return an additional set of constraints that specify the conditions that must be met for the generated variant to be used[5]. The runtime then memoizes the variant along with the necessary constraints for using it so that it can be re-used where possible for future task executions.

It is important to note that generator functions also provide a useful level of indirection for DSL compilers that target Legion. Generator functions can base their Terra code generation off of any intermediate representation that they choose, including, but not limited to, strings, assembly code, LLVM IR, or Terra abstract syntax trees. This level of indirection decouples the Legion runtime interface from any one DSL compiler infrastructure, allowing DSL compilers to be built in any framework, as long as they can emit Lua-Terra generator functions.

8.1.6 Addressing Critical Paths with Priorities

The last aspect of mapping a task is determining the *priority* for a task. While most applications written for Legion are limited by throughput of tasks or data movement, there are still phases in many of these applications that contain a critical path that impacts the performance of the overall application. Under these circumstances it is important that mappers have a means for indicating to the runtime that tasks and copies are on a potentially critical path to ensure that the execution of these operations is not obstructed by non-critical operations. To make this possible, the mapping interface is allowed to assign an integer to a task that indicates its priority. By default, the priority for all operations is set to zero. Mappers can assign a positive integer to increase priority or a negative integer to decrease priority. One interesting open problem is how to normalize between priorities assigned by different mappers. For example, one mapper might assign priorities on a scale from -10 to 10, while another uses a scale from -100 to 100. For such cases, it would be beneficial for the

[5]This set of constraints can be a subset of the constraints passed to the generator function as the generator function can be used to emit more general code that satisfies the necessary constraints and also may satisfy other constraints as well.

two priority scales to be equated and normalized to ensure that tasks are scheduled appropriately. Currently, the default mapper makes no attempt to guess priorities for tasks and assigns the default zero priority to all operations.

The priority that is assigned to a task applies to both the execution of the task as well as any mapping operations that are generated as part of mapping the task. This ensures that data movement that may reside on a critical path is also prioritized.

8.2 Load Balancing

For many of the dynamic applications for which Legion is designed, load balancing is an important performance issue. Legion supports several different features for allowing mappers to customize load balance at runtime. These features can be categorized into two areas: support for load balancing within a node and load balancing between nodes. We briefly describe how load balancing within a node can be achieved in Section 8.2.1 and then describe the two features for managing load balancing between nodes in Sections 8.2.2 and 8.2.3 respectively.

8.2.1 Processor Groups

To support load balancing within a node, Legion permits mappers to create *processor groups*. A processor group is effectively a name for a task queue that is shared between a set of processors. Tasks that are mapped to a processor group are placed in the task queue for the processor group. As soon as any processor in the processor group is available for executing a task, it will pull a task off the queue and execute it. Using processor groups, mappers can easily load balance task execution across a set of processors.

The runtime does enforce three requirements on the use of processor groups. First, all processors in a processor group must be on the same node. The reason for this is the processor group must share a common task queue data structure that cannot be implemented in a distributed environment without considerable communication overhead. Second, all processors in a processor group must be of the same processor

kind. This is necessary to ensure that tasks have selected the proper variants for running on the processor group. Finally, all regions that have been mapped for tasks launched on a processor group must be visible to all processors in the processor group. This is necessary to ensure that when the task is run it will run correctly regardless of the processor that the task is assigned.

8.2.2 Task Stealing

There are two possible mechanisms for performing inter-node load balancing: one based on a *pull* methodology and one based on *push* methodology. In this section, we describe how task stealing can be used to pull work between nodes while the next section covers how work can be pushed between nodes.

To support task stealing, as part of every scheduler invocation, the Legion runtime invokes the `target_task_steal` mapper call, which queries each mapper to see if it would like to target any other processors in the system for task stealing. The mapper is free to target no processor for stealing or to target any subset of processors. If steal requests are made, the runtime sends the necessary messages to the same kind of mappers on the remote node. When these requests arrive, they trigger an invocation of the `permit_task_steal` mapper call. In Legion, tasks are not stolen automatically. Mappers that own tasks must explicitly permit them to be stolen. The reason for this is that most task stealing systems operate in shared memory environments (e.g. [30]), and there is minimal data movement as a result of stealing.

In Legion, stealing primarily occurs between nodes, and the cost of moving data is much higher. We therefore give the owning mapper the prerogative to reject steal requests. The receiving node of a steal request is only permitted to allow tasks currently in its ready-queue to be stolen. Tasks that have been mapped onto a processor are not eligible for stealing. If a task is permitted to be stolen, then its meta-data is moved to the node where the steal request originated, and the task itself is added to the ready queue of the requester mapper. It is important to note that this approach only allows task stealing between mappers of the same kind, which guarantees that only a mapper of the kind assigned to the task will perform the

mapping.

In order to avoid communication overheads associated with repeated steal attempts, the runtime tracks which steal requests fail. If a steal request from processor P_1 to processor P_2 fails for a mapper of kind M, then P_2 is added to the blacklist for mapper M of processor P_1. The blacklist prevents any additional steal requests from P_1 being sent to P_2. At the same time, the runtime keeps track of failed steal requests for P_2. As soon as new tasks are added to the ready queue of mapper M on P_2, then an advertisement is sent to processor mapper M of P_1 to alert it to the new tasks that may be eligible for stealing. The advertisement then clears P_2 from the blacklist of P_1 for mapper M. The idea behind this approach is that mappers are unlikely to permit stealing of tasks that were already rejected for stealing once. However, with new work, the mapper has acquired additional execution responsibility and may make a different mapping decision.

One important detail about stealing is that stealing is permitted within a node as well as between nodes. However, within a node the more effective approach is likely to be the use of processor groups as they provide a finer-grained approach to load balancing that works better within a single node.

Since the default mapper has no information about the structure of an application, in normal conditions it avoids stealing. Stealing is easily enabled with a command line flag. When enabled, the default mappers on various processors randomly choose a processor from which to attempt to steal. Random choices avoid stampedes where all mappers attempt to steal from the same processor at the same time. Clearly there is room for a more advanced stealing scheme, but for now it is left for future work.

8.2.3 Mapper Communication

The other approach to performing load balancing is to implement a push-based model where mappers coordinate to balance load. To allow mappers to coordinate with each other, the mapper interface provides a mechanism for mappers to send messages to other mappers of the same kind on other processors. A message consists of a pointer to an untyped buffer and a size of the number of bytes to copy. The runtime makes a

copy of this buffer and transmits it to the target node. On the target node the runtime invokes the message handler mapper call `handle_message`. Mappers are permitted to send messages from inside of any mapper call including the message handler mapper call.

Using messages, mappers of the same kind can orchestrate dynamic load balancing patterns that can be re-used for long epochs of application execution. For example, in an adaptive mesh refinement code, a custom mapper implementation could have mappers communicate the load of tasks that they receive after each refinement. Based on load information for different processors, each mapper can independently compute a load balancing scheme and determine where to send all the tasks it is initially responsible. The mappers can memoize this result and re-use it until a new refinement occurs or an old refinement is deleted. The granularity at which load balancing schemes are computed will vary with the size of the machine and the amount of work being generated, but these kinds of performance considerations are explicitly left to the mapper by design.

8.3 Mapper Feedback

One of the most important aspects of the mapper interface is that it contain mechanisms for mapper objects to either receive or request feedback about how the application is actually mapped onto the target architecture. Without this feature, mappers would be able to make decisions that affect performance, but would have no mechanism with which to reason about how those decisions impacted performance. In this section we cover the aspects of the mapper interface that permit mapper objects to determine how mapping decisions are impacting application performance.

8.3.1 Mapping Results

The first way that mappers can receive feedback about mapping decisions is through the `notify_mapping_result` and `notify_mapping_failure` mapper calls. Anytime that a task, copy, or inline mapping fails to map (usually because physical instances

for the requested regions could not be allocated) the runtime notifies the mapper that made the mapping decisions by invoking the `notify_mapping_failure` mapping call. This call passes a pointer to the operation that failed to map. The runtime annotates all of the region requirements for the operation indicating whether they succeeded or failed to map. This mapper call is a passive function in that the mapper has no responsibility regarding the operation (it is immediately added back into the ready queue by the runtime), but the mapper object can record the result and use it the next time the operation attempts to map.

To further gain insight into how mapping decisions impact operation performance, mappers can request the runtime notify them when operations are successfully mapped using the `notify_mapping_result` mapper call. Each of the `map_task`, `map_copy`, and `map_inline` mapper calls require the mapper to return a boolean indicating whether they would like to be notified of the mapping result if the mapping succeeds. If the mapper returns `true` then the runtime calls `notify_mapping_result` if the mapping for that operation succeeds. The runtime annotates each region requirement of the operation with the name of the physical instance that was mapped and the memory in which it was located. Mappers can then record this information and use it in conjunction with the profiling information for a task (described in Section 8.3.2) to gauge how the resulting mapping impacted performance. Mappers can then use this information when mapping future instances of the task.

8.3.2 Performance Profiling

While knowledge about how tasks and other operations are mapped is a necessary condition for understanding the performance of an application, it is not sufficient. Mappers also need access to profiling information to understand the performance implications of mapping results. The runtime therefore provides a mechanism for allowing mappers to request profiling results for tasks by setting the `profile_task` flag in the `select_task_options` call. Setting this flag indicates to the runtime to check the profiling options that mappers can request for a task in the `ProfilingOptions` structure that is available for all task instances. By setting flags in this structure,

mappers can ask for profiling options including the execution time of a task, total execution time of a task and its children, and hardware instruction and memory counters such as loads and stores issued and cache hits/misses. After the task has completed, the runtime fills in the requested fields in the `ProfilingOptions` data structure and invokes the `notify_profiling_info` mapper call to report the profiling results. By coupling these results with the mapping decision results reported by the mapper calls described in Section 8.3.1, mappers can infer the performance effects that different mapping decisions might have on performance. Profiling information closes the loop in the mapping process, giving mappers the ability to drive future mapping decisions based on the performance of previous mapping results.

8.3.3 Introspecting Machine State

While performance is one metric by which mappers might choose to make mapping decisions, another is resource utilization. For example, a mapper might opt to omit a specific memory from a ranking for a region requirement in the `map_task` mapping call because the memory is nearly full, and the mapper knows that the space needs to be reserved for a later task. Discerning such information requires mappers to query state information about different kinds of resources in the system. To make this possible, as part of the mapper interface the runtime provides calls for mappers to inquire about the state of different resources.

The first resource that mappers can query is the current usage of different memories. Mappers can ask the runtime how much memory is already allocated or already free within a given memory. The result of this call should be interpreted as a point sample and doesn't guarantee that the same amount of space will be available in the immediate future. The reason for this is that allocations in memories are handled by the low-level runtime. Consequently there may be other tasks from both the same node as well as remote nodes that may be issuing allocation requests as part of their mapping process to the same memory in parallel. However, by using this query, mappers can gain insight into the underlying state of different memories.

The second resource that mappers can query is the current number of outstanding

tasks mapped onto different processors. This metric gives an indication as to the current workload of different processors, allowing mappers to do a better job of load balancing. Similar to the memory usage query, this query should always be interpreted as a sample that is subject to change immediately as other operations may be mapped onto the processors in parallel.

While the runtime currently only supports querying these two resources, there is significant opportunity to include profiling for additional resources. For example, the runtime routinely knows that copy operations are pending which should provide sufficient information to gauge future load on data movement pathways such as interconnect pipes and PCI-Express buses. While the implementation of mappers that could leverage this information is some ways off, it is likely that intelligent future mappers will be capable of leveraging this information to achieve higher performance.

Optimizing for resource usage is important, but also very difficult to do locally by individual mappers making mapping decisions for a single operation at a time. As part of future work we plan to investigate mechanisms for mapping many operations together as groups. This would allow for mappers to better balance trade-offs for resource usage and to make better global decisions with additional information. Mapping operations in groups would also allow for other interesting optimizations such as fusing operations that could reduce resource utilization, and allow dynamic program analysis in the Lua-Terra framework to further optimize generated code.

8.3.4 Managing Deferred Execution

The final area where the mapper can receive feedback about the state of an application is through the dynamic data flow graph. Legion makes available the input and output dependences for all operations, allowing the mapper to explore the same of the dynamic dataflow graph. The runtime also provides a mechanism for mappers to query which operations have already been mapped and which ones are still un-mapped. By combining access to the shape of the graph along with profiling information, mappers can infer critical paths that are important for assigning priorities. Furthermore, by monitoring the location of the wavefront of mapped tasks within the

graph, mappers can determine how far ahead of actual application execution mapping is occurring, thereby giving the mapper the feedback necessary to accurately manage deferred execution.

Chapter 9

Relaxed Coherence

In this chapter we describe *relaxed coherence modes* as an extension to the Legion programming model. Relaxed coherence modes improve both the expressivity of the programming model and enable higher performance for some Legion programs. The need for relaxed coherence modes stems from a very simple observation: not all programs require direct data dependence ordering on access to logical regions. In many cases it is permissible for tasks with interfering region requirements to be re-ordered as long as the runtime guarantees a weaker property such as atomicity. Relaxed coherence modes also allow applications to be more precise regarding the kinds of properties that the Legion runtime must guarantee when performing dependence analysis between tasks. In this chapter we discuss the introduction of two relaxed coherence modes and their implementation in the Legion runtime.

9.1 Relaxed Coherence Modes

By default, all region requirements specified in a Legion program use *exclusive* coherence. Exclusive coherence guarantees that at most one task can be accessing the logical region named by a region requirement. Furthermore, exclusive coherence ensures that interfering tasks are executed based on program order (the order in which they were issued to the runtime). This guarantees that all data dependences are

obeyed and program execution abides by serial execution semantics. To allow programmers to relax the constraints describing how the Legion programming model handles tasks with interfering region requirements, we introduce two relaxed coherence modes: *atomic* and *simultaneous* coherence.

9.1.1 Atomic Coherence

Atomic coherence is a relaxed coherence mode that allows Legion applications to specify that the accesses performed by a sub-task (or other operation) on a logical region are required to be serializable with respect to other operations. Serializability only ensures that interfering operations on the same logical region are performed atomically with respect to each other, and not necessarily that they are performed in strict program order. This gives the runtime the additional freedom to re-order tasks with interfering region requirements as long as both have requested atomic coherence. If either of the region requirements had requested exclusive coherence then re-ordering would not be sound as it would violate the semantics of exclusive coherence for one of the tasks. Atomic coherence is therefore useful for tasks that need to make atomic updates to a given region, but permit re-orderings of operations with respect to program order for higher performance.

9.1.2 Simultaneous Coherence

Simultaneous coherence is another relaxed coherence mode that permits two tasks with interfering region requirements to either be re-ordered or potentially executed at the same time. The semantics of simultaneous coherence allow re-ordering similarly to atomic coherence, making no guarantees about serializability. Instead, simultaneous coherence permits two potentially interfering tasks to execute at the same time. The one requirement on concurrent execution is that if any interfering tasks are executing at the same time, then they all need to immediately see updates from all the other executing tasks. For this to be the case, the runtime must guarantee that simultaneously running tasks that interfere on some region requirements have been

mapped to the same physical instance[1], ensuring that updates are immediately visible to other interfering tasks. We describe the necessary changes to the physical region tree traversal for supporting the semantics of simultaneous coherence in Section 9.4.2.

One important property of simultaneous coherence is that it permits applications to effectively cut through some of the standard Legion abstractions when programmers want more control over expressing dependences (or lack thereof) between different operations. Consequently, programmers can express other programming paradigms without the overhead and support of the Legion runtime. For example, an application may have stored a data structure such as a tree inside of a logical region. If the application attempts to launch many fine-grained tasks with atomic coherence to make modifications to the tree, it can result in considerable runtime overhead. Instead, the application can launch fewer tasks with simultaneous coherence on the region and rely on one of the finer-grained synchronization primitives supported for simultaneous coherence that we discuss in Section 9.2.

9.1.3 Composability and Acquire/Release Operations

Unlike privileges on logical regions that can be passed from a parent task to a child task, relaxed coherence modes apply only between sibling tasks. Consider, for example, two sibling tasks s_1 and s_2 that both request simultaneous coherence on the same logical region. If s_1 and s_2 execute in parallel, then they can each launch child tasks with their own coherence modes. Interference test are only performed between tasks within the same context. It is the responsibility of the application to handle synchronization between tasks in different contexts accessing logical regions with simultaneous coherence.

While it is the application's responsibility to manage synchronization in these cases, it is the Legion runtime's responsibility to ensure the invariants of simultaneous coherence are met; specifically all updates by a task (including all updates from its child tasks) must be made visible in the simultaneous mapped region immediately. In general, to provide the expected behavior, child tasks of a task with a region using

[1]In the case of cache coherent memories, there is some flexibility where the runtime can rely on the hardware to maintain coherence.

simultaneous coherence must map to the same physical instance to guarantee that updates are immediately visible. Any region requirements for child-tasks (or other operations) that derive their privileges from a parent task region that was mapped with simultaneous coherence is automatically marked by the runtime as being *restricted*. This flag informs the mapper that, regardless of any attempts to map the task to a different physical instance, the runtime will force the use of the same physical instance as the parent task. This maintains two important invariants of the Legion programming model: all updates for regions with simultaneous coherence are immediately visible and no mapping decisions can impact the correctness of application execution.

The restricted property provides for the enforcement of simultaneous coherence semantics, but there are many cases where application developers know that it is safe to elide such restrictions. To communicate this information to the runtime, Legion provides an explicit system of coarse grained software coherence using *acquire* and *release* operations. Issuing an acquire operation for a logical region that the parent task has mapped with simultaneous coherence removes the restricted property, allowing sub-tasks to map physical instances of the logical region in any potential memory. A corresponding release operation indicates that the restricted property should be enforced again. This invalidates all existing physical instances except the one initially mapped by the parent task and flushes all dirty data back to the originally mapped instance, thereby making changes visible to other concurrent tasks using the same physical instance. Like all operations in Legion, acquire and release operations are issued asynchronously and the runtime automatically determines the necessary dependences for actually triggering execution of the operations. There are no coherence annotations associated with acquire and release operations. Both kinds of operations behave as if they were performed with exclusive coherence.

Unlike simultaneous coherence, atomic coherence does not suffer from the same composability issues. Since each parent task is guaranteed to have atomic access to logical regions with atomic privileges, then all of its child tasks can run and map in any possible memory. Under these circumstances the naturally hierarchical semantics of Legion task execution will flush any observable changes back to the mapped physical instance before the parent task completes and any of its siblings can observe the

changes.

9.2 Synchronization Primitives

When using simultaneous coherence with logical regions it is the responsibility of the application to perform any necessary synchronization in order to avoid data races. There are two potential avenues for implementing synchronization protocols in conjunction with simultaneous coherence. First, users can implement their own synchronization primitives and store them in logical regions that are also accessed with simultaneous coherence. Simultaneous coherence guarantees that if two tasks are executing at the same time with privileges on interfering logical regions, then they are using the same physical instance with reads and write being immediately visible. Along with an atomic operation (e.g. compare-and-swap), this condition is sufficient to construct locks and similar synchronization primitives. The alternative approach to implementing synchronization in conjunction with simultaneous coherence is to use one of the two built-in synchronization primitives provided by Legion: *reservations* (Section 9.2.1 and *phase barriers* (Section 9.2.2).

9.2.1 Reservations

Reservations are the synchronization primitive provided by the Legion runtime for guaranteeing atomicity when accessing data in a region with simultaneous coherence. Applications issue *requests* for reservations that may or may not be immediately granted. If a task with simultaneous coherence issues a request, then it must block and wait for the request to be granted before continuing. Legion is able to preempt a blocked task and schedule additional tasks until the reservation is granted in order to hide the latency. Alternatively, the task might issue child tasks and request reservations on behalf of the child tasks. Legion adds the reservation request as part of the prerequisites for the child task and ensures that the child task is not scheduled until the reservation is granted. Reservations can be released at any point after an acquisition.

Reservations support being acquired in different *modes* and with different *exclusivity* guarantees. When a reservation request is made, the application specifies the mode (e.g. an integer). A reservation can be held in at most one mode at a time. Furthermore, the exclusivity requirement states whether other users are also permitted to hold the reservation in the given mode at the same time. If exclusivity is requested, then at most one user can hold the reservation at a time. However, if exclusivity is not requested, then multiple users all requesting the same mode (without exclusivity) are permitted to hold the reservation at the same time. Exclusivity is useful for supporting read-only cases where multiple users of a given mode are acceptable because they are all reading a data structure.

9.2.2 Phase Barriers

The other synchronization primitive provided by the Legion runtime for use with simultaneous coherence are phase barriers. Phase barriers provide a way of ordering access to logical regions in simultaneous coherence. Unlike traditional barriers that block waiting for all of the users of a resource to arrive, phase barriers can be used to encode producer-consumer relationships between subsets (or simply pairs) of users. When phase barriers are created, the application specifies the number of *arrivals* that must be observed before the *generation* of the barrier advances. Arrivals are non-blocking operations performed by tasks to signal that they have completed (produced) some value and stored it in a logical region mapped with simultaneous coherence. Similarly, other tasks can block waiting on a specific generation of a phase barrier. When a barrier advances to a new generation, it triggers all the tasks waiting on the new generation, indicating that it is safe to consume the results in a logical region mapped with simultaneous coherence. It is the responsibility of the application to maintain the association of phase barriers with simultaneously mapped logical regions.

Using phase barriers, applications can encode producer-consumer relationships between tasks with access to logical regions in simultaneous coherence. This is especially useful for capturing common communication patterns such as exchanging ghost cells through logical regions with simultaneous coherence as we will show in Chapter 11.

In order to use phase barriers safely, applications need a crucial guarantee from the runtime regarding the concurrent execution of tasks. Phase barriers require that all tasks participating in the synchronization operation are executing at the same time to avoid deadlock. In Section 9.3 we discuss the specific mechanism that Legion provides for applications to express this constraint of an application to the runtime.

9.3 Must Parallelism Epochs

When developing programs with simultaneous coherence and explicit synchronization between tasks, it is often necessary for tasks to synchronize with each other as part of their execution to make forward progress. As an example, consider the S3D application described in Chapter 11. S3D creates explicit regions for storing ghost cells and uses phase barriers to synchronize between different tasks that have mapped the explicit ghost cell logical regions with simultaneous coherence. In order to avoid livelocking, the application requires a mechanism for communicating to the runtime that some tasks must be guaranteed to be executing at the same time. Legion provides support for these cases with *must epochs*.

A must epoch is a collection of tasks that all must execute concurrently on different processors, thereby making it possible for tasks to synchronize with each other. In some ways, this mirrors the semantics of traditional SPMD programming models such as MPI which guarantee that all processes run concurrently and can therefore synchronize with each other using barriers. However, unlike traditional SPMD programming models, the phase barriers and reservations available as synchronization primitives do not block the Legion deferred execution model. Must epoch task launches also differ in that they can be used at several different levels in the task tree. In practice, must epoch task launches are used infrequently and mainly only when relaxed coherence modes are necessary.

Must epoch task launches can consist of an arbitrary collection of individual and index space task launches. The tasks that are launched proceed through the standard dependence analysis independently, but the runtime captures the set of simultaneous coherence constraints that exist between tasks within the must epoch launch. When

all of the tasks within a must epoch launch have had all their mapping dependences satisfied, the runtime invokes the `map_must_epoch` call to map all of the tasks within the must epoch. As part of the mapper call, the runtime passes in all the simultaneous coherence constraints that must be satisfied (e.g. region requirements that must all be mapped to the same physical instance) in order for the must epoch to be successfully mapped. The mapper then performs the equivalent of the `map_task` mapper call for each of the tasks. It is the responsibility of the mapper to satisfy all of the mapping constraints or else the mapping will fail. The mapper is also responsible for assigning each task in the must epoch to a different target processor so that all the tasks can execute concurrently. The runtime uses the results of the `map_must_epoch` call to perform the mapping for each of the tasks. If all the tasks successfully map, all the tasks are assigned to different processors, and all the simultaneous coherence constraints are satisfied, then the must epoch mapping will be executed.

The introduction of must epoch launches has an interesting consequence for the design of the Legion programming model. Must epoch launches allow applications to be written in a way that permits longer running tasks that operate in parallel, communicating data through regions with simultaneous coherence and synchronizing with reservations or phase barriers. This property makes it possible to write applications that behave similarly to the SPMD applications discussed in Section 1.2. The primary difference being that Legion tasks written in this style are still executing in a deferred execution model that confers considerably more latency hiding ability that existing SPMD approaches.

Long running tasks using regions mapped with simultaneous coherence can also be used to implement applications that require pipeline parallelism (e.g. [47]), with different long running tasks performing the operation of different pipeline stages and logical regions mapped with simultaneous coherence storing queues of data. Must epochs give Legion the power to capture these patterns and thereby increase the expressivity of the programming model.

9.3.1 A Complete Example

Initially all of the various features of relaxed coherence modes may seem disparate and unrelated. However, for many applications, all of the different components are essential for using relaxed coherence modes to achieve high performance. As a minimal example of how all these features compose, we introduce a very simple stencil example.

Consider a simple one dimensional stencil computed using explicit ghost cell regions. In this scenario, many parallel tasks are launched, each of which requests privileges on the logical region of local cells that it owns (`local_lr`), two logical regions of explicit ghost cells that it owns (`left_lr` and `right_lr`), and the left and right ghost cell regions from its right and left neighbor respectively. The region `local_lr` is requested with exclusive coherence, but the other regions are requested with simultaneous coherence to enable all the tasks to run at the same time. In order to permit the tasks to synchronize using phase barriers, the tasks are all launched as part of the same must epoch. Since they are all launched in the same must epoch, the launching node maps them all at the same time. Each task is assigned a different processor and maps its `local_lr` and neighbor `left_lr` and `right_lr` into memories local to the target processor. The idea behind this mapping is that tasks will push their data to the explicit copies of their owned `left_lr` and `right_lr` on remote nodes.

Figure 9.1 illustrates the execution of the stencil tasks for several iterations. When the tasks run, they may perform other computations on `local_lr`, but eventually they need to perform the stencil computation. To perform the stencil computation they first issue explicit copies from their `local_lr` logical region to the ghost cell regions they own (which have been mapped into memories on remote nodes). Phase barriers are triggered contingent upon the completion of the copies to mark when the data in the owned logical regions are valid. After issuing the explicit copies, the tasks then perform an acquire request for the neighbor explicit ghost cell regions contingent on the phase barriers from each of their neighbors have triggered. The acquire operation is necessary to gain coherence for the explicit ghost cell logical regions so they can be moved if necessary. In the specific case of Figure 9.1, the acquire operation is necessary so the stencil computations themselves can be run

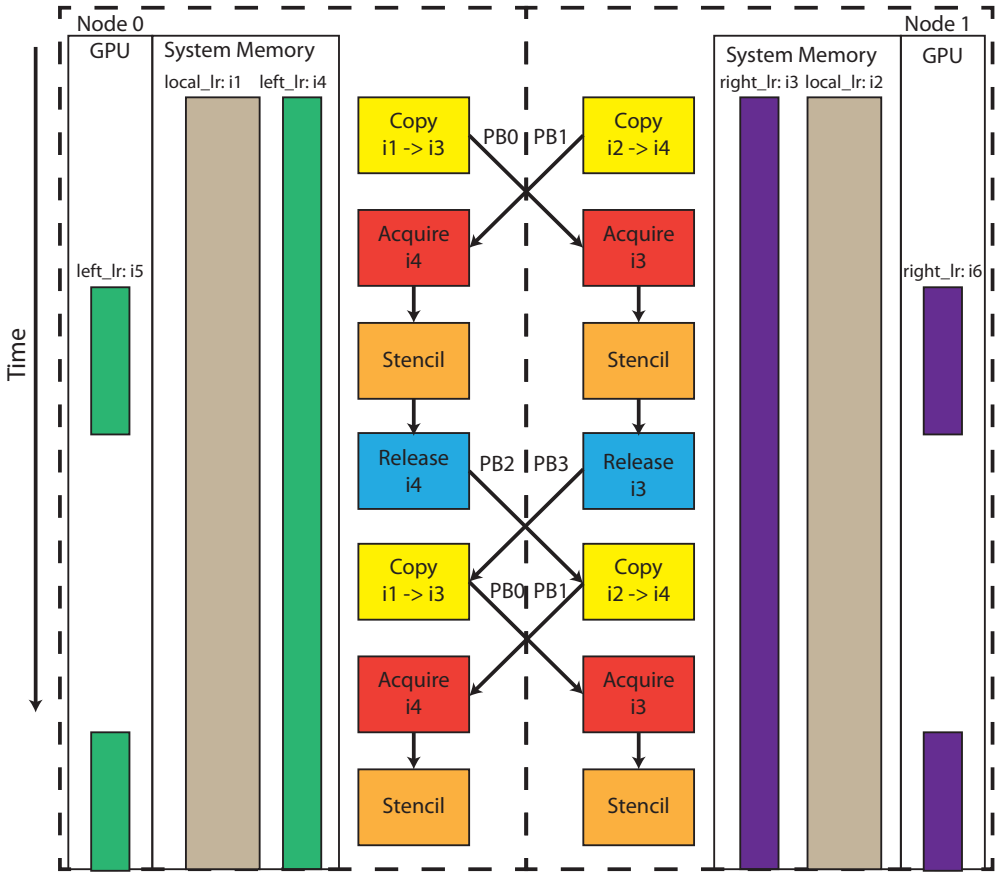


Figure 9.1: Example Use of Simultaneous Coherence and Related Features

on a GPU. After the acquire operation is issued, the explicit stencil computation is launched (Legion automatically computes the dependence based on region usage). After the stencil is performed, then a release operation is issued to invalidate any copies of the explicit ghost cell regions that were created. A second phase barrier is triggered upon completion of the release operation to indicate that the results in the explicit ghost cell regions have been consumed. Each of these phase barrier triggers become a precondition for the next explicit copy request that is performed into the explicit ghost cell regions to prevent write-after-read races.

Figure 9.1 has three important properties. First, once this stencil pattern of communication is set up, tasks can run for arbitrarily long[2], re-using the phase barriers

[2]Until the phase barriers exhaust their 2^{31} generations.

over and over again. While the pattern expressed here is sometimes difficult for human programmers to set up, it is trivial to generate either as part of a library and or a higher-level language that targets Legion.

The second interesting property of this example is that the operations necessary for computing the stencil never block, thereby enabling continued deferred execution of the tasks. This is especially important if the tasks have significant additional work to do on `local_lr` which is mostly non-interfering with the stencil computation (this will be the case with S3D as we will see in Chapter 11). The non-blocking nature of the operations, even with their composition with phase barriers, allows Legion to find additional work to do while the long latency copy operations are being performed.

Third, while the pattern set-up in this example is relatively simple, more complex double- and triple-buffered algorithms can be employed for further latency hiding. The choice of the buffering depth is often application and machine dependent and it is easy to parametrize the buffering depth as a tunable variable that can be changed by the mapper. Legion applications therefore have a rich set of possibilities available to them for exchanging data between executing tasks using explicit logical regions.

This stencil example shows how all the different components associated with relaxed coherence modes compose naturally and can be used to write high performance distributed code. In practice, all of these features will be used as part of our implementation of the S3D combustion simulation in Chapter 11.

9.4 Tree Traversal Updates

In order to support relaxed coherence modes, several changes need to be made to both the logical and physical region tree traversal algorithms from Chapters 4 and 5. In this section we describe the general modifications to both the traversal algorithms as well as the additional state added to the region tree data structures for supporting these additional traversals.

9.4.1 Logical Tree Updates

There are several modifications that need to be made to the logical region tree traversal algorithm. First, there is a slight modification to the definition of interference for relaxed coherence modes. While both atomic and simultaneous coherence modes prevent interference on region requirements that would normally be interfering if exclusive coherence was used, for the logical traversal it is still important to detect any potential interference and register a mapping dependence. The reason is that relaxed coherence modes only prevent non-interference if the two region requirements are ultimately mapped to the same physical instance. If they fail to map to the same physical instance, then the runtime will need to inject a copy to move the data that will ultimately serialize the execution of the two tasks and elide the need for relaxed coherence. In order to detect whether the two potentially interfering region requirements both map to the same physical instance or not, the mapping of the two different region requirements therefore must be serialized by a mapping dependence.

To detect these additional mapping dependences the majority of the region tree traversal remains the same. Traditional tests for non-interference based on logical regions, fields, and privileges all remain the same. However, for cases where non-interference would normally be detected based on relaxed coherence modes, mapping dependences are recorded. It is possible that these mapping dependences ultimately will not result in any event dependences in the event graph, but the result depends on the chosen mapping. While these additional mapping dependences for relaxed coherence modes do serialize some aspects of the region tree traversal, in practice we have found the latency hiding capabilities of Legion's deferred execution model to be sufficient for eliminating any additional overhead.

The second modification that must be made to the logical region tree traversal algorithm is the addition for tracking whether mapping of any region requirements are restricted because an ancestor task initially mapped a region with simultaneous coherence. The initial part of detecting mapping restrictions is done before traversal begins. Each parent task records which of its physical instances were restricted when it mapped. If the parent task was either restricted or directly requested simultaneous coherence, then any child tasks with region requirements that subsume their privileges

from the restricted region must also be restricted. If either of these cases hold, then the appropriate region requirements of the child task are also marked restricted.

After a child task has completed its logical dependence analysis for a region requirement, it then checks to see if the chosen requirement is also potentially restricted. If the requirement is restricted due to simultaneous coherence, then an additional traversal is necessary to detect whether there are any outstanding acquire operations that might remove the restriction. To support detecting such cases, we first annotate each node on the logical region tree with a field mask for storing which fields for that particular logical region have acquired user-level coherence. When an acquire performs its logical dependence analysis, it records no dependences[3], but instead mutates all the fields in the user level coherence mask in the logical node for which user-level coherence was requested. The acquire operation then recurses down all sub-trees from the acquired logical region annotating all user-level coherence field masks as well. Release operations have the opposite effect of unsetting the flags in the user-level coherence bit masks as later operations. Unlike acquire operations, release operations undergo the standard mapping dependence traversal for its region requirement as it will be required to actually perform operations on the physical region tree as we will discuss in Section 9.4.2.

Using the user-level coherence bits, operations that might otherwise be restricted can detect if they actually have acquired user-level coherence and can therefore avoid any mapping restrictions. After an operation completes its normal logical dependence analysis traversal for a region requirement, if the region requirement is possibly restricted based on analysis of its parent task's regions, then the operation can traverse the path from where the parent task has privileges to where the region requirement is requesting privileges to detect if user-level coherence has been acquired on any of the logical regions. If all of the fields for the region requirement can be shown to be in an acquired state, then the restriction on mapping due to simultaneous coherence can be removed.

[3]The acquire operation does not need to record any mapping dependences as it simply modifies the state of the logical region tree.

9.4.2 Physical Tree Updates

There are fewer modifications to the physical dependence analysis than for the logical dependence analysis. The most obvious change involves region requirements that are restricted due to simultaneous coherence. In these cases, any input from the mapper is ignored and the runtime proceeds to use the physical region mapped by the parent task. If the physical region is not visible from the target processor then the mapping will fail and the runtime reports that an invalid target processor was selected.

The next modification that is made is for supporting the instance view traversal algorithm. In this case, the interference test for users is modified such that interference between two users both requesting atomic coherence, or two users both requesting simultaneous coherence is elided so that no interference results. Recall that an instance view only stores data for users of the same physical instance; therefore two users testing for interference within the same instance view have already satisfied the constraint for relaxed coherence modes that they use the same physical instance. If two tasks with a mapping dependence based on relaxed coherence modes map to different physical instances, then the copy generated by the Legion runtime is sufficient to properly serialize the execution of the two tasks.

In the case where the two tasks both request atomic coherence, the runtime uses the low-level runtime reservations to mediate atomic access to the physical instance. Due to the large number of fields associated with many physical instances, the runtime uses a few reservations per physical instance and maps different fields onto different reservations. While this may in some cases result in unnecessary synchronization between tasks, in general it is not problematic and is better than maintaining a single reservation used for each physical instance which could lead to considerable unnecessary synchronization.

9.5 Implications for Hardware

Relaxed coherence has many potential implications for the design of hardware systems for both performance and power efficiency. In this section, we discuss how two

different kinds of hardware improvements could be automatically employed by Legion and the potential benefits for hardware.

9.5.1 Transactional Memory

Currently, the implementation of atomic coherence in Legion is based on the software-level reservation primitive provided by the low-level runtime. This leads to a very coarse-grained approach to synchronizing access to physical instances. Another approach would be the use of either a hardware or hybrid transactional memory system to allow multiple tasks to execute concurrently using the same physical instance. If conflicts between the tasks resulted in an execution that was not serializable the transactional memory could automatically abort the affected tasks and restart them. This could considerably improve the throughput of tasks that use atomic coherence and would also remove any potential for false serialization resulting from the shared reservation scheme described in Section 9.4.2.

9.5.2 Hardware Coherence

Another potential improvement afforded by relaxed coherence modes is the ability to elide hardware coherence for large ranges of virtual memory. Legion coherence modes are precise enough to know exactly which memory allocations need support from the hardware for providing cache coherence. Specifically, only physical instances using simultaneous coherence and internal runtime data structures would require hardware cache coherence. For all other physical instances (which for most applications is the majority of memory) hardware cache coherence could be disabled. The large power cost of hardware cache coherence could be considerably lessened, allowing for more efficient hardware. Furthermore, giving a programming system like Legion the ability to manage hardware caches using software primitives could ultimately lead to higher performance by explicitly mapping different logical regions into different caches for specific tasks, resulting in higher performance and lower power due to less data movement.

Chapter 10

Speculation and Resilience

In modern hardware out-of-order processors, branches can have a significant impact on performance. Waiting until the branch condition evaluates can stall execution and result in significant bubbles in the instruction pipeline. To avoid the performance degradation associated with branches, out-of-order processors rely on branch predictors and speculation to continue executing beyond a branch even before it has been evaluated. If a branch is mis-predicted, then the speculatively executed instructions are flushed, and execution restarts at the mis-predicted branch. With accurate branch predictors, speculative execution results in considerable performance gains.

The Legion programming model has an analogous problem to dynamic branching. In many applications, as a parent task is launching sub-tasks, determination of which sub-tasks to launch can be dependent on dynamic values computed by earlier sub-tasks. For example, an iterative numerical solver needs to continue executing sub-tasks until it converges. The test for convergence often is dependent upon earlier sub-tasks, and the decision about whether to continue execution (e.g. the `if` or `while` statement) must be evaluated before execution can continue. Usually this involves waiting on a future that blocks execution, causing the Legion pipeline to drain.

To avoid blocking on futures for evaluating these dynamic branches, the Legion programming model enables task execution to be predicated. In many cases this allows applications to execute past branches, and discover additional parallelism. We discuss predication in more detail in Section 10.1. To further increase performance,

Legion also permits mapper-directed speculation of predicated execution. By allowing tasks to execute speculatively, further work can be uncovered and performed. We discuss speculative execution in Section 10.2.

An important added benefit of supporting speculation is that the same machinery used for recovering from mis-speculation can be re-purposed to handle recovery from faults. The dynamic dependence graph computed as part of dependence analysis (see Chapter 4) permits the Legion runtime to precisely scope the effects of both faults and mis-speculation. We discuss how this symmetry along with an additional task pipeline stage can be used to seamlessly provide resilience support for Legion in Section 10.3.

10.1 Predication

To avoid waiting on future values to evaluate branches, the Legion programming model allows applications to launch predicated sub-tasks. Predicates can either be assigned based on a computed boolean value, or created directly from a boolean future. Predicates can also be composed to create additional predicates using standard boolean operators (e.g. `and`, `or`, and `not`). By default all tasks in Legion are launched with the constant predicate representing `true`. Applications are free to change the predicate to any arbitrary predicate.

When a predicated task is launched, it goes into the standard dependence queue along with all other tasks launched within the same context. Dependence analysis is performed in sequential program order as described in Chapter 4. Predicated tasks register dependences on other operations, just like non-predicated tasks. Similarly, other operations can record dependences on predicated tasks. The difference for predicated operations occurs once all of its mapping dependences have been satisfied. A predicated task is not immediately placed into the ready queue for a mapper until its predicate value has been resolved (unless the mapper decides to speculate on the predicate which we discuss in Section 10.2). Once the predicate value is resolved, the execution of the task can continue. In the case that the predicate is evaluated to true, the task is placed in the ready queue for its target mapper and

proceeds like a normal task. In the case when the predicate evaluates to false, the task immediately progresses through the rest of the task stages to the complete stage without executing (in the process of completing the mapping stage it also triggers all its mapping dependences). While predicated execution may be slightly limited in allowing predicated tasks to execute before the predicate resolves, the ability to use predicates to evaluate past dynamic branches dependent on future values allows the Legion runtime to continue discovering additional parallelism, thus resulting in higher performance.

The Legion programming model currently only allows some operations to be predicated. Specifically, only sub-tasks, explicit copies, acquire, and release operations are permitted to be predicated. For operations such as inline mappings predication does not make sense because the launching task will need to wait for the physical region to be ready anyway, so waiting on the future predicate introduces no additional latency beyond what would be observed anyway. For operations that are permitted to be predicated and that return future values, the runtime requires that applications provide default values for futures in case the operation is predicated false. Without a default future value, futures could be mistakenly used when they would never be completed. This would be especially bad in cases where an empty future was used to create a predicate. To provide a default future value, applications can either give a concrete value, or another future of the same type. By providing a default value for futures in the case of false predication, the programming model maintains the invariant that all futures will eventually be completed with a specific value.

10.2 Speculation

While predication allows Legion to discover additional independent non-predicated tasks to execute while waiting for predicates to resolve, it does not allow Legion to execute predicated tasks until their predicates have resolved. For truly high performance, speculative execution is necessary to begin executing tasks before their predicates resolve. This is especially important for applications such as iterative solvers in which nearly all tasks are predicated on the previous loop iteration having

not converged, resulting in large chains of dependently predicated tasks. Especially in this scenario, predicting the value of predicates is straightforward: the solver never converges until the last iteration so always predict non-converged. By speculating on predicate values, Legion can begin mapping and executing tasks and other operations before predicates resolve.

In this section we cover the machinery necessary for handling both speculation as well as mis-speculation. Section 10.2.1 covers how mappers can decide to speculate on a predicate and how the runtime can detect mis-speculation. In order to recover from mis-speculation, the runtime maintains a scheme for versioning logical regions that we cover in Section 10.2.2. In Section 10.2.3, we detail how the Legion runtime recovers from mis-speculation once it has been detected.

10.2.1 Speculative Execution

The decision about whether to speculatively execute a task is purely a performance decision and has no impact on application correctness. Therefore all decisions about when and how to speculate are resolved by the mapper object specified for an operation. For each predicated operation, if its predicate has yet to be resolved when all its mapping dependences are satisfied, the runtime queries the mapper to determine if it would like to speculatively execute the task. There are three valid responses: the mapper can choose not to speculate, it can speculate `true`, or it can speculate `false`. In the case where speculation is not requested, then the operation executes like a standard predicated operation as described in Section 10.1. If the mapper predicates true, the task is placed onto the ready queue and proceeds to execute as if the predicate had resolved to true. If the mapper elects to speculate false, then the default false value for the future is set and the task does not execute. If the speculative value chosen by the mapper proves to be prescient, then tasks will proceed through to the completion stage as though they were not even predicated. We discuss how the runtime recovers from mis-speculated predicate values in Section 10.2.3.

One of the important invariants maintained by the runtime is that no operation is allowed to proceed to the completion stage in the pipeline until all possible speculative

inputs have been resolved. It is important to note that this applies even to tasks and other operations that are not speculative. For example, if a task B has a data dependence on a logical region that is produced by a task A, then if A is speculatively executed B cannot be considered to be non-speculative until A resolves its speculation. To track this invariant we add an additional kind of dependence between operations in the dynamic dependence graph for tracking *speculative dependences* and incorporate a *resolve speculation* stage into the operation pipeline before the completion stage.

Speculative dependences are registered as part of the logical dependence analysis described in Chapter 4. For every mapping dependence registered on an operation that has not resolved its speculative state, an additional speculative dependence is registered between the operations. Speculative operations can also be registered another way. If a task is predicated, then a task cannot resolve its speculative state until all the future values used to compute the predicate are also non-speculative, thereby entailing that the operations that compute those futures are also non-speculative. It is important to note that this definition of non-speculative is inductive allowing Legion to automatically handle cases of nested speculative execution (similar to modern hardware out-of-order processors). An operation is considered no longer speculative once all of its incoming speculative dependences are resolved and it has resolved any speculation that it performed. Having resolved its speculative state the operation can then notify all its outgoing speculative dependences and progress to the completion stage of the operation pipeline.

An important detail regarding the resolve speculation stage is that it permits future values to be propagated for computing predicates, but it does not permit futures to actually be completed and made visible to the enclosing parent task. There is an important reason for this. First, we want speculative execution to enable additional speculative execution. For this reason we allow future values to propagate through a predicate chain to aid in this process. While this may lead to additional speculative execution, it is the kind that Legion is capable of recovering from internally without requiring a restart of the enclosing parent task (see Section 10.2.3 for more details). We do not permit speculative future values to be completed and made visible to the enclosing parent task. If we did, then these speculative values escape the ability

of the runtime to recover from mis-speculation as the future value may be used for performing arbitrary computation in the enclosing parent task. Any mis-speculation could then ultimately lead to re-execution of the parent task, which is a poor recovery model from a performance standpoint. Therefore we allow futures from tasks still in the resolve speculation stage to propagate internally, but not to be visible to the enclosing parent task until speculation is resolved.

10.2.2 Data Versioning

As part of speculative execution, the Legion runtime needs a versioning mechanism for tracking logical region data in order to be able to recover from mis-speculation. There are two different kinds of versioned data that must be tracked. First, versions of physical instances must be explicitly tracked from each logical region perspective. Second, the runtime must track different versions of the physical state meta-data in order to be able to re-map mis-speculated executions. We now describe how each of these two versioning systems work in detail as well as how the tasks record the necessary versioning information for when they need to be re-issued (see Section 10.2.3).

The first kind of versioning done by the runtime is for logical regions and physical instances. Versioning of logical regions and physical instances is necessary to know when speculative execution has potentially corrupted the data in a physical instance, and therefore it cannot be re-used when an operation is re-mapped. To detect this case, we maintain a version number with each logical region and physical instance. Version numbers are assigned to logical regions as part of the logical dependence analysis using the following scheme. Every write applied to a logical region increments the version number of the region by two. The write also updates the version numbers of any ancestor logical regions. If an ancestor logical region has an even version number, its version number is incremented by one, otherwise no operation is taken if the version number is odd. Odd numbered versions therefore represent partial writes where at least one child logical region has dirty data. When a close operation is performed, the target logical region version number is incremented by two if it is even and one if it is odd. When a logical region is opened, it checks its version number

against its parent version number, and the larger of the two is taken as the updated version number. This scheme is used independently on each field to avoid ambiguity.

When tasks perform their dynamic dependence analysis for each of their logical regions, they record the version numbers for all required fields along the path from where their parent task had privileges to the logical region on which they are requesting privileges. When performing the traversal of a physical instance to record writers, the same versioning scheme is used to track the version number for each field in an instance view object. If the version number of a logical regions matches the version number of a physical instance, then we know the data contained in the physical instance is correct. However, if an operation is re-issued because of mis-speculation and the version number for a field in a physical instance is larger than the version number recorded by the task as part of the logical dependence analysis, then the physical instance can no longer be considered a valid copy of the data for that logical region.

In addition to versioning logical regions and physical instances, the runtime must also version the physical states of the logical region tree nodes. As we will discuss in Section 10.2.3, logical dependence analysis only needs to be performed once regardless of speculation, but physical dependence analysis needs to be performed each time an operation is re-issued as part of mis-speculation. To support the re-issuing of tasks, we also version the physical state associated with logical region for all fields so that the physical dependence analysis can be performed. We modify the physical state object stored for each physical context at every logical region node in region tree (and described in Chapter 5) to store two additional kinds of information. First, in addition to storing a list of physical instances with field masks to describe the valid instances, we also keep a list of all previously valid physical instances. Associated with each element in this list is a mapping from from version numbers to field masks describing which fields where valid for a physical instance under which versions. The second additional data structure that we store for versioning is a list of open children. For each open child, we maintain a map from version numbers to field masks describing which fields were open for each child under each version number. Version numbers for physical state objects follow the same versioning protocol as for logical regions and physical instances. Even though read operations mutate physical state objects, they

simply create additional valid physical instances and do not fundamentally change the nature of the physical state, permitting the same versioning scheme to be used. The recorded information regarding the version numbers of logical regions as part of the dynamic dependence analysis in conjunction with the additional stored information on physical states can automatically be used to regenerate the physical state for any logical region node when an operation is re-issued due to mis-speculation.

10.2.3 Mis-speculation Recovery

When mis-speculation is detected, it is the responsibility of the Legion runtime to automatically recover from the mis-speculation. There are several parts to recovering from mis-speculation and we cover them in detail sequentially. First, once a mis-speculated operation has been identified, the runtime must compute the set of dependent operations that are impacted by the mis-speculation. Fortunately, based on the dynamic dependence graph computed by the logical dependence analysis, this process is straight-forward. The runtime starts at the node representing the mis-speculated operation and walks forward through the graph following all the unsatisfied speculation edges, recording all nodes encountered along the way. We refer to nodes impacted by the mis-speculation as *poisoned* nodes. As part of the traversal, the runtime resets all mapping dependence edges to poisoned nodes that were impacted by the mis-speculation.

In the process of traversing the dynamic dependence graph, the runtime sorts poisoned nodes into two different classes: nodes whose operations have already been triggered to map and those that have yet to have all their mapping dependences satisfied. Operations that are yet to progress to the mapping stage are easy to recover by simply resetting the satisfied mapping dependences to the unsatisfied state. Alternatively, if an operation has begun to map, already mapped, already been issued to the low-level runtime, or potentially already run, then recovery is more challenging. Importantly, we know that it is impossible for any of the poisoned operations to have progressed past the resolve speculation stage of the operation pipeline because of the transitive nature of speculation dependences discussed in Section 10.2.1.

Recovering operations that have progressed beyond mapping requires a multi-stage shoot-down process. First, we mark the operation as having been poisoned. This prevents it from taking any additional steps in the pipeline, allowing us to begin the recovery process. Next, we begin recovering the effects of each of the stages in the pipeline in reverse order. If an operation has already been issued to the low-level runtime then we immediately poison its post-condition event. If the operation has yet to run, we also poison its precondition event. By poisoning these events, the low-level runtime receives immediate information that this subset of the event graph and all its dependences can be poisoned and elided from execution.

Progressing backwards from the mapping execution stage, if a task has already been distributed to a remote node, then a message is sent to the target node the task was sent to, instructing it to initiate the recovery process. It is possible that as part of the mapping process that the task was stolen or sent to another remote node. The runtime maintains forwarding information for all distributed tasks that it has observed until it receives notification that the operation has committed, allowing it to continue to forward these messages to the eventual destination. When a remote node receives the notification, it performs the same recovery process as the origin node of the task.

After the distribution stage has been recovered, the runtime proceeds to recover the mapping stage. All valid references held by the task operation are removed, permitting the physical instances to be potentially collected. Note that we do not need to roll back the physical state of the region tree because it is captured in the versioned state of the physical instances at each point in time. Poisoned operations do remove any references that they hold to different versions of the physical states of region tree nodes that they held, potentially allowing these states to be recovered since they are no longer valid.

Once all of the poisoned operations have been recovered, it is now safe to begin the re-execution of operations. Unfortunately, it is not safe to immediately begin re-execution of the task that was initially mis-speculated. In the process of speculatively executing tasks, we may have corrupted physical instances that contained the data necessary for performing the speculation initially. In order to determine if we still

have access to a valid instance for each logical region of the initial mis-speculated task, the runtime traverses the physical states of the region tree nodes at the versions required by the initially mis-speculated task. It then checks all the valid instance views to see if any of their version numbers have been advanced. If they have, then they can no longer be considered valid copies of the data and are invalidated. If they still have the same version number then they can be re-used. If at least one valid instance can be found for each logical region, then we still have valid copies of all the data and can begin re-execution.

If a valid physical instance for any logical region required by the mis-speculated task cannot be found, then it is the responsibility of the runtime to re-issue the previous tasks that generated the valid instance view data. The runtime walks backwards through the dynamic dependence graph for each region requirement that does not have a valid instance. At each node, it performs the same test as on the mis-speculated operation to see if valid physical instances exist to re-execute the operation. If valid physical instances of the right version exist, then the operation is re-issued and the traversal terminates. Otherwise, the runtime recursively invokes this process to re-issue all necessary operations for re-computing data.

There are two conditions under which the recovery process will fail. First, if the recovery process reaches the beginning of the dynamic dependence graph for a context and the initial instances used by the parent task have been corrupted by mis-speculation, the runtime stops recovery within the current context. It then elevates the recovery process to restarting the parent task with its enclosing context. We discuss the benefits of hierarchical recovery in further detail in Section 10.3.3, but the same process is applied to the parent task in order to restart it. The second condition that can cause the recovery process to fail within a context and be elevated to the parent task is if any of the tasks that need to be re-issued are non-idempotent (see Section 2.5.8). Non-idempotent tasks have side effects that the runtime cannot reason about, requiring the the handling be elevated to the parent task (possibly recursively triggering a restart of the entire program in some cases). In practice, most Legion tasks are idempotent and only have side effects on logical regions about which the runtime can reason.

While recovery may seem as if it can be very expensive, it is important to note that the performance of recovery is directly influenced by mapping decisions. Mapper calls are informed about when they are mapping speculative tasks. Mappers then have a natural space-time trade-off to make when mapping speculative tasks. Mappers can choose to re-use existing physical instances when mapping a speculative task that reduces memory usage, but at the potential cost of more-expensive (and deeper) recovery operations when mis-speculation occurs. Alternatively, mappers can prevent the corruption of physical instances for speculative execution by forcing the runtime to either create new instances or re-use instances based on the same speculative predicates. This will result in faster recovery, but at the cost of additional memory usage. Since this is a standard performance trade-off, we ensure that the mapper can directly make this decision without interference from the runtime.

After the runtime determines that recovery can be done for a mis-speculated task, then all poisoned tasks, as well as any additional tasks needed to re-generate physical instances for the mis-speculated task, are re-issued into the operation pipeline by the runtime. These operations are re-issued past the dependence analysis stage because re-computation of the dynamic dependence graph is unnecessary. Since no future values escape to the parent task until they are non-speculative, we know that the shape of the dynamic dependence graph is invariant under speculation. Therefore all operations have the same dependences as they did before. The only difference is that the predicate values may impact how the dynamic dependence graph is executed. Once all the tasks have been re-issued, then execution proceeds as if the mis-speculation had not occurred.

10.3 Resilience

One of the benefits of supporting speculation in Legion is that it enables a nearly free implementation of hierarchical resilience in Legion. The same machinery described in Section 10.2 for recovering from speculation can be used for recovering from both hard and soft faults. While there are a few additional features required to support fault tolerance, they have a minimal impact on performance. As with many fault

tolerance mechanisms, additional memory will be required.

We assume in this section that Legion is purely responsible for recovering from faults and not for detecting them. We will describe how Legion can handle many different kinds of faults reported at any of the hardware, operating system, or application. For the purposes of this discussion we detail how Legion handles faults that impact application data, and do not describe any mechanisms for handling faults that impact the Legion runtime meta-data or execution. Handling faults within the runtime is left for future work.

10.3.1 Commit Pipeline Stage

Recall from Section 3.2.2 that the final stage of the Legion pipeline is the *commit stage*. In a version of Legion that is not resilient, the commit stage is a no-op, allowing all operations to proceed immediately from completion to being retired. The commit stage is necessary for resilience as it prevents operations in the pipeline from retiring when it is possible that they may still need to be re-issued due to a fault. In many ways, this stage is similar to the resolve speculation stage from Section 10.2, except that the commit stage is performed after the completion stage. Placing the commit stage after the completion stage is important because it allows dependent operations to begin running (the completion event has triggered) and completes any future values making them available to the enclosing parent task. This decision reflects an important design concept: Legion prevents speculation from polluting a parent task's execution context, but provides no such guarantee for faults. The motivation behind this design decision is that speculation is likely to be common in Legion applications, while faults should be significantly less frequent.

After an operation has finished the completion stage of the pipeline, it progresses to the commit stage. There are three conditions under which an operation can finish the commit stage and retire. The first way that an operation can be committed is if the mapper determines that all the output region requirements have had physical instances placed in memories of a sufficient degree of *hardness*. The low-level runtime assigns all memories a hardness factor that indicates their level of resilience. For

example, a memory backed by a RAID array has a higher degree of hardness then a normal disk which has a higher degree of hardness than flash memory which has a higher degree of hardness than DRAM, etc. After each operation with output region requirements have been mapped, the runtime invokes the `post_map_operation` mapper call that asks the mapper to both set a minimum default hardness for each region requirement, and also permits the mapper to request additional copies of logical regions be made in hardened memories after the operations has completed.

In additional to being placed in hardened memories, the runtime also needs to validate that the data in the physical instances is valid. To validate the data, mappers can specify a validation task to be run using the physical instance that must return a boolean indicating whether the data is validated. Alternatively, Legion tasks are responsible for checking their own input logical regions for inconsistencies. If a consuming task of a logical region (e.g. one with at least read-only privileges) completes without reporting an error (see Section 10.3.2 for details on reporting errors), then the runtime can conclude that the data generated by the task has been validated.

If all of the region requirements for an operation have instances in a sufficiently hardened memory and they have all been validated, then the runtime knows that it will never need to re-execute the operation (because valid physical instances exist for all the data), and therefore it can retire the operation. If an instance is corrupted in a hardened memory and the operation has already been retired, then the runtime will automatically invoke the hierarchical recovery mechanism described in Section 10.3.3.

The second way that an operation can be retired is if all of the operations that depend on it have been retired. In Section 4.4, we described how commit edges were added from all operations to each of the predecessors on which they held mapping dependences (i.e. commit edges are reverse mapping dependences). When an operation commits, it triggers all of its commit edges, indicating to the predecessors of the operation that it will never need to be re-issued in order to recover from a fault. If at any point all commit edges for an operation have been satisfied, then the runtime knows that an operation will never need to re-issue and allows it to commit. Intuitively, commit edges create slices of the dynamic dependence graph that will never need to be re-issued and can therefore be committed.

The third and final way that an operation can be committed is that the mapper can explicitly direct the runtime to commit an operation. For each invocation of the `select_tasks_to_schedule` mapper call, the runtime also makes an invocation of the `select_tasks_to_commit` mapper call that asks the mapper to identify any operations to commit early. The need for early committing of tasks by the mapper is necessary to manage a trade-off between recovery time from faults with the size of the dynamic dependence graph for a context. If the mapper makes no attempt to harden the results of operations, then the dynamic dependence graph maintained by the runtime for recovery purposes can grow arbitrarily large. Since the decision about how much of the dynamic dependence graph to maintain is purely a performance decision, we directly expose the decision using the `select_tasks_to_commit` mapper call. The mapper can choose to maintain a large dynamic dependence graph for shorter recovery times, or it can eagerly prune the graph, possibly requiring a restart of the enclosing parent task if a task needs to be re-issued that has already been committed.

10.3.2 Fault Detection and Reporting

At the beginning of this section we scoped the problem of resilience in Legion to only recovering from errors. We now describe how errors are reported to Legion and how recovery is performed. There are two kinds of faults that can be reported to the runtime: task faults and physical instance faults. Either of these faults can be the result of soft or hard faults in hardware or software. We now discuss the recovery process from each of these kinds of faults in turn.

In the case of task failure, a hard or soft fault causes a task to fail. This might occur because of a hard fault where the hardware processor running the task fails, or a soft fault such as a bit flip that changes the execution of the task and results in a failure. Regardless of the fault, the runtime is notified of the failure (either by the hardware or the operating system via a mechanism in the low-level runtime). When a task faults, the same recovery mechanism described in Section 10.2, for recovering from mis-speculated tasks is employed. This process prunes out any tasks impacted by

the fault and re-issues any operations necessary for recomputing the starting regions for the task that faulted. If, during the recovery process, an operation that needs to be re-issued has already been committed, then hierarchical recovery is invoked (see Section 10.3.3).

The second kind of fault handled by the Legion runtime is physical instance faults. This kind of fault is a corruption of the data stored in a physical instance by either a soft or a hard fault, possibly occurring while no tasks are accessing the data in the physical instance. Physical instance faults can be reported from any level of the software stack. If the instance resides in a DRAM memory with error detection codes, then the hardware might report a fault. If the instance is in a software managed memory such as a disk, then the operations system might trigger a fault during a consistency check (e.g. RAID). Finally, the application can report invalid input data for a physical instance using a runtime call invoked during task execution[1]. Regardless of the fault reporting mechanism, the runtime examines the version number of the physical instance. Unless the reporting mechanism explicitly specifies the exact range of invalidated data, the runtime conservatively assumes that the entire physical instances has been corrupted. If a range for invalid data in the physical instances is specified, the runtime can more precisely scope the consequences of the corrupted data in terms of logical region(s) of the physical instance. The runtime then determines all the consuming operations that have mapped the current version number of the physical instance for their execution. (Note that the current version number is different from the most-recent version number. The most recent version number reflects the farthest point of the mapping process.) All operations using the current version number are determined to have faulted and the recovery mechanism from Section 10.2 is again employed. If one of the faulted operations has already been committed (i.e. because the mapper committed it early), then it cannot be recovered and the hierarchical recovery mechanism is invoked.

[1]This final fault reporting mechanism is also used for inferring the valid property discussed in Section 10.3.1.

10.3.3 Hierarchical Recovery

One of the important properties of the recovery mechanism in Legion is that it is hierarchical. If at any point a task can no longer be recovered, the parent task faults and the recovery mechanism is invoked on the parent task within the grandparent task context. This can be applied recursively whenever necessary to ensure that recovery from faults is possible. In the worst case scenario, the entire program will be restarted. While the Legion approach does not guarantee forward progress, it does guarantee sound execution of Legion programs that will continue recovering from faults as they occur.

Another important benefit of hierarchical recovery is that it operates in a distributed fashion. Unlike global checkpoint and recovery mechanisms that require all nodes to checkpoint the majority of data simultaneously before continuing, hierarchical recovery allows checkpointing to be done independently on individual tasks without any synchronization. Furthermore, the decision about which data to checkpoint is completely under control by the mapper objects that specify which instances have checkpoints made, and to which memories. By taking a hierarchical approach and making all performance decisions related to resilience mapper-controlled, the Legion approach to resiliency has the potential to be significantly more scalable.

Chapter 11

A Full-Scale Production Application: S3D

Instead of evaluating Legion on many small benchmark applications, we decided to evaluate Legion on a single full-scale production application. The reason for this decision is two-fold. First, evaluating Legion features in isolation is difficult because many of them only make sense within the context of a Legion implementation. Testing our Legion implementation in an end-to-end fashion at scale demonstrates that our design decisions lead to an efficient implementation and that all components of Legion operate in concert to achieve high performance.

Our second reason for performing an end-to-end evaluation of Legion at scale is that evaluating benchmarks at smaller scales often fails to translate to high performance for applications run over thousands of nodes on real machines. At such scales, many features of a system are tested and bottlenecks are discovered that are not obvious at smaller scales. Due to this phenomenon, we decided that a better evaluation of the capability of Legion would be a comparison to an existing application written using current tools that has been developed and optimized by experts. By performing a comparison at full production scale, we would truly test the advantages of the Legion programming model.

For our comparison application we selected S3D [21]. S3D is a combustion simulation used by researchers from the United States Department of Energy to study

the chemical and physical properties of fuel mixtures for internal combustion engines. S3D simulates the turbulent processes involved in combustion by performing a direct numerical simulation (DNS) of the Navier-Stokes equations for fluid flow. The simulation for S3D operates on regular Cartesian grid of points with many values being stored on each point. The application iterates through time using a six stage, fourth order Runge-Kutta (RK) method. The RK method requires that the *right-hand side function* (RHSF) be evaluated six times per time step, requiring considerable computational resources. There are many (often independent) phases involved in computing the RHSF function for all points, resulting in considerable task- and data-level parallelism.

While the framework of S3D is general, S3D is parametrized by the kind of *chemical mechanism* being simulated. A chemical mechanism is constituted of a collection of species (molecules) and reactions that govern the chemistry of the underlying simulation. For actual combustion, there are usually on the order of thousands of species and tens of thousands of reactions. Performing a computational simulation of a mechanism with perfect fidelity is computationally intractable. Instead, *reduced* mechanisms are developed that represent tractable approaches to simulating combustion. Much of the original science done using S3D involved simulating the `H2` mechanism consisting of 9 chemical species and 15 reactions. More recently S3D is being used to simulate larger chemical mechanisms. Figure 11.1 shows details of the chemical mechanisms used for our experiments involving Legion. Each chemical mechanism models a specific number of species and reactions. In many cases, species with similar behavior are aliased resulting in a smaller number of *unique* species. In some cases, certain groups of species need to appear to be in a quasi-steady state, requiring a quasi-steady state approximation (QSSA). Additional computation will be required for QSSA species. Stiff species are those whose rates of change are often volatile and are therefore subject to additional damping computations.

S3D has been developed over the course of thirty years by a number of scientists and engineers. There are currently two primary versions of S3D. The first version of S3D is the baseline version developed primarily by the scientists and is used for active research. This version consists of approximately 200K lines of Fortran code and uses

Mechanism	Reactions	Species	Unique	QSSA	Stiff
DME	175	39	30	9	22
Heptane	283	68	52	16	27
PRF	861	171	116	55	93

Figure 11.1: Summary of Chemical Mechanisms Used in Legion S3D Experiments

MPI [45] for performing inter-node communication when necessary. The choice of Fortran as the base language for S3D has allowed scientists to take advantage of the many advances made by Fortran compilers (developed by both Cray and Intel) for generating both multi-threaded and vectorized code. Many of the larger computational loops are automatically partitioned across threads and auto-vectorized leading to good performance on homogeneous CPU-only clusters. While this version of S3D was sufficient for performing science on traditional clusters, it is challenging to scale to modern machines and is currently unable to take advantage of the heterogeneous processors (e.g. GPUs) available on most modern supercomputers.

The second version of S3D developed more recently combines MPI with OpenACC [3] to target modern heterogeneous supercomputers [38]. This version of S3D was ported from the baseline version by a team consisting of both scientists as well as engineers from Cray and NVIDIA. By leveraging both CPUs and GPUs, this version of S3D was made considerably faster than the baseline version of S3D. Due to its higher performance we chose this version of S3D to serve as our baseline for performance comparisons with Legion.

In order to avoid porting all 200K lines of S3D into Legion, we decided instead to port the RHSF function to Legion and leave the remaining start-up/tear-down code as well as the RK loop in the original Fortran. The RHSF function represents between 95-97% of the execution for a time step in S3D (depending on chemical mechanism and hardware), thereby representing the bulk of S3D's computational workload. Ultimately we ported approximately 100K lines of Fortran into Legion C++ code. By only porting a portion of S3D into Legion, we also showcase Legion's ability to act as a library capable of inter-operating with existing MPI code. By developing a path for inter-operability we also demonstrate an evolutionary path for

applications to be ported into Legion that does not require all code to be moved into Legion.

The rest of this chapter describes our implementation of S3D in Legion. We begin by describing the machine-independent implementation of the S3D in Legion based on logical regions and tasks in Section 11.1. In this section we also describe how Legion is able to inter-operate with MPI so only a portion of S3D is required to be ported into Legion. We cover the details of how the Legion mapping interface allowed us to experiment with different mapping approaches in Section 11.2. Finally, we illustrate the performance benefits of Legion when compared to MPI and OpenACC in Section 11.3.

11.1 S3D Implementation

Our Legion implementation of the RHSF function in S3D operates differently from a stand-alone Legion application. Since most of the S3D code for starting and analyzing an S3D run is in MPI-Fortran, we use MPI to start-up a separate process on each node (in contrast to having GASNet start-up a process on each node for the normal Legion initialization procedure). Our Legion implementation of S3D is implemented as a library and we add a Fortran call into our library that triggers the initialization of the necessary data structures on each node. Consequently, Legion is initialized within the same process as MPI. While this is less than an ideal situation (we would have preferred cooperating processes with isolation), alternative approaches are limited by the poor quality of software tools (e.g. mpirun and aprun) for supporting cooperating jobs with different binaries.

11.1.1 Interoperating with Fortran

The call to initialize the Legion runtime triggers the execution of the top-level task for our S3D Legion application. Unlike a standard Legion application, our top-level task for S3D doesn't immediately start execution of S3D. Instead it begins by launching a sub-task on every processor in the Legion system that synchronizes with the MPI

process on the target node. These sub-tasks synchronize with the MPI thread in each process and determine the mapping from Legion processors (and therefore memory and nodes) to MPI processes. This mapping is necessary for ensuring proper inter-operation with MPI. Further sub-tasks are then launched to distribute the mapping result to each of the different mappers within the system.

After computing and distributing the mapping of Legion processors to MPI ranks, the top-level task prepares to launch a separate sub-task for each of the different MPI ranks. We refer to these different sub-tasks as *distributed* sub-tasks. Whereas normal Legion tasks actively execute an application, these tasks instead are long-running tasks that are responsible for synchronizing with the MPI ranks of S3D and only performing the RHSF functions for a time step when directed to do so by the MPI code. It is often the case that these distributed sub-tasks need to communicate with each other for exchanging information such as ghost cell data. To support this, the top-level task creates explicit ghost cell regions (that we describe in more detail in Section 11.1.2) and uses simultaneous coherence in conjunction with a must epoch index space task launch to ensure that all the task are executed at the same time (see Section 9.3 for details on must epoch task launches).

Each distributed sub-task begins execution by creating logical regions for storing the necessary simulation data. We cover the the creation of these regions in further detail in Section 11.1.3. Once these regions are created, each of the distributed sub-tasks launches an `AwaitMPI` sub-task that is responsible for synchronizing with the MPI code and copying in data from Fortran arrays into logical regions. To perform the synchronization, each distributed sub-task creates a *handshake* object that mediates coordination between MPI and Legion. For each time step of an S3D simulation there are two handshakes between Legion and MPI. In the first handshake, MPI signals to the `AwaitMPI` task that it is safe to begin executing a time step. As part of this handshake the `AwaitMPI` task also receives pointers to the Fortran array in the local rank containing data that needs to be copied over to logical regions. The `AwaitMPI` sub-task requests privileges for writing into the target logical regions, allowing the distributed sub-tasks to run ahead launching additional sub-tasks that will naturally wait based on region dependences for the `AwaitMPI` task to finish.

The second handshake occurs after a time step has completed. The MPI rank will block waiting on the second handshake from the Legion side of the application. At the end of a time step, each distributed sub-task launches a `HandoffToMPI` task that requests read-only privileges the necessary logical regions containing state. Naturally, Legion dependence analysis ensures these tasks do not run until all data dependences have been satisfied. When run, the `HandoffToMPI` tasks copy data from logical regions back to Fortran arrays and then completes the handshake by signaling to the Fortran side, that it is safe to continue execution.

The handshaking model is an effective one that allows for part of an application to be ported to Legion without requiring that all code be transitioned. The down-side to this approach is that either all of the computational resources of the machine are being used for performing Legion work or MPI work, but never both at the same time. For some future applications it may be beneficial to provide a means for taking advantage of parallelism between both Legion and MPI computations simultaneously.

While the handshaking model is currently performed using an approach that is specific to S3D, we believe that it can be easily generalized to additional existing MPI codes. Further support by the Legion runtime for automatically computing the mapping between processors and MPI ranks would greatly aid in porting applications to Legion. In the future, we plan to provide direct support for MPI and Legion interoperability based on experiences with S3D.

11.1.2 Explicit Ghost Regions

In order to provide a communication substrate for exchanging ghost cell data between distributed tasks, our Legion implementation uses explicit ghost cell regions mapped with simultaneous coherence. S3D has a straight-forward nearest neighbors communication pattern between each rank. Each neighbor must communicate ghost cell information with its six adjacent neighbors (two in each dimension). Based on this communication pattern, our Legion implementation creates six ghost cell logical regions for each of the distributed tasks.

The ghost cell logical regions are much smaller than the logical regions that will

be created to store the state of the simulation for each distributed task (see Section 11.1.4). For the general S3D simulation, most ghost cell regions need only be four cells deep (for fourth-order stencils) on each face. Each explicit ghost cell region only needs to allocate fields in its field space for fields that will be the subject of stencil computations. In practice this normally is on the order of three times the number of species as exist in the mechanism (there are three primary stencil phases in S3D). Since all explicit ghost cell regions are used for the same stencils, they can all share the same field space.

In addition to creating explicit ghost cell regions, the top-level task also creates phase barriers (see Section 9.2.2) for synchronizing ghost cell exchange between different distributed tasks. Two phase barriers are created for every field on each explicit ghost cell region. The first phase barrier is used to describe when a field has been written by the producer distributed task. The second phase barrier is used for communicating that the results have been finished being used by the consumer task and it is safe to write the next version of the ghost cells. Both phase barriers are registered as having a single producer necessary to trigger the barrier. Unlike MPI barriers, Legion phase barriers are first class objects that are stored in their own logical region. Each distribute task also receives privileges on the phase barrier regions necessary for synchronization of its sub-task operations.

When the distribute tasks are launched, projection region requirements are used for computing which explicit ghost cell regions are assigned to each point task. Each distributed task receives twelve ghost cell regions: the six that it owns as well as the six from its corresponding neighbors. Simultaneous coherence is used to remove any interference between the different sub-tasks. A must epoch launch is performed to guarantee that all of the tasks are executing at the same time, thereby ensuring that phase barriers can be used for synchronization between the distributed tasks.

11.1.3 RHSF Logical Regions

When the distribute tasks initially begin running they first create logical regions for storing the state necessary for running S3D. Each distribute task creates four local

logical regions for storing state: a region r_q for storing the input state, a region r_{rhs} for storing the output state, a region r_{state} for storing state that it is iteratively preserved between time steps, and a region r_{temp} that stores temporary data used within each time step. Each of these regions is sized by a three dimensional index space of the same size as the Fortran code (usually with 48^3, 64^3, or 96^3 elements). While each of these logical regions persists throughout the duration of the distributed tasks, they are commonly represented by many different logical sub-regions.

Each of the different state logical regions for a distributed task have different field spaces. The r_q and r_{rhs} regions have the smallest field spaces containing $N + 6$ fields where N is the number of species in the chemical mechanism being simulated. The r_{state} and r_{temp} logical regions contain between hundreds and thousands of fields for storing all the temporary values created as part of the execution of an S3D time step. For example, the r_{temp} logical region requires 630, 1048, and 2264 fields for the DME, Heptane, and PRF mechanisms respectively. The number of fields required for these logical regions motivated many of the optimizations discussed in Chapters 4 and 5 for optimizing the representation of fields with field masks.

Each of the distributed tasks also partition each of the four state logical regions in two different ways. First, a disjoint partition is made of each logical region for describing how to partition cell data for assignment to different CPUs within the local node. The second partition is also a disjoint partition that describes how to break up data for each of the different GPUs available within a node. Both partitioning schemes are performed using tunable variables (see Section 2.3.1) for selecting the number of sub-regions in each partition. Note that because each of the four state logical regions use the same index space, the two partitions only need to be computed once and the necessary sub-regions already exist for each of the different state logical regions.

11.1.4 RHSF Tasks

After each of the distribute tasks are initialized they begin launching sub-tasks for performing time steps. Each time step begins with the launch of an `AwaitMPI` task that

will block waiting for synchronization with the MPI side of the computation. Each `AwaitMPI` task requests privileges on the r_q logical region and its fields, ensuring that other tasks that depend on this data cannot begin running until data is copied from the Fortran arrays into the r_q logical region by the `AwaitMPI` task. The distributed tasks continue executing and in the process launch hundreds of sub-tasks for each time step. Each sub-task names different logical regions and fields with various privileges that it requires to execute. Based on region usage, field usage, and privileges the Legion runtime automatically infers considerable task and data level parallelism. At the end of each time step, the distributed task also issues a `HandoffToMPI` task to synchronize again and pass both data and control back to the MPI Fortran part of S3D. The natural back pressure applied by the Legion runtime by controlling the maximum number of outstanding tasks in a context and the back pressure of the mapping process all limit how far the distributed task can run in advance of actual application execution.

When stencil computations are performed, the distributed tasks use explicit region-to-region copy operations to perform the data movement between the state logical regions and instances of the explicit ghost logical regions on remote nodes. Phase barrier triggering is directly tied to the result of the copy, ensuring that it too is performed asynchronously. When using the explicit ghost cell regions to perform stencil computations, acquire and release operations are employed to remove any restrictions on mapping decisions associated with the use of simultaneous coherence (see Section 9.1.3). The second phase barrier for marking that it is safe to issue the next copy operation on each field is also triggered as part of the completion of the release operation for a given field. The computation of each stencil therefore directly mirrors the simple stencil example shown in Figure 9.1 from Section 9.3.1.

Tasks that are independent of any specific chemical mechanism are represented as separate classes in C++. Each task class represents a different kind of task and static member functions are used to provide the implementation of different variants of the code. The static nature of the tasks reflects that tasks in Legion can be written in an imperative language, but are primarily functional with side-effects on logical regions. Task classes also have other static member functions explicitly used for registering

task variants as well as for constructing region requirements and launching sub-tasks. This approach results in very modular code where all the components of a Legion task including the registration, launching, and implementation are all co-located. When contrasted with the original MPI-Fortran code, Legion code is significantly easier to both read and maintain.

While many of the tasks in S3D are independent of the chemical mechanism or easily parametrized by the number of fields, some task implementations are dependent upon the chemical mechanism. For these tasks we developed a simple DSL compiler called Singe that takes as input combustion specific files (a CHEMKIN reaction file and two files containing tables of thermodynamic and transport coefficients for different species) and generates optimized implementations of the necessary kernels for both CPUs and GPUs. The Singe compiler performs many interesting optimizations that are beyond the scope of this thesis [12].

11.2 S3D Mapping

In Chapter 1, we claimed that decoupling an application's specification from how it is mapped to a target architecture is important for both performance and portability. S3D is an excellent example of an application that showcases the need for this decoupling. The dynamic dependence graph for S3D is sufficiently complex that it would be very difficult for any human programmer to directly pick the best mapping of tasks onto CPU and GPU processors on their first or even second guess. By using the explicit mapping interface provided by Legion, we were able to explore more than 100 different mapping possibilities for each combination of chemical mechanism and target architecture in order to find the best performing versions. As we will shortly see, the best mapping for S3D depends directly upon the machine being targeted.

The first step in mapping S3D is to compute the mapping for the distributed tasks to processors on different nodes. We compute the mapping of these tasks based on the relationship between processors and MPI ranks. We send each distributed task to a processor on the node with the same MPI rank ID as the point for the distributed task in the index space task launch. We also are careful about mapping region instances

so that we satisfy all the restricted constraints on explicit ghost regions for the must epoch task launch. In general we attempt to map all of the physical instances for the explicit ghost regions into *registered memory* (pinned memory accessible directly by NIC hardware for performing one-sided asynchronous sends and receives) and fall back to *system memory* (e.g. DRAM) when space is not available. In some cases, the explicit ghost cell regions are actually mapped into memories that are not directly visible from the processor running the distributed tasks. The runtime permits this behavior because the application sets a flag indicating that the distributed task implementations will never request accessors to these regions, but instead only access them using explicit region-to-region copies.

One downside to inter-operating with MPI is that there are cases where mapping decisions can impact correctness. For example, failing to map distributed tasks to a processor on the node with the corresponding MPI rank can result in incorrect exchanges of ghost cell data. This is an unfortunate consequence of inter-operability. When an application is entirely contained within Legion, we can guarantee that mapping decisions do not impact correctness. However, when inter-operating with another programming system, mapping decisions must be made in such a way as to align with the implicit mapping decisions made in the other programming system.

As the distribute tasks begin running, they start performing time steps and launching sub-tasks for each of the different phases of an S3D time step. The mapping decisions made for these many sub-tasks strongly impact the ultimate performance of an S3D run. However, the complex nature of the dependence graph makes it very difficult to easily infer the best mapping decisions. Specifically, it is very difficult to determine which tasks should be assigned to CPUs and which should be scheduled on the GPUs. Furthermore, determining the best placement of data is not entirely obvious. GPU framebuffers are regularly much smaller than a node's system memory, requiring that applications with larger working sets actively manage which data is placed in the framebuffer. To explore these many trade-offs we relied heavily on the programmability afforded by the Legion mapping interface.

To test different mapping strategies, we developed a collection of specialized mappers, each of which deployed a different mapping strategy for assigning tasks to CPUs

and GPUs as well as assigning data to either the system memory on the CPU-side of the PCI-E bus or the framebuffer memory on the GPU-side of the PCI-E bus. In general, we found that there was one particular set of fields whose placement significantly impacted the performance of a mapping scheme. The decision about whether to place the derivatives of the diffusive flux fields on the CPU-side or GPU-side of the PCI-E express bus tended to dictate the mapping of the tasks that depend on them. There are $3N$ diffusive flux derivatives in the number of species per cell, requiring a considerable amount of memory to store regardless of the problem size. In general, we found that there was insufficient work to hide the latency of moving these fields across the PCI-E bus. Therefore, all three of the different chains of dependent tasks that require the diffusive flux derivatives were always executed on the the same side of the PCI-E bus as where the values were generated.

Ultimately, we referred to these two different mapping approaches for S3D as *mixed* and *all-GPU* mapping strategies. Figures 11.2 and 11.3 illustrate the differences for the mixed and all-GPU mapping strategies respectively. Each picture shows the performance of the utility processors, CPUs, and GPUs on a single Titan node for one RHSF invocation of the Heptane mechanism[1]. In the mixed strategy, the diffusive flux derivatives are computed on the CPU-side of the PCI-E bus, leading to the majority of the tasks that depend on those values to also be mapped to CPU processors. Only the most computationally expensive kernels (e.g. transport and chemistry kernels) are mapped to the GPU. While this leads to a relatively balanced workload between CPUs and GPUs (depending on the relative performance between the CPUs and GPUs), in general, the CPUs are always busy while the GPUs are under-loaded. The alternative approach is to map the diffusive flux derivative computations onto GPU processors, which generates the values in the framebuffer. This then moves the majority of dependent tasks down to the GPU. Since both these tasks as well as the computationally expensive tasks are all on the GPU, the GPU processors are much more heavily loaded, while the CPU processors have very few tasks to perform (hence

[1]Interestingly, in the case of the utility processors, they are actually performing the dynamic analysis for the RHSF invocation that occurs two steps later in the Runge-Kutta method. This demonstrates the ability of Legion's deferred execution model to permit the runtime to perform analysis well in advance of where the actual application is executing.

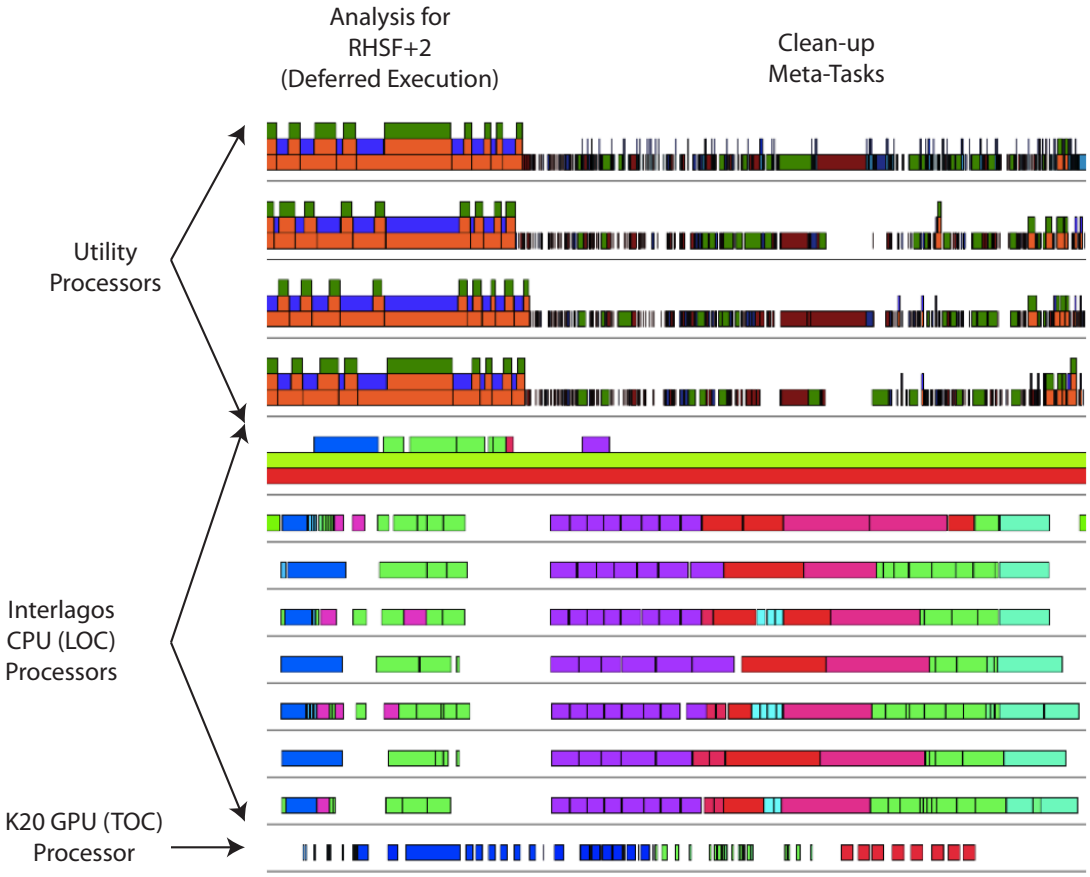


Figure 11.2: Example Mixed Mapping Strategy for S3D on a Titan Node

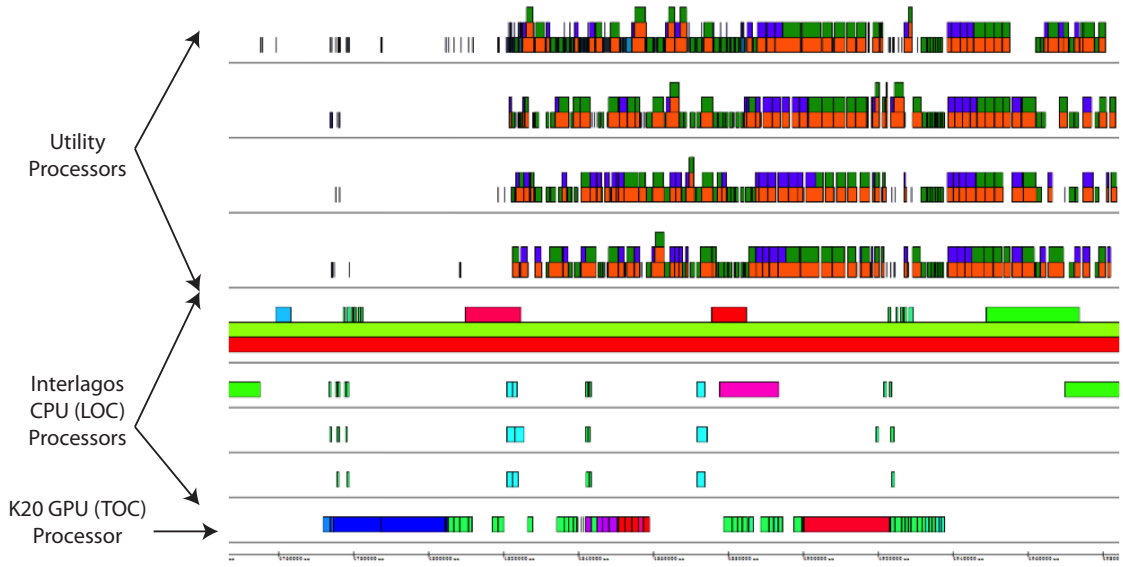


Figure 11.3: Example All-GPU Mapping Strategy for S3D on a Titan Node

the all-GPU mapping name).

Importantly, one commonly observed problem when mapping code to accelerators also occurs in S3D. In many cases, problem sizes are sometimes too large to fit in the limited amount of framebuffer memory for an accelerator[2]. For many codes, this will result in crashes due to out-of-memory errors. However, in the case of Legion, it is easy to adjust to new larger problem sizes with dynamic mapping strategies. Figure 11.4 shows an approach to mapping a 96^3 cells per node Heptane problem onto a Titan node. Only a limited number of tasks can be mapped to the GPU as there is limited space in the 6 GB framebuffer memory. Under this scenario, the field-aware nature of Legion is crucial. Legion understands precisely which fields are necessary for executing the tasks mapped to the GPU, which means only those fields are moved down to the GPU framebuffer. The mapper can therefore dynamically manage the working set of the GPU and ensure that the code can run unlike the case today where significant refactoring would be required.

The crucial insight provided by our experience mapping S3D is that we would

[2]As of the writing of this document, most accelerators have between 6 and 12 GB of either DRAM or GDDR5 memory.

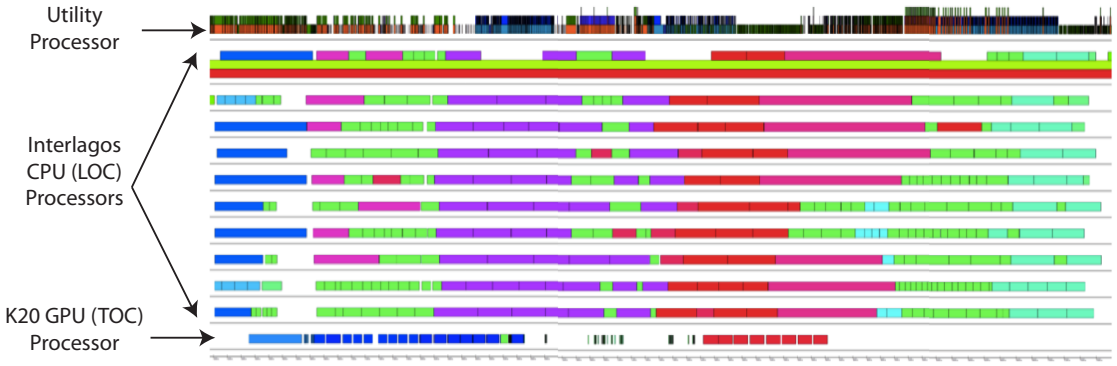


Figure 11.4: Mapping Strategy for a Large S3D Problem Size

never have discovered either of these mapping strategies without the decoupled nature of the Legion mapping interface. We tried more than a hundred different mapping implementations before settling on the two primary approaches described above. The Legion mapping interface provided the flexibility to try many different mapping strategies without having to be concerned with correctness. Furthermore, as we will show in Section 11.3, the determination of which mapping strategy performs better is architecture dependent.

11.3 Performance Results

In order to evaluate our implementation of S3D in Legion, we ran experiments on two different supercomputers: Keeneland [50] and Titan [7]. Keeneland is a 250 node cluster with an Infiniband interconnect. Each Keeneland node contains 2 CPU sockets with an 8-core Intel Sandybridge chip in each socket, 3 NVIDIA Fermi M2090 GPUs, and 32 GB of DRAM partitioned between the two sockets. Titan is the number two supercomputer in the world (as of the writing of this document) consisting of more than 18000 nodes with a Cray Gemini Interconnect. Each Titan node has one CPU socket with a 16-core AMD Interlagos chip, one NVIDIA Kepler K20 GPU, and 32 GB of DRAM partitioned across four primary Interlagos NUMA domains. There is an interesting mismatch between CPU and GPU pairings as we will see. The CPUs and GPUs on Keeneland are relatively evenly matched, while on Titan there is a difference

of several generations between the CPU cores and GPU cores. These differences will show different performance under different mapping strategies as we will observe.

For both or our target machines, we experimented with three different chemical mechanisms (DME, Heptane, and PRF) for three different problem sizes. The numerical formulation of the S3D simulation is designed for weak-scaling (keeping a fixed problem size per node and trying to maintain uniform throughput per node). For the DME and Heptane mechanisms, we experimented with problem sizes of 48^3, 64^3, and 96^3 cells per node. For the larger PRF mechanism we used smaller problem sizes of 32^3, 48^3, and 64^3 cells per node. For different problem sizes, we also experimented with each of our two mapping strategies.

To test for weak scaling we ran experiments starting with one node and scaling up by powers of two. (Our implementation of Legion S3D performs equally well for node counts that are not powers of two.) For Keeneland, we show scaling through 128 nodes, while for Titan we show scaling through 8192 nodes. As a baseline version, we use the combination OpenACC and MPI version of S3D that was independently hand optimized by engineers from NVIDIA and Cray over the course of a year.

For all experiments run on 1024 nodes or more, we found that a node re-ranking script was necessary for taking into account network topology performance effects. We use the genetic algorithm re-ranking script described in [5] to perform the re-ranking. The script uses a genetic algorithm to discover better assignments of ranks to nodes to minimize latency for S3D (and applications with a nearest neighbors communication pattern). While we could have computed a topologically aware re-ranking in our custom S3D Legion mappers, we use the same re-ranking script for both MPI/OpenACC and Legion versions of S3D so we can compare the latency hiding abilities of both versions directly.

Figures 11.5 and 11.6 plot the per-node throughput of the DME and Heptane mechanisms respectively for several different problem sizes on both the Titan and Keeneland supercomputers. Higher lines show better performance and flat lines are indicative of good weak scaling (tail-offs are inefficiencies). For the Legion runs, performance at a given problem size is shown for the best performing mapping strategy. On Keeneland, the best mapping strategy is to use the mixed CPU-GPU approach to

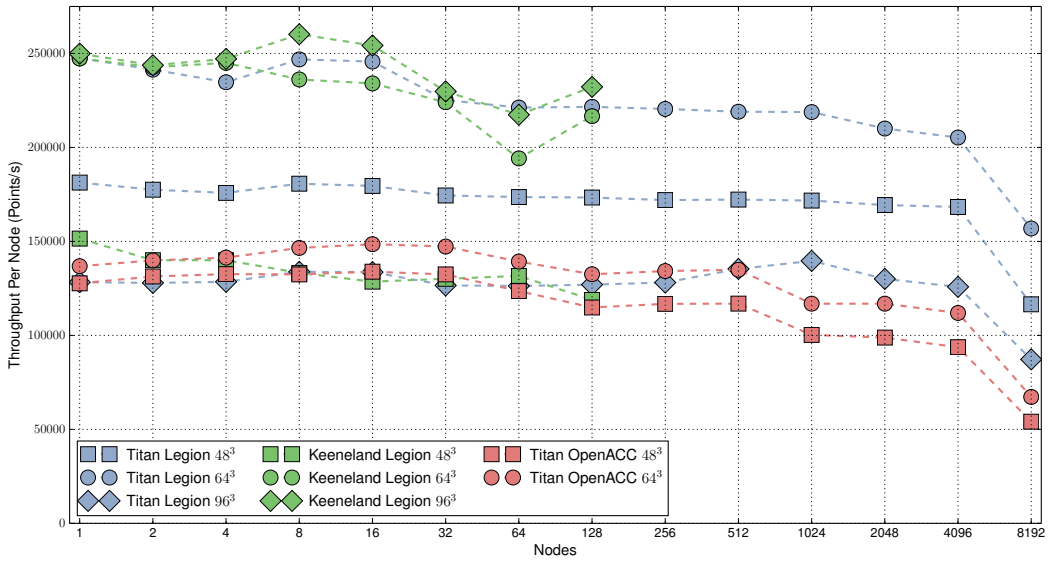


Figure 11.5: S3D Weak Scaling Performance for DME Mechanism

leverage the relatively balanced computational capabilities of the CPUs and GPUs. On Titan, the best mapping strategy is to use the all-GPU approach due to the extreme imbalance between the CPUs and the GPUs: the Kepler K20 GPU is many times more powerful than all the Bulldozer CPU cores combined. One important detail is that because the MPI-OpenACC version of the code effectively has its mapping hard-coded to place the majority of work on the GPU, it is unable to run a 96^3 problem size without exhausting the available memory in the GPU framebuffer.

As we can see in Figures 11.5 and 11.6, in general larger problem sizes achieve higher throughput as both the MPI-OpenACC and Legion versions of S3D can use the additional work to better overlap computation and communication. The one exception to this trend is with the 96^3 problem size for Legion on Titan. In this case, in order to fit the problem size within a Titan node, we had to use the mixed CPU-GPU mapping, which is not as efficient due to the slow Bulldozer CPU cores. We therefore notice a fall off in performance compared to the 64^3 problem size.

Overall, Legion significantly outperforms the MPI-OpenACC code. Between 1024 and 8192 nodes, when compared to the MPI-OpenACC code, the Legion version

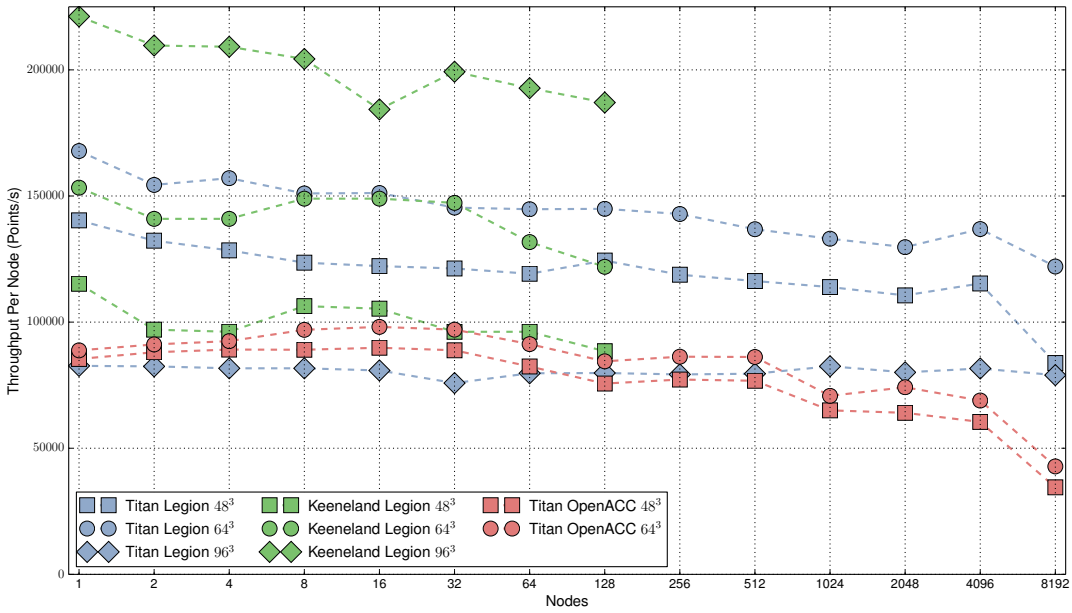


Figure 11.6: S3D Weak Scaling Performance for Heptane Mechanism

of S3D is 1.71-2.33X faster for the DME mechanism and 1.75-2.85X faster for the Heptane mechanism. On the Heptane mechanism, Legion does particularly well at discovering additional parallel work and using it to better overlap computation and communication (both inter-node as well as intra-node over the PCI-E bus). Both the Legion and MPI-OpenACC versions experience a falloff in performance at 8192 nodes. This is due to the failure of the genetic algorithm re-ranking script to fully explore the space of possible re-rankings given a fifteen minute time limit. Interestingly, for the Heptane mechanism, Legion is able to mitigate this effect by discovering further work to hide the additional latency. Note that this effect is even more pronounced for larger problem sizes.

Figures 11.7 and 11.8 illustrate the performance of different Legion mapping strategies for all problem sizes on Titan. On Titan, the severe performance mismatch between the CPUs and the GPUs results in the best mapping being to place as much work as possible on the GPUs. In contrast, on Keeneland, the best mapping

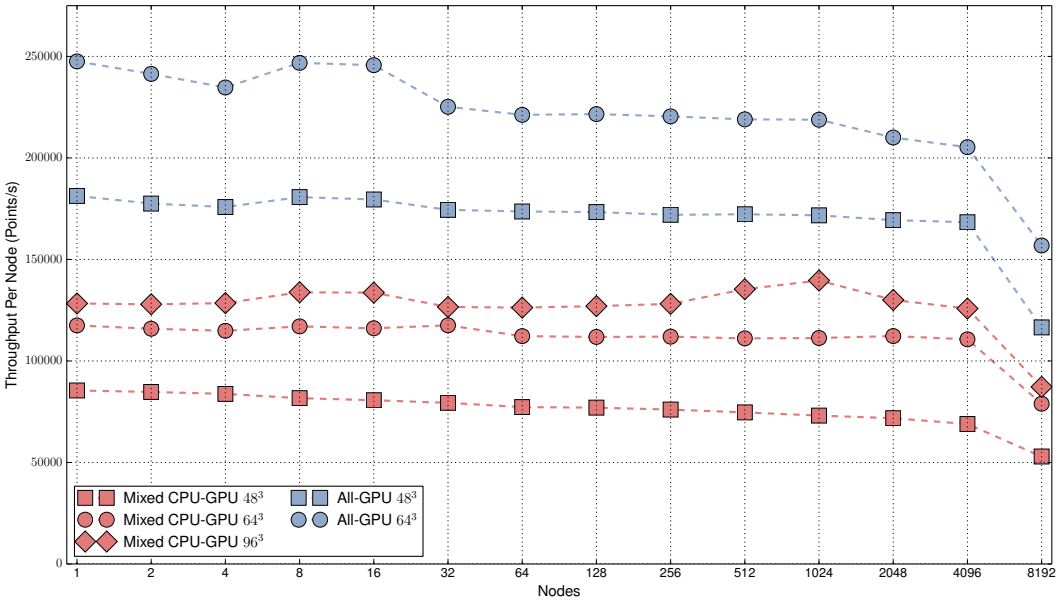


Figure 11.7: Mapping Strategies for the S3D DME Mechanism

strategy is always to evenly distribute work between CPUs and GPUs to ensure that all processing elements are fully utilized. Mismatches in processor capabilities are just another form of heterogeneity that programmers need to contend with as part of writing code for current and future architectures. By leveraging the Legion mapping interface we were easily able to deal with this form of heterogeneity without needing to make any code changes to the machine independent Legion code; we simply wrote different mappers for Titan and Keeneland.

In addition to the performance gains on existing mechanisms, Legion is also able to support the PRF mechanism that is much larger than the DME and Heptane mechanisms currently supported by the MPI-OpenACC code. Each new mechanism that is significantly different in size or computational intensity requires the MPI-OpenACC code to be refactored in order to tune for performance, often necessitating many man hours of work. Our Legion version of S3D requires no refactoring and only slight modifications to the customized mapper (no more than 10 lines of code). (We do use the Singe DSL compiler to generate new leaf task kernel implementations for

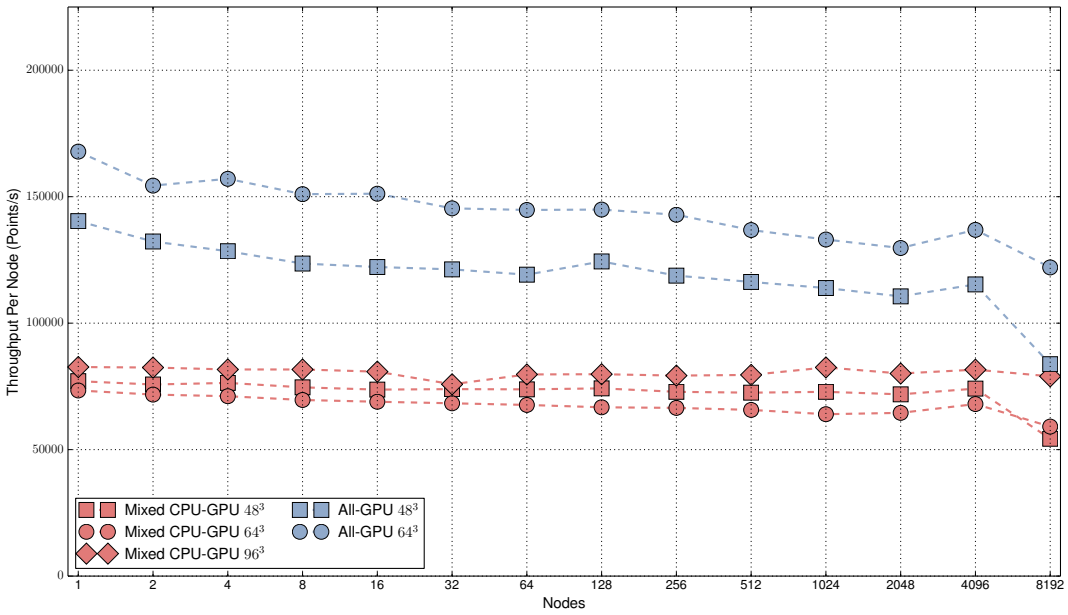


Figure 11.8: Mapping Strategies for the S3D Heptane Mechanism

the additional chemistry, but the base S3D code is unmodified.) Figure 11.9 shows performance results for the PRF mechanism running on Titan for three different problem sizes with different mapping options. Note that, unlike smaller mechanisms, there is no performance cliff at 8192 nodes for the PRF mechanism, as Legion is able to discover sufficient work in the PRF mechanism to automatically hide all of the communication latency. Being able to adapt to new mechanisms and automatically scale with just a few changes to a custom mapper truly illustrates the power of the Legion programming model.

11.4 Programmability

The final criteria on which we evaluate our Legion implementation of S3D is programmability. While this is qualitative metric, it is an important one, as the cost of programmer productivity and software maintenance can often exceed the cost of

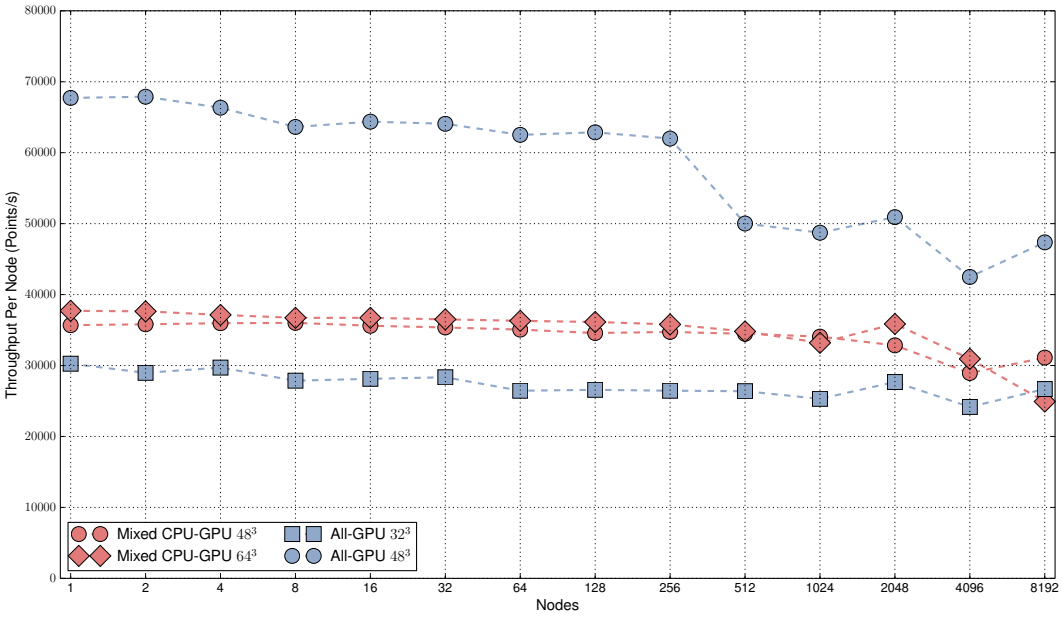


Figure 11.9: Performance of Legion S3D for PRF Mechanism

doing production runs. In the process of tuning our Legion version of S3D for both the Titan and Keeneland machines, we were able to try hundreds of different mapping strategies all without needing to modify the machine-independent specification of S3D. This demonstrates the power of our core Legion design goal to decouple an application specification from its mapping. In the future, we anticipate this property will allow us to easily port S3D to new architectures simply by developing new mapping strategies.

Chapter 12

Related Work

The history of programming supercomputers is extensive. In this chapter we attempt to cover the most recent work that is closely related to Legion. Many of the systems covered here are primarily academic systems and not in wide use in the larger scientific community. For a more comprehensive overview of the way that most computational scientists program supercomputers we refer readers to Section 1.2.

12.1 Array-Decomposition Languages

One of the more recent trends in academic research on supercomputing has been the focus on array decomposition languages. These programming systems have an understanding of the structure of program data through the use of arrays. Using their knowledge of arrays, these systems can aid programmers by providing easy ways of partitioning data and performing optimizations to hide communication latencies. We now look in detail at several of these array decomposition languages.

One of the most closely related systems to Legion in the academic literature is its spiritual ancestor Sequoia[28]. Sequoia is an array decomposition language targeted at programming hierarchical memory. Sequoia couples tasks with arrays to be accessed and provided mechanisms for launching sub-tasks that operated on distinct subsets of the arrays that the parent task was given (similar in several ways to Legion's privilege system). By recursively decomposing tasks into sub-tasks and arrays into sub-arrays,

applications could be generated that would target deep memory hierarchies. While there are many similarities between Legion and Sequoia, there are two primary differences. First, Sequoia is a static language that required all decisions about how to partition arrays and map computations onto the target architecture to be known at compile-time whereas Legion is a totally dynamic system. Second, Sequoia couples the decomposition of tasks with the decomposition of data, ultimately constraining both to be decomposed isomorphically to the target memory hierarchy. In Legion, the decomposition of tasks and regions are decoupled which makes application development more flexible. We will return to Sequoia's mapping interface shortly in Section 12.8.

Another array decomposition language related to Legion is Chapel[19]. Chapel provides *domains* as the fundamental abstract data type for describing collections of data. Using domains, applications can create arbitrary collections of data in a way that is machine independent. To map data onto a specific machine, Chapel uses *locales*, which are abstract locations in the machine that are ultimately mapped onto specific machine locations. Using *domain maps* programmers can specify how different domains are mapped onto different locales. This gives Chapel more flexibility to dynamically decide how program data is mapped to the target hardware. However, domain maps only support a single decomposition of data onto the target architecture that is then fixed for the remainder of application's execution. Using logical regions, Legion allows applications to describe many different views onto the same data which cannot be accomplished with domains. Furthermore, Legion allows different logical regions to be mapped into different memories throughout an application's lifetime yielding considerably more flexibility to the programmer than Chapel.

Like MPI, Co-Array Fortran[40] is another instance of an SPMD programming model. However, unlike MPI, Co-Array Fortran provides explicit support for describing arrays that are distributed across all of the processes in a Co-Array Fortran program. In order to help with the distribution of data, Co-Array Fortran supports many different distribution schemas, allowing for applications to tailor these distributions based on which data is going to be accessed by different nodes. However,

similar to Chapel, once these distributions are set, the arrays are fixed and the distribution cannot be modified. This is sufficient for applications that only have a single communication pattern, but for more complex applications with multiple phases and different communication patterns it can often lead to poor performance.

Unified Parallel C (UPC)[17, 2] is another SPMD programming model that instead presents a partitioned global address space (PGAS) abstraction to programmers. In this model, there is a single address space that is designed to make it easier for programmers to develop applications. However, it is implicit that each piece of data in the PGAS address space is associated with a single node. Reading or writing data is then either fast or slow depending on whether the data is local to a node or resides on a remote node. While PGAS abstractions regularly make it easier to achieve functional programs, they are often difficult to tune for performance as decisions about how data is mapped onto different nodes is either implicit or baked into application code.

While determining whether data is local or remote is usually difficult in UPC, Titanium[53] is another array decomposition language that attempts to make this information explicit through its type system. Titanium is an extension to the Java programming model that also supports distributed program execution with a PGAS model. Unlike UPC, Titanium's type system encodes whether data is local or remote, forcing programmers to understand the performance consequences of their programs. Furthermore, Titanium also supports different array decomposition patterns for distributed data between different nodes. By leveraging its static knowledge of the type system and array decompositions, the Titanium compiler is capable of performing many interesting optimizations such as aggregation of communication and overlapping computation with communication which is difficult to perform in the more general UPC model. However, these benefits come at a cost. Needing to reason about local and remote types can often lead to an overly conservative analysis that lowers performance when data is actually local. Furthermore, the need to statically decide the partitioning of data limits the distribution patterns that can be expressed. Finally, much like Chapel, only supporting a single view onto an array of data is limiting for applications with multiple phases and different communication patterns.

12.2 Region Systems

The term *region* has many connotations in the systems literature of which Legion is just one example. In this section we will cover region systems that in some ways are related to Legion.

Deterministic Parallel Java (DPJ) [14, 15] is the closest region system related to Legion. Regions in DPJ are used to statically describe the sets of data being operated on by different tasks. By leveraging this information, the DPJ compiler is able to infer where there is interference between different tasks as well as where parallelism exists. Thus DPJ can extract very low overhead parallelism from applications that appear to be sequential, which provides a simpler programming model for programmers to use when writing applications. This approach to automatically extracting parallelism based on region usage is very similar to Legion's approach. However, DPJ requires that tasks statically partition data into regions and statically name the regions to be accessed by tasks. This limits the kinds of computations that can be expressed and restricts data to being partitioned in a single way that is often limiting to applications with different phases. One benefit of this static approach is that the low overhead enables DPJ to reason about considerably smaller regions and to exploit very fine-grained parallelism that is difficult for Legion to achieve at runtime. The last difference between DPJ and Legion is that DPJ still relies upon the Java virtual machine, which limits applications to shared memory architectures, while Legion is capable of running on distributed machines.

Cyclone[32] is another region system designed for providing high performance and type safety for the C language. Cyclone introduces lexically scoped regions as a mechanism for allocating and storing data. By applying region annotations to functions applications can designate different access to regions similar to how Legion tasks declare privileges on logical regions. In addition, the regions in Cyclone provide a level of indirection for decoupling the specification of data from how it is laid out in memory. Logical regions in Legion provide a similar flexibility, but with a greater range of potential layouts.

One of our early inspirations for using logical regions to describe sets of data

was RC[31]. The RC language uses dynamically allocated regions for handling memory management and garbage collection in C. The dynamic hierarchy of regions in RC in some ways is related to the hierarchy of logical regions represented by region trees. Garbage collection of regions in RC is done using a reference counting scheme that bears some resemblance to the hierarchical reference counting scheme that Legion's garbage collector uses for collecting physical instances, with the caveat that RC operates in a shared memory address space while Legion works in a distributed environment.

12.3 Task-Based Systems

While most of the previous work that we have addressed has centered around abstractions for decomposing data, there are also many systems that are oriented around tasks. In this section we detail related work that is primarily based on extracting parallelism through descriptions of parallel tasks.

SSMP[41] is a task based system that in some ways is similar to Legion. SSMP allows for the creation of tasks that describe data access patterns to shared memory data structures. Using these access patterns, SSMP discerns which tasks have data dependences and which can be run in parallel. This is similar to how Legion extracts parallelism based on the logical region usage of different tasks. In some ways SSMP can be easier to use than Legion because describing access patterns can be easier than allocating data in logical regions. However, this is only possible because SSMP operates within shared memory architectures. Logical regions enable Legion to understand the structure of program data and not just the set of data accessed by different tasks, which is a necessary prerequisite for operating in a distributed memory environment. Another difference is that all SSMP tasks are checked for non-interference. This is only scalable on a shared memory system whereas Legion relies on its independence principle to perform scalable non-interference tests in a distributed environment. This approach also allows Legion to extract nested parallelism while SSMP is unable to take advantage of any hierarchy of its tasks.

Perhaps the closest related work to Legion was the design and implementation

of the Jade language[44]. Jade is a task based language where tasks are issued in sequential program order. Tasks dynamically declare the accesses that they are to perform to various shared objects. The Jade runtime then dynamically determines data dependences based on the data accesses performed to the shared objects. This implicit extraction of parallelism from dynamic information is very similar to Legion. Furthermore, unlike SSMP, Jade is capable of performing this analysis in a distributed system at the very fine granularity of individual objects. It is important to note that this is possible because Jade was developed at a time when communication was significantly faster than processor speeds, which made complex distributed algorithms requiring much more communication tractable. Jade tracks versions of shared objects in a way that is similar to how Legion tracks versions of logical regions. Legion uses logical regions as coarser objects over which to perform analysis to amortize the cost of dependence analysis. Legion also relies on its containment principle in order to make analysis hierarchical, which reduces the communication that needs to be performed by the runtime. Jade has no such restriction as runtime communication was not a performance bottleneck on the the class of machines it was targeting.

Another task based system based on an initial sequential programming model is the Cilk language[30]. Cilk is an extension of the C language that allows function calls to be spawned off as independent tasks. Using *sync* statements, programmers can then specify when parallel work must be completed. One of the benefits of Cilk is that the elision of all the Cilk extensions to a program results in a valid C program, which makes it easy to extend existing code. One downside to the Cilk programming model is that unlike Legion, Jade, and SSMP, Cilk programmers are directly responsible for specifying where parallelism and synchronization need to happen. This makes it easy to introduce race conditions and bugs both when writing and maintaining code. The primary reason for this is that the Cilk runtime has no knowledge about the structure of program data and therefore must trust the programmer to correctly specify parallelism and synchronization. Another consequence of this is that Cilk can only operate in a shared memory environment as it has no knowledge of how to move program data necessary.

A more recent task-based system is the Galois programming system[42]. The

Galois programming model and runtime is designed for extracting task based parallelism from irregular computations that are difficult to parallelize and load balance. To handle this class of applications, Galois provides several special kinds of data structures such as ordered queues and un-ordered queues. By combining efficient implementations of these data structures with small tasks operating on data structures (generalized as *morph* operators), Galois can extract fine-grained task parallelism from applications with irregular data structures. In Legion, equivalent behavior is obtained through the use of relaxed coherence modes such as atomic and simultaneous. However, because Galois operates only within a node and requires no knowledge of the underlying data structures it can handle much finer-grained tasks than Legion.

12.4 Place-Based Programming Models

While most systems for distributed machines use SPMD programming models, some encourage a more asynchronous approach based on the abstraction of *places*. Places are either abstract or concrete machine locations that enable application developers to name where either computation or data is to be assigned. Instead of having computations implicitly start running across all the nodes within a machine, an application must specify where different computations as well as the data that they require are assigned. We cover several the several systems that illustrate the evolution of the place-based programming model here.

The origin of place-based programming models is the X10 programming language[20]. X10 is an extension of the Java language that creates abstract places for naming where computations and data can be placed. A later deployment step maps abstract places onto concrete locations in the machine. The X10 programming model is mostly asynchronous and encourages the creation of many independent asynchronous operations. This approach aids in extracting significant parallelism from X10 programs, but it suffers from two fundamental problems. First, there is no way in X10 for programmers to compose operations (e.g. issue a asynchronous task launch dependent on an asynchronous copy completion). As a result programmers must explicitly manage synchronization, which yields code that suffers from the same modularity problems

as MPI codes. Second, programmers have to explicitly determine the mapping of applications onto places as part of their code. This hard-codes the effective mapping decisions into the code, making it very difficult to try new mapping decisions later without significant refactoring, which can potentially introduce correctness bugs.

One of the shortcomings of X10 places is that the abstract places were organized as a flat graph with little structure to provide programmers with any meaningful information about the underlying hardware. To rectify this problem, Hierarchical Place Trees[52] were introduced. The Hierarchical Place Trees model differs from X10 by organizing places into a tree. Places higher in the tree and closer to the root correspond to slower memories with less processing power, while places lower in the tree and closer to the leaves correspond to smaller faster memories with more processing power (and more processors because of the larger number of nodes). While hierarchical place trees make it possible to write X10 programs that express abstract locality and better describe data movement through the memory hierarchy, they still suffer from the same fundamental problems regarding hard-coded mappings as X10 programs using flat places.

Most recently, work on X10 and hierarchical place trees has been extended to create Habanero Java[18]. Habanero Java presents a more structured approach to programming with places. A richer set of primitives for launching asynchronous tasks and performing synchronization makes it possible for the language to make provable guarantees regarding deadlock and determinism. Specifically the *phaser* construct in Habanero Java resembles in some ways the phase barrier synchronization primitive used with relaxed coherence modes in Legion. Despite the additional programming features and new guarantees made by Habanero Java, it too directly encodes the mapping of an application to places in a way that is difficult to tune and modify for new architectures.

12.5 Actor Models

Another common approach to programming machines with distributed memory systems is to use an actor model. In an actor model each object operates as a separate

entity that both handles incoming messages and can send out its own messages to other actors in the system. We highlight two actor models for scientific programming here.

The canonical instance of an actor model being used in a supercomputing context is the Charm++ language[35]. Charm++ is an object-oriented programming model for distributed memory machines. In Charm++ any object can invoke a method on any other object. If both objects reside on the same node, then the method invocation occurs as it would in a shared memory machine. However, if the objects reside on different nodes, then the method invocation is automatically converted into a message that is sent to the node where the target object is stored. Charm++ programs rely on this feature to perform all communication between nodes. In order to reduce communication costs, the Charm++ runtime attempts to discover the underlying communication pattern and re-arrange object placement to minimize communication. Despite this approach to minimizing communication, Charm++ still suffers from the fundamental problem that communication patterns consist of sending many very small messages between nodes. This was acceptable in the decade when Charm++ was designed because communication networks were significantly faster than processors. However, in the two decades that have elapsed since, processors have become significantly faster than the communication networks. Ultimately, Charm++ programs suffer from low utilization of the network due to many small messages and an inability to scale their runtime algorithms for re-arranging objects to today's large machines.

Another example of an actor model is the Tarragon programming model[22]. While not a pure actor model, Tarragon combines elements of static dataflow with actor-like operations. Tarragon allows programmers to construct static graphs of tasks. Edges between tasks specify where communication between tasks is allowed to occur. At runtime tasks can then send arbitrary messages to other tasks with which they share an edge. In this way Tarragon can know the communication pattern of an application statically, while still permitting the application to engage in a dynamic actor-like communication framework. By leveraging the static knowledge of the communication graph, the Tarragon compiler can optimize many of the communication paths at compile-time.

12.6 Dataflow Models

One of the closer sets of related work to Legion involves dataflow programming models. In dataflow models, programmers explicitly specify the data that must be passed from one operation to the next. In some ways these systems are related to Legion as the resulting dependence graphs constructed by Legion are similar to the dataflow graphs constructed by these systems.

In their infancy, dataflow languages were initially designed for expressing very fine-grained operations that could be easily mapped onto the hardware. One example of these kinds of dataflow languages is the Id programming language[9]. Id allowed developers to write programs consisting of fine-grained functions with explicit data dependences between functions. The Id compiler then mapped the dataflow program down onto a specialized dataflow processor. In many ways Id was a successful demonstration of how dataflow could be leveraged to extract significant fine-grained parallelism from applications. However, as the disparity between processor speeds and the available memory bandwidth grew, dataflow architectures could no longer support the large bandwidths required of fine-grained dataflow programs making it much harder to maintain performance improvements with each successive processor generation.

To handle the complications encountered by Id, many dataflow languages moved to a coarser granularity of operations. One particular example of this transition is the Lucid dataflow system[34]. Lucid introduced coarse-grained operations that explicitly specified data usage, and then allowed the system to implicitly extract the dataflow inherent between different operations. In many ways this is similar to how Legion extracts data dependences between tasks based on region usage. The primary different is that Legion's logical regions are routinely much larger than the dataflow edges described in Lucid programs.

More recently, dataflow systems have been undergoing a renaissance as a way to deal with modern heterogeneous architectures. For example, CGD is an example of dataflow system designed for converting SPMD programs into dataflow

applications[46]. By leveraging its understanding of data layouts in distributed systems CGD automatically computes dataflow edges between different tasks and then attempts to hide communication latency through its knowledge of which tasks are safe to run in parallel. Legion performs many of the same optimizations as CGD, but does so based on its understanding of logical regions. Since Legion is not a pure dataflow system like CGD it is more expressive, but the underlying representation of tasks and their data dependences are similar in many ways to dataflow systems.

12.7 Low-Level Runtimes

The Legion runtime system is primarily a higher-level programming system because it provides higher-level abstractions such as logical regions that aid in the development of Legion applications. However, there are also runtime systems that are more accurately categorized as low-level programming systems that provide the minimal number of intrinsics for executing parallel applications. Low-level runtime systems are more closely related to Legion's low-level runtime system Realm[48], but we cover them here for completeness.

StarPU is a low-level runtime system that allows application developers to dynamically construct a task graph with dependences[10]. The StarPU runtime is then free to optimize the execution of this graph in any way that is consistent with the specified dependences. In many ways this is similar to the semantics of the Realm runtime. There are two primary differences. First, Realm provides generational events that alleviate the need for the Legion runtime to garbage collect event objects, which is part of the StarPU interface. This significantly simplifies the implementation of the Legion runtime. Second, StarPU does not support the reservation and phase barrier intrinsics supplied by Realm, which are crucial for implementing relaxed coherence modes.

Another related low-level runtime system to Realm is the Threaded Abstract Machine (TAM)[23]. Unlike Realm and StarPU, TAM is a hybrid static/dynamic system. At very fine granularities, TAM requires static dependences to be specified between tasks in order to allow for the TAM compiler to schedule the resulting tasks

at compile-time. This approach is necessary to deal with fine-grained parallelism. At coarser granularities, TAM is dynamic, with the graph of dependences between *frames* evolving at runtime. TAM was designed in an era of much smaller machines than modern heterogeneous supercomputers and therefore leaves much of the execution of the dynamic dependence graph to the client. TAM also has limited support for synchronization primitives analogous to reservations and phase barriers.

An even lower-level runtime system is GASNet[16]. GASNet provides the basis for the Realm runtime system and aids in portability by providing different conduits for various interconnect architectures such Infiniband, Cray Gemini and Aires, and Blue Gene L/P/Q. GASNet provides primitives for supporting communication using one-sided RDMA puts and gets as well as an active message interface. Most inter-node communication done in Legion is performed using GASNet active messages while bulk data transfer is handled using the one-sided RDMA puts. While GASNet provides many additional operations with its extended API, but Legion does not use them. This suggests that the GASNet interface could potentially be simplified and further optimized.

12.8 Mapping Interfaces

One of the more novel aspects of Legion is its dynamic mapping interface which provides a way of decoupling the specification of how an application is mapped from the algorithm. While the Legion design is unique, there are several other systems that use related ideas for decoupling algorithms from how they are mapped.

To the best of our knowledge the first programming system with an explicit mapping interface was the previously mentioned Sequoia language[28]. In addition to taking source files, the Sequoia compiler also required applications to provide mapping and machine files. A machine file gave a description of the target architecture and the mapping file directed the Sequoia compiler how to map the machine-independent source code onto the target machine. This approach made Sequoia code very portable. The only downside to this approach was that all mapping decisions had to be made statically in keeping with Sequoia's static language semantics. While static mapping

enabled many of the Sequoia compiler's global optimizations, it does not permit applications to dynamically react to the dynamic nature of either an application or the underlying hardware.

Another programming system that makes use of an explicit mapping interface is the Halide language and compiler[43]. Halide is a language and compiler for describing and optimizing image processing pipelines. Halide programs describe operations that are performed on two dimensional images and the Halide compiler optimizes the implementation of these pipelines for different architectures. To perform these optimizations, the Halide compiler attempts to find high-performance *schedules* for a given pipeline. A schedule is analogous to a mapping in Legion or Sequoia as it specifies which computations are executed on different processors as well as how data movement is performed. Much like Sequoia, schedules in Halide are static and can either be specified or searched for using an auto-tuner. The fine-grained nature of Halide image pipelines necessitates that these schedules be determined statically.

While Legion has an explicit mapping interface for mapping applications onto target hardware, another approach to performing dynamic mappings is to leverage meta-programming to just-in-time (JIT) compile code that is specific to the target architecture at runtime. One of the more recent ways to accomplish this is with two-stage programming in Lua-Terra[26]. Lua is a dynamically typed scripting language designed for productivity while Terra is a C-like statically typed performance language. The Terra compiler allows for meta-programming to be done in Lua so that Lua can construct Terra functions at runtime and JIT them either to specialize for the dynamic behavior of an application or to target specific hardware. Using meta-programming Lua-Terra programs can dynamically specialize and map a function for a target architecture. This feature ultimately convinced us that using meta-programming in Lua-Terra to create generator functions was the right approach to specializing Legion tasks for specific architectures and region instance layouts. Using Lua-Terra generator tasks, Legion applications can be dynamically specialized for any potential target architecture and mapping decision.

Chapter 13

Conclusion

Programming modern supercomputers is currently a daunting task for even the most expert programmers. For application scientists and engineers without considerable programming experience, the task will soon become completely intractable as machines become increasingly complex and heterogeneous. If left unaddressed, this trend could potentially stunt the advancement and progress of many areas of science as users are no longer capable of fully leveraging the potential of new machines.

Unfortunately, combating this problem cannot be achieved by a simple evolution of existing software tools at the same level of abstraction. Instead, a new approach to supercomputing is necessary to raise the level of abstraction and allow software systems such as runtimes and compilers to manage the complexity of targeting code to new architectures.

In this thesis we have presented Legion as one possible candidate for achieving this end. Legion raises the level of abstraction in two ways. First, Legion provides a way of decoupling policy decisions from how they are implemented, allowing the Legion runtime to manage all of the complexity associated with carrying out an implementation. Second, Legion provides a programming model that allows applications to be specified in a machine-independent manner which decouples how applications are written from how they are mapped onto a target machine. Ultimately, this is essential for allowing codes to be written once and re-targeted to new architectures as hardware changes, thereby ensuring that application developers never have to re-write

their code from scratch when a new machine is released. By exposing all performance decisions through the Legion mapping interface, codes can be easily ported without application developers needing to fight the Legion runtime in order to control performance.

As we have shown throughout this thesis, the crucial innovation that permits Legion to raise the level of abstraction is the introduction of logical regions. Unlike most current programming systems, logical regions and operations on them provide a data model for the application to communicate information about the structure of program data to the programming system in a machine independent fashion. Specifically logical regions support many of the same operations as relations, a fundamental data abstraction for many areas of computer science. The relational data model supported by logical regions provides a familiar way to describe both how data is partitioned and how it is used by computations. Furthermore, Legion can leverage its knowledge of the structure of program data and how computations use it to automatically version it to support transparent resiliency and speculation.

The primary result of this thesis is a demonstration of the power that logical regions confer on a programming system when used in executing real scientific computing applications. Our implementation of Legion illustrates that a significant amount of complexity that currently burdens application developers can be cleanly encapsulated within a reasonable abstraction. While the Legion implementation is nontrivial, with proper software engineering techniques it can be maintained by several expert programmers, thereby insulating application developers from the complexity and allowing them to focus on actual science and research.

To prove that Legion is a real programming system, we ported S3D, a full production combustion simulation to use Legion and compared it to a highly optimized version of S3D tuned independently by experts. Our Legion version is between 2-3X faster when run at full scale compared to the baseline on the world's number two supercomputer. The relational data model supported by logical regions allowed us to express computations as tasks that operate over thousands of fields. Furthermore, the Legion mapping interface allowed us to experiment with many different mapping strategies for different supercomputers to discover higher performance versions of the

code while requiring no changes to the machine-independent specification of the application. Our version of Legion S3D is now being used for production runs by real combustion scientists and may in the immediate future become the canonical version of S3D.

In the future we expect to see the evolution of many other programming systems that mirror the characteristics of Legion. We contend that in order to be successful, these programming systems will need to have many of the same characteristics of Legion. First, they must be capable of expressing codes in a machine-independent manner, and then later mapping the code onto the target machine. Second, they should decouple policy from mechanism, thereby allowing applications to specify the best way of performing important operations such as data partitioning and communication. In order to achieve these goals, these programming systems will need to be able to abstract the structure of data in a way that is at least as powerful as the relational data model supported by logical regions.

In order to continue to scale applications for modern and future supercomputers, a disruptive shift in the way applications are developed is necessary. Raising the level of abstraction so that advanced programming systems can hide much of the complexity associated with writing codes for target machines will be the only way forward. We believe that Legion provides a crucial first look at some of the techniques necessary for making this change a reality. By already demonstrating considerable improvements over current programming systems when running production codes at scale on modern machines, we have shown that the Legion design is a viable way forward. In the future, we anticipate Legion continuing to grow and evolve to support additional applications on new architectures, thereby improving the productivity of the scientists and researchers that use Legion.

Bibliography

[1] The Gemini Network. Technical report, 2010.

[2] UPC language specification v1.2. `upc.lbl.gov/docs/user/upc\_spec\_1.2.pdf`, 2011.

[3] OpenACC standard. `http://www.openacc-standard.org`, 2013.

[4] CUDA programming guide 5.5. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`, February 2014.

[5] Genetic algorithm for MPI placement. `https://github.com/ramanan/gampi`, 2014.

[6] Variable smp - a multi-core cpu architecture for low power and high performance. Technical report, 2014.

[7] Top 500 supercomputers. `http://www.top500.org`, June 2014.

[8] M. Adams, P. Colella, D. T. Graves, J. N. Johnson, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo software package for AMR applications - design document. Technical Report LBNL-6616E, 2014.

[9] K. Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, March 1990. ISSN 0018-9340. doi: 10.1109/12.48862.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011. ISSN 1532-0626. doi: 10.1002/cpe.1631.

[11] 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, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5.

[12] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: Leveraging warp specialization for high performance on gpus. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 119–130, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555258.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Structure slicing: Extending logical regions with fields. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, Los Alamitos, CA, USA, 2014. IEEE Computer Society Press. ISBN 978-1-4799-5500-8.

[14] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640097.

[15] Robert L. Bocchino, Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, pages 535–548, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926447.

[16] Dan Bonachea. Gasnet specification v1.1. Technical Report UCB/CSD-02-1207.

[17] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. UC Berkeley Technical Report: CCS-TR-99-157, 1999.

[18] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. doi: 10.1145/2093157.2093165.

[19] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl*, pages 1094–3420, 2007.

[20] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852.

[21] J H Chen, A Choudhary, B de Supinski, M DeVries, E R Hawkes, S Klasky, W K Liao, K L Ma, J Mellor-Crummey, N Podhorszki, R Sankaran, S Shende, and C S Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, page 015001, 2009.

[22] Pietro Cicotti and Scott B. Baden. Asynchronous programming with Tarragon. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06,

New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188620.

[23] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 164–175, New York, NY, USA, 1991. ACM. ISBN 0-89791-380-9. doi: 10.1145/106972.106990.

[24] Leonardo D. and Ramesh M. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998. ISSN 1070-9924. doi: http://dx.doi.org/10.1109/99.660313.

[25] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063396.

[26] 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, pages 105–116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462166.

[27] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108.

[28] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188543.

[29] Charles R. [Los Alamos National Laboratory] Ferenbaugh. *The PENNANT Mini-App: Unstructured Mesh Hydrodynamics for Advanced Architectures (U)*. Jan 2013. URL `http://www.osti.gov/scitech/servlets/purl/1059398`.

[30] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277725.

[31] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 70–80, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378815.

[32] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563.

[33] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 38–49, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993504.

[34] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading*, pages 113–129. IEEE Computer Society Press, 1995.

[35] 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, pages 91–108, New York, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi: 10.1145/165854.165874.

[36] Khronos. The OpenCL Specification, Version 1.0. The Khronos OpenCL Working Group, December 2008.

[37] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control. *CoRR*, abs/0907.0929, 2009.

[38] J. Levesque, R. Sankaran, and R. Grout. Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond. SC '12, pages 15:1–15:11, 2012.

[39] M. Lijewski, A. Nonaka, and J. Bell. Boxlib. `https://ccse.lbl.gov/BoxLib/index.html`, 2011.

[40] R.W. Numrich and J. Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN FORTRAN Forum*, 17(1998):1–31, 1998.

[41] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Handling task dependencies under strided and aliased references. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 263–274, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810122.

[42] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993501.

[43] 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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176.

[44] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, May 1998. ISSN 0164-0925. doi: 10.1145/291889.291893.

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

[46] A Soviani and J.P. Singh. Optimizing communication scheduling using dataflow semantics. In *Proc. ICPP*, pages 301–308, 2009.

[47] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1):4:1–4:11, February 2009. ISSN 0730-0301. doi: 10.1145/1477926.1477930.

[48] Sean Treichler, Michael Bauer, and Alexander Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Parallel Algorithms and Compilation Techniques*, PACT '14.

[49] Sean Treichler, Michael Bauer, and Alex Aiken. Language support for dynamic, hierarchical data partitioning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 495–514, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509545.

[50] J.S. Vetter et al. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Computing in Science Engineering*, pages 90 –95, 2011.

[51] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012. ISBN 0321842685.

[52] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 172–187, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13373-8, 978-3-642-13373-2. doi: 10.1007/978-3-642-13374-9_12.

[53] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. volume 10, pages 825–836, 1998.