REGENT: A HIGH-PRODUCTIVITY PROGRAMMING LANGUAGE FOR IMPLICIT PARALLELISM 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

Elliott Slaughter
August 2017

Abstract

Modern supercomputers are dominated by distributed-memory machines. State of the art high-performance scientific applications targeting these machines are typically written in low-level, explicitly parallel programming models that enable maximal performance but expose the user to programming hazards such as data races and deadlocks. Conversely, implicitly parallel models isolate the user from these hazards by providing easy-to-use sequential semantics and place responsibility for parallelism and data movement on the system. However, traditional implementations of implicit parallelism suffer from substantial limitations: static, compiler-based implementations restrict the programming model to exclude dynamic features needed for unstructured applications, while dynamic, runtime-based approaches suffer from a sequential bottleneck that limits the scalability of the system.

We present Regent, a programming language designed to enable a hybrid static and dynamic analysis of implicit parallelism. Regent programs are composed of tasks (functions with annotated data usage). Program data is stored in regions (hierarchical collections); regions are dynamic, first-class values, but are named statically in the type system to ensure correct usage and analyzability of programs. Tasks may execute in parallel when they are mutually independent as determined by the annotated usage (read, write, etc.) of regions passed as task arguments. A Regent implementation is responsible for automatically discovering parallelism in a Regent program by analyzing the executed tasks in program order.

A naive implementation of Regent would suffer from a sequential bottleneck as tasks must be analyzed sequentially at runtime to discover parallelism, limiting scalability. We present an optimizing compiler for Regent which transforms implicitly parallel

programs into efficient explicitly parallel code. By analyzing the region arguments to tasks, the compiler is able to determine the data movement implied by the sequence of task calls, even in the presence of unstructured and data-dependent application behavior. The compiler can then replace the implied data movement with explicit communication and synchronization for efficient execution on distributed-memory machines. We measure the performance and scalability of several Regent programs on large supercomputers and demonstrate that optimized Regent programs perform comparably to manually optimized explicitly parallel programs.

Acknowledgments

As with many large research efforts, Regent has been made possible through the work of many people. Michael Bauer and Sean Treichler lead the development of the Legion and Realm runtime systems, which Regent uses as a code generation target. Regent draws heavily in its design from Legion and can be seen as a "Legion language" in some sense. (Before Regent had a name, this is in fact what we called it.) Sean Treichler wrote much of the original type system for Core Legion from which Regent evolved. During the hectic days leading up to the submission of the original Regent paper, Wonchan Lee stepped in to contribute the vectorization optimization that enabled Regent to match the performance of hand-tuned vectorized codes in the initial experiments. Wonchan also contributed the OpenMP optimization. The author was otherwise response for all aspects of the work described herein. Portions of this work appeared previously as [62] and [63].

In the design of any programming language, it is critical to choose an initial set of applications for which the language should work well. Patrick McCormick put me on the right path here during my internship at Los Alamos National Lab, and introduced me to Charles Ferenbaugh, the author of the PENNANT proxy application. PENNANT was the first substantial application that I wrote in Regent, and like any first test of a complex system, made apparent a number of flaws in the design and implementation of the language. Wonchan Lee took the lead on developing a number of additional Regent applications including Circuit, MiniAero, and Soleil-X (the last along with Manolis Papadakis), which have been repeatedly used in our experiments.

During my time at Stanford, I have been fortunate to be working in a vibrant research group. Manolis Papadakis, Lázaro Clapp and I all joined in the same year,

and we did several rotations together before all joining Alex Aiken's group. Michael Bauer, Sean Treichler, Wonchan Lee, Zhihao Jia, Todd Warszawski and the other members of the Legion team have been inspiring, encouraging, and hard working (as appropriate for the situation). We have to this day kept a sign in our office which reads: "This Would Be So Much Easier In Haskell", a sentiment inherited from Michael Bauer's days on the Sequoia project which we appeal to from time to time when we run into an especially hard problem to solve. This gives you at least a small sense of the flavor of the group. And, of course, throughout all of this Alex Aiken has been responsible for keeping us all fed and happy and on track to graduate (no small feat in itself).

Throughout all of this I have relied on my family for support. I'd like to thank my wife Cathy for waiting patiently for me to graduate. My daughter Jane has been racing against time to see if she can walk before I complete my dissertation. (She won this round.) And my brother Matthew, parents Susan and David, and all my grandparents have been supportive of me throughout my years of education.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computation, and computational performance, has become an important driver of scientific progress over the last several decades. Computer-based simulations are used, for example, to test the validity of models of physical phenomena and to rapidly explore the possible design space for physical systems. Such simulations are often performance-constrained. That is, higher-fidelity simulations require more operations to be performed by the computer, while the time allotted to compute a solution is limited, and thus the performance of the computer system determines the maximum fidelity of a simulation that can be attempted.

Fortunately, performance-constrained scientific simulations are also often highly *parallelizable* (i.e., can be implemented efficiently on a parallel computer). For such applications, simulation fidelity is limited primarily by the scale of the parallel computer on which it can be efficiently executed. At the upper end of this scale, *supercomputers* provide the highest possible performance available in a single, massively-parallel computer. Applications written specifically for supercomputers are called *high-performance* applications.

Supercomputers differ from conventional personal machines in two key ways. First, they employ massive numbers of processors in order to accelerate the computation of parallel applications. Second, they feature *distributed* as opposed to *shared* memories. Briefly, the shared memory abstraction provides the illusion that a single logical memory is accessible from all processors, whereas under distributed memory each

processor only has direct access to a local memory and must access the remote memories of other processors indirectly via an interconnect. Thus the distributed-memory abstraction trades increased complexity for the potential to achieve superior performance on these machines. Programming models can choose whether or not to pass this complexity on to the programmer, but in high-performance application development, performance is generally the top priority and thus the de facto standard programming models in use on modern supercomputers are *explicitly parallel*. That is, these models expose the parallel (and distributed) semantics of the hardware to the user, permitting maximal performance at the cost of higher complexity. Ironically, as a supercomputers grow in scale and complexity, the cost of achieving the absolute highest possible performance within these explicit models can become prohibitive, and thus production codes often stop short of this goal [14].

An attractive alternative to the explicit approach is *implicit parallelism*, in which the user sees sequential execution, and the programming system is responsible for achieving parallelism and distribution. By definition, the implicitly parallel approach requires the system to perform a *program analysis*, or analysis of the possible effects of a program, to determine where parallelism exists in the code. When it works, implicit parallelism offers the best of both words: ease of use, and high performance. However, outside of domain-specific settings, the implicitly parallel approach has faced roadblocks which prevent it from scaling efficiently to large node counts with certain classes of computations, such as simulations on unstructured meshes, in which the required program analysis is challenging.

This dissertation aims to show that leveraging coarse-grained tasks (functions) with strict privileges (denoting what data a task reads and writes) and user-visible partitioning (permitting the user to specify the relevant sets of data and their relationships) enables implicitly parallel programs with sequential semantics to be compiled to efficient SPMD code that scales to large numbers of nodes. For certain classes of codes, such as simulations on unstructured meshes, this permits implicitly parallel implementations to achieve practical levels of scalability.

The following sections establish in more detail the various points in the design space of parallel programming systems, and lay out a motivating example that demonstrates

some of the challenges in these designs.

1.1 Background

While a number of programming models for supercomputers have been developed, by far the most successful is the SPMD, or single-program multiple-data, programming model exemplified by MPI [64]. MPI is an explicitly parallel and distributed abstraction layer that (by design) closely reflects the capabilities of the underlying hardware. An executing MPI program consists of a set of *ranks*, or independent copies of the code, that run on distinct processors with distinct memories and that have the ability to send *messages* to communicate data between ranks. No other method of accessing the contents of remote memories is provided. This approach has the advantage that because the hardware capabilities are directly exposed, experienced programmers are able to map applications directly to this hardware in a way that reliably achieves high performance.

However, the approach taken by MPI also has a number of disadvantages. First, while the abstractions of MPI closely resemble the hardware, they do not generally reflect the way that applications themselves are conceived, and thus there is some work required to map an application into the MPI model in the first place. Second, a result of MPI's design is that the programmer is exposed to a number of *programming hazards*, or potential mistakes a programmer can make in the implementation of an application in MPI. These include traditional pitfalls of parallel programming (e.g. data races, deadlocks, non-determinism) as well as ones specific to MPI (e.g. mismatched sends and receives). Third, MPI achieves performance only when the underlying machine resembles at least to a first approximation the class of machines for which MPI was originally designed: i.e. ones with homogeneous processors with reasonable network latencies and a reasonably uniform interconnect. Increasingly, modern and future machines diverge from this design in one or more dimensions (scale, hierarchy, heterogeneity, etc.). Although MPI can be augmented to create a number of "MPI+X" programming models to account for e.g. the presence of heterogeneous processors, these approaches introduce additional complexity and hurdles which make

it challenging to achieve performance.

As described above, an attractive alternative is to employ an *implicitly parallel* programming model where programs appear to execute with sequential semantics but can be parallelized automatically by the system. Classic parallelizing compilers [19, 37, 43], HPF [45, 56] and the data parallel subset of Chapel [27] are canonical examples; other examples include MapReduce [32] and Spark [74] and task-based models such as Sequoia [35], Legion [13], StarPU [10], and PaRSEC [22].

Because implicitly parallel programming models employ sequential execution semantics, programs in these models are easy to read and write for both expert and non-expert users, and also avoid by definition the various programming hazards associated with explicit parallel and distributed programming. Furthermore, such systems can be designed to exploit information about the structure of the program control and data, permitting the system to automatically schedule such programs for execution on machines with heterogeneous processors and automatically manage data movement across the deep memory hierarchies present in such systems.

For the programming system to automatically find the parallelism in an implicitly parallel program—where by definition that parallelism is not explicitly specified by the user—some amount of program analysis is required. This analysis can be performed at different times: either statically, without specific knowledge of the runtime inputs to the program, or dynamically, when such inputs are available. Both approaches have fundamental limitations.

Due to the halting problem (and more generally, Rice's theorem), static analysis of non-trivial program properties such as parallelism is challenging. For the analysis to be tractable, systems must sacrifice either soundness or completeness.

A system that sacrifices *soundness* admits programming hazards into the model—i.e. it's possible for the user to make a mistake and the system won't (and can't) know. At the far end of this spectrum is OpenMP [31], which implicitly trusts user assertions about parallelism; the system has no ability whatsoever to check the correctness of these assertions. As another example, Cilk [2] relies on shared memory to avoid the need for a precise analysis of the side effects of tasks. The lack of soundness limits the ability of a compiler to perform aggressive optimizations, making it challenging to

scale these models efficiently to distributed memory.

On the other hand, a system that sacrifices *completeness* must restrict the possible input programs—meaning that there will in general be valid programs that would execute correctly if permitted, but are disallowed by the system. As a somewhat extreme example, Cilk (described above) can be used without support for shared memory (and without relying on a distributed shared-memory subsystem), but in such cases tasks are required to be pure functions, making the programming model substantially more restrictive. However, the same principle is visible to varying degrees in many implicitly parallel programming systems.

This is clearly the case with domain-specific languages (DSLs), where the intended domain is explicitly identified. For example, in the Ebb [16] DSL the bodies of functions (kernels) are restricted to a class of forall-style parallel loops with stencil-like memory access patterns. The advantage of these restrictions is that they permit a more robust and predictable implementation. The memory access patterns permitted by Ebb can be checked by a straightforward static analysis which is included in the Ebb type system. As a result, any Ebb program which compiles successfully is guaranteed to be free of parallel programming hazards such as deadlocks and data races. Furthermore, because of the restrictions in the programming model, Ebb programs can be compiled automatically for efficient execution on GPUs and vectorized execution on CPUs. However, Ebb's restrictions on programs limit the applicability of the technique to certain classes of applications.

A risk with more general-purpose languages is that there can be a gap between what the compiler nominally supports, and what it supports well. For example, the HPF language [45, 56] provides rather ambitious support for automatic parallelization of general loops via the `DO` loop construct. In practice, HPF compilers can reliably parallelize affine loops. More general loops may be difficult or impossible for the compiler to parallelize, even if it is obvious to the programmer that those loops have no loop-carried dependencies. This is a reflection of the difficulty of static analysis of fine-grained memory accesses in general-purpose programs.

An alternative is to consider programs on coarse-grained tasks and partitions of data. Under this approach, the user explicitly identifies the relevant units of compute

and data, potentially simplifying the required analysis.

Sequoia [35] is an example of such a language that also makes extensive use of static analysis. Sequoia programs consist of a decomposition of a sequential program into a tree of tasks. *Leaf tasks* (i.e. that do not call other tasks) are permitted to contain arbitrary code, as they need not be analyzed by the compiler. However, *inner tasks* (i.e. that call other tasks) must be analyzed by the compiler, and thus have many restrictions. The only permitted data structures are (multi-dimensional) arrays, and the partitioning of such arrays is restricted so that sub-arrays must also be dense. The sizes of all data structures, including inputs, must be supplied to the compiler at compile-time, and the exact number of tasks to be executed must also be fixed statically. Furthermore, a specification of the target machine, and a mapping of the application to that target machine, must also be provided to the compiler. With all this information, a compiler has a complete view of the execution of the program and can perform a number of powerful optimizations [46]. However, the restrictions (though not applicable to leaf tasks) are substantial and prevent the system from being applied to less regular problems such as unstructured meshes. Again, static analysis (when soundness is an objective) forces the programming model to place significant restrictions on the possible input programs.

Thus, with approaches based on static analysis, there is a fundamental trade-off between expressiveness and tractability of the analysis. Programming systems must explicitly or implicitly limit the set of programs that are supported, assuming safety is a goal, or else give up safety entirely and rely on programmer assertions about key program properties.

Dynamic analysis avoids many of the limitations of static analysis, but presents a different set of challenges.

A classic approach used to parallelize sequential programs with fine-grained loops is the inspector/executor (I/E) method [53,54]. Under the I/E method, loop dependencies are determined by instrumenting the program to record the exact read and write sets of each loop iteration at runtime. Because the approach relies only on a dynamic, rather than static, analysis it can be applied to very general classes of loops. However, the information being recorded about the program is so fine-grained that at large problem

sizes and large scales, the program may run out of memory before it even begins to run. Implementations that attempt to save space in the dynamic analysis by approximating the information collected lose precision and thus impose additional communication overheads resulting in worse overall scalability [47]. As with static analysis, there is a trade-off between fine and coarse-grained approaches, where fine-grained approaches lose sight of many higher-level program properties that could help reduce analysis cost.

Legion [13] is a programming model with coarse-grained tasks, similar in its basic outlines to Sequoia, but which employs an entirely dynamic analysis. Thus Legion allows substantially more dynamic behavior than Sequoia, permitting, for example, dynamic, input-dependent, and possibly unstructured or sparse data structures, dynamic numbers of and dependencies between tasks, and a dynamic mapping of the application to a target machine. Legion permits very expressive partitioning of user data structures, allowing the user to specify with precision the important subsets of data, avoiding the need for expensive (in time and memory) fine-grained dynamic analysis. However, these benefits come at the cost of a less expensive but still non-trivial coarse-grained dynamic analysis. In Legion, this analysis happens off the critical path of the computation, so the cost is hidden as long as the application running time (total running time of tasks divided by number of processors) is greater than the time required for the dynamic analysis. However, as the analysis time for common programming idioms is itself proportional to the number of tasks, while the ratio of running time to processors generally stays constant (when weak scaling), the overhead generally exceeds running time at some scale. In our experience this happens between 10 to 100 nodes on typical HPC applications.

This dissertation presents *Regent*, a programming language for task-based implicit parallelism. Regent takes a hybrid approach that allows it to carefully and robustly navigate the trade-offs described above. Regent programs are composed of tasks, and tasks take regions as arguments. Regent borrows many aspects from Sequoia and Legion: for example, the effects of tasks are soundly summarized by the privileges (read, write, etc.) declared on parameters to tasks, thus avoiding the need for fine-grained static analysis at the level of individual memory accesses as in OpenMP, Ebb,

HPF, and I/E methods. And language support for partitioning enables the user to identify the important subsets of data and their relationships.

The default semantics of Regent are sequential, as this is the easiest and most productive way to use the language, and is intended to be the way the vast majority of users use the language. Regent also provides a variety of "escape hatches" that enable explicit parallelism within the language that are primarily intended to be used in situations where the compiler and runtime are unable to achieve performance on a straightforwardly written implicitly parallel program. This also has a side benefit of making it easy to describe Regent's optimizations, as all code transformations produce valid Regent code (though possibly requiring the use of Regent's explicitly parallel subset). However, as the goal of this work is to make the implicitly parallel subset as effective as possible, all references to "Regent" should be assumed to refer to the implicitly parallel subset unless otherwise stated.

Regent allows highly dynamic behavior. The number, values of arguments to, and dependencies between tasks are all dynamically determined, and the data model is very expressive. Regent's regions (containers of elements) may be partitioned into dynamically determined numbers of subregions and these subregions may include arbitrary, dynamically computed subsets of elements.

However, unlike a pure dynamic runtime system, Regent checks various properties of the input program via static analysis, allowing Regent to achieve a number of static safety guarantees. For example, privileges on region arguments to tasks are checked statically, as is the safety of pointer dereferences within regions.

Furthermore, Regent's amenability to static analysis enables a number of optimizations that enable efficient scalability to large numbers of nodes. One optimization of particular importance is *control replication*. For programs with repetitive loops of task launches, Regent is able to apply aggressive static optimizations to achieve an algorithmic reduction in the cost of dynamic analysis compared to I/E methods and Legion, while preserving Regent's ability to handle substantially dynamic behavior. Although Regent may not be able to fully characterize the dependencies between tasks at compile-time, and thus may not be able to completely eliminate the overhead of dynamic analysis, Regent is able to apply a transformation of the code to produce

SPMD-style *shards* of execution that distribute this analysis across multiple nodes to significantly improve the scalability of Regent codes.

We demonstrate that control replication achieves up to 99% parallel efficiency on 1024 nodes (12288 cores) on the Piz Daint supercomputer, on a set of applications with tasks on the order of milliseconds to tens of milliseconds. For several classes of applications that we consider, such as simulations on unstructured meshes, this is to the best of our knowledge the first automatic technique capable of achieving practical levels of scalability for these applications where the application source code is written in a general-purpose language with sequential semantics. Control replication is not limited to Regent, but is enabled by the careful selection of features that Regent provides, and is one of the key contributions of this work.

To expand on the motivation of Regent and particularly control replication, we now consider a more concrete example in the context of the control replication optimization.

1.2 Motivating Example

Consider the code in Figure 1.1a, an example of the implicitly parallel style described above. Assuming there are no loop carried dependencies, the parallelization of this program is straightforward: The iterations of each of the two inner loops can be executed in parallel on multiple processors in a fork-join style. As illustrated in Figure 1.1c, a main thread launches a number of worker threads for the first loop, each of which executes one (or more) loop iterations. There is a synchronization point at the end of the loop where control returns to the main thread; the second loop is executed similarly. Because the second loop can have a completely different data access pattern than the first (indicated by the arbitrary function `h` in `B[h(j)]`), complex algorithms can be expressed.

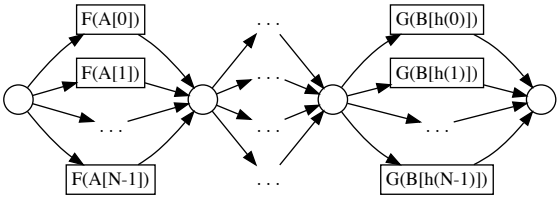
As already suggested, in practice programmers don't write highly scalable, high-performance codes in the implicitly parallel style of Figure 1.1a. Instead, they write the SPMD-style code in Figure 1.1b. Here the launching of a set of worker threads happens once, at program start, and the workers run until the end of the computation. We can see in Figures 1.1c and 1.1d that conceptually the correspondence between

```
1 for t = 0, T do
2     for i = 0, N do -- Parallel
3         B[i] = F(A[i])
4     end
5     for j = 0, N do -- Parallel
6         A[j] = G(B[h(j)])
7     end
8 end
```

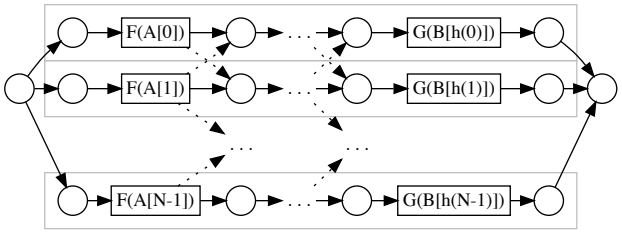
(a) Original program.

```
1 for i = 0, N do -- Parallel
2     for t = 0, T do
3         B[i] = F(A[i])
4         -- Synchronization required
5         A[i] = G(B[h(i)])
6     end
7 end
```

(b) Transposed program.



(c) Implicitly parallel execution of original program.



(d) SPMD execution of transposed program.

Figure 1.1: Comparison of implicit and explicit parallelism.

the programs is simple. Where Figure 1.1a launches N workers in the first loop and then N workers in the second loop, Figure 1.1b groups sequences of worker tasks into larger parallel threads in the obvious way.

While Figure 1.1a and Figure 1.1b are functionally equivalent, they have asymptotically different scalability. To see this, consider what happens in Figure 1.1a as the number of workers N (the "height" of the execution graph in Figure 1.1c) increases. Under weak scaling, the time to execute each worker task (e.g., `F(A[i])` in the first loop) remains constant, but the main control thread does $\mathcal{O}(TN)$ work to launch $2TN$ workers. Thus, for some N, the runtime overhead of launching workers exceeds the individual worker's execution time and the program ceases to scale. While the

exact scalability in practice always depends on how long-running the parallel worker tasks are, our experience is that many implicitly parallel programs don't scale beyond 10 to 100 nodes when task granularities are on the order of milliseconds to tens of milliseconds. In contrast, the SPMD program in Figure 1.1b launches only N workers, avoiding the cost of repeated launches on every time step (the `T` loop). Programs written in SPMD style can scale to thousands or tens of thousands of nodes.

However, from a usability perspective implicit parallelism is provides clear benefits over SPMD. While it is not possible to give precise measurements, it is clear that the difference in productivity is large: In our experience an implicitly parallel program that takes a day to write will require roughly a week to code in SPMD style. The extra programming cost is incurred because the individual workers in Figure 1.1b each compute only a piece of the first loop of Figure 1.1a, and thus explicit synchronization is required to ensure that all fragments of the first loop in all workers finish before dependent parts of the second loop begin. Furthermore, because the access patterns of the two loops in Figure 1.1a need not be the same, data movement is in general also needed to ensure that the values written by the various distributed pieces of the first loop are communicated to the threads that will read those values in the distributed pieces of the second loop. In most SPMD models (and, specifically, in MPI) this data movement must be explicitly written and optimized by the programmer. The synchronization and the data movement are by far the most difficult and time consuming parts of SPMD programs to get right, and these are exactly the parts that are not required in implicitly parallel programs.

Control replication allows us to both "have our cake and eat it", extending the performance range of the implicitly parallel style so that programs written in this style can achieve scalability and performance comparable to hand-written SPMD code. Control replication leverages a combination of static and dynamic analysis to generate long-running shards that amortize the overhead of executing large numbers of tasks. Intuitively, the control flow of the original implicitly parallel program is *replicated* across the shards, with each shard maintaining enough state to mimic the decisions of the original control thread. An important feature of control replication is that it is a local transformation, applying to a single collection of loops. Thus, it need not be

applied only at the top level, and can in fact be applied independently to different parts of a program and at multiple different scales of nested parallelism.

As suggested above, the heart of the control replication transformation depends on the ability to analyze the implicitly parallel program with sufficient precision to generate the needed synchronization and data movement between shards. Similar analyses are known to be very difficult in traditional programming languages. Past approaches that have attempted optimizations with comparable goals to control replication have relied on either very sophisticated, complex and therefore unpredictable static analysis (e.g., HPF) or have relied much more heavily on dynamic analysis with associated run-time overheads (e.g., inspector-executor systems [54]).

A key aspect of our work is the interaction between the programming language and control replication. Regent leverages recent advances in parallel programming model design that greatly simplify and make reliable and predictable the static analysis component of control replication. Many parallel programming models allow programmers to specify a *partition* of the data, to name different subsets of the data on which parallel computations will be carried out. Recent proposals allow programmers to define and use *multiple* partitions of the same data [13, 20]. For example, returning to our abstract example in Figure 1.1a, one loop may be accessing a matrix partitioned by columns while the other loop accesses the same matrix partitioned by rows. Control replication relies on the programmer to declare the data partitions of interest (e.g., rows and columns). The static analysis is carried out only at the granularity of the partitions and determines which partitions may share elements and therefore might require communication between shards. The dynamic analysis optimizes the communication at runtime by computing exactly which elements they share.

An important property of this approach is that the control replication transformation is guaranteed to succeed for any programmer-specified partitions of the data, even though the partitions can be arbitrary. Partitions name program access patterns, and control replication reasons at the level of those coarser collections and their possible overlaps. This situation contrasts with the static analysis of programs where the access patterns must be inferred from individual memory references; current techniques, such as polyhedral analyses, work very well for affine index expressions [21], but do not

address programs with more complex accesses.

We present an optimizing compiler for Regent, which includes the control replication optimization. We evaluate Regent, and control replication, using five codes: a circuit simulation on an unstructured graph, an explicit solver for the compressible Navier-Stokes equations on a 3D unstructured mesh, a Lagrangian hydrodynamics proxy application on a 2D unstructured mesh, a stencil benchmark on a regular grid, and a turbulence and particle solver on a 3D structured grid. Our implementation achieves up to 99% parallel efficiency on 1024 nodes (12288 cores) on the Piz Daint supercomputer [7] while providing overall performance comparable hand-tuned MPI(+X) reference codes (where available).

This work makes the following key contributions:

- Chapter 2 presents the design of Regent, a practical programming language for implicitly parallel, task-based programming with logical regions.
- Chapter 3 describes control replication. To the best of our knowledge, we are the first to demonstrate the impact of programming model support for multiple partitions on a compiler analysis and transformation. We show that this feature can be leveraged to provide both good productivity and scalability.
- Chapter 4 discusses the translation from Regent programs to the Legion runtime API. Regent is the first, to the best of our knowledge, to target a dynamic task-based runtime in this way.
- Chapter 5 considers novel optimizations required to ensure that the translation from Regent to Legion is efficient.
- Chapter 6 presents an implementation of the Regent programming language and discusses various details of this implementation.
- Chapter 7 provides a qualitative evaluation of Regent by presenting from first principles the design of an application in Regent.
- Chapter 8 contains a quantitative evaluation of Regent. For the class of applications considered, we are the first (to the best of our knowledge) to provide

practical levels of scalability for general-purpose implicitly parallel programs. We demonstrate scaling to 48x more cores than the previous best known techniques (12288 vs. 256 cores) while maintaining performance competitive with hand-written MPI(+X) codes.

Following this, Chapter 9 describes Regent in the broader context of related work, and Chapter 10 concludes.

Chapter 2

Programming Model

Regent is a programming language with support for both implicit and explicit parallelism, making it possible to describe both the base language and the output of various optimizations, such as control replication transformation, entirely within one system. In particular, Regent's support for multiple partitions of data collections enables a particularly straightforward analysis of data movement required for efficient SPMD code generation. In this section, we discuss the Regent programming model, focusing on features relevant to the control replication transformation. Initially, we describe the implicitly parallel subset, then consider extensions for explicit parallelism.

2.1 Regent Example

Continuing from the example in Section 1.2, we consider a implementation of the same algorithm in Regent and discuss the features used in the Regent implementation of the code.

Figure 2.1 shows a Regent version of the program in Figure 1.1a. The two inner loops with calls to point functions `F` and `G` have been extracted into tasks `TF` and `TG` on lines 1-6 and 8-13, respectively. These tasks identify the granularity at which Regent will consider a parallel execution of the code. In this case, we have selected tasks in the obvious way by partitioning the iteration spaces of the loops into blocks of an appropriate size. The main simulation loop has been rewritten to call these

```
1 task TF(B : region(SU, ...), A : region(SU, ...))
2 where reads writes(B), reads(A) do
3     for i in SU do
4         B[i] = F(A[i])
5     end
6 end
7
8 task TG(A : region(SU, ...), B : region(_, ...))
9 where reads writes(A), reads(B) do
10    for j in SU do
11        A[j] = G(B[h(j)])
12    end
13 end
14
15 -- Main Simulation:
16 var U = ispace(0..N)
17 var I = ispace(0..NT)
18 var A = region(U, ...)
19 var B = region(U, ...)
20 var PA = block(A, I)
21 var PB = block(B, I)
22 var QB = image(B, PB, h)
23 for t = 0, T do
24    for i in I do
25        TF(PB[i], PA[i])
26    end
27    for j in I do
28        TG(PA[j], QB[j])
29    end
30 end
```

Figure 2.1: Regent version of program with aliasing.

tasks. The arguments to tasks have also also been partitioned to identify the subsets of the data being passed to each parallel task.

A central concern of the Regent programming language is the management and partitioning of data. Data in Regent is stored in *regions*. A region is a (structured or unstructured) collection of objects and may be *partitioned* into subregions that name subsets of the elements of the parent region.

Lines 18 and 19 declare two regions `A` and `B` that correspond to the arrays by the

same name in the original program. These regions contains elements of some data type indexed from 0 to `N` $-$ 1. (The element data type does not matter for the purposes of most Regent optimizations and analysis.) The declaration of the *index space* `U` on line 16 gives a name to the set of indices for the regions; symbolic names for sets of indices are helpful because in general regions may be structured or unstructured, and are not necessarily indexed contiguously. Note that memory allocation for regions is lazy. No actual memory allocation occurs at lines 18-19. Instead the program proceeds to partition the regions into subregions so that the eventual memory allocations are distributed across the machine.

Lines 20-22 contain calls to partitioning operators. The first two of these, on lines 20 and 21, declare block partitions of the regions `A` and `B` into roughly equal-sized subregions numbered 0 to `NT` $-$ 1. (As before, a variable `I` is declared on line 17 to name this set of indices.) The variables `PA` and `PB` name the sets of subregions created in the respective partitioning operations. For convenience, we name the object which represents a set of subregions a *partition*.

Line 22 declares a second partition `QB` of the region `B` based on the image of the function `h` over `PB`. That is, for every element `b` of region `B`, `h(b)` $\in$ `QB[i]` if `b` $\in$ `PB[i]`. This partition describes exactly the set of elements that will be read inside the task `TG` on line 11. Importantly, there are no restrictions on the form or semantics of `h`. As a result, `QB` may not be a partition in the mathematical sense; i.e. the subregions of `QB` are not required to be disjoint, and the union of subregions need not cover the entire region `B`. In practice this formulation of partitioning is extremely useful for naming the sets of elements involved in e.g. halo exchanges.

Regent supports a number of additional operators as part of an expressive sub-language for partitioning, described in more detail in Section 2.3.2. In the general case, Regent partitions are extremely flexible and may divide regions into subregions containing arbitrary subsets of elements. For the purposes of control replication and most other Regent optimizations, the only property of partitions that is necessary to analyze statically is the disjointness of partitions. A partition object is said to be *disjoint* if the subregions can be statically proven to be non-overlapping, otherwise the partition is statically *aliased*. For example, the block partition operators on lines

20-21 produce disjoint partitions as the subregions are always guaranteed to be non-overlapping. For the image operator on line 22, the function `h` is unconstrained and thus Regent assumes that the subregions may contain overlaps, causing the resulting partition to be considered aliased.

Note that the subregions of a statically aliased partition may not necessarily overlap at runtime; i.e., due to the nature of the conservative approximations in the language, a statically aliased partition might not be dynamically aliased. As the compile-time optimizations in Regent must rely on this static approximation, it is possible for some compiler optimizations to fail statically that might succeed if more dynamic information were available. However, Regent is quite lenient with respect to statically aliased partitions, and in practice with control replication and other optimizations, this has not been a limiting factor. In addition, as described in Section 6.1, in cases where static optimizations are not possible, Regent falls back to a runtime-based implementation which is able to recover most parallelism dynamically.

The main simulation loop on lines 23-30 then executes a sequence of task calls with the appropriate subregions as arguments. Tasks declare privileges on their region arguments (*read*, *write*, or *reduce* on an associative and commutative operator). Execution of tasks is apparently sequential. Two tasks may execute in parallel as long as they operate on disjoint regions, or with compatible privileges (e.g. both read, or both reduce with the same operator). Regent programs are typically written such that the inner loop can execute in parallel; in this case the loops on lines 24-26 and 27-29 both execute in parallel.

Note that in Regent, unlike in the fork-join parallel execution of Figure 1.1c, there is not an implicit global synchronization point at the end of each inner loop. Instead, Regent computes the dependencies directly between pairs of tasks (as described above) and thus tasks from different inner loops may execute in parallel if doing so preserves sequential semantics.

An important property of Regent tasks is that privileges are *strict*. That is, a task may only call another task if its own privileges are a superset of those required by the other task. Similarly, any reads or writes to elements of a region must conform to the privileges specified by the task. As a result, a compile-time analysis such as control

replication need not consider the code inside of a task. All of the analysis for control replication will be at the level of tasks, privileges declared for tasks, region arguments to tasks, and the disjointness or aliasing of region arguments to tasks.

2.2 Execution Model

Now we consider the Regent programming model in more detail. Regent (in its default mode) is an implicitly parallel programming language; Regent programs have sequential semantics and the system (compiler and runtime) is responsible for automatically discovering parallelism in the program. In a naive sense this is sufficient for the system to achieve parallel execution of the program. However, in order to achieve efficient execution, additional information is required from the user. In particular, the user must decompose the program into *tasks* and *regions* of appropriate granularity for execution on the target machine. Tasks divide program control (recursively) into subtasks, while regions divide program data (recursively) into subregions. These decompositions of the program often influence one another and are thus frequently decided together. For clarity of the following discussion, we consider tasks (and division of control) first, then regions (and division of data).

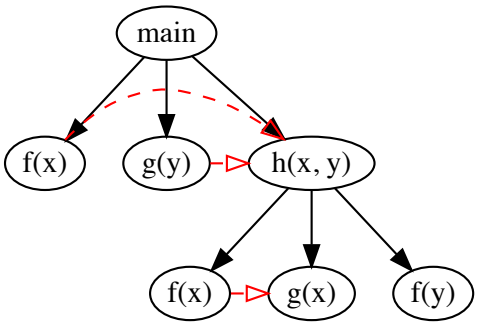
2.2.1 Tasks

A *task* is the fundamental unit of control in Regent. Tasks resemble functions in traditional programming languages with formal parameters and a body. Figure 2.2a shows an example program with four tasks.

Execution in the example begins at the top of the task `main` and follows standard sequential semantics. Tasks may call subtasks (lines 13-15), and subtasks may recursively call their own subtasks. Thus the program can be seen as being decomposed into a tree of tasks as shown for the example in Figure 2.2b. Potential parallelism exists between the children of a given parent task in the tree. Two sibling tasks are allowed to run in parallel when the system can prove them to *non-interfering*. The non-interference of two tasks is determined (conceptually) by comparing all pairs of

```
1 task f(a : region) where reads writes(a) do ... end
2 task g(b : region) where reads writes(b) do ... end
3 task h(c : region, d : region)
4   where reads writes(c, d) do
5   f(c)
6   g(c)
7   f(d)
8 end
9
10 task main()
11   var x = region(...)
12   var y = region(...)
13   f(x)
14   g(y)
15   h(x, y)
16 end
```

(a) A program with four tasks.



(b) A task tree. Parent-child relationships are shown with solid arrows, and sibling dependencies are shown with dashed red arrows.

Figure 2.2: A Regent program and corresponding task tree.

actual arguments to those tasks for interference. Specifically, a pair of task arguments is non-interfering when:

1. the two regions are disjoint,
2. the privileges requested are both read or both reduce with the same operator,
3. or the privileges name disjoint *fields* within the objects of the two respective regions.

Conversely, whenever the arguments of two tasks cannot be proven to be non-interfering, a dependence exists between the tasks. In practice, this formulation is more useful, as dependencies form a DAG which can directly guide the execution of a program. Dependencies are computed automatically using the rules for non-interference between tasks. Two restrictions in the programming model allow this analysis to be tractable: First, tasks in Regent are only permitted to access data passed via formal parameters, so any data accesses can be safely determined by examining the arguments supplied to tasks. Second, the ways in which data are used are identified explicitly via a task's privileges.

Note that this analysis can be performed at either compile time or at runtime. At compile time, the compiler may not have full information about aliasing in the program, but even a conservative analysis can often enable useful optimizations. Regardless of the result of any compile-time analysis, the runtime will determine the precise dependencies at runtime to exploit any latent parallelism which may be available.

Dependencies between the tasks in the example are shown as dashed red lines in Figure 2.2b. The two calls to `f` and `g` from `main` are independent, because they use distinct data. (In this example, we assume that `x` and `y` are regions that do not share any common elements.) The subsequent call to `h` is dependent on both previous tasks. The call to `g` from `h` is dependent on the call to the same task from `main` because dependencies for parent tasks apply transitively to children.

In general, the arguments passed to tasks may be complex expressions naming non-trivial collections of elements, and the language for describing such collections of elements is quite expressive. As such the problem of finding dependencies between

tasks requires in the general case a dynamic analysis which can be expensive when the number of subtasks being executed is large. Chapter 3 considers an optimization which is able to leverage partial static information reduce the cost of this analysis. First, however, we consider the language features for data partitioning which enable Regent to be so expressive.

2.3 Data Model

Regent programs consist of a decomposition of control into tasks, and a parallel decomposition of data into *regions*, or collections of data elements. Regions are typically used as arguments to tasks, and thus designate the sets of elements those tasks are intended to access. On distributed-memory machines, regions also frequently denote sets of elements to be allocated in on-node memories, or to be communicated between memories on distant nodes, both of which can have a significant impact on performance. As such, it is of the utmost importance for Regent to provide sufficiently expressive features for describing the sets of elements contained in regions.

2.3.1 Regions

Regions are containers of data elements, similar to arrays of objects in traditional programming languages or relations in relational databases. Regions map indices in an *index space* to objects consisting of multiple fields in a *field space*. Index spaces may be structured collections of multi-dimensional points (e.g. 1D, 2D, or 3D), or unstructured (i.e. sets of unordered elements). With these abstractions regions are able to describe a variety of data structures such as regular grids, unstructured meshes, and graphs.

Regions are called *logical* data structures because unlike traditional data structures such as arrays they are not allocated immediately in memory and may be moved as necessary between nodes or even exist simultaneously on multiple nodes. At runtime the data which is logically contained in a region is stored in zero or more *physical instances* of the region. When a region is created, there will be zero instances of the

region until the contents are initialized by calling a task with write privilege on the region. Thereafter, there will always be at least one physical instance with valid data somewhere in the machine. The mapping from logical regions to physical instances is managed automatically by the Regent implementation and is not exposed to the user at the level of the Regent source code. Instead, all performance decisions, including control over the placement of physical instances, are made available to the user via a mapping interface discussed further in Section 4.1.3.

2.3.2 Partitions

Partitions name sets of subregions, where subregions each name subsets of the elements of a given parent region. Partitioning can be applied recursively, resulting in a tree-shaped hierarchical decomposition of the data structures in an application. Typically, partitions are used to name the subregions to be used by data-parallel subtasks, and in a distributed machine the elements shared in common between overlapping subregions will need to be communicated over the network. As a result, it is extremely important for efficiency that Regent provide a set of expressive constructs for describing the sets of elements that belong to the respective subregions of a partition.

Partitioning in Regent is very expressive; in general, the subregions of a partition may contain arbitrary subsets of the elements of the region being partitioned. This expressivity enables Regent to handle a great many classes of problems including dynamic or unstructured data structures and to precisely identify the sets of elements that are required for communication in such problems.

Regent features two distinct interfaces for partitioning. The first, based on *colorings*, allows the user to construct an explicit map from colors (small integers) to sets of points naming the contents of the respective subregions [68]. Examples of this style of partition are shown in Figure 7.1; each of the regions is divided into three subregions denoted by the colors red, orange and blue. In the case of the upper-left coloring, the three colors map to non-overlapping sets of elements, thus the resulting partition is *disjoint*. On the other hand, the bottom-right coloring assigns multiple colors to some of the elements, resulting in an *aliased* partition where the subregions of the partition

overlap. The disjoint or aliased property of a partition is important enough that it must be named explicitly by the user and is tracked in the Regent type system. This approach enables maximum flexibility in partitioning, but is also completely opaque to the compiler, and thus any invariants on the subregions of the partition (most importantly, disjointness) must be checked dynamically.

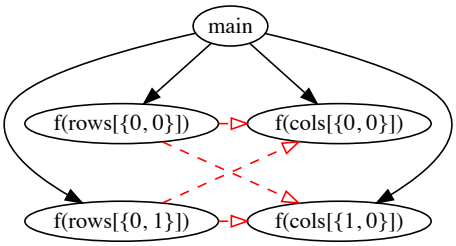
The second interface for partitioning provides an expressive sublanguage of operators for computing partitions [70]. An example of this style of partitioning is shown in Figure 2.1, where the `block` and `image` operators are used to create a blocked partition, and then to derive an image partition from the initial blocked partition via some arbitrary function. In particular, this sublanguage has been carefully chosen to permit various forms of static analysis on the partitioning code, which enables Regent to check more partitioning invariants at compile-time. The sublanguage includes operators for computing disjoint-by-construction partitions from e.g. field data stored in a region (useful when calling the ParMetis [60] graph partitioning library), for computing partitions through functions or the fields of regions (as in the `image` operator shown in the figure), and for common set operations (union, intersection, etc.). These operations are sufficient for many classes of computations such as unstructured meshes and graph applications which perform nearest-neighbor accesses on pointer data structures.

In practice, these operators are often the simplest and most convenient way to define partitions. For example, the *equal* partitioning operator divides a region into a number of equal sized subregions. Figure 2.3a uses equal partitioning to divide a 2-dimensional grid into chunks of rows and columns (lines 7 and 8). These partitions are trivially disjoint due to the nature of equal partitioning, thus the tasks called in the loop on line 9 are able to run in parallel with respect to each other (respectively line 10).

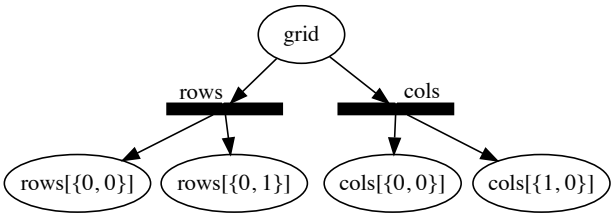
Note also that the partitions on lines 7 and 8 both come from the same parent region. Regent allows a given region to be partitioned an arbitrary number of times. Because the subregions of different partitions may overlap (as they in fact do in Figure 2.3a), tasks on such subregions are considered to be interfering (if either task writes its region). Thus, the tasks on line 10 depend on the tasks on line 9.

```
1 task f(r : region(...)) where reads writes(r) do ... end
2
3 -- Main Simulation:
4 var N = 8
5 var B = 2
6 var I = 0..N × 0..N
7 var R = 0..1 × 0..B
8 var C = 0..B × 0..1
9 var grid = region(I, ...)
10 var rows = partition(equal, grid, R)
11 var cols = partition(equal, grid, C)
12 for i = 0, B do f(rows[{0, i}]) end
13 for j = 0, B do f(cols[{i, 0}]) end
```

(a) Code for an `N` × `N` region partitioned into `B` rows and `B` columns.



(b) Task tree for code sample.



(c) Region tree for code sample.

Figure 2.3: A partitioning scheme for rows and columns of a grid.

Figure 2.3b shows the dependencies that result from the execution of this program. Each dependency in the execution will result in data movement if the program is executed on a distributed machine.

2.3.3 Privileges

Privileges describe the ways in which a task may use a region argument: may the task *read*, *write*, or *reduce* (with an associative and commutative reduction operator) the elements of the region? Privileges are expressed in the declaration of a task and enable both type checking and tractable and effective static and dynamic analysis of Regent programs. The use of privileges in the language ensures that the compiler need not attempt to analyze bodies of tasks in order to determine the effects of that task (i.e. all possible reads and write a task may perform), as a task's effects are completely described by the task's privileges. This also improves the performance of dynamic analysis of Regent programs, as even when runtime information is available, it may be prohibitively expensive to consider all possible pointer reads and writes in a task.

While regions and partitions are first-class values in Regent, privileges are not. The privileges in a task's declaration must completely describe the behavior of a task. Thus, in general, any called subtasks (or other operations such as accesses to region elements) must use a subset of privileges of the parent task. The only caveat to this rule is that when a task creates an entirely new region, the task responsible for creating the region gains full read-write privileges on the region. These rules enforce a form of stack discipline in the usage of regions in Regent programs. It is worth noting that this stack discipline also enables the composability of Regent programs; in general it is always safe to call a task that may itself recursively call subtasks, as the task will be held to the privileges it declares. This is in contrast to explicitly parallel programming paradigms, where composability is not guaranteed to be safe when using nested parallel constructs.

In the presence of partitions, this leads to an additional slight complication. In Figure 2.3a at line 9, why is it ok for `main` to call `f(rows[0, i])`? The access to `rows[0, i]` is safe because it is a subregion of `grid`, which was created on line 6 of the same task (and therefore has read-write privileges available). This is in turn known because `rows[0, i]` is a subregion of the partition `rows`, which itself partitions `grid`.

Regent tracks these relationships between regions via a *region tree* which captures parent-child relationships between regions. Figure 2.3c shows the region tree the compiler builds for the code sample. Region trees are a useful tool in the analysis of

regions, and are discussed further in Section 3.2.

2.4 Features for Explicit Parallelism

In normal usage, Regent programs execute with sequential semantics. Regent also provides features that extend Regent's sequential programming model with explicitly parallel features for synchronization and communication. Although these features can be used for many purposes, they are most commonly used to reduce dynamic analysis cost by manually (or automatically) sharding Regent programs over multiple long-running tasks. In this way, a Regent program can achieve constant analysis overhead while scaling to large numbers of nodes (and therefore large numbers of tasks).

Chapter 3 describes the technique required to apply this transformation to a Regent program automatically. In the remainder of this section, we consider the features which make this transformation (whether automated or manual) possible.

2.4.1 Must-Parallel Epochs

In Regent, two tasks are permitted to run in parallel when they are mutually non-interfering; however, in standard Regent there is no mechanism for specifying that two tasks *must* run in parallel. This property is critical, for example, when two tasks are involved in manual synchronization. In this case, a failure to execute the tasks concurrently could result in deadlock. For example, consider a case where the available memory limits Regent to running only one task at a time. If one task attempts to synchronize with the other, the system will deadlock. (In sequential Regent code this situation cannot occur because all synchronization is implicit and managed by the system.) Thus, before allowing the user to write code with explicit synchronization and communication, Regent must first provide a way to specify when tasks must run in parallel.

A *must-parallel epoch* specifies a set of tasks which must run in parallel. These tasks must be mutually non-interfering. Some additional techniques described below permit

tasks to be non-interfering even when multiple tasks in a must-parallel epoch request write privilege on the same region. Non-interference is required so that the tasks can be scheduled simultaneously by the runtime. To avoid a resource deadlock as described above, all resource constraints for the tasks must also be satisfied simultaneously. This places some constraints on the mapping of tasks in a must-parallel epoch. Mapping is discussed in more detail in Section 4.1.3.

2.4.2 Coherence

A must-parallel epoch requires that all tasks within the epoch be non-interfering. However, under Regent's normal semantics, two tasks that request write privilege on the same region are necessarily interfering. Thus, in order to write explicitly parallel programs in Regent it is necessary to relax this constraint.

Coherence modes specify the degree of consistency required of region arguments two tasks. Regent provides four coherence modes:

- *Exclusive* coherence is the default and follows Regent's standard sequential semantics. A sequence of tasks with exclusive coherence are guaranteed to execute in a manner which is indistinguisahble from sequential execution.
- *Atomic* coherence permits tasks to be reordered. Two tasks with atomic coherence can execute in either order, but must still execute one at a time.
- *Simultaneous* coherence permits tasks to be executed concurrently. Region arguments with simultaneous coherence guarantee shared-memory semantics, although it is still the responsibility of the tasks to synchronize individual memory accesses (e.g. with atomic instructions).
- *Relaxed* coherence permits tasks to be executed concurrently, and there are no guarantees on the semantics of regions. All synchronization must be provided by the user for safe execution.

The use of coherence modes other than exclusive results in a progressive relaxation of the dependencies between tasks, and thus permit increasing degrees of reordering

or concurrency among sibling tasks in order to achieve explicitly parallel execution. The permitted interleavings of various combinations of coherence modes is described in [68].

Note that Regent considers dependencies only between sibling tasks of a single parent task. This is safe under Regent's standard implicitly parallel semantics because the children of distinct non-interfering parent tasks are themselves trivially non-interfering (by the inclusion property of privileges described above). This enables, among other things, a safe distributed analysis of task dependencies among non-interfering tasks. However, in the presence of coherence modes other than exclusive, this property can be violated. Regent makes no attempt to compute dependencies between the grandchildren tasks of two sibling tasks using e.g. simultaneous coherence—these dependencies become the user's responsibility. Thus, when writing explicitly parallel code, it is important for the user to consider what dependencies are or are not being tracked automatically by Regent, and in cases where those dependencies are not tracked automatically, to add manual synchronization and communication.

Among explicitly parallel Regent programs, the most commonly used coherence mode is simultaneous. This mode permits concurrent execution, but requires shared-memory semantics for any regions. This is a useful abstraction because it means that two concurrently executing tasks both have access to a single instance of that region in memory. Regent provides explicit copy operations that can be used to copy data to and from instances of regions located on remote nodes.

2.4.3 Explicit Copies

In implicitly parallel Regent programs, copies between distant memories are scheduled automatically whenever a true dependence exists between tasks that execute on distant processors. However, when using simultaneous or relaxed coherence, the computation of dependencies is relaxed, and therefore these copies must be scheduled manually by the user (or by compiler transformation).

Regent supports explicit copy operations for this purpose. While copies are most often used in explicitly parallel Regent programs, they are well-defined in the sequential

case as well. A copy operation behaves like a task which reads elements from one region (the source) and writes the values to corresponding elements of the destination region. However, a copy is permitted to operate on regions stored in a remote memory, whereas a task would only be permitted to operate on local copies of remote regions. When used in combination with simultaneous or relaxed coherence, copy operations must be explicitly synchronized with tasks that read or write their results, otherwise data races can occur.

2.4.4 Phase Barriers

Explicit copies permit the movement of data between distant memories. However, accesses to this data are not safe from data races unless additional synchronization is used. The user is free to use whatever synchronization mechanism they prefer. However, Regent offers a built-in synchronization mechanism, called a *phase barrier*, which is attractive for this use case.

Phase barriers are a reusable, non-blocking, N-M producer-consumer synchronization mechanism. Phase barriers are unlike MPI barriers in that the use of phase barriers never blocks the currently executing task. Instead, operations (tasks or copies) are given phase barriers as preconditions or postconditions. For example, a task may be issued such that it does not start until the barrier has triggered. The task is said to *wait* on the barrier, although this does not block execution in the traditional sense. (There may generally be M of these waiting operations.) Similarly, a task may be issued with the barrier as a postcondition, in which case it is said to *arrive* on the barrier. The arrival is deferred until the task actually completes. A barrier is considered to be triggered when N operations arrive at the barrier. The arrival count may be modified at runtime as long as the task attempting to alter the arrival count is itself responsible for at least one arrival. Similarly, the set of waiting tasks may be entirely dynamic.

Barriers may be reused; the *advance* operation returns the subsequent generation of the current barrier. Operations can be scheduled on barriers multiple generations in advance without blocking the current task, allowing all operations to be scheduled

asynchronously from the actual execution of the program.

2.4.5 Dynamic Collective

In many applications, it is necessary to perform reductions on the values of scalar variables, for example to compute a new value for *dt* for the subsequent time step in a simulation loop. In explicitly parallel, SPMD-style implementations of such applications, this requires an additional synchronization primitive. A *dynamic collective* is a variation on a phase barrier where the tasks which arrive on the collective are permitted to supply values, which are then reduced and broadcast to all tasks waiting on the collective. This allows a dynamic collective to achieve behavior similar to an `MPI_Allreduce`, except that as with phase barriers the use of a collective is simply as a precondition or postcondition to a task and thus does not block the main thread of execution of the application.

Chapter 3

Control Replication

Control replication is an optimization that transforms implicitly parallel programs with sequential semantics into scalable and efficient SPMD code, even in the presence of dynamically determined partitioning of data and communication patterns.

In the absence of this optimization, the overhead from launching increasingly large sets of tasks comes to dominate execution time at large node counts. This is a direct consequence of Regent's sequential semantics: tasks must be analyzed in program order in order to preserve the original semantics of the code. However, for repetitive programs where the parallelism in the inner loop of task launches can be determined statically, control replication can be used to avoid a sequential bottleneck in the analysis of tasks. The goal of control replication is to automatically generate a set of *shards*, or long-running tasks, which are each responsible for a subset of the tasks in the original program. Shards execute in an explicitly parallel, SPMD-style fashion. Among the subtasks launched by a single shard, Regent's normal sequential semantics applies. However, whenever a subtask of a shard depends on data produced by another shard, the compiler must generate explicit synchronization and communication to preserve that dependence. Thus, much of the focus in control replication is on discovering and generating code for efficient synchronization and data movement.

While control replication relies on the compiler to statically determine when loops are able to execute in parallel, the technique notably does *not* require the compiler to statically determine the precise patterns of communication in the application. In

particular, while performing control replication, the compiler need not be aware of exactly which shards will need to communicate during the program's execution, or exactly what sets of elements they will exchange. Instead, control replication reasons at the level of partitions, which have been explicitly identified by the user to name the relevant sets of elements in the application. The use of language-level partitioning enables control replication to be applied to classes of codes which have historically been difficult to analyze and optimize in a compiler, such as unstructured mesh codes where the exact structure of the mesh (and therefore communication pattern of the application) cannot be known until the program's input is read. The analysis of the precise communication pattern of the application is deferred until runtime, when the structure of the application's partitions is known. Because this analysis is only performed once, before the creation of shards, it does not impact the overall scalability of long-running applications.

3.1 Target Programs

For the purposes of control replication we consider programs containing forall-style loops of task calls such as those on lines 24-26 and 27-29 of Figure 3.1 (duplicated, for ease of reference, from Figure 2.1). Control replication is a local optimization and need not be applied to an entire program to be effective. The optimization is applied automatically to the largest set of statements that meet the requirements described below. In the example, control replication will be applied to lines 23-30 of the program.

Control replication applies to loops of task calls with no loop-carried dependencies except for those resulting from reductions to region arguments or scalar variables. Arbitrary control flow is permitted outside of these loops, as are statements over scalar variables.

No restrictions are placed on caller or callee tasks; control replication is fully composable with nested parallelism in the application. The compiler analysis for control replication need not be concerned with the contents of called tasks because the behavior of a task is soundly approximated by the privileges in the task's declaration.

```
1 task TF(B : region(SU, ...), A : region(SU, ...))
2 where reads writes(B), reads(A) do
3     for i in SU do
4         B[i] = F(A[i])
5     end
6 end
7
8 task TG(A : region(SU, ...), B : region(_, ...))
9 where reads writes(A), reads(B) do
10    for j in SU do
11        A[j] = G(B[h(j)])
12    end
13 end
14
15 -- Main Simulation:
16 var U = ispace(0..N)
17 var I = ispace(0..NT)
18 var A = region(U, ...)
19 var B = region(U, ...)
20 var PA = block(A, I)
21 var PB = block(B, I)
22 var QB = image(B, PB, h)
23 for t = 0, T do
24    for i in I do
25        TF(PB[i], PA[i])
26    end
27    for j in I do
28        TG(PA[j], QB[j])
29    end
30 end
```

Figure 3.1: Regent version of program with aliasing.

Similarly, any caller task is completely agnostic to the application of control replication because any possible transformation of the code must be consistent with the task's privileges.

The region arguments of any called tasks must be of the form `p[f(i)]` where `p` is a partition, `i` is the loop index, and `f` is a pure function. Any accesses with a non-trivial function `f` are transformed into the form `q[i]` with a new partition `q` such that `q[i]` is `p[f(i)]`. Note here that we make essential use of Regent's ability to

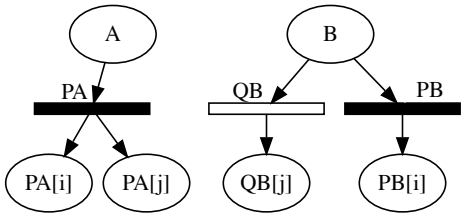


Figure 3.2: Region tree for the example. Filled boxes are disjoint partitions.

define multiple partitions of the same data.

3.2 Region Trees

An analysis of aliasing between regions is critical for determining what tasks can be permitted to execute in parallel. The semantics of Regent enables a straightforward implementation of such an analysis based on the relationships between regions and partitions. To determine whether two regions may alias, the compiler constructs a *region tree* that describes these relationships. This tree is a compile-time adaptation of the runtime data structure described in [13].

Figure 3.2 shows the region tree for the code in Figure 3.1. Note that regions in this formulation are *symbolic*, that is, the indices used to identify subregions are either constants or unevaluated loop variables. A dynamic evaluation of this program would result in an expansion of this tree for the various iterations of the loops (resulting in e.g. `PA[0]`, `PA[1]`, ..., `PA[NT-1]` under the `PA` partition). However, the number of iterations is not available at compile-time, making the symbolic version necessary.

The region tree is convenient because it provides a natural test to determine whether any two regions may alias: For any pair of regions R and S, find the least common ancestor A with immediate children R' and S' (along the path to R and S, respectively). If A is a disjoint partition and R' and S' are indexed by constants, then R and S are guaranteed to be disjoint regions at runtime; otherwise they may alias.

Region trees can be constructed by walking the program source from top to bottom. Each newly created region becomes the root of a fresh region tree. Partitions are inserted under the region they partition, and expressions that access subregions

of partitions result in the corresponding subregion nodes, tagged with the index expression used.

3.3 Program Transformation

In this section we describe the program transformations that comprise the control replication optimization. Over the course of the optimization, the program is restructured to avoid the assumption of the sequential semantics that Regent normally provides, such that the long-running shards that are finally generated operate with explicit distributed memory, maintaining the coherence of regions explicitly via explicit communication and synchronization.

Consider a subregion S and its parent region P. Semantically, S is literally a subset of P: an update to an element of S also updates the corresponding element of P. There are two natural ways to implement this region semantics. In the *shared memory implementation* the memory allocated to S is simply the corresponding subset of the memory allocated to P. In the *distributed memory implementation*, S and P have distinct storage and the implementation must explicitly manage data coherence. For example, if a task writes to region S, then the implementation must copy S (or at least the elements that changed) to the corresponding memory locations of P so that subsequent tasks that use P see those updates; synchronization may also be needed to ensure these operations happen in the correct order. Intuitively, control replication begins with a shared memory program and converts it to an equivalent distributed memory implementation, with all copies and synchronization made explicit by the compiler.

3.3.1 Data Replication

The first stage of control replication is to rewrite the program so that every region and subregion has its own storage, inserting copies between regions where necessary for correctness. We use the shorthand $R_1 \leftarrow R_2$ for an assignment between two regions: R_1 is updated with the values of R_2 on the elements $R_1 \cap R_2$ they have in common.

```
1 -- Initialization:
2 for i in I: PA[i] ← A
3 for i in I: PB[i] ← B
4 for i in I: QB[i] ← B
5
6 -- Transformed code:
7 for t = 0, T do
8     for i in I: TF(PB[i], PA[i])
9     for i, j in I × I: QB[j] ← PB[i]
10    for j in I: TG(PA[j], QB[j])
11 end
12
13 -- Finalization:
14 for i in I: A ← PA[i]
15 for i in I: B ← PB[i]
```

(a) Code after data replication.

```
1 -- Initialization:
2 for i in I: PA[i] ← A
3 for i in I: PB[i] ← B
4 for i in I: QB[i] ← B
5 var IQPB = {i, j|QB[j] ∩ PB[i] ≠ ∅}
6
7 -- Transformed code:
8 for t = 0, T do
9     for i in I: TF(PB[i], PA[i])
10    for i, j in IQPB: QB[j] ← PB[i]
11    for j in I: TG(PA[j], QB[j])
12 end
13
14 -- Finalization:
15 for i in I: A ← PA[i]
16 for i in I: B ← PB[i]
```

(b) Code with intersections.

```
1 -- Initialization:
2 for i in I: PA[i] ← A
3 for i in I: PB[i] ← B
4 for i in I: QB[i] ← B
5 var IQPB = {i, j|QB[j] ∩ PB[i] ≠ ∅}
6
7 -- Transformed code:
8 for t = 0, T do
9     for i in I: TF(PB[i], PA[i])
10    barrier()
11    for i, j in IQPB: QB[j] ← PB[i]
12    barrier()
13    for j in I: TG(PA[j], QB[j])
14 end
15
16 -- Finalization:
17 for i in I: A ← PA[i]
18 for i in I: B ← PB[i]
```

(c) Code with synchronization.

```
1 -- Shard task:
2 task shard(SI, SIQPB, PA, PB, QB)
3 where reads writes simult(PA, PB, QB) do
4     for t = 0, T do
5         for i in SI: TF(PB[i], PA[i])
6         barrier()
7         for i, j in SIQPB: QB[j] ← PB[i]
8         barrier()
9         for j in SI: TG(PA[j], QB[j])
10    end
11 end
12
13 -- Initialization as before
14 -- Transformed code:
15 var X = ispace(0..NS)
16 var SI = block(I, X)
17 must_parallel_epoch for x in X do
18    var SIQPB = {k, j|k, j ∈ IQPB ∧ k ∈ SU[x]}
19    shard(SI[x], SIQPB, PA, PB, QB)
20 end
21 -- Finalization as before
```

(d) Code with shards.

Figure 3.3: Regent program at various stages of control replication.

Figure 3.3a shows the core of the program in Figure 3.1 after three sets of copies have been inserted. Immediately before the code to which the optimization is applied (lines 7-11), the various partitions are initialized from the contents of the parent regions (lines 2-4). Symmetrically, any partitions written in the body of the transformed code must be copied back to their respective parent regions at the end (lines 14-15). Finally, inside the transformed code, writes to partitions must be copied to any aliased partitions that are also used within the transformed code. Here `PB` and `QB` are aliased (i.e. subregions of `PB` may overlap subregions of `QB`), so `PB` must be copied to `QB` on line 9 following the write to `PB` on line 8. Note that `PA` is also written (on line 10) but can be proven to be disjoint from `PB` and `QB` using the region tree analysis described in Section 3.2, thus no additional copies are required.

3.3.2 Copy Placement

The placement of the copies in Figure 3.3a happens to be optimal, but in general the algorithm described in Section 3.3.1 may introduce redundant copies and place those copies suboptimally. To improve copy placement, we employ variants of partial redundancy elimination and loop invariant code motion. The modifications required to the textbook descriptions of these optimizations are minimal. Loops such as lines 8-10 of Figure 3.3a are viewed as individual statements operating on partitions. For example, line 8 is seen as writing the partition `PB` and reading `PA` (summarizing the reads and writes to individual subregions). Note that the use of standard compiler techniques is only possible because of the problem formulation. In particular, aliasing between partitions is removed by the data replication transformation in Section 3.3.1, and program statements operate on partitions which hide the details of individual memory accesses.

3.3.3 Copy Intersection Optimization

Copies are issued between pairs of source and destination regions, but only the intersections of the regions must actually be copied. The number, size and extent of such intersections are unknown at compile time; this is an aspect of the analysis that

is deferred until runtime. For a large class of high-performance scientific applications, the number of such intersections per region is $\mathcal{O}(1)$ in the size of the overall problem and thus for these codes an optimization to skip copies for empty intersections is able to reduce the complexity of the loop on Figure 3.3a line 9 from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. Figure 3.3b shows the code following this optimization.

To avoid an $\mathcal{O}(N^2)$ startup cost in comparing all pairs of subregions in the computation of intersections at line 5 in Figure 3.3b, we apply an additional optimization (not shown in the figure). The computation of intersections proceeds in two phases. First, we compute *shallow* intersections to determine which pairs of regions overlap (but not the extent of the overlap). For unstructured regions, an interval tree acceleration data structure makes this operation $\mathcal{O}(N \log N)$. For structured regions, we use a bounding volume hierarchy for this purpose. Second, we compute *complete* intersections between these known-intersecting regions. Following the creation of shard tasks in Section 3.3.5 these operations are performed inside the individual shards, making them $\mathcal{O}(M^2)$ where M is the number of non-empty intersections for regions owned by that shard.

In practice, at 1024 nodes, the impact of intersection computations on total running time is negligible, especially for long-running applications. Section 8.3.6 reports running times for the intersection operations of the evaluated applications.

3.3.4 Synchronization Insertion

When moving to a distributed-memory semantics, it is necessary to synchronize on copies performed between remote nodes. A naive version of this synchronization is shown in Figure 3.3c. The copy operations on line 11 are issued by the producer of the data. Therefore, on the producer's side only, copies follow Regent's normal sequential semantics. Explicit synchronization is therefore only required for the consumer. A naive implementation of this synchronization could be performed with traditional barriers as shown in Figure 3.3c. Two barriers are used in the example on lines 10 and 12. The first barrier on line 10 preserves write-after-read dependencies and ensures that the copy does not start until all previous consumers of `QB` (i.e. `TG` tasks from the previous iteration of the outer loop) have completed. The second barrier on line 12

preserves read-after-write dependencies and ensures that subsequent consumers of `QB` (i.e. subsequent `TG` tasks) do not start until the copy has completed.

As an additional optimization (not shown), the traditional barriers are replaced with point-to-point synchronization via phase barriers (described in Section 2.4.4). In particular, the tasks which require synchronization are exactly those with non-empty intersections computed in Section 3.3.3. A simple dataflow analysis determines all consumers of `QB` preceding the copy on line 11 and all those following; these tasks synchronize with copies on line 11 as determined by the non-empty intersections computed in `IQPB`. This form of synchronization in Regent has the additional benefit that the phase barriers can be added as direct preconditions or postconditions to tasks and therefore do not block the main thread of control as would a traditional barrier.

3.3.5 Creation of Shards

In the final stage of the transformation, control flow itself is replicated by creating a set of shard tasks that distribute the control flow of the original program. Figure 3.3d shows the code after the completion of the following steps.

First, the iterations of the inner loops for `TF` and `TG` must be divided among the shards. Note this division does not determine the mapping of a task to a processor for execution, which is discussed in Section 6.2. This simply determines ownership of tasks for the purposes of runtime analysis and control flow. The assignment is decided by a simple block partition of the iteration space on line 14. Second, the compiler transforms the loops so that the innermost loops are now over iterations owned by each shard, while the new outermost loop on line 15 iterates over shards.

Third, the compiler extracts the body of the shard into a new task on lines 2-11. This task is called from the main loop on line 19. Note that the shard task requests the use of simultaneous coherence (described in Section 2.4.2) in order to permit the shards to execute in parallel, despite the conflicts between the read-write privileges of the various tasks. The main task uses a must-parallel epoch (described in Section 2.4.1) to assert that the tasks must be scheduled for parallel execution on the machine.

3.4 Implementation

In discussing control replication, we have been largely concerned with a limited subset of a Regent features. Most additional features of Regent are straightforward to implement within control replication, though a couple warrant special attention.

3.4.1 Region Reductions

Control replication permits loop-carried dependencies resulting from the application of associative and commutative reductions to region arguments of tasks. These reductions require special care in an implementation of control replication.

The partial results from the reductions must be stored separately to allow them to be folded into the destination region, even in the presence of aliasing. To accomplish this, the compiler generates a temporary region to be used as the target for the reduction and initializes the contents of the temporary to the identity value (e.g., 0 if the reduction operator is addition). The compiler then issues special *reduction copies* to apply the partial results to any destination regions which require the updates.

3.4.2 Scalar Reductions

In control replication, scalar variables are normally replicated as well. This ensures, for example, that control flow constructs behave identically on all shards in a SPMD-style program. Assignments to scalars are restricted to preserve this property; for example, scalars cannot be assigned within an innermost loop (as the iterations of this loop will be distributed across shards during control replication).

However, it can be useful to perform reductions on scalars, for example, to compute the *dt* for the next time step in a code with dynamic time stepping. To accommodate this, control replication permits reductions to scalars within inner loops. Scalars are accumulated into local values that are then reduced across the machine with a dynamic collective (described in Section 2.4.5), an asynchronous collective operation that supports a dynamically determined number of participants. The result is then broadcast to all shards.

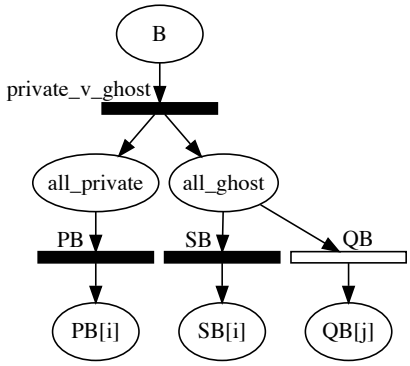


Figure 3.4: Region tree with hierarchical partitions.

3.4.3 Hierarchical Region Trees

Regent permits recursive partitioning of regions. Among many other uses, this feature enables a common idiom in which the programmer constructs a top-level partition of a region into two subsets of elements: those which are guaranteed to never be involved in communication, and those which may need to be communicated. This design pattern, in combination with the region tree analysis described in Section 3.2, enables an important communication optimization that reduces data movement for distributed-memory execution, and also substantially reduces the cost of the dynamic computation of intersections described in Section 3.3.3.

Figure 3.4 shows a possible modification to the region tree from Figure 3.2 that uses this optimization. The top-level region `B` has been partitioned into two subregions that represent all the *private* elements (i.e. those never involved in communication) and *ghost* elements (i.e. those that are involved in communication). The new partition `SB` represents the subset of elements of the original `PB` partition involved in communication. Similarly, the new `PB` and `QB` partitions have been intersected with the regions `all_private` and `all_ghost`.

Notably, the top-level partition in this new region tree is disjoint, and thus by consulting the region tree the compiler is able to prove that the partition `PB` is disjoint from `QB` and `SB`. As a result, the compiler is able to prove that the subregions of `PB` are not involved in communication (as they are provably disjoint from all other subregions), and can avoid issuing copies for `PB`. Additionally, because `PB` has been excluded from

the set of partitions involved in communication, the compiler is able to skip any intersection tests with `PB` and other partitions. As in most scalable applications the set of elements involved in communication is usually much smaller than those not involved in communication, so placing the private data in its own disjoint subregion can reduce the runtime cost of computing intersections.

An application of this technique to a more full-featured application code is described in detail in Section 7.2.

Chapter 4

Translation to Legion

Regent programs may be highly dynamic: in particular, the precise dependencies between tasks, sets of elements contained in regions (and overlapping elements between pairs of regions), and the scheduling of tasks on processors and placement of regions in distributed memories may be dynamic and not amenable to static analysis. To leverage this dynamic behavior, Regent targets a dynamic runtime for task-based parallelism, Legion [13].

Legion, though also a task-based system, provides lower-level abstractions compared to Regent. Several features which are implicit in Regent, such as the management of the memory associated with a region, are more explicit in Legion. The Regent compiler is responsible for managing the translation into this more explicit model.

In this chapter we describe the relevant features of the Legion runtime system and how they differ from Regent, and then present a translation from the Regent language to Legion runtime APIs. Note that several aspects of the translation described in this chapter result in potentially suboptimal performance. These are addressed through further optimizations described in Chapter 5.

4.1 Features of the Legion Runtime System

Legion is implemented as a *software out-of-order processor*. Tasks (like instructions) are scheduled for possibly out-of-order execution on a set of physical resources. Similarly,

regions (like registers) are virtualized. A given region may be mapped to zero or more *physical instances* in memory at any given point in the program execution. Thus in Legion there is a distinction between between *logical* and *physical* abstraction layers in Legion that does not exist in Regent. Separating the logical and physical levels permits important patterns, such as having multiple copies of read-only data, to be expressed directly.

The Legion C++ API allows programmers to write efficient task-based programs that run out-of-order, asynchronously, and in a distributed fashion. However, because Legion is written in C++, which does not understand the semantics of tasks and regions, the Legion API is forced to expose functionality beyond the logical layer of the programming model. Programmers must generally write Legion programs with some awareness of both the logical and physical levels. In contrast, Regent only exposes the logical level, and the compiler is responsible for managing the translation from logical to physical abstractions.

As a running example used throughout this chapter, Figures 4.1 and 4.2 show excerpts from an implementation of the proxy application PENNANT in Regent and Legion. The implementation of PENNANT is described in more detail in Chapter 7. For the purposes of the present discussion, these code comparisons serve to highlight the substantial differences in usability between Regent and Legion. The specific aspects of the code samples are discussed along with the relevant features, below.

4.1.1 Regions

A region is the product of an index space (set of indices) and a field space (set of fields). Like an array of structs in a traditional language, a region holds a value for every index in the index space, for each field in the field space. At any given point in the apparently sequential execution of the program, a region can be as a snapshot mapping indices to values. However, unlike an array in a conventional programming language, there is not necessarily a one-to-one mapping between a region and its representation in physical memory.

As noted above, a region may correspond to zero or more physical instances (actual

```
task adv_pos_full(points : region(point), dt : double) where
  reads(points.{x0, u0, f, maswt}), writes(points.{x, u})
do
  var fuzz = 1e−99
  var dth = 0.5 ∗ dt
  for p in points do
    var pap = (1.0 / max(p.maswt, fuzz))∗p.f
    var pu = p.u0 + dt∗pap
    p.u = pu
    p.x = p.x0 + dth∗(pu + p.u0)
  end
end
```

(a) PENNANT leaf task implementation in Regent.

```
void adv_pos_full(const Task ∗task,
                  const std::vector<PhysicalRegion> &regions,
                  Context ctx, HighLevelRuntime ∗runtime)
{
  PhysicalRegion points0 = regions[0];
  Accessor<double, SOA> points_x0_x(points0, PX0_X);
  Accessor<double, SOA> points_x0_y(points0, PX0_Y);
  Accessor<double, SOA> points_u0_x(points0, PU0_X);
  Accessor<double, SOA> points_u0_y(points0, PU0_Y);
  Accessor<double, SOA> points_f_x(points0, PF_X);
  Accessor<double, SOA> points_f_y(points0, PF_Y);
  Accessor<double, SOA> points_maswt(points0, PMASWT);
  PhysicalRegion points1 = regions[1];
  Accessor<double, SOA> points_x_x(points1, PX_X);
  Accessor<double, SOA> points_x_y(points1, PX_Y);
  Accessor<double, SOA> points_u_x(points1, PU_X);
  Accessor<double, SOA> points_u_y(points1, PU_Y);
  Future f0 = task−>futures[0];
  double dt = f0.get_result<double>();
  double fuzz = 1e−99;
  double dth = 0.5 ∗ dt;
  IndexIterator it(points0.get_logical_region().get_index_space());
  while (it.has_next()) {
    size_t count;
    ptr_t start = it.next_span(count);
    ptr_t end(start.value + count);
    for (ptr_t p = start; p < end; p++) {
      double frac = (1.0 / max(points_maswt.read(p), fuzz));
      double pap_x = frac ∗ points_f_x.read(p);
      double pap_y = frac ∗ points_f_y.read(p);
      double pu_x = points_u0_x.read(p) + dt ∗ pap_x;
      double pu_y = points_u0_y.read(p) + dt ∗ pap_y;
      points_u_x.write(p, pu_x);
      points_u_y.write(p, pu_y);
      points_x_x.write(p, points_x0_x.read(p) +
                       dth∗(pu_x + points_u0_x.read(p)));
      points_x_y.write(p, points_x0_y.read(p) +
                       dth∗(pu_y + points_u0_y.read(p)));
    }
  }
}
```

(b) PENNANT leaf task implementation in Legion C++ API.

Figure 4.1: PENNANT leaf tasks in Regent and C++.

```
var dt, dtmax = conf.dtmax, conf.dtmax
var dthydro = 0.0
var time, tstop = 0.0, conf.tstop
while time < tstop do
  dt = calc_global_dt(dt, dtmax, dthydro, time, tstop)
  for i = 0, conf.npieces do
    adv_pos_full(points_all_private_p[i], dt)
  end
  for i = 0, conf.npieces do
    dthydro min= calc_dt_hydro(zones_all_p[i], dt, dtmax)
  end
  time += dt
end
```

(a) Excerpt from PENNANT main simulation loop in Regent.

```
Future dt = Future::from_value<double>(conf.dtmax);
Future dthydro = Future::from_value<double>(0.0);
double dtmax = conf.dtmax;
Future time = Future::from_value<double>(0.0);
double tstop = conf.tstop;
runtime->unmap_region(ctx, pr_points_all_private);
while (time.get_value<double>() < tstop) {
  double buffer[2];
  buffer[0] = dtmax;
  buffer[1] = tstop;
  TaskArgument global_args0((void *)&buffer[0], sizeof(buffer));
  TaskLauncher launcher0(CALC_GLOBAL_DT, global_args0, ArgumentMap());
  launcher0.add_future(dt);
  launcher0.add_future(dthydro);
  launcher0.add_future(time);
  dt = runtime->execute_task(ctx, launcher0);

  Domain domain = Domain::from_rect<1>(
    Rect<1>(Point<1>(0), Point<1>(conf.npieces - 1)));
  IndexLauncher launcher1(ADV_POS_FULL, domain, TaskArgument(), ArgumentMap());
  launcher1.add_region_requirement(
    RegionRequirement(points_all_private_p, 0 /* identity projection */, READ_ONLY, EXCLUSIVE, points_all_private));
  launcher1.add_field(0, PX0_X);
  launcher1.add_field(0, PX0_Y);
  launcher1.add_field(0, PU0_X);
  launcher1.add_field(0, PU0_Y);
  launcher1.add_field(0, PF_X);
  launcher1.add_field(0, PF_Y);
  launcher1.add_field(0, PMASWT);
  launcher1.add_region_requirement(
    RegionRequirement(points_all_private_p, 0 /* identity projection */, READ_WRITE, EXCLUSIVE, points_all_private));
  launcher1.add_field(1, PX_X);
  launcher1.add_field(1, PX_Y);
  launcher1.add_field(1, PU_X);
  launcher1.add_field(1, PU_Y);
  launcher1.add_future(dt);
  runtime->execute_index_space(ctx, launcher1);

  TaskArgument global_args2((void *)&dtmax, sizeof(double));
  IndexLauncher launcher2(CALC_DT_HYDRO, domain, global_args2, ArgumentMap());
  launcher2.add_region_requirement(
    RegionRequirement(zones_all_p, 0 /* identity projection */, READ_ONLY, EXCLUSIVE, zones));
  launcher2.add_field(0, ZDL);
  launcher2.add_field(0, ZVOL0);
  launcher2.add_field(0, ZVOL);
  launcher2.add_field(0, ZSS);
  launcher2.add_field(0, ZDU);
  launcher2.add_future(dt);
  dthdyro = runtime->execute_index_space(ctx, launcher2,
                                         REDOP_ADD_DOUBLE);

  TaskLauncher launcher3(ADD_DOUBLE,
                         TaskArgument(), ArgumentMap());
  launcher3.add_future(time);
  launcher3.add_future(dt);
  time = runtime->execute_task(ctx, launcher3);
}
```

(b) Excerpt from PENNANT main simulation loop in Legion C++ API.

Figure 4.2: Excerpt from PENNANT main simulation loop in Regent and C++.

instantiations of the region in memory). However, the full correspondence of a region involves some additional layers of complexity that must be managed by the application:

- One logical region
- corresponds to zero or more region requirements
- which each correspond to one or more physical instance
- containing one or more fields.
- And each field of a physical instance corresponds to exactly one accessor.

A *region requirement* (`RegionRequirement` in the Legion API) is the fundamental unit of privilege in Legion. A region requirement names a region, a privilege (read, write, etc.), and a set of fields. Most Legion APIs that involve an effect on a regions actually take region requirements. For example, the API for launching a task takes a list of region requirements rather than regions. Legion's internal mechanisms for tracking privileges operate at the level of region requirements. Examples of region requirements can be seen in Figure 4.2b lines 21-35 and 41-47; they are also implicit in the regions argument of Figure 4.1b.

Each region requirement corresponds to one or more physical instances. Physical instances are the actual unit of allocated memory in the system. As the objects contained inside regions may consist of multiple fields, a physical instance contains memory for a set (or subset) of fields of the elements in the region.

Each physical instance has a *layout* in memory: struct-of-arrays (`SOA`), or array-of-structs (`AOS`), etc. In certain cases, it may be beneficial to use different layouts for different sets of fields. To support this, Legion permits a region requirement to be mapped to multiple physical instances.

Finally, *accessors* are used to actually access the data contained in a physical instance. In C++ API, the `Accessor` type makes extensive use of C++ templates in order to amortize the cost of computing values such as strides and base pointers for accessing the physical instance, and to provide constant-folding of compile-time

information about instance layouts (such as strides, in certain layouts). These can be seen in Figure 4.1b lines 6-17.

A Legion programmer is responsible for managing the correspondence of a region into its constituent parts, and packing those pieces as necessary to call tasks, etc. Furthermore, when a parent task calls a subtask with a subregion of one of its regions, Legion requires that the programmer explicitly specify the parent region on which the parent task has privileges. Thus the programmer is responsible for tracking all parent-child relationships between regions. As described in Section 4.2.1, in Regent these correspondences are managed transparently by the compiler.

4.1.2 Index and Region Trees

In Legion, the distinction between an index space and a region on that index space is more explicit than it is in Regent. This distinction is particularly noticeable when partition a region into subregions. In Legion, partitioning is performed only on index spaces. Partitioning can be seen as dividing an index space into a tree of subspaces. This tree is mirrored implicitly in the region tree. That is, creating a partition of an index space implicitly creates a parallel partition on all regions of the index space.

4.1.3 Mapping

Regions are logical containers and must be *mapped* to one or more physical instances prior to use. Mapping can be critical to performance, and a correct decision can depend on architecture or application-specific factors. As such, Legion chooses to expose these decisions to the user via the `Mapper` API. The `Mapper` object is queried during the program execution whenever the runtime needs to map a region to a physical instance. Legion also provides a default implementation of the mapper that uses heuristics to provide a reasonable out-of-the-box experience.

There is an additional issue with respect to mapping and subtasks. By default, at the start of a task Legion automatically maps each region used by the task, and when the task ends each of those regions is unmapped. Before launching a subtask a parent task must also *unmap* (release access to the memory of) any region that the child task

needs to use. As the parent and child execute concurrently, this is needed to avoid data races due to concurrent access to regions shared between the parent and child. By default, the Legion runtime unmaps all of the parent's regions before calling a child and remaps them when the child terminates. While this default behavior guarantees correct execution, if the parent and child have interfering privileges for a region (e.g., both can write the region) then the parent will block until the child terminates, as the parent's call to map the region following the call must block until the child finishes. Blocking in the parent task is potentially harmful to performance as it prevents the parent task from running ahead of execution and thus restricts the parallelism which is available for Legion to exploit in the execution of the application. In the worst case, excessive mapping and unmapping can serialize the execution of the application.

For optimal performance, Legion programmers must explicitly manage the mapping of regions through explicit *map* and *unmap* calls provided by the Legion interface. By unmapping a region, the programmer notifies the runtime that the data in that region is not required by the parent task until a corresponding map call is issued. The map call causes the runtime to query the `Mapper` object to choose a new (or existing) physical instance, and then blocks the parent task until that instance becomes valid. In typical usage, programmers unmap all regions before entering a main loop, and remap all regions once the loop completes, which ensures that the runtime can avoid blocking when issuing tasks within that loop. An example of such an unmap call can be seen in Figure 4.2b line 6.

4.1.4 Tasks

Tasks are the fundamental unit of control in both Regent and Legion. Tasks are issued in program order, exactly as they are written in the text, and every possible program execution is guaranteed to be indistinguishable from serial execution. As discussed in Section 4.1.1, tasks specify the regions they use via a set of region requirements (consisting of a region, privilege, and set of fields).

Legion performs an analysis to determine when tasks *interfere* (i.e. perform conflicting operations on the same or aliased regions), building a dependence graph over the

tasks in the program as the program executes. Legion's dynamic dependence analysis imposes a cost with every task launched. This cost is proportional to the number of region requirements used to describe a task, and thus the calling convention used for tasks has a non-trivial impact on the overhead incurred by the Legion runtime.

To ensure that the runtime overhead stays off the critical path as much as possible, the Legion runtime is itself asynchronous and performs its analysis in parallel to the execution of the application [13]. The goal is for the runtime to run ahead of the application, issuing tasks and analyzing task interference in advance of when those tasks can actually run. Thus the overhead of the runtime analysis of tasks only becomes a factor in the overall execution time of the application if the total duration of the runtime analysis exceeds the total running time of the application, or if a blocking operation causes the analysis to be exposed on the critical path of the application. Pipeline stalls, blocking operations, and excessive analysis costs can all cause the runtime to fall behind relative to the application and hurt the performance of the application. Legion mitigates these issues by providing more sophisticated abstractions which can result in higher performance, but also have more complex semantics. These features are discussed below.

Leaf Tasks

Task execution and analysis in Legion is pipelined. In general, a task must complete a pipeline stage before it passes to the next stage. If a given stage stalls for any reason, that task and any tasks that depend on it also stall. Mapping, described in greater detail in Section 4.1.3, is one pipeline stage. When a task is mapped, physical instances are chosen for each of its region requirements.

Because tasks can execute subtasks, Legion must wait for all subtasks to map before it can consider a parent task to have completed mapping. In general the only way to know that a parent task cannot issue more subtasks is if the parent task has terminated. This can result in unnecessary pipeline stalls when the task in question is one that never intends to launch any subtasks.

Legion allows users to annotate tasks as *leaf tasks* if they launch no subtasks, a mechanism inherited from Sequoia [35]. In Legion, the runtime considers the mapping

of a leaf task to be complete once the task itself is mapped (even if the task has not begun execution), avoiding unnecessary pipeline stalls for dependent operations. Users of the Legion C++ API must manually annotate leaf tasks to avoid such stalls.

Futures

Tasks can produce results in one of two ways: via direct return values, or as a side-effect on a region argument. In Legion, operations can block whenever a parent task consumes the direct return value produced by one of its child tasks. Since blocking a parent task is undesirable, the Legion runtime provides ways of avoiding blocking on both kinds of task results.

When a task produces a direct return value, Legion returns immediately with a *future* representing the not-yet-produced result of the task. Parent tasks can block to obtain the value of a future, but Legion also supports passing futures as inputs to other subtasks without blocking in the parent task. In this way, the programmer can describe the flow of values between subtasks without blocking, allowing the runtime to run further ahead and hide runtime analysis costs. Futures are visible in the C++ sample codes in Figure 4.1b lines 18-19 and Figure 4.2b lines 13-15, 36, 48, and 54-55. Note that in tasks that take *both* future and immediate arguments, such as `calc_global_dt` in Figure 4.2b on lines 8-16, the future arguments must be explicitly filtered and added to the task launch separately from immediate arguments; the user is responsible for maintaining the consistency of this code with the implementations of the tasks themselves.

Index Launches

Even when execution does not stall in the runtime or block in the application, if the throughput of the dynamic analysis itself is not sufficient, the runtime can still fall behind. Legion provides a number of features that can be used to mitigate the cost of this analysis. Most notably, an *index space task launch* (or index launch) can substantially reduce the cost of analysis of a repetitive set of tasks.

Conceptually, an index launch simply represents a loop of task launches. Figure 4.2a

lines 6-8 and 9-11 show two examples of Regent loops that can be transformed into index launches. The corresponding C++ code is shown in Figure 4.2b lines 18-37 and 39-50. Because the tasks in an index launch are all similar, the cost of analyzing these tasks can be reduced. Note that the Legion runtime places several structural restrictions on index launches to ensure that they are well-behaved:

1. A *launch domain* (an index space) must be explicitly specified. One task is launched for each index in the launch domain.

2. Arguments to all tasks in the index launch must be computed outside the launch, guaranteeing that arguments are available and that no arguments depend on side-effects from tasks within the launch.

3. Futures, if any, are added to the launch as a whole, not to individual tasks.

4. Region requirements can be in one of two forms:

   - Individual region requirements name a single region to be used by all tasks in the launch.
   - Partition requirements name a subregion of the partition per task in the launch (such as `p[i]` for each index `i` in the launch domain). If the index expression is non-trivial (e.g. `p[f(i)]` for a non-identify function `f`), then the user must supply a *projection functor* implementing `f`.

5. Because an index launch implies parallel execution, all the tasks must be non-interfering. That is, the region requirements must be mutually disjoint, or use non-interfering privileges (read-only, or reductions).

6. If tasks within the launch return a value, then the launch as a whole is allowed to either return a map with all the resulting futures, or to reduce the futures into a single value via a user-specified reduction operator.

When executing an index launch, the runtime still performs dynamic checks to ensure that the tasks within the launch are non-interfering. However, Legion is able

to amortize these checks across the entire index launch instead of performing them individually. Note that, while this reduces the cost of analysis for N tasks to $\mathcal{O}(1)$, the overall cost of launching N tasks is still $\mathcal{O}(N)$ because a single node is still responsible for all tasks. In particular, on a distributed-memory machine, this requires one node to send N messages to other nodes informing them of what tasks to execute. While index launches improve the scalability of Regent programs, control replication, described in Chapter 3 is generally required for scaling to very large numbers of nodes.

4.1.5 Variants

Instance layouts can have a significant impact on the performance of tasks. Furthermore, the optimal instance layout (and corresponding implementation of a task) may depend on the architecture of the machine. Legion permits multiple *variants* of a task to be registered simultaneously. During execution, the mapper is able to dynamically select the appropriate variant to execute for each task.

Variants are distinguished from each other by a set of *layout constraints* describing the layout that the variant expects. In general, high-performance variants are expected to specify their layout constraints in great detail so that the implementation can constant-fold information such as the strides of the physical instances into the implementation code. Legion programmers are responsible for supplying variants, describing the associated layout constraints, and ensuring that the layout constraints match the expectations of the variant implementations.

4.2 Code Generation from Regent into Legion

Regent only exposes the logical aspects of the programming model to the user. Features such as physical instances, futures, and variants are managed transparently by the compiler. In Regent, unlike in Legion, users really can think of the program as simply a sequential code with tasks as function and region as arrays of structs.

The initial translation from Regent to Legion does not attempt to be optimal. Instead, a number of features are provided by subsequent optimizations to the code.

These optimizations are described in detail in Chapter 5.

4.2.1 Regions

The constituent parts of regions, index spaces and field spaces, must be managed through a series of API calls in Legion. In Regent, these are much more streamlined. Index spaces are created automatically for each region based on the size of the region. Field spaces are created automatically from the element type of the region. In cases where a region contains a nested struct (e.g. a struct containing a struct), Regent automatically flattens the nested struct in the resulting field space; Legion does not support nested fields.

When partitioning a region, the corresponding index space is implicitly partitioned; Regent hides the parallel index space and region trees from the user.

Regent manages the correspondence between regions, region requirements, physical instances, and accessors transparently on behalf of the user. This mapping is tracked in the code generator of the compiler and thus imposes no additional runtime overhead. These differences are illustrated in the difference between Figure 4.1a and Figure 4.1b.

When creating region requirements, Regent can automatically determine the correct parent region from which to derive privileges by consulting the compiler's static representation of the region tree for the task, along with the task's declared privileges. Region trees were previously discussed in Section 3.2.

4.2.2 Variants

Instance layouts can have a significant impact on the performance of Regent tasks. For performance, Regent generates high-performance variants of each task that are heavily optimized for specific instance layouts. By default, Regent generates a single variant for each task that assumes a default instance layout appropriate for vectorization and for constant-folding of the strides of physical instances. In future work we are interested in investigating the use of a static mapping language, which parallels the Legion mapping API, to allow the user to generate additional variants of tasks in Regent.

Each variant of a Regent task only works for a specific layout. Regent generates the appropriate layout constraints for Legion automatically to ensure that Legion generates instances in the correct layout for each task.

4.2.3 Task Calling Convention

Regent tasks are simply functions that operate on region arguments, and task calls are written as normal function calls.

In Legion, task calls are somewhat more involved. The various kinds of arguments—regions, futures, and immediates—must be separated and are attached separately to the task launch. Region arguments to tasks must be decomposed into region requirements where each region requirement is a tuple of a region, privilege, and a set of fields. As the cost of dynamic analysis in Legion is a function of the total number region requirements, rather than the number of task launches, the calling convention used has an impact on the cost of analysis of tasks. A naive approach, which simply builds a region requirement per region, privilege, and field, would have cost on the order of $\mathcal{O}(RF)$ where R is the number of regions used in task arguments and F is the average number of fields used per region. Instead Regent uses an optimized calling convention below which reduces this cost to $\mathcal{O}(R)$, which is the optimum.

Regent's calling convention operates as follows. Regent enumerates the privileges and fields declared in the target task. For each parameter, Regent maps the parameter region to the actual argument being passed to the task. Regent then collects 3-tuples containing (region, privilege, field) for all region arguments, and sorts these lexicographically. Regions are ordered by the position of the parameter in the original task. Privileges are ordered in the following way: reads, then writes, then reductions by reduction operator. Fields are ordered by the position of the field in the original field space. This ensures a stable, deterministic, and predictable ordering to region requirements. Regent then applies a group-by operator (borrowed from relational algebra) to group region requirements by region and privilege. This produces a set of fields for each region and privilege. Finally, due to a requirement in the Legion runtime, reduction privileges are split out into individual requirements, one per field.

Note that the predictability and determinism of the calling convention is also a necessary precondition to designing a useful foreign function interface for Regent. Such an interface is described in more detail in Section 6.3.

Futures also require some attention. Regent tasks may generally be called with either future or immediate (non-future) arguments. In the Legion C++ API, futures are passed separately from immediates, and the coordination of which arguments are passed in which mode is left to the programmer. Although it would be possible in the Regent compiler to generate different variants of a task for each call site, such an approach could potentially lead to an exponential increase in the amount of code compiled. Instead, Regent follows a calling convention where either futures or immediates can be passed to the same task. To support both future and immediate arguments to the same task, Regent tasks use an extra immediate argument to encode which subsequent arguments are being passed as futures. This argument is a bitmask where each bit represents a subsequent argument, if the bit is set to 1 then the argument is passed as future, otherwise 0. When unpacking arguments, Regent maintains a count of the number of future arguments unpacked up to that point, and increments the count each time a future argument is unpacked.

The correspondence between Regent and hand-written Legion code can be seen, for example, in Figure 4.2a line 5 and Figure 4.2b lines 8-16. The primary difference between the hand-written Legion code and the Regent calling convention is that the hand-written Legion code does not use an additional bitmask field to describe which parameters are passed in futures.

4.2.4 Additional Optimizations

Other aspects of optimal code generation are left to subsequent optimizations. The placement of map and unmap calls is discussed in Section 5.1, declaration of leaf tasks in Section 5.2, generation of index launches in Section 5.3, and futures in Section 5.4.

Chapter 5

Optimizations

As illustrated in Section 4, Regent simplifies the Legion programming model and provides a higher level of abstraction that is concerned only with logical, rather than physical, constructs. The Regent compiler is able to manage the correspondences between logical and physical constructs in a way that achieves performance comparable to a hand-written Legion C++ implementation, and significantly better than a naive compiler. This section describes a number of optimizations that together allow the Regent compiler to achieve performance comparable to hand-tuned code written to the Legion C++ API.

5.1 Mapping Elision

Regent frees programmers of the burden of managing physical instances of regions by statically computing correct and optimal placements of map and unmap calls. The Regent type system guarantees that the compiler has complete information about what regions can be accessed within any task. The compiler uses this information to perform a flow-sensitive analysis over the AST to determine the spans over which regions are used and inserts the map and unmap calls at the boundaries of spans when switching between usage in a parent and a child task. Redundant map and unmap calls resulting from repeated task launches are eliminated entirely, and the placements of map and unmap calls is chosen such that the blocking map calls occur

as late as possible in the program. In the case where a region is not used at all within a task, the compiler issues a single unmap call at the top of the task and leaves the region unmapped for the entire duration of the task's execution. In contrast, the Legion runtime, in the absence of manually placed calls to map and unmap, is forced to continue to map and unmap the region throughout the task's execution.

5.2 Leaf Tasks

As discussed in Section 4.1.4, correctly identifying leaf tasks is an important optimization for Legion programs, as otherwise the Legion runtime must consider the mapping of a task still in progress until it can be certain all child tasks have mapped (which is only known to be the case when the task itself finishes executing, as a task can in general continue to launch subtasks as long as it is still executing). Regent automatically infers at compile time which tasks are leaf tasks. The compiler knows all call targets and is therefore able to determine, using a flow-insensitive analysis, whether a given task calls any subtasks. These annotations are guaranteed to be correct and precise, in contrast to the user-provided leaf task annotations in Legion.

5.3 Index Launches

Whenever possible, the Regent compiler transforms loops of task launches into index space task launches. The analysis for this optimization proceeds in multiple phases:

1. The compiler begins with a structural analysis of the code to determine whether the loops in question are eligible for transformation into an index space launch. Currently all simple loops containing single task launches are considered eligible.

2. For each loop, the compiler determines whether the body of the loop (aside from the task call itself) is side-effect free. In particular, the loop body must not read or modify data that the task itself might read or modify. Doing so would introduces a loop-carried dependence and shows the loop is not fully parallelizable.

3. For each argument to the task launch, the compiler determines whether the argument in question is eligible to be transformed into an argument for an index task launch. Arguments must be one of:

   - a non-region value;
   - a region value that is provably loop invariant;
   - a region value that is provably an analyzable function of the loop index; i.e., it is an expression such as a partition access $p[i]$ indexed by the loop variable i.

4. The compiler then performs a static variant of Legion's dynamic non-interference analysis. For each region-typed argument, the compiler determines whether it is statically non-interfering with other region-typed arguments. As with the dynamic analysis, the compiler has several dimensions along which to prove non-interference:

   - disjointness, either because the region types are incompatible, or because the compiler can statically prove disjointness through the static region tree (Section 3.2);
   - field disjointness, because the arguments use different fields; or
   - privileges, because both arguments use compatible privileges (e.g. both read-only, or both reductions with the same reduction operator).

If the analysis determines that a task launch is eligible for optimization, the compiler emits the code to perform the index task launch.

It is worth noting that while this optimization looks similar in its basic outlines to forall-style constructs in other languages and programming models, it behaves quite differently in many respects. In particular, when index launch optimization fails (because any of the properties above cannot be established), that does not imply the resulting code runs sequentially. The Legion runtime will perform its standard dynamic analysis, and will parallelize all tasks that are dynamically non-interfering, regardless of whether the compiler performs the optimization or not. This optimization

simply allows the runtime to amortize the dynamic analysis costs in cases where the loops can be analyzed statically. Thus, Regent has a much more forgiving fallback for when static analysis is insufficient than language implementations that rely solely on static analysis.

5.4 Futures

In Legion, tasks can return futures, which can be passed to other tasks without blocking, allowing applications to build chains of asynchronous operations ahead of the actual computation. The Regent compiler can automatically lift variables and simple operations to futures to take advantage of these benefits. This optimization has three phases:

- The compiler first performs a flow-insensitive analysis to determine which variables are assigned to futures at any point within each task. Any such variables are automatically promoted to hold futures.
- The compiler then issues calls to automatically wrap and unwrap futures when storing a concrete value into a future-typed variable, or when reading a future-typed value because a concrete value is required. Tasks do not require arguments to be concrete, and can therefore be issued in advance of when the concrete values of futures are ready.
- Finally, the compiler emits tasks to allow simple side-effect free operations (such as arithmetic) to be performed directly on futures.

Note that the calling convention for Regent, discussed in Section 4.2.3 permits future and immediate arguments to be used interchangeably, and in any number and order. Thus this optimization can be applied aggressively without needing to be concerned for the number of futures that might or might not be passed to tasks.

5.5 Pointer Checks Elision

As noted in [68], static type checking of Legion programs allows certain classes of pointer checks to be elided. Regent preserves all the properties of the type system which make this possible. In particular, all pointer types in Regent explicitly contain one or more regions that they point into. Regent checks these annotations to ensure correctness at compile time, and elides the dynamic pointer checks, which are often prohibitively expensive at runtime.

5.6 Dynamic Branch Elision

In addition, Regent is able to elide certain classes of dynamic branches when accessing pointers in Legion. Pointers that can point into multiple different regions (e.g., private or ghost points in PENNANT, as described in Section 7.2) carry some dynamic tag bits encoding the region the pointer currently points to. In some cases, however, the memory for the two regions is actually co-located at runtime (e.g. because of a decision to map both regions to the same physical instance), allowing the dynamic branches on the tag bits to be elided. The compiler emits code that automatically detects such cases at runtime and selects the fast path when it is available.

5.7 Vectorization

Regent leaf tasks frequently feature loops over regions. In many cases, the Regent compiler is able to vectorize these loops automatically, often exceeding performance provided by traditional autovectorizers.

Regent performs runtime code generation to LLVM [48] via Terra [33]. While LLVM provides an autovectorizer, the low level of abstraction of the LLVM IR means that the vectorizer frequently misses vectorization opportunities or chooses the wrong optimization strategies for its vector code. Regent's native understanding of regions allows the vectorizer to make these decisions with improved precision. Regent uses Terra's built-in vector types to produce explicit vector instructions for LLVM, resulting

in significant performance gains in many cases.

Regent derives this advantage in precision from two sources. First, Regent has improved information about aliasing through type, field, and region-based analysis. In particular:

- While accesses for composite types are ultimately expressed as array accesses to fundamental types (integers, double-precision floating point, etc.), Regent is able to compare the original types to determine if there is potential for aliasing.
- Furthermore, even for identical types, Regent knows which fields are accessed and may be able to use this information to prove independence.
- Finally, when two accesses are to different regions, Regent may be able to use its knowledge of region disjointness to prove that accesses are independent.

Beyond this, Regent has access to implicit information about the costs of potential vectorization opportunities through regions. Regions are hierarchical and distributed data structures intended to provide opportunities for parallelism. Therefore, when Regent sees an outer loop over a region, and an inner loop (over something other than a region), Regent can infer with high confidence that the outer loop is the better opportunity for vectorization. In some cases, largely because it lacks comparable information for its cost model, LLVM chooses to vectorize the inner rather than outer loop, resulting in degraded performance.

5.8 OpenMP

In cases where Regent can generate vectorized code, the compiler can also automatically generate code to target other programming models, such as OpenMP. The primary benefit to using OpenMP in Regent is to reduce runtime overhead by reducing the number of tasks (e.g. producing one task per node instead of one task per core). Note that Regent is *not* a source-to-source compiler and does not make use of the C++ compiler in any way to generate OpenMP code. Instead, Regent directly targets the OpenMP ABI. This means that Regent can automatically generate OpenMP

tasks without using any user-level pragmas, and that Regent's OpenMP support, unlike OpenMP implementations in C++ and Fortran, is sound and cannot result in erroneous parallel execution of code for which such execution is not safe.

In practice, the underlying OpenMP implementation used in Regent is provided by Realm, the portability layer that Legion targets. Thus, Regent OpenMP code is not even linked against a conventional OpenMP runtime. This implementation strategy means that OpenMP tasks execute as tasks in the normal way, preserving task parallelism in the application. In addition, Realm's support for OpenMP permits multiple instances of OpenMP tasks to be executing simultaneously on different sets of processors, for example to take advantage of NUMA properties of the machine.

Chapter 6

Implementation

We have implemented an optimizing Regent compiler using Terra [33], a low-level programming language with semantics comparable to C, but with extensive and sophisticated support for metaprogramming via multi-stage programming [67]. Terra is embedded inside Lua [42], a high-level scripting language with first-class functions. Lua plays the same role for Terra that C++ templates play for C++, and provides many of the same benefits. However, Lua/Terra provides superior ease of use, because the metaprogramming language is a full programming language rather than C++'s restricted template language.

Terra uses LLVM [48] to provide efficient JIT compilation of Terra functions to fast machine code. As noted in Section 5.7, Terra makes it possible to perform vectorization and specialization with full awareness of the vector instruction set supported by the machine. The use of LLVM as the JIT compiler also allows both Terra and Regent functions to call and link easily against native C libraries.

Regent is implemented as a co-embedded language within Terra. The Terra API provides support for extending the parser with additional keywords, which when seen in the source program text cause Terra to invoke the embedded language compiler. Regent overloads a number of keywords—most notably, the `task` keyword—allowing the Regent language to interoperate seamlessly with both Lua and Terra. Regent tasks may call Terra functions and have access to all data types supported by Terra, including structs, arrays, and explicit vector types. The Regent compiler uses this

information to provide automatic structure slicing [14] for struct types stored inside logical regions. Regent tasks may also be dynamically specialized, using Lua, to provide multiple implementations, which are JIT compiled prior to starting the Legion runtime.

6.1 Runtime Support

In non-control replicated Regent programs, Legion discovers parallelism between tasks by computing a dynamic dependence graph over the tasks in an executing program. Control replication removes the need to analyze inter-shard parallelism, but Legion is still responsible for parallelism within a shard as well as any parallelism in the code outside of the scope of control replication.

A notable feature of Legion is its deferred execution model. All operations (tasks, copies, and even synchronization) execute asynchronously in the Legion runtime. This is an important requirement for supporting task parallelism, as it guarantees that the main thread of execution does not block and is subsequently able to expose as much parallelism as possible to the runtime system.

Legion targets Realm, a low-level runtime that supports execution on a wide variety of machines [69]. Realm uses GASNet [73] for active messages and data transfer.

6.2 Mapping

All tasks in Regent, including the shard tasks produced by control replication, are processed through the Legion *mapping interface* [13]. This interface allows the user to define a *mapper* that controls the assignment of tasks to physical processors, assignment of regions to physical instances in specific memories, and the layouts of instances. At the user's discretion, these decisions may be delegated to a library implementation. Legion provides a default mapper which provides sensible defaults for many applications.

When using control replication, a typical strategy is to assign one shard to each node, and then to distribute the tasks assigned to that shard among the processors

of the node. However, substantially more sophisticated mapper implementations are also possible; in general mappers are permitted to be stateful and/or dynamic in their decision making.

Regent and control replication are mostly agnostic to the mapping used. The exception is that Regent task variants place some constraints on the layouts of instances used, as described in Section 4.2.2. However, these constraints are provided by Regent to the runtime and thus any valid mapper decision is guaranteed to produce an acceptable layout for use in Regent.

6.3 Foreign Function Interface

In any practical system it is necessary to be able to interoperate with components written in other languages. Regent provides a foreign function interface (FFI) for this purpose. The Regent FFI supports two main use cases: calling C functions from Regent tasks, and calling Legion APIs.

6.3.1 Calling C Functions

Fortunately, Terra provides much of the support required for calling C functions. Terra can parse C header files via the `includec` built-in function, and can link dynamic shared objects with `linklibrary`. This is sufficient for calling functions of simple values such as `sin` and `printf`. Note that this also enables calling languages such as Fortran which can support a C calling convention.

However, it is also important to support calling C functions that operate on the contents of regions. For example, rather than implement matrix multiply in Regent, it would be more productive (and likely more efficient) to call an optimized version of the `dgemm` function. Regent provides a number of mechanisms to support this.

Typically, C functions that manipulate memory expect to receive pointers, possibly with strides or other layout information. Regent provides two functions to assist in this. The `__physical(R)` function returns an array of physical instances for the region `R`, one per field for which the current task has privileges. The `__fields(R)` function

returns an array of Legion field IDs for `R`. Regent does not provide a direct way to obtain a pointer to a field of `R` as there are potentially many ways to do so, and the choice will generally be application specific. However, with the physical instances and field IDs, applications can use the Legion APIs as described below to obtain accessors and/or raw pointers to memory.

6.3.2 Calling Legion APIs

Legion is written in C++, but also provides C functions that wrap the important entry points for the API. These are exposed in Regent via `regentlib.c`, and can be directly called inside Regent tasks.

Many Legion API calls require either a reference to the Legion runtime object or to the context handle of the current task. These can be accessed in Regent via the functions `__runtime` and `__context`. In addition Regent provides the function `__raw(R)` which returns the C API handle for many types of objects `R` such as regions, index spaces, partitions, etc.

With these functions, it is possible to write code in Regent which uses the Legion API directly. This enables Regent tasks to call Legion tasks written in C++ or other languages, and (along with the functions for obtaining instances above) to call non-Legion functions which support a C API.

6.3.3 Calling Regent Tasks from C++

No special support is required to call Regent tasks from Legion C++ code. The calling convention for Regent tasks is documented in Section 4.2.3 and can be followed in a straightforward way to generate calls to Regent tasks. The arguments to Regent tasks are packed, along with the bitmask to signal the use of futures, in a struct and follow the normal C++ rules for alignment of fields. In all other respects Regent tasks operate as normal Legion tasks: regions, futures, phase barriers, and dynamic collectives are all passed in the normal Legion manner.

6.3.4 Interactions with Optimizations

A number of FFI features interact with Regent's optimizations. For example, using the `__physical` function provides access to physical regions which Regent is not able to track, and thus using the `__physical` function inhibits the use of Regent's inner task optimization. Similarly, the use of `__context` inhibits Regent's leaf optimization as the context object enables the use of a large number of Legion API calls which might be invalid inside a leaf task. Notably, this restriction does not apply to the `__runtime` call as the runtime object alone is not sufficient to execute such API calls; this permits leaf tasks to use Legion API calls which query the region tree, but not modify it.

6.3.5 Generating Object Files

When integrating Regent with external applications, two approaches can be taken: either the Regent tasks can be compiled separately and linked into the external application, or the external application code can be compiled first and linked into Regent. Regent supports both approaches.

Regent provides a `regentlib.saveobj` function that compiles Regent tasks and produces either an executable binary (which can be useful for running Regent applications on machines where a dependence on LLVM would be problematic), or an object file. An object file can be linked in to an existing application as desired.

If external C or C++ code is to be used in Regent, this code must be wrapped in a pure C API and compiled as a shared library. Regent can then use the code by loading the appropriate header file with `includec` and then linking a shared library with `linklibrary`.

6.4 Metaprogramming

Metaprogramming is a technique for performing programmatic code generation. Terra supports metaprogramming via the Lua scripting language, in which it is embedded. Thus it is natural to extend Regent to support metaprogramming as well.

While not strictly a required feature, the advantage of metaprogramming in Regent is that it provides a natural way to support code generation for parallel programs and languages. For example, a compiler for a domain specific language could use Regent to automatically generate efficient and scalable parallel code. In contrast, a more traditional approach would be to generate code to a lower-level, explicitly parallel programming API such as pthreads or MPI. However, such an approach can be time consuming, as the semantic gap between the domain-specific language and these low-level parallel APIs can be quite large, and error-prone, as the compiler author is exposed to the same potential pitfalls as users of explicit parallelism are generally exposed. Worse, explicitly parallel APIs are typically not composable, making it difficult to construct systems out of multiple domain-specific languages. Even when targeting a dynamic, implicitly parallel runtime system such as Legion, there are a number of ways in which naive code generation can lead to poor performance. Regent provides easy-to-use sequential semantics and takes on responsibility for discovering parallelism in the program, and for any optimizations required to achieve performance. Regent also enables composability: calls to tasks from a domain-specific language can be inserted into arbitrary Regent code while preserving the intended semantics, enabling parallelism within and between domain-specific languages and other Regent libraries or user code.

In the remainder of this section we describe features available in Regent that enable metaprogramming.

6.4.1 Symbols, Quote and Escape

Regent provides three key operators for generating code and composing programs to produce larger programs.

Symbols name Regent variables, and can be generated with `regentlib.newsymbol(type, name)`. Both arguments are optional. If a type is not supplied, it will be inferred from the type of the variable initializer used in the variable declaration. A type is always required if the symbol is to be used as a parameter to a task.

Quotes represent ASTs for Regent expressions or statements. Regent provides

two quote operators: `rexpr ... end` and `rquote ... end` for expressions and statements, respectively. For example, the expression `rquote x += 1 end` describes a Regent statement that increments the variable `x`. As in Terra, `rexpr` and `rquote` are Lua expressions, and are intended to be used inside a Lua script to build up ASTs for the bodies of tasks.

Quotes can be composed using the *escape* operator `[...]`. The expression `...` inside the brackets is a Lua expression which is evaluated at the time that the quote itself is evaluated in Lua. The rules for lexical scoping of Regent quotes and escapes are identical to those used in Terra [33]. In the following code, the reference to `x` inside the escape is well-defined and refers to a Regent symbol: `quote var x = 0; [do_something(x)] end`. Quotes can composed to produce larger sets of expressions and statements, allowing entire tasks to be constructed programmatically.

6.4.2 Task Generation

Regent tasks need not be defined at the top level of the program, and can be created within arbitrary Lua code. This means that Regent tasks may be dynamically generated based on the inputs to the program. Note however that the `regentlib.start` call which begins execution of the Regent program does not return, and thus dynamic task generation is currently limited to the initial phase of the program execution, before any tasks have begun to execute.

Task bodies, parameter lists, and privileges can be generated dynamically with quotes. A task body must be a Regent expression or statement. Task parameters must be Regent symbols, or lists of symbols. Privileges are constructed via the function `regentlib.privilege(mode, region, field)` where mode is one of `regentlib.reads`, `regentlib.writes`, or `regentlib.reduces(op)`, region is a Regent symbol naming a region parameter, and field (optional) is a string naming a field within the region.

To assist in debugging, the Regent compiler provides a mode in which all tasks are automatically pretty-printed to the console. This can be used to inspect the generated code to ensure that the code being produced has the desired effect. This

mode can also be used to inspect the result of any optimizations performed by the Regent compiler—to ensure that all desired optimizations are being applied, and if necessary to debug issues in the Regent optimizations themselves.

6.4.3 Type, Dimension, and Field Polymorphism

Metaprogramming can also be used to achieve a variety of kinds of polymorphism which are otherwise not possible in non-metaprogrammed Regent. In particular, the types of parameters, types and dimensionality of regions, and privileges and sets of fields used in tasks, may all be customized via metaprogramming.

In Regent, all references to types (e.g. the `T` in `var x : T`) are Lua expressions. This is also true of types that appear in the declarations of task parameters, and of course in the type arguments to Regent symbols. This enables type polymorphism in the language, as any type expression can be replaced by a Lua variable or expression as needed. Lua functions can also be used to generate multiple copies of a task for different types, and Lua code can be used to generate the appropriate call sites for such tasks.

While the size and extent of regions in Regent is dynamic, the number of dimensions is a static property of the region's type in order to ensure that Regent can generate high-performance code. As a result, codes that aim to be polymorphic over dimensions should use metaprogramming as above to customize the types of regions to account for startup-time dynamic numbers of dimensions in regions.

The sets of fields in field spaces, and fields accessed in tasks, can also be customized via metaprogramming. This can be useful, for example, when a common task or piece of code is used repeatedly with different fields. The names of fields in Regent are represented as strings. Most places which accept a single field can also be used in metaprogramming with a list of fields.

Chapter 7

Case Study

In this chapter we consider the design and implementation of a non-trivial proxy application, PENNANT, in Regent. As a proxy application, PENNANT is designed to reflect the patterns of computation and memory accesses typical of a broader class of applications (in this case, unstructured mesh codes), while being small enough to allow the code to be ported easily to a variety of architectures and programming models. The reference PENNANT implementation, provided by Los Alamos National Lab, is written in C++ and can be configured to use MPI, OpenMP, MPI+OpenMP, or none of the above (for sequential execution), and has been heavily tuned for performance. The code is approximately 2500 lines (ignoring blank lines and comments). The initial Regent implementation of PENNANT was completed in under two weeks, including time to learn and understand the structure of the reference code. As PENNANT was also the first non-trivial code to be written in Regent, this also included time to make a number of minor adjustments to the Regent language. The implementation of PENNANT in Regent drove a number of design and implementation decisions in the compiler and motivated a number of optimizations that Regent provides.

7.1 PENNANT Overview

PENNANT is a Lagrangian hydrodynamics proxy application for unstructured meshes from Los Alamos National Laboratory [36]. PENNANT implements a subset of the

functionality in FLAG [25], a shock-hydro code used in production at Los Alamos. Compared to FLAG, PENNANT is restricted to 2D unstructured meshes (rather than 2D or 3D), and simulates only a subset of the physics in FLAG, explicitly excluding the shock portion of the code.

PENNANT simulates hydrodynamics using a 2D unstructured mesh. The fundamental constituents of the mesh in 0, 1, and 2 dimensions are called *points*, *edges* and *zones*. Because the mesh is unstructured, zones are polygons with an arbitrary number of edges. Rather than manage dynamically-sized lists of edges for each zone, PENNANT performs most operations on intermediary data structures called *sides* which represent the triangular area between an edge and the center of a zone. Sides contain pointers to zones and points, but not vice versa. Because PENNANT is a Lagrangian code, the points in the mesh move in the simulation space over the duration of the simulation, causing the mesh to deform. However, the logical structure of the mesh (i.e. the pointers between the mesh elements) does not change over time.

PENNANT includes several types of physics, which are computed in phases within each time step in the simulation:

1. First, the dt for the time step is determined, and the positions of points in the mesh are advanced halfway (i.e. by $\frac{1}{2}dt$ in time).

2. Second, various properties of sides and zones are computed, culminating in an accumulation of forces from sides into points.

3. Third, this force is used to accelerate points and compute the fully advanced positions of points (by the remaining $\frac{1}{2}dt$ in time).

4. Finally, zones are again updated, and the dt for the next time step is computed. In simulations such as PENNANT, the size of a time step depends on the physical properties of the mesh, and as the mesh can deform, these must be recomputed on each time step. This requires the use of a scalar reduction to compute dt, which has the potential to be a bottleneck at large node counts.

In a parallel and distributed implementation of PENNANT, the mesh must be

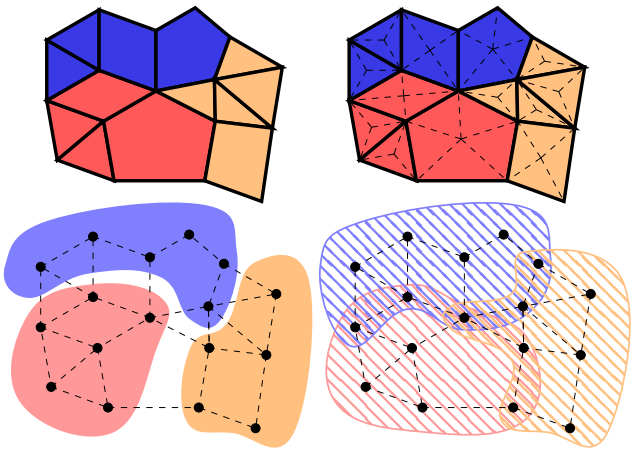


Figure 7.1: Naive PENNANT data partitioning: zones (left), sides (middle), and points: write sets of phases 1, 3 (bottom left) and read/reduce sets of phases 2, 4 (bottom right).

partitioned into submeshes in order to distribute work between the various processors of the machine. Figure 7.1 shows an example of a mesh that could be used with PENNANT; in this case the mesh has been divided into three submeshes. In both reference and Regent implementations, the primary partitioning of the mesh is over zones. Every zone belongs to exactly one submesh, as the access patterns in PENNANT do not require the values of zones to be communicated. This also leads to a straightforward partitioning of sides, as each side belongs exclusively to a zone, and again need not be communicated. However, the points of the mesh require communication as different phases of the application require different sets of points as shown on the bottom of Figure 7.1. Phases 1 and 3 of the simulation perform writes to the sets of points shown on the bottom left. Phases 2 and 4 perform either reads or reductions to sets of points on the bottom right; note that points at the boundaries between submeshes exist in the read/reduce sets of multiple tasks. Because of these overlapping access patterns, multiple processors may race to update the forces on these aliased points. In a shared-memory machine, these accesses may be mediated through atomic operations or other synchronization; no explicit communication is required. In a distributed-memory environment, the partial sums of forces must be communicated and synchronized. Both of these implementation details can be seen as a necessary

outcome of the fundamentally overlapping access patterns in the application. Thus a central concern in the design of an implementation of PENNANT is the management of the communication implied by such aliased access patterns.

In the OpenMP implementation of PENNANT, the required synchronization is handled by the implicit barrier at the end of each parallel loop; the accumulation of forces into points is managed by computing partial sums over sides and using a separate parallel loop to read these partial sums and compute the final forces on points. (An implementation using atomic operations would also be possible, but this approach is not taken in the OpenMP reference, in part for better consistency with the MPI implementation.) OpenMP relies on shared-memory semantics to avoid any need for explicit partitioning or data movement.

The MPI implementation requires additional work as the mesh must be explicitly distributed throughout the machine. Again, the partitioning of zones and sides is straightforward as these access patterns do not overlap and there is no need for communication. Points, however, require communication. Points at the boundaries between submeshes are duplicated, and one of the duplicates of each point is named the *master*. Other copies are named *slaves*. All computations on points are performed on the master copy and results communicated to the slaves. In the phase where forces are accumulated onto points, partial sums are computed on sides, these partial sums are communicated to the master, and the final sum computed for each point and then broadcast back out to slaves.

7.2 Regent Implementation

The primary concerns in the design of a Regent implementation are the decomposition of the program control into tasks, and the partitioning of regions into subregions that accurately name the elements to be used by the various tasks. A straightforward implementation would exploit Regent's sequential semantics to maintain the consistency of a single (conceptually shared) copy of the mesh. In this sense, the Regent implementation—despite the use of explicit partitioning—resembles a shared-memory implementation more than it does an explicitly distributed implementation such as in

MPI. In Regent, points that exist at boundaries between submeshes may be included in multiple aliased subregions, but are not duplicated and explicitly communicated as in MPI.

A Regent program achieves parallelism by dividing the computation into tasks. In this case, the program might consist of a task per phase of the simulation per chunk of the mesh. More fine-grained tasks are also possible, and would expose additional task parallelism in the application, but as described in Section 7.4, various factors push us towards an implementation where tasks are fused to the maximum extent possible, resulting in exactly one task per phase.

In Regent, data structures are stored in regions. For PENNANT, each kind of mesh element (zone, side, or point) is stored in a separate region, and elements of sides contain pointers to elements of the other two regions. For parallel and distributed execution, these top-level regions must be partitioned into subregions naming the sets of elements needed by the various tasks in the application. For zones and sides, these subregions correspond to the colored submeshes shown in Figure 7.1. For points, a simple partitioning scheme could simply use two partitions of the points, naming the write and read/reduce sets shown on the bottom left and bottom right of the figure, respectively.

While the naive partitioning in Figure 7.1 is appealing in its simplicity, in practice this partitioning scheme leads to unnecessary data movement and analysis cost at runtime. The root of the problem is that the naive scheme does not take care to separate elements that will be communicated from those not involved in communication. Because this information is not provided by the user, a Regent implementation is forced to choose between two undesirable options: either it can perform potentially unnecessary data movement (i.e. copying all the shaded points of a given color, rather than just those with multiple colors), or it can perform a dynamic analysis to determine which elements are involved in communication (i.e. requiring an all-pairs comparison of the sets of elements in subregions). Fortunately, Regent's support for hierarchical partitions allow the user to express this information directly, avoiding the need for the compiler and runtime to second-guess the user's decisions.

Figure 7.2 shows the result of this hierarchical partitioning applied to the original

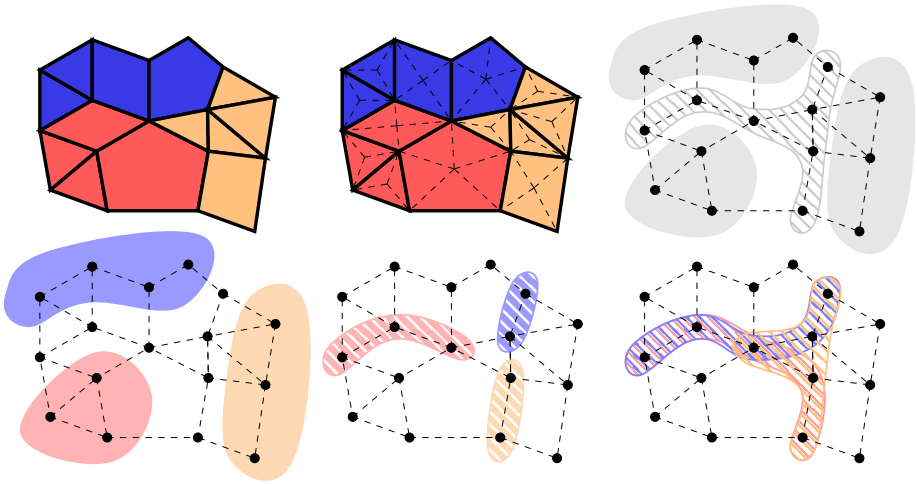


Figure 7.2: Hierarchical PENNANT data partitioning: zones (top left), sides (top middle), and points: all private vs. all ghost (top right), private (bottom left), master (bottom middle) and ghost (bottom right).

example mesh. Instead of using two partitions of points, the new scheme uses four partitions. The initial partition, which is shown in the upper right of Figure 7.2, divides points into two sets: *all private* (the solid-shaded points, which are not involved in communication) and *all ghost* (the hatched points, which may be involved in communication at some point in the application execution). This stage is critical because it identifies to the Regent compiler which points are or are not involved in communication. Note the two subregions (all private and all ghost) are disjoint. Because the partition is disjoint, there will be no subsequent need to consider any interactions between private and ghost points; private points will never be communicated, and in control replication no intersections between private and ghost subregions need be considered.

The subregions of this top-level partition are then further partitioned an additional three times. On the bottom right of Figure 7.2, the private points are partitioned to identify the subsets that belong to each of the three submeshes. This partition is again disjoint, which guarantees that the Regent compiler need not consider interactions between the private points belonging to each of the submeshes. In the bottom middle, the *master* partitions plays a role similar to the master points created in MPI. During

certain phases of the application, various computations will be performed on the master partition, and this partition must be disjoint to permit these computations to proceed in parallel. On the bottom right, the ghost partition identifies the points that need to be read from each submesh, and as a given point may need to be read from multiple submeshes, this partition is aliased.

Thanks to Regent's sequential semantics, this partitioning is relatively transparent to the application. Although the sets of elements in each subregion must be explicitly identified, the consistency of subregions is managed by the Regent compiler and runtime. Subregions behave as views or aliases of the parent region. If an object is contained in multiple overlapping subregions, any update to that object is visible to any task that follows in program order and that references the object through any other aliased subregion. This means that references through pointers between objects (e.g. from sides to points) are automatically valid as long as the task in question has requested privileges on the appropriate regions. This is in contrast to MPI, where overlapping access patterns such as the one in PENNANT require the application to explicitly create duplicate sets of master and slave points and manage the consistency of those duplicates.

Figure 7.3 shows an excerpt from an implementation of PENNANT that follows this strategy. The code shows phases 2 and 3 of the application, where forces are accumulated onto points and then forces used to update the velocity and position of the points themselves. For simplicity, and because only points are involved in communication, the code only shows data usage for points, and only for a single *field* `f` of the point objects representing the force exerted on each point.

In Regent, the effects of a task are completely described by the arguments and privileges in the task's declaration. Thus, for the purposes of the top-level design of an application, it is sufficient to declare the tasks as in lines 1-4 of Figure 7.3, and leave the bodies of the tasks to be implemented later. The task `calc_forces` on lines 1-2 reads and writes the `f` field of the private subregion belonging to each submesh, and applies reductions over `+` to the ghost subregion. (For regions of user-defined data types, a privilege may optionally name a specific field within the elements of the region.) The task `adv_pos_full` on lines 3-4 reads the accumulated forces from

```
1 task calc_forces(private : region(point), ghost : region(point))
2   where reads writes(private.f), reduces +(ghost.f) do ... end
3 task adv_pos_full(private : region(point), master : region(point))
4   where reads(private.f, master.f) do ... end
5
6 -- inside the main simulation task:
7 var points = region(..., point)
8 var private_vs_ghost = partition(disjoint, points, ...)
9 var private = partition(disjoint, private_vs_ghost[0], ...)
10 var master = partition(disjoint, private_vs_ghost[1], ...)
11 var ghost = partition(aliased, private_vs_ghost[1], ...)
12 while t < T do
13   dt = dtnext
14   for i = 0, N do
15     calc_forces(private[i], ghost[i])
16   end
17   -- implied communication from ghost to master
18   for i = 0, N do
19     adv_pos_full(private[i], master[i])
20   end
21   dtnext = ... -- computed via scalar reduction
22   t += dt
23 end
```

Figure 7.3: Excerpt from PENNANT Regent implementation control flow.

private and master subregions and (not shown) writes the velocity and position fields of both regions.

Lines 8-11 of Figure 7.3 show the partitioning calls used to create the hierarchical partitions shown in Figure 7.2. Line 8 creates the initial top-level partition between all private and all ghost points in the mesh. Lines 9 and 10-11 then create nested partitions of the respective subregions: a disjoint partition of the private points, and disjoint and aliased partitions of the ghost points. These calls show the use of the original style Regent partitioning calls using arbitrary coloring objects that map colors (small integers) to sets of points. (The colorings themselves are not shown.) It is also possible to compute the same partitions via the sublanguage of dependent partitioning operators described in Section 2.3.2. However, PENNANT was originally written at a time when these operators were not available in Regent.

7.3 Leaf Tasks

After the top-level control flow and partitioning scheme is decided, the remaining work in a Regent implementation consists of defining leaf tasks which perform the actual work in the application. Fortunately, leaf tasks are usually straightforward to implement. Figure 7.4a shows the implementation of the `adv_pos_full` task declared above. The task loops over points, reading the forces (`f`) computed from the previous step and finally updating velocity (`u`) and position (`x`).

For comparison, an equivalent C++ implementation that uses the Legion runtime API is shown in Figure 7.4b. The primary differences include:

1. Physical instances are explicitly unpacked from the task's arguments (lines 5 and 13).
2. Accessors for each field are constructed (lines 6-12 and 14-17).
3. Arguments passed as futures are explicitly unpacked (lines 18-19).
4. The C++ code explicitly iterates over spans of contiguous elements so that the inner loop can avoid a call into the iterator (lines 22-27).
5. Fields of complex types are explicitly expanded into multiple fields of basic types (lines 29-30, 31-32, 33-34, and 35-38).

All of this complexity is hidden and managed automatically by the compiler in the Regent implementation as described in Chapter 4.

7.4 Cache Blocking

The reference implementation of PENNANT employs a critical cache-blocking optimization which prevents the application from becoming memory-bound and allows it to scale with reasonable efficiency to the cores within a node. In the reference code, the inner loops have been strip-mined to form double-nested loops where the outer loops iterate over *chunks* of elements that are small enough to fit in the L2 cache of a

```
task adv_pos_full(points : region(point), dt : double) where
  reads(points.{x0, u0, f, maswt}), writes(points.{x, u})
do
  var fuzz = 1e−99
  var dth = 0.5 ∗ dt
  for p in points do
    var pap = (1.0 / max(p.maswt, fuzz))∗p.f
    var pu = p.u0 + dt∗pap
    p.u = pu
    p.x = p.x0 + dth∗(pu + p.u0)
  end
end
```

(a) PENNANT leaf task implementation in Regent.

```
void adv_pos_full(const Task ∗task,
                  const std::vector<PhysicalRegion> &regions,
                  Context ctx, HighLevelRuntime ∗runtime)
{
  PhysicalRegion points0 = regions[0];
  Accessor<double, SOA> points_x0_x(points0, PX0_X);
  Accessor<double, SOA> points_x0_y(points0, PX0_Y);
  Accessor<double, SOA> points_u0_x(points0, PU0_X);
  Accessor<double, SOA> points_u0_y(points0, PU0_Y);
  Accessor<double, SOA> points_f_x(points0, PF_X);
  Accessor<double, SOA> points_f_y(points0, PF_Y);
  Accessor<double, SOA> points_maswt(points0, PMASWT);
  PhysicalRegion points1 = regions[1];
  Accessor<double, SOA> points_x_x(points1, PX_X);
  Accessor<double, SOA> points_x_y(points1, PX_Y);
  Accessor<double, SOA> points_u_x(points1, PU_X);
  Accessor<double, SOA> points_u_y(points1, PU_Y);
  Future f0 = task−>futures[0];
  double dt = f0.get_result<double>();
  double fuzz = 1e−99;
  double dth = 0.5 ∗ dt;
  IndexIterator it(points0.get_logical_region().get_index_space());
  while (it.has_next()) {
    size_t count;
    ptr_t start = it.next_span(count);
    ptr_t end(start.value + count);
    for (ptr_t p = start; p < end; p++) {
      double frac = (1.0 / max(points_maswt.read(p), fuzz));
      double pap_x = frac ∗ points_f_x.read(p);
      double pap_y = frac ∗ points_f_y.read(p);
      double pu_x = points_u0_x.read(p) + dt ∗ pap_x;
      double pu_y = points_u0_y.read(p) + dt ∗ pap_y;
      points_u_x.write(p, pu_x);
      points_u_y.write(p, pu_y);
      points_x_x.write(p, points_x0_x.read(p) +
                       dth∗(pu_x + points_u0_x.read(p)));
      points_x_y.write(p, points_x0_y.read(p) +
                       dth∗(pu_y + points_u0_y.read(p)));
    }
  }
}
```

(b) PENNANT leaf task implementation in Legion C++ API.

Figure 7.4: PENNANT leaf tasks in Regent and C++.

CPU core. Because the mesh is unstructured and contains pointer data structures with internal consistency properties that must be maintained, the optimization is challenging to implement in a general-purpose compiler and thus must be implemented manually. The Regent implementation follows the same pattern as the reference code, although the definition of a chunk has been tweaked slightly to keep more elements in cache when switching from looping over different kinds of data structures (from zones to sides or vice versa). Fortunately, this optimization has a minimal impact on the signature of a task, and thus all of Regent's higher-level optimizations (most notably control replication) are not impacted by the manual application of this optimization in the code.

In the Regent implementation, the chunks used for cache blocking are represented as an addition level of partitioning in the region tree (a second level for zones and sides, and a third level for points). No tasks are ever launched on the individual chunks; instead entire partitions of chunks are passed to the leaf tasks, and those tasks contain double-nested loops first over chunks and then over elements of chunks. The partitions themselves are subsumed by the existing region arguments to tasks and thus do not impact privileges the tasks require.

Due to the multiple types of physics it uses, PENNANT does have some task parallelism available, which could potentially enable flexibility in the scheduling of tasks. However, exploiting this task parallelism interferes with the cache blocking optimization described above. Because the cache blocking optimization prevents PENNANT from being memory-bound, it is much more important to preserve that optimization than to expose task parallelism in the application. In the Regent implementation of PENNANT, we manually fused the tasks in order to preserve the contents of the cache. The result of this fusion is that the Regent implementation of PENNANT uses one task per phase of the computation as described above.

Chapter 8

Evaluation

In this chapter we follow up on the qualitative evaluation of Regent in Chapter 7 and attempt to quantify the impact of Regent on programmer productivity and performance. To conduct these experiments, we ported five small applications into Regent: three unstructured and two structured codes. These applications, ranging from approximately 1000 to 4000 lines of code, are intended to represent meaningful subsets of larger applications or classes of applications. Three of the five applications have hand-written and manually tuned C or C++ implementations that employ either MPI or some variety of MPI+X for parallelism. For one version of one of the four applications we considered a C++ Legion reference implementation of the same code.

To quantitatively evaluate Regent's productivity, we compare the number of lines of code in the Regent and reference implementations. While lines of code comparisons have some limitations, this provides some empirical evidence that Regent provides meaningful productivity benefits for programmers.

The performance benchmarks evaluate Regent's progress towards two distinct goals. First, is Regent capable of generating kernels with performance competitive with hand-tuned C and C++ code? Note that this is not strictly necessary, because Regent tasks can always call C or C++ functions directly. However, Regent's productivity benefits are much more significant when entire codes can be entirely written in Regent.

Second, is Regent capable of matching the single- and multi-node scalability of reference implementations written in well-known parallel programming models? In

all cases, we report absolute performance, as this is the measure that is ultimately relevant to end users, though parallel efficiency is also of interest, particularly with control replication.

The experiments in this chapter were originally conducted in two parts, and thus are described separately. There are some differences between the experiments. Most notably, the initial experiments were conducted without control replication, and thus evaluate performance only on single or small numbers of nodes. The second set of experiments evaluate the effectiveness of control replication specifically and focus primarily on scaling to large numbers of nodes.

In the following section, we describe the benchmarks used in the experiments. Then we consider each of the sets of experiments.

8.1 Benchmarks

We evaluate Regent versions of five applications: a circuit simulation on a sparse unstructured graph; MiniAero, an explicit solver of the compressible Navier-Stokes equations on a 3D unstructured mesh; PENNANT, a Lagrangian hydrodynamics simulation on a 2D unstructured mesh; a stencil benchmark on a regular grid; and Soleil-X, a turbulence and particle solver on a 3D structured grid. Each application is described below.

8.1.1 Circuit

Circuit, introduced in [13], is a distributed simulation of an electrical circuit, operating over an arbitrary, unstructured graph of nodes and wires. The simulation consists of three phases. The first phase reads the voltages of nodes from the previous phase and determines the current moving along each wire using an iterative solution to the differential equations of the RLC model of the circuit. The second phase reads the current on each wire and computes the resulting charge that accumulates on each node in the circuit. The third phase computes updated voltages based on the charge at each node. Of these, the first stage dominates overall execution time, and because

the iterative method is compute-limited, the application has high compute intensity.

We consider two variations on the circuit simulation design. The original C++ Legion implementation from [13] is used in the initial experiments, to enable a direct apples-to-apples comparison with Regent. The C++ implementation uses hand-written SSE vector intrinsics in the compute-limited portions of the code, and is highly optimized. The Regent version relies entirely on Regent's auto-vectorizer to achieve the same performance. The input problem tested was a randomly generated circuit with some internal structure. Specifically, the nodes in the input graph are divided into subgraphs which are densely connected, with fewer wires (5%) crossing between subgraphs. However, while a smaller number wires cross between subgraphs, the choice of which subgraph to connect to is still random, and thus the overall structure of the communication of the application is still dense: almost every compute node in the machine can be expected to communicate with every other node. As a result, this version of the application is inherently communication-bound at higher node counts and thus not appropriate for scaling studies at very large numbers of nodes.

In the second set of experiments, the structure of the graph was modified to permit scaling the simulation to large numbers of nodes. Specifically, the new simulation modifies the random function used in selecting which subgraph an external wire should connect to. Instead of using an unconstrained random function, the function is constrained so that each subgraph touches at most six other subgraphs, and external wires are randomly assigned from among these. This results in an overall communication graph that is sparse. In addition, the percentage of external wires was increased to 20% from 5%, which increases the volume of communication to offset the decrease in the amount of connectivity between subgraphs.

Both Legion and Regent implementations of Circuit take advantage of the hierarchical partitioning structure described in Section 3.4.3. In Circuit, because only a fraction of the wires are involved in communication, the set of nodes connected to those wires is bounded and can be determined at initialization time. Circuit uses a top-level partition to separate all private nodes, which are internal to a subgraph, from shared nodes, which exist at the boundaries between subgraphs. Following this the two regions of private and shared nodes are further subdivided by subgraph to identify the

halos and sets of internal nodes in each subgraph. This structure identifies explicitly the elements that are involved in communication, enabling further optimizations in Legion and Regent.

8.1.2 PENNANT

PENNANT is a Lagrangian hydrodynamics proxy application for unstructured meshes from Los Alamos National Laboratory [36]. The reference implementation of PENNANT is written in C++ and supports configurations that use OpenMP, MPI, and MPI+OpenMP. The Regent implementation of PENNANT is discussed in detail in Chapter 7.

PENNANT includes a dynamic computation of the dt which is used to increment the simulation time on every time step. This requires the use of a scalar reduction, which has the potential to be a bottleneck at large node counts. Unfortunately, this scalar reduction is completely exposed as there is no additional task parallelism available to hide the additional latency. Fortunately however, the stop condition at the top of the main simulation loop depends only on the previous, rather than current, dt, so this scalar reduction does not cause the control thread to block. In the Regent implementation, tasks are issued one iteration ahead of execution, which is sufficient to hide the latency of the dynamic runtime analysis.

PENNANT employs a cache-blocking optimization which prevents the application from becoming memory-bound and enables reasonable scaling to the cores within a node. This optimization does not impact the application of control replication because the details of the cache blocking are subsumed by and hidden behind the signature of a task.

However, a result of this cache-blocking optimization is that the tasks in the application are fused to the maximum extent possible. Thus PENNANT exposes no task parallelism at all to the Legion runtime, which causes PENNANT to be somewhat more sensitive to runtime overhead than other applications.

As in Circuit, PENNANT also applies the hierarchical partitioning scheme from Section 3.4.3. In the case of PENNANT, points are involved in communication, and

thus the points are partitioned hierarchically. This partitioning is described in detail in Section 7.2. Zones and sides need not be partitioned hierarchically as they are not involved in communication; a single disjoint partition suffices for each.

8.1.3 MiniAero

MiniAero is a computational fluid dynamics mesh proxy application from the Mantevo suite [38] developed at Sandia National Laboratories. MiniAero uses uses a Runge-Kutta fourth-order time marching scheme to solve the compressible Navier-Stokes equations on a 3D unstructured mesh. The reference version of the application is written in a hybrid style, using MPI for inter-node communication and Trilinos Kokkos [34] for intra-node parallelism. (Kokkos is a portability layer for C++ that compiles down to pthreads (on CPUs), also developed at Sandia.)

The mesh elements in MiniAero are cells and faces. The mesh is divided into submeshes via a simple disjoint partitioning of cells. Cells at the boundaries between submeshes are members of the halos of other submeshes, and are thus involved in communication. The Regent implementation of MiniAero uses a hierarchical partitioning scheme to separate communicated and non-communicated elements. Faces are duplicated at submesh boundaries, and thus are not involved in communication.

MiniAero is mostly memory-bound, and thus is sensitive to optimizations that improve locality. When implementing MiniAero in Regent, we noticed that locality, and thus performance, benefits substantially from using a hybrid data layout, where some fields are stored in SOA layout and others are stored in AOS layout. The versions of Legion used in the experiments did not support this kind of hybrid layout, and thus the initial Regent implementation of MiniAero uses arrays to achieve the same effect. However, this served as a proof of concept for the value of these hybrid layouts (with different fields simultaneously in SOA or AOS layout), and support for these layouts has since been added to Legion.

8.1.4 Stencil

Stencil is a 2D structured benchmark from the Parallel Research Kernels (PRK) [71,72]. Note that, as support for structured regions was not available in Regent at the time of the initial experiments, this benchmark is included only in the latter experiments for control replication.

The code performs a stencil of configurable shape and radius over a regular grid. Our experiments use the default configuration: a radius-2 star-shaped stencil on a grid of double-precision floating point values. In our experiments we compare a Regent implementation against the MPI and MPI+OpenMP reference codes provided by PRK.

Because Stencil is a structured application, it is possible to identify the grid elements involved in communication even more precisely than in the unstructured applications above. At each step in the computation, halos of grid elements must be communicated with the subgrids to the north, south, east and west. (In a star-shaped stencil, no elements are communicated diagonally.) These communication patterns are all independent, and thus a top level partition divides the regions five ways: one way for elements not involved in communication at all, and four ways for communication in each of the cardinal directions. Then, each of the four subregions involved in communication is then partitioned two ways to identify producers and consumers of values in each direction. This structure identifies to Regent and Legion precisely which elements must be communicated. Note that, under control replication, only four sets of intersections must be computed: one for each of the cardinal directions, between the producer and consumer partitions of each of those directions. No other intersections must be considered at all, because of the static disjointness information that is inherent in this region tree structure.

For this application, we use the Regent foreign function interface to call into C implementations of the kernels. These kernels are called from Regent tasks which declare the appropriate privileges, thus optimizations such as control replication need not reason about the effects of calls to C functions. We have also built a pure Regent implementation which uses metaprogramming to construct optimized kernels for the

stencil; however the Regent auto-vectorizer does not yet have full support for multi-dimensional regions and thus for these experiments we only consider the version which uses C kernels.

8.1.5 Soleil-X

Soleil-X is a turbulence solver that uses a 3D structured grid of cells with particles that track the movement of the fluid in a bidirectionally-coupled simulation. Soleil-X is therefore a hybrid structured/unstructured code: the fluid computation is entirely structured, while the particles move freely and are represented with an unstructured region.

Of all the applications in this evaluation, Soleil-X is the only application *not* written directly in Regent. Instead, Soleil-X is written in a domain specific-language Ebb [16] for physical simulations that targets Regent. Ebb supports forall-style parallel loops with stencil-like access patterns. Ebb programs do not contain any explicit data partitioning or tasks; these are generated automatically by the Ebb compiler. We used an implementation of Ebb for Regent to evaluate the performance of Soleil-X on distributed-memory machines. Regent was in this case an enabling technology, allowing a efficient and scalable implementation of Ebb to be created quickly, without needing to write a compiler to directly produce low-level distributed code. Ebb was also able to take advantage of Regent's support for generating high-performance kernels.

Soleil-X also depends on Regent's support for structured grids and thus is included only in the second set of experiments.

8.2 Initial Experiments

In our initial experiments, we consider the productivity and performance of Regent on three unstructured benchmark applications. First, to quantify productivity, we compare the lines of code of each Regent implementation against a reference. Second, we explore the impact of the optimizations described in Chapter 5. And third, we

		Regent		
Application	Reference	Total	Mapper	Partitioning
Circuit	1701	969	144	159
PENNANT	2416	1789	244	163
MiniAero	3993	2836	193	51

Figure 8.1: Non-comment, non-blank lines of code for Regent and reference implementations.

consider the performance and scalability of each Regent application. Note that control replication was not available at the time these experiments were performed, thus in these experiments we consider performance only on a single node or small number of nodes.

The experiments were done on the Certainty supercomputer [1]. Each node has two sockets with an Intel Xeon X5650 per socket for a total of 12 physical cores per node (24 threads with hyperthreading). Nodes are connected with Mellanox QDR Infiniband. The Legion runtime and all three C++ reference codes have been compiled with GCC 4.9.2. Regent uses LLVM 3.5 for code generation.

8.2.1 Lines of Code

We evaluate the productivity of Regent by comparing the number of lines of codes in each Regent implementation against each reference. Figure 8.1 summarizes the results.

It is worth acknowledging up front the limitations of such measurements. Not all benefits can be accounted for by way of lines of code. Regent's sequential semantics is an example of a benefit with enormous impact which does not necessarily translate to a direct reduction in lines of code. Beyond this, the benchmarks chosen do not make substantial use of task parallelism. If additional task parallelism were added to any of these benchmarks, the OpenMP and MPI versions would require substantial rewriting to take advantage of it, while the Regent implementations would all exploit this parallelism with no additional effort.

When comparing to OpenMP and MPI in particular, note that Regent (somewhat counterintuitively) requires the user to be more explicit about organization and

placement of data. Despite this, the Regent codes evaluated were all shorter than their corresponding reference versions. In addition, two improvements to the Regent language have been identified that have the potential to dramatically reduce the lines of code associated with certain activities (specifically partitioning and mapping). These improvements were not available at the time of the original experiments, but have been called out separately in the results to indicate the potential upside. These are also discussed in more detail below.

Of the three applications, Circuit is shows the largest difference: the Regent implementation is 43% smaller than the Legion C++ implementation. This is mostly due to the use of SSE vector intrinsics in the Circuit source code. In our experience, C++ compilers are unable to generate code of the same quality automatically, while Regent's auto-vectorization achieves the performance of the hand-written vector code. The Regent implementation did not use any explicit vectors or vector intrinsics.

PENNANT in Regent is approximately 25% shorter than the OpenMP reference code (43% when excluding mapper and partitioning code). Note that the line counts in Figure 8.1 exclude common code used in both applications—specifically, the mesh initialization code, which the Regent implementation borrowed from the reference.

The Regent implementation of MiniAero is about 30% shorter than the MPI+Kokkos reference code (35% when excluding mapper and partitioning code).

In all cases, the lines of code reported for Regent include a mapper written in C++ that targets the Legion mapper API. The mapper column under Regent in Figure 8.1 reports the lines of code contained in mapper implementations for each application. In practice these mappers are quite simple and the C++ code to implement one is needlessly verbose. In the future, we are interested in investigating the possibility of a domain-specific language for mappers that could potentially replace the C++ implementations used in these experiments. We have an initial prototype of such a language, and the preliminary results are encouraging, suggesting that the lines of code associated with mapping can be significantly reduced.

Each of the Regent implementations also includes code to partition regions into subregions. The number of lines of code associated with partitioning is reported in the partitioning column under Regent in Figure 8.1. The numbers reported here use

Key	Optimization
map	Mapping Elision
leaf	Leaf Task Optimization
idx	Index Launch Optimization
fut	Future Optimization
dbr	Dynamic Branch Elision
vec	Vectorization
all	All of the Optimizations Above

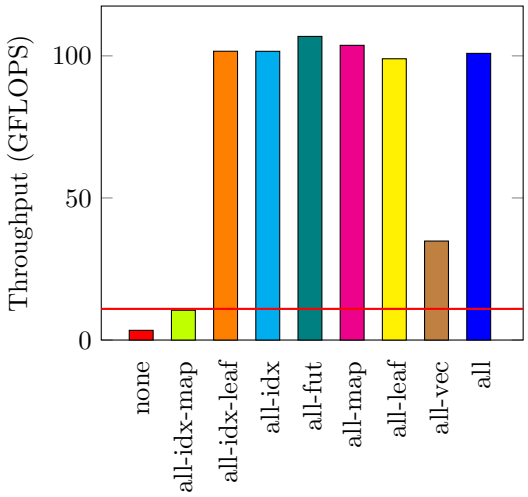
Figure 8.2: Legend key for knockout experiments.

the current Legion partitioning API, which is known to be verbose. As noted in [70], a more expressive sublanguage for partitioning can dramatically reduce the size of this code. In fact, for each of the applications above, less than 10 lines of code are required with the partitioning sublanguage. Once this support is available in Legion, this more expressive sublanguage for partitioning will also be made available in Regent.

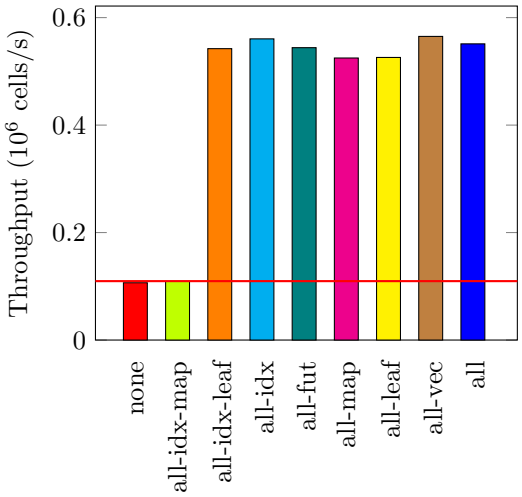
8.2.2 Impact of Optimizations

Next, to demonstrate the impact of the compiler optimizations performed by Regent, we perform *knockout experiments* for each application, disabling each optimization presented in Chapter 5 in turn. In addition, we perform double knockout experiments, measuring performance with all possible pairs of two optimizations disabled, and call out a few interesting combinations. As several of the optimizations impact the achieved parallelism, we evaluate each configuration in a parallel configuration and compare against the best sequential performance achieved by Regent. The labels for the various optimizations are described in Figure 8.2. Pointer check elision has been previously demonstrated to have a significant impact on performance [68] and has been left out of the knockout to reduce clutter.

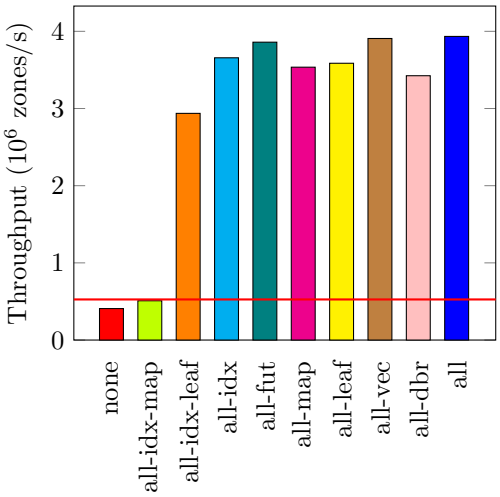
Figure 8.3 summarizes the results. Two classes of effects are visible. Some optimizations (or pairs of optimizations), when disabled, result in a loss of parallel execution. The combination of index launch and mapping optimizations is an example of such a pair. Fortunately, there are relatively few such combinations. Most other optimizations have a smaller impact, about 10-15%. While these effects may seem inconsequential compared to a loss of parallelism, they are still important to overall



(a) Circuit (10 CPUs).



(b) MiniAero (8 CPUs).



(c) PENNANT (10 CPUs).

Figure 8.3: Knockout experiments. The red line in each graph shows the best sequential Regent performance.

application efficiency. An application without any of these optimizations would lose a total of about 50%, an amount which is often considered unacceptable in high-performance application development.

As mentioned above, certain optimizations impact the parallelism available in the

application; index launch optimization and mapping elision are two such optimizations. When both are disabled simultaneously, the code runs sequentially. (The red line on each graph indicates the best performance on a single-thread.) As described in Section 5.1, the Legion runtime, in the absence of the map and unmap calls placed by the compiler, must copy back the results of each task execution before returning control to caller. This creates an effective barrier between consecutive tasks, but the effect is not noticeable as long as index launch optimization is able to parallelize the task launches. Disabling both optimizations serializes the code. But if either optimization is disabled by itself, the application continues to run in parallel at somewhat reduced throughput.

This redundancy allows Regent to be much more robust in the presence of dynamic behavior. Traditional optimizations for parallelism can fail in situations where the independence of tasks cannot be proven statically. In these situations, Regent is able to fall back on the Legion runtime to discover parallelism dynamically. As a result, most optimizations for parallelism, when disabled individually, have only a 10-15% impact on overall performance. This impact is due to either unnecessary blocking, stalls in the runtime analysis pipeline, or increased overhead, as noted in Section 4.1.4, and thus is more noticeable in applications where the runtime overhead is already more exposed. PENNANT is such an application, because the dynamic computation of *dt* at the end of the time step loop prevents the runtime from running more than one iteration ahead of the application. However, the result of a failed optimization in traditional static compilers would be sequential execution, and in Regent this only occurs when at least two optimizations fail.

Some more subtle effects are also visible in the knockout results. PENNANT's pattern of task launches is such that when leaf optimization alone is disabled, the Legion runtime must stall for mapping to complete in order to ensure that all the dependencies are correctly captured. Circuit and MiniAero are structured differently from PENNANT and therefore are not impacted significantly by the absence of leaf optimization (in combination with index launches or otherwise).

PENNANT also shows the most benefit from eliminating dynamic branches. In contrast to Circuit and MiniAero which are generally compute or memory bound,

certain performance critical kernels in PENNANT contain long chains of dependent math instructions, which in turn depend on conditional memory accesses (when dynamic branch elision is not enabled). At 10 cores, throughput improves by 15% if dynamic branches can be eliminated. Dynamic branch elision does not have a significant impact on the other applications and is hidden in those graphs to reduce clutter.

8.2.3 Performance

We now consider the performance of Regent implementations of the three applications against the various reference codes. Figure 8.4 shows the absolute performance of each of the three applications while strong scaling.

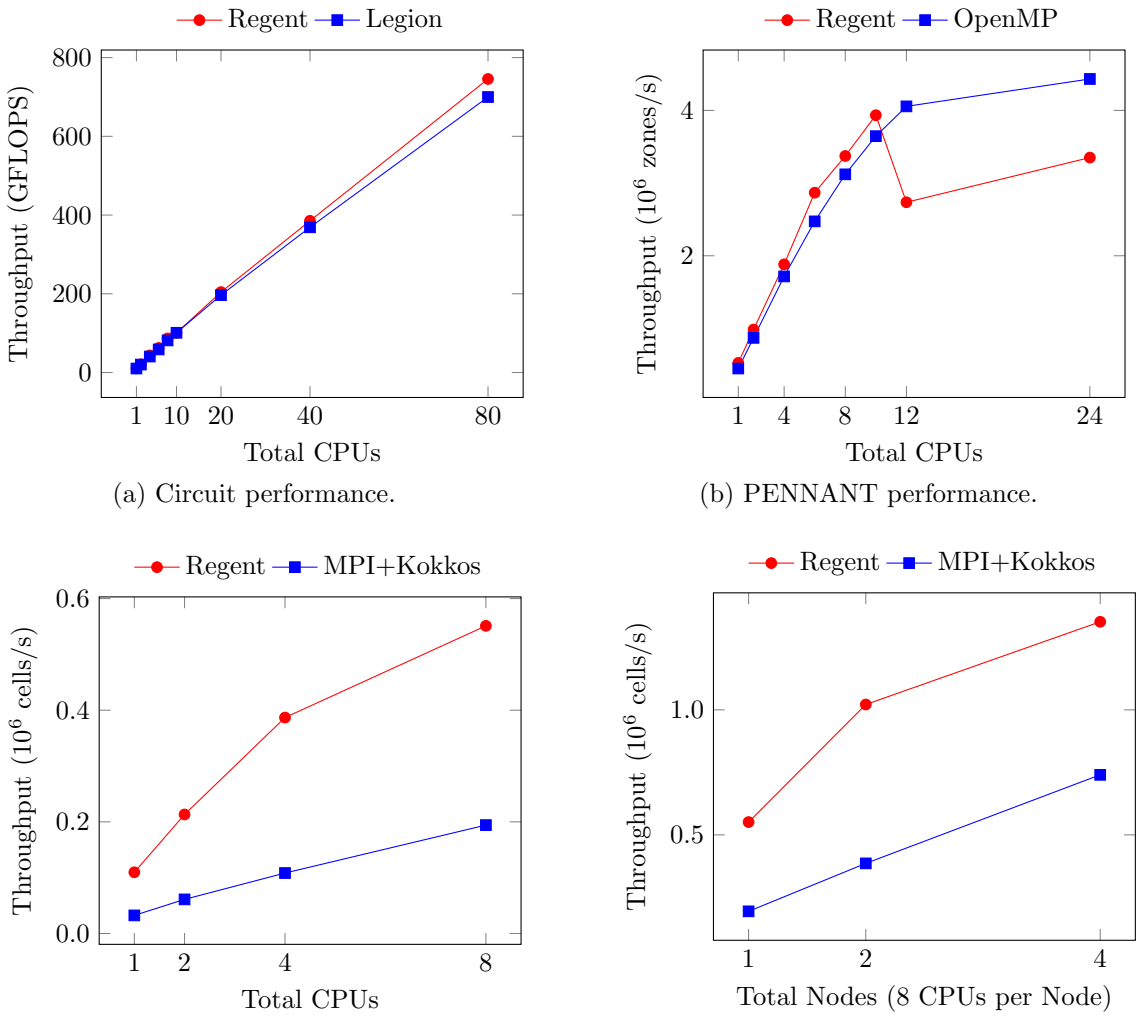
Circuit

We compare the performance of Regent against a hand-tuned and manually vectorized CPU implementation written to the C++ Legion API. We evaluate both implementations on a graph with 800K wires connecting 200K nodes. Figure 8.4a shows the strong scaling performance of Regent against the baseline C++ Legion implementation running on up to 8 nodes on Certainty. Notably, the fully-optimized Regent implementation—which is written in a straightforward way with no use of explicit vectors or vector intrinsics, and is less than half the total number of lines of code—achieves performance comparable to the manually vectorized C++ code, exceeding the performance that can be achieved by using the LLVM 3.5 vectorizer alone.

PENNANT

Figure 8.4b evaluates Regent against an OpenMP implementation of PENNANT for strong scaling a problem containing approximately 2.6M zones.

Regent performs better than OpenMP for all core counts up to 10, surpassing OpenMP by 8% at 10 cores. Starting at 12 cores, Regent performance degrades because the additional compute threads interfere with threads Legion uses for dynamic



(a) Circuit performance.

(b) PENNANT performance.

(c) MiniAero single-node performance.

(d) MiniAero multi-node performance.

Figure 8.4: Initial strong-scaling performance.

dependence analysis and data movement. The Legion runtime is also unable to exclusively allocate physical cores for each thread and abandons pinning altogether, leading to increased interference between application threads.

PENNANT performance is sensitive to the NUMA architecture of the machine. OpenMP performance was substantially impacted by CPU affinity, and a manual assignment of threads to cores was needed for optimal performance. Regent automatically binds threads to cores when possible and round robins threads between NUMA

domains, and thus performs well with minimal manual tuning.

MiniAero

Figures 8.4c and 8.4d compare strong scaling performance between a Regent implementation and the baseline MPI+Kokkos version on a problem size with 4M cells and 13M faces running on up to 4 nodes on Certainty.

Regent outperforms MPI+Kokkos on 8 cores by a factor of 2.8X through the use of a hybrid SOA-AOS data layout, as noted in Section 8.1.3. The improved data layout substantially boosts cache reuse and improves utilization of memory bandwidth.

8.3 Control Replication Experiments

We evaluate performance and scalability of control replication in the context of Regent with the five applications described in Section 8.1. For each application we consider a Regent implementation with and without control replication and when available a reference implementation written in MPI or a flavor of MPI+X.

For each application, we report weak scaling performance on up to 1024 nodes of the Piz Daint supercomputer [7], a Cray XC50 system. Each node has an Intel Xeon E5-2690 v3 CPU (with 12 physical cores) and 64 GB of memory. Legion was compiled with GCC 5.3.0. The reference codes were compiled with the Intel C/C++ compiler 17.0.1. Regent used LLVM for code generation: version 3.8.1 for Stencil and PENNANT and 3.6.2 for MiniAero and Circuit.

Finally, we report the running times of the dynamic region intersections for each of the applications at 64 and 1024 nodes.

8.3.1 Circuit

We evaluate the weak scaling performance of a sparse circuit simulation based on [13]. The implicitly parallel version from [13] was already shown to be substantially communication bound at 32 nodes and would not have scaled to significantly more nodes, regardless of the implementation technique. The input for this problem was a

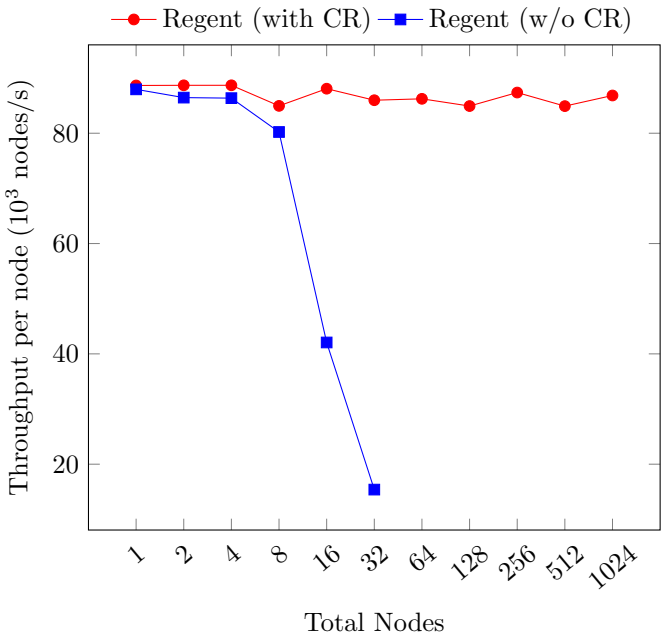


Figure 8.5: Weak scaling for Circuit.

randomly generated sparse graph with 100k edges and 25k vertices per compute node as described in Section 8.1.1; the application was otherwise identical to the original.

Figure 8.5 shows weak scaling performance for the simulation up to 1024 nodes. (In the legend control replication is abbreviated as CR.) Regent with control replication achieves 98% parallel efficiency at 1024 nodes. Regent without control replication matches this performance at small node counts (in this case up to 16 nodes) but then efficiency begins to drop rapidly as the overhead of having a single master task launching many subtasks becomes dominant, as discussed in Section 1.2.

8.3.2 PENNANT

Figure 8.6 shows weak scaling performance for PENNANT on up to 1024 nodes, using a problem size of 7.4M zones per node. The single-node performance of the Regent implementation is less than the reference because the underlying Legion runtime requires a core be dedicated to analysis of tasks. This effect is noticeable on PENNANT because, due to the cache blocking optimization in the implementation

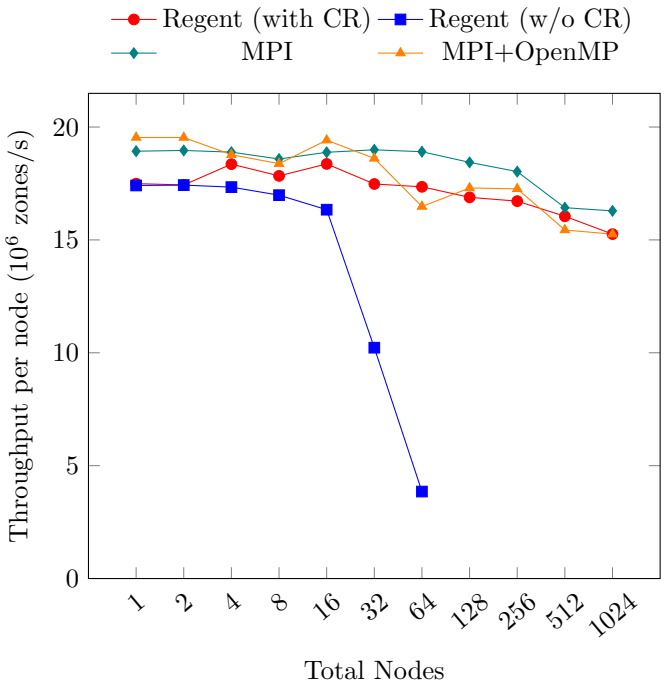


Figure 8.6: Weak scaling for PENNANT.

of PENNANT (described in Section 8.1.2), the code is mostly compute-bound. This optimization impacts even the data structure layouts, as the (otherwise unordered) mesh elements are grouped into chunks to be processed together. In spite of this, control replication applied seamlessly to the code, as the details of the cache blocking optimization are limited to the structure of the region tree (which subsumes the chunk structure of the original code) and the bodies of tasks (whose details are accurately summarized by the privileges declared in the task declaration).

However, the performance gap which is visible at a single node closes at larger node counts as Regent is better able to achieve asynchronous execution to hide the latency of the global scalar reduction to compute the dt in the next time step of the application. At 1024 nodes, control replication achieves 87% parallel efficiency, compared to 82% for MPI and 64% for MPI+OpenMP.

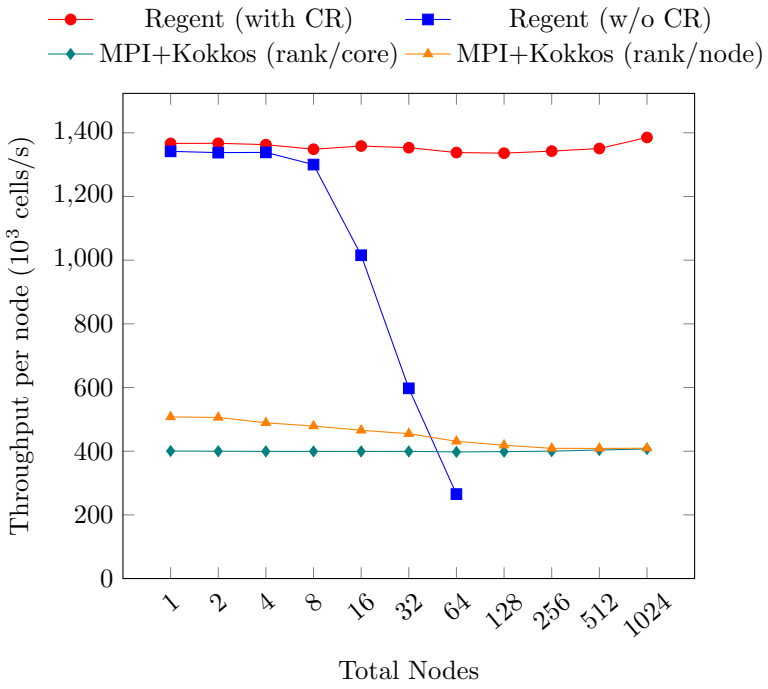


Figure 8.7: Weak scaling for MiniAero.

8.3.3 MiniAero

As described in Section 8.1.3, MiniAero is a 3D unstructured mesh proxy application that includes an explicit solver for the compressible Navier-Stokes equations. The reference is written in MPI+Kokkos. In these experiments, we ran the reference in two configurations: one MPI rank per core, and one MPI rank per node (using Kokkos support for intra-node parallelism).

Figure 8.7 shows weak scaling absolute performance for the various implementations of MiniAero on a problem size of 512k cells per node. As described in Section 8.2.3, Regent out-performs the reference MPI+Kokkos implementations of MiniAero on a single node, mostly by leveraging the improved hybrid data layout features of Legion [14].

Control replication achieves slightly over 100% parallel efficiency at 1024 nodes due to variability in the performance of individual nodes; as before, Regent without control replication struggles to scale beyond a modest number of nodes. Although the

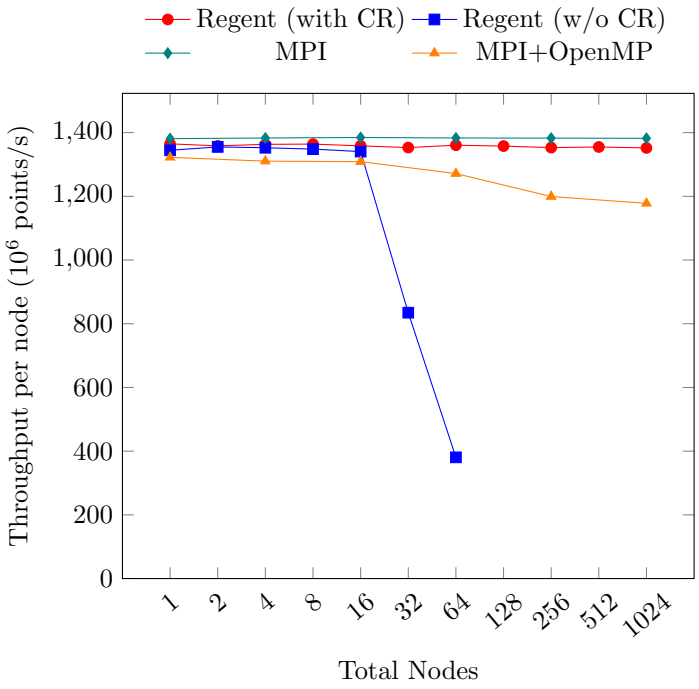


Figure 8.8: Weak scaling for Stencil.

rank per node configuration of the MPI+Kokkos reference provides initial benefits to single-node performance, performance eventually drops to the level of the rank per core configuration.

8.3.4 Stencil

We test the Stencil benchmark from the PRK suite in its default configuration: a radius-2 star-shaped stencil on a grid of double-precision floating point values. We evaluate weak scaling performance on $40k^2$ grid points per node, comparing Regent with and without control replication against the MPI and MPI+OpenMP reference codes provided by PRK. Both reference codes require square inputs and thus were run only at node counts that were even powers of two.

As noted in Section 3.1, all analysis for control replication was performed at the task and region level. Control replication was able to optimize code containing affine access patterns, without requiring any specific support for affine reasoning in the

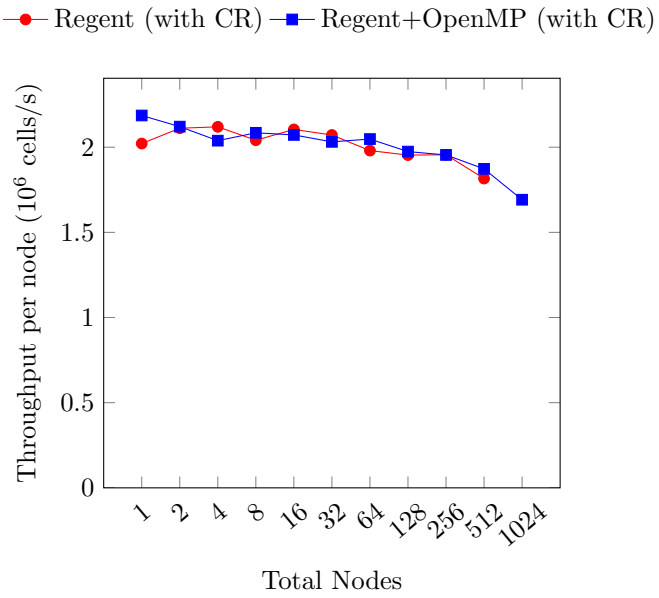


Figure 8.9: Weak scaling for Soleil-X.

compiler.

Figure 8.8 shows weak scaling performance for Stencil up to 1024 nodes. Control replication achieved 99% parallel efficiency at 1024 nodes, whereas Regent without control replication rapidly drops in efficiency when the overhead of launching an increasing number of subtasks begins to dominate the time to execute those subtasks.

8.3.5 Soleil-X

Figure 8.9 shows the weak scaling performance of Soleil-X on up to 1024 nodes. Soleil-X was configured to use a grid of 256^3 cells and 1M particles per machine node.

We ran Soleil-X in two configurations: with a task per core, as programs normally do in Regent; and with a task per node, using Regent's support for OpenMP. Note that this did *not* involve a C++ compiler in any way, or the use of OpenMP pragmas. Instead, when configured to use OpenMP, the Regent compiler directly generates calls to the OpenMP ABI for loops within tasks that can be executed in parallel (as described in Section 5.8). Notably, Regent only optimizes loops in this way when it can prove that the iterations are safe to execute in parallel, unlike traditional OpenMP, in

Application	Nodes	Shallow (ms)	Complete (ms)
Circuit	64	7.8	2.7
	1024	143	4.7
MiniAero	64	15	17
	1024	259	43
PENNANT	64	6.8	14
	1024	125	124
Stencil	64	2.7	0.4
	1024	78	1.3

Table 8.1: Running times for region intersections on each application at 64 and 1024 nodes.

which user-provided pragmas are trusted and thus unsound. Furthermore, on Legion, the OpenMP ABI is provided by Realm, Legion's performance portability layer. Thus for the purposes of Regent, OpenMP is used to expose parallelism below the level of Legion, and permits Regent to reduce the number of tasks that must be exposed to the Legion runtime.

Soleil-X achieved 77% parallel efficiency at 1024 nodes with Regent support for OpenMP enabled. Without OpenMP support Regent achieves similar parallel efficiency up to 512 nodes. At 1024 nodes, the configuration without OpenMP experiences a crash in the GASNet active messaging layer used by Legion.

Soleil-X also uses a dynamic time step as described previously with respect to PENNANT, requiring the use of a scalar reduction to compute *dt* for each time step.

8.3.6 Dynamic Intersections

As described in Section 3.3.3, dynamic region intersections are computed prior to launching a set of shard tasks in order to identify the communication patterns and precise data movement required for control-replicated execution. Table 8.1 reports the running times of the intersection operations measured during the above experiments while running on 64 and 1024 nodes. Shallow intersections are performed on a single node to determine the approximate communication pattern (but not the precise sets of elements that require communication); these required at most 259 ms at 1024 nodes (15 ms at 64 nodes). Complete intersections are then performed in parallel on each

node to determine the precise sets of elements that must be communicated with other nodes; these took at most 124 ms. Both times are much less than the typical running times of the applications themselves, which are often minutes to hours.

Chapter 9

Related Work

Parallelism research has a long history, both within the high performance computing community and more broadly. To keep the following discussion manageable, we focus on two areas. First, in Section 9.1, we consider parallel programming systems which provide *implicit parallelism*, where the system is responsible to some degree for the management of parallelism in the application. Second, Section 9.2 surveys *explicitly parallel* programming systems where parallelism is instead the responsibility of the user. Regent provides aspects of both styles, though the primary thrust of the design and our key contributions focus on the implicitly parallel aspects of the language.

9.1 Implicit Parallelism

Broadly speaking, implicitly parallel programming models are ones which provide some form of sequential (imperative, functional or declarative) semantics. The system is responsible, at least to some degree, for finding parallelism in the program, and for generating correct parallel code that obeys the original program semantics. As a result, implicitly parallel programming models typically rely at least to some degree on static and/or dynamic program analysis to determine what parallelism is available in the program. Note that parallelism in such programming models may still be user-visible to some degree (e.g. the user may be asked to identify the portions of code that are appropriate for parallel execution), though typically the use of sequential semantics

isolates the user from traditional pitfalls of explicitly parallel programming as data races and deadlocks.

9.1.1 Automatic Parallelizing Compilers

Automatic parallelizing compilers [19, 37, 43] attempt to generate parallel code from programs written in traditional, sequential programming languages. This problem has proven to be extremely challenging when the programs in question were written without regard to parallelism—i.e. so-called "dusty-deck" programs, usually written in Fortran. More success has been achieved for sequential programs in restricted domains. In particular, programs consisting of affine loops are amenable program analysis via the polyhedral method and can be automatically optimized by a compiler for distributed memory, as shown in [21]. However, in practice this limitation is quite restrictive, and many high-performance computing applications such as unstructured mesh codes cannot be expressed in this form. The general problem remains unsolved for a static compiler analysis and thus some form of dynamic analysis, changes to the programming model, or both, are required. Regent uses a combination of carefully selected language features and a hybrid static/dynamic program analysis and optimization, allowing it to effectively address codes such as simulations on unstructured meshes.

9.1.2 Inspector/Executor Methods

Inspector/executor (I/E) methods have been used to compile a class of sequential programs with affine loops and irregular accesses for distributed memory [53, 54]. As in control replication, a necessary condition for I/E methods is that the memory access patterns are fixed within the loop, so that the inspector need only be run once. Use of an inspector allows the read/write sets of program statements to be determined dynamically when the necessary static analysis is infeasible in the underlying programming language, enabling distributed, parallel execution of codes written in conventional languages. This approach has been demonstrated to scale to 256 cores. However, the time and space requirements of the inspector limit scalability at very large node counts. Also, the I/E approach relies on generic partitioning

algorithms such as automatic graph partitioning [26, 60].

Kwon et al. describe a technique for compiling OpenMP programs with regular accesses to MPI code [47] that is similar to the inspector/executor method. A hybrid static/dynamic analysis is used to determine the set of elements accessed by each parallel loop. For efficiency, the dynamic analysis maintains a bounded list of rectangular section fragments at communication points. As a result, non-affine accesses cause analysis imprecision that results in replicated data, increased communication, and limited scalability. The approach has been demonstrated to scale to 64 cores.

Like the two approaches above, Regent's control replication optimization uses a combined static/dynamic analysis to obtain precise information about access patterns. At a high level, the key difference is that control replication leverages a programming model with explicit support for coarse-grain operations (tasks), data partitioning (of regions into subregions), and the simultaneous use of multiple partitions of the same data. Control replication performs analysis at this coarsened level rather than at the level of individual loop iterations, resulting in a more efficient dynamic analysis and in-memory representation of the access patterns of each loop without any loss of precision. Furthermore, hierarchically nested partitions enable control replication to skip analysis at runtime for data elements not involved in communication (further reducing memory usage for the analysis). Finally, explicit language support for partitioning allows control replication to leverage application-specific partitioning algorithms, which are often more efficient and yield better results than generic algorithms. As a result, control replication is able to support more complex access patterns more efficiently, resulting in better scalability.

9.1.3 Loop-Level Parallelism

In many parallel programs, it is common for the available parallelism to be contained in loops. OpenMP [31] is a language extension for exploiting the form of parallelism that is now in widespread use in high-performance computing. Programs in traditional sequential languages can be incrementally converted into OpenMP programs by adding compiler directives to loops instructing the compiler to execute these loops in parallel.

OpenMP is explicitly unsound and does not attempt (and in fact cannot check) the correctness of user-specified compiler directives. In general, OpenMP relies heavily on shared memory to avoid the need for the user to describe or the compiler to understand data movement in the application. OpenMP was originally intended for use in single-node, shared-memory machines. Distributed implementations of OpenMP are possible but challenging; of these the most successful to date has been the one by Kwon et al. described in Section 9.1.2.

A number of other efforts to support OpenMP on distributed-memory machines target software distributed shared-memory (DSM) systems [11, 39, 59]. These approaches have reduced implementation complexity compared to approaches such as I/E that leverage dedicated compiler and runtime technology, but have limited scalability due to the limitations of general-purpose, page-based DSM systems.

In contrast, Regent leverages a sound type system [68, 70], and thus can offer more aggressive static and dynamic optimizations, allowing Regent to execute seamlessly and efficiently in a distributed environment despite providing sequential semantics.

9.1.4 Fork-Join Parallelism

Fork-join parallelism is a style of parallelism where the application *forks* to execute parallel work and then *joins* to wait on the completion of that work. As with Regent in the absence of control replication, fork-join parallelism suffers from a sequential bottleneck on the repeated creation and destruction of parallel workers during fork and join operations. A compiler-assisted approach for generating SPMD code from fork-join parallel programs—via the insertion of barriers—is well known [30]. However, this approach depends on the use of shared memory, and generalizing the approach to distributed memory requires a precise analysis of the memory accesses in the application. It is exactly this analysis of memory accesses that Regent addresses via control replication. By exploiting the structure of user-defined partitions, control replication is able to achieve an effective and reliable transformation of implicitly parallel programs into SPMD code for distributed-memory machines. In addition, control replication uses point-to-point synchronization (rather than barriers), and

preserves task parallelism to the extent that it exists in the application.

Cilk [2] is a well-known language for fork-join parallelism on shared-memory machines. Cilk extends the C language with the keywords *spawn* to fork a task and *sync* to join on forked tasks. The sequence of spawn and sync statements can be viewed as defining a dependence graph between tasks, though the structures that can be expressed are limited as the sync call blocks on all locally spawned tasks and does not permit the specification of individual dependencies between tasks. Memory accesses are not tracked by Cilk, making the language unsound, and thus the user is responsible for ensuring that the necessary synchronization is in place; otherwise data races may occur. Previous versions of Cilk supported distributed-memory machines but with the restriction that there be no memory accesses at all except to parameters or return values of tasks, severely limiting the expressiveness of the programming model. In contrast, Regent employs a sound type in which the privileges (and thus side-effects) of tasks are explicit, allowing the implementation to seamlessly provide distributed-memory execution.

Cilk employs a work-stealing scheduler for tasks. This is also available in Regent and is exposed via the mapper; i.e. in Regent the user can choose to use work-stealing for tasks, or another generic or application-specific placement scheme, at their option.

9.1.5 Data Parallelism

Data-parallel languages are a subclass of general implicitly parallel languages that restrict programs to data-parallel operators over collections of objects such as arrays. Within the constraints of this subset, data parallel languages can make it very easy to express certain classes of parallel algorithms and, as with other implicitly parallel languages, avoid by construction pitfalls of explicit parallel programming such as data races and deadlocks.

Efforts in data-parallel languages such as High Performance Fortran (HPF) [45, 56] pioneered compilation techniques for a variety of machines, including distributed-memory. In HPF, a single (conceptual) thread of control creates implicit data-parallelism by specifying operations over entire arrays in a manner similar to traditional

Fortran. This data parallelism is then mapped to a distributed-memory system via explicit user-specified data distributions of the arrays—though the compiler is still responsible for inferring *shadow regions* (i.e. halos that must be communicated) from array accesses. Several implementations of HPF achieved good scalability on structured application [58, 61]. The HPF specification provides extensions for very limited support for sparse data in CSR format (and other similar formats), but no implementations are available for these extensions and no extensions address more general unstructured applications. Regent provides support for both structured and unstructured applications, and Regent's support for multiple partitions enable a more effective hybrid static/dynamic analysis of the intersections of partitions, in Regent's control replication optimization, which serve a similar purpose to HPF's shadow regions.

The Chapel [27] language supports a variety of styles of parallelism, including implicit data parallelism and explicit PGAS-style parallelism. This *multiresolution* design reduces the burden placed on the compiler to optimize Chapel's data parallel subset because users can incrementally switch to other forms of parallelism as needed. However, use of Chapel's explicitly parallel features expose users to the hazards of traditional explicitly parallel programming.

Compared to Regent, Chapel's data parallel subset (which is most similar to Regent's implicit parallelism) only supports a single, static distribution of data, and limited task parallelism. Regent's support for multiple and hierarchical partitions is critical for control replication to optimize implicitly parallel programs for efficient execution on distributed memory machines.

9.1.6 Functional Parallelism

Parallelism has also been explored in the context of functional programming languages. Functional languages provide a number of advantages in this regard. First, programs are composed of functions that are side-effect free, and thus any functional programs (not just those that are data-parallel) are trivially safe to execute in parallel. Second, support for first-class and higher-order functions leads to a natural expression of

parallel pattern such as *map* and *reduce*. In particular, these patterns enable certain forms of data parallelism to be expressed naturally.

MapReduce [32] and Spark [74] are functional programming models that provide support for data parallelism in distributed-memory environments. MapReduce provides support for only two operators, map and reduce, and only in a very specific configuration where each function is only called once, and in a specific order (map and then reduce). Despite the restrictiveness of this model, MapReduce is useful for a variety of data processing workloads. However, for iterative applications, MapReduce can be very inefficient as data is read from persistent storage (such as disk) on each iteration. Spark provides a broader set of operators and is designed so that intermediate results in iterative applications can be maintained in memory and need not be written to disk. MapReduce and Spark were both originally intended for use in industrial data centers, and thus have been tuned for applications with very different performance characteristics and more coarse-grained tasks than typical high-performance computing applications. In order to efficiently parallelize an application, tasks in MapReduce and Spark must generally be on the order of seconds or larger, whereas Regent and other systems for high-performance computing are generally optimized for tasks on the order of milliseconds or tens of milliseconds. Regent (with control replication) has been demonstrated to efficiently schedule tasks of at this granularity on 1024 nodes (12288 cores).

The use of execution templates to reduce control overhead [49] has been explored as a way to improve the scalability of a centralized scheduler. Execution templates can be created, modified, and executed dynamically. Thus execution templates permit substantially more flexibility than Regent's control replication optimization. However, execution templates still require a centralized control to trigger execution, and thus the overhead is $\mathcal{O}(N)$ (or $\mathcal{O}(\log N)$) where N is the number of nodes rather than $\mathcal{O}(1)$ as it is in Regent with control replication.

9.1.7 Nested Parallelism

NESL [18] is a language for nested data parallelism. Nested parallelism provides two advantages over traditional (flat) data parallelism. First, for applications with irregular parallelism (where the iterations of the outermost parallel loop are themselves parallel and take variable time), nested parallel implementations may be able to achieve superior performance by exploiting better load balancing across processors. Second, nested parallel languages promote composability of parallelism, which is increasingly important as supercomputer architectures make increasing uses of deep memory hierarchies.

Regent makes heavy use of nested data and task parallelism. Tasks may recursively launch subtasks to expose additional parallelism to the system. Control replication and other Regent optimizations apply locally to a task and thus are fully composable with nested parallelism.

9.1.8 Implicit Task Parallelism

Task-based parallel systems attempt to overcome the limitations of traditional data parallel languages by focusing on parallelism at the granularity of user-defined *tasks*. Although there is considerable variation in the design of these systems, the unifying feature of these systems is a directed acyclic graph of dependencies between tasks. This dependence graph captures both data and task parallelism in an application and leads naturally to asynchronous execution and aggregation of data transfers, both of which are essential for performance on modern supercomputers. A key question in the design of task-based systems is whether the specification of the dependence graph is implicit or explicit. This section describes implicit task-parallel systems; explicit systems are described in Section 9.2.3.

Among implicit task-parallel systems, dependencies between tasks are typically determined automatically based on the arguments passed to tasks along with privileges declared on task arguments. A key design question for these systems is how precisely to track accesses to data. In particular, does the system support multiple partitions of a given region, or only a single disjoint partition? We will first consider systems

that support multiple partitions.

Models with Multiple Partitioning

Legion [13] is a runtime system for implicit task parallelism that leverages dynamic program analysis to compute a dynamic dependence graph from programs with sequential semantics. Legion provides extensive support for data partitioning [68, 70], and in particular supports dynamic, multiple, hierarchical, and overlapping partitions of a given region. This data model is very expressive and allows the programmer to specify with precision the exact data required in each task. As a result, the dependence graph Legion computes is an accurate representation of the data movement in the program, enabling efficient distributed execution. However, due to its implementation as a runtime system, implicitly parallel Legion programs suffer from a sequential bottleneck in the analysis of tasks (and particularly the data usage of tasks) that can inhibit scalability at large numbers of nodes.

Legion also supports the use of more involved explicit communication constructs that enable scaling to very large node counts [14]. However, the explicit approach can be time-consuming and error-prone, and was identified in a recent study [15] as a challenge for this class of programming systems.

Regent targets the Legion runtime, but provides additional static checks that are not possible in a dynamic runtime system written in a traditional language. Regent leverages these static guarantees to offer control replication, which distributed Legion's dynamic analysis across multiple nodes so the overhead remains constant in the number of nodes. Regent also allows the explicit style, although implicit parallelism is strongly preferred. Regent (with control replication) can be seen as greatly increasing the performance range of the implicit style, allowing more whole codes and subsystems of codes to be written in this more productive and more easily maintained style.

Sequoia [35] is a language for array-based implicitly task-parallel programs. Sequoia supports multiple, hierarchical, and aliased partitions of an array; however in Sequoia the task tree and the sizes of arrays and partitions must be completely determined at compile-time. The Sequoia compiler [46] thus has access to complete static information about the structure of the program and data and can generate efficient code for

distributed-memory targeting a low-level runtime system [41]. A number of additional features allow the language to express certain classes of irregular parallelism [12], however the data model of Sequoia is still restrictive and does not adapt well to unstructured applications. This is in contrast to Regent, which allows substantially more flexible behavior: in particular, the number of tasks, values of arguments to tasks, and precise dependencies (and exact set of elements that must be communicated) between tasks are all permitted to be dynamic, even when using control replication.

Sequoia, like Regent, provides a user-visible mapping interface which allows the user to tune the execution of a Sequoia application. Unlike Regent, a mapping in Sequoia is a static file and must be provided as an input to the compiler. Sequoia provides an autotuner for the mapping interface [55] that is able to relieve the user of the burden of manually mapping an application. In future work we would like to explore similar facilities for Regent.

DPJ [20] is a language for region-based implicit task parallelism based on Java. DPJ is similar to Regent in that it employs a region-based type system to track the effects of tasks. However, unlike Regent, DPJ's regions are not first class and are simply static names for sets of objects. Parallelism is identified statically by the compiler, and the compiler must be conservative in cases where disjointness cannot be statically proven. As a result, DPJ's support for partitioning is also more restrictive. DPJ permits multiple, hierarchical partitions of arrays, but not aliased partitions, and the set of supported partitioning operators is very limited. DPJ was developed for shared-memory systems and has not been demonstrated on distributed-memory machines.

Models with Single Partitioning

In contrast to the above, several task-based parallel systems allow only a single, disjoint partition of a given region (or equivalently, no partitioning, in which case all objects are disjoint by construction). This approach favors implementation simplicity; in particular, a subregion may be identified by the index of the first element of the subregion, making many of the sophisticated checks in Regent unnecessary. However, this decision comes with a cost, as it significantly reduces the expressivity of the model.

This makes it substantially more difficult to describe certain classes of applications in a natural way, and critically reduces the precision of the information the system has about data movement in the application.

StarPU [10] is a runtime system for implicit task parallelism on single (possibly heterogeneous) node. Task dependencies are computed based on privileges and arguments to tasks. Two extensions to StarPU enable distributed execution. StarPU-MPI [9] adds support for explicit message passing via MPI. This approach exposes the programmer to the hazards of explicit distributed programming. An alternative extension generates MPI calls implicitly based on data usage in the program [8]. However, in this approach the sequence of tasks must be executed on all nodes, and thus the time to submit tasks is not $\mathcal{O}(1)$ with the number of nodes (as with control replication) but grows with the number of nodes. StarPU supports only a single partition of a region at a time; this partition may be changed during the execution of the program but must be changed simultaneously on all nodes, requiring global communication to shuffle the data in addition to a global synchronization point. In contrast, Regent supports multiple simultaneous partitions and only requires data movement when the data is actually required by a remote task.

PaRSEC [22] is a runtime system for explicit task parallelism with a frontend compiler that adds support for implicit task parallelism. The frontend compiler takes as input programs with affine accesses over arrays and generates an explicit task-parallel program targeted at the PaRSEC runtime API. Unlike Regent, PaRSEC does not provide support for implicit task parallelism for more general languages, thus limiting the extent to which the approach can be applied to programs with sequential semantics. PaRSEC only supports a single, disjoint partition of arrays.

OpenMP version 4.0 [4] provides support for shared-memory task-based parallelism in traditional languages via compiler directives. OpenMP tasks declare privileges *in*, *out*, or *inout* on task parameters. Dependencies over tasks are computed based on the privileges and the values of arguments passed to tasks; arguments may be pointers to individual elements or dense array sections. However, as with OpenMP's parallel loop constructs, programmer assertions about task privileges are implicitly trusted, and unlike Regent the compiler does not and cannot check these assertions soundly.

Arguments to tasks are not permitted to overlap; thus OpenMP's data model is equivalent to a single, disjoint partitioning of data. For distributed-memory execution, OpenMP tasks must be composed with an explicitly distributed programming model such as MPI. Direct implementations of OpenMP with support for tasks on distributed memory have not been demonstrated.

Jade [57] is an older programming language for implicit task parallelism. Jade does not support data partitioning; instead objects are the units of data movement in the system. This is equivalent to a system that allows only a single, disjoint partition of a given region. Jade natively supports distributed-memory execution, but due to inefficiency inherent in a model with only single partitioning has not been demonstrated to scale efficiently on modern supercomputers. Regent employs multiple partitions specifically to address these challenges.

9.1.9 Speculative Parallelism

Thread-level speculation (TLS) [52, 65] is a hybrid hardware and software approach to parallelizing sequential programs. In TLS, iterations of loops run *speculatively* in parallel, even when the compiler cannot prove statically that it is safe to do so. Conflicting memory accesses are caught dynamically during execution and the iterations that issued those accesses rolled back and re-executed. The mechanism for determining conflicts relies on hardware support and piggybacks on the hardware's cache coherence mechanism. Therefore, TLS can be expected to scale as well as cache coherence protocols in multi-core processors, i.e. to single nodes but not to the scale of modern supercomputers. Regent's control replication places more restrictions on the program source, but in exchange provides scalability to large numbers of nodes.

9.1.10 Domain-Specific Languages

Domain-specific languages (DSLs) have a number of potential advantages over general-purpose languages and runtimes. In particular, by restricting the input domain of the system, DSL implementations may be able to exploit deep domain knowledge to automatically parallelize programs with sequential or declarative semantics. However,

DSLs by definition restrict the input domain of the language and thus any given DSL cannot be expected to be applicable to all possible problems of interest.

Ebb [16] is a DSL for physical simulations. Ebb supports forall-style parallel loops and is able to generate code for CPUs and GPUs via LLVM. Because Ebb and Regent are both implemented in Terra [33], an implementation of Ebb using Regent is also straightforward. Ebb programs are able to automatically take advantage of control replication for efficient execution on distributed-memory machines.

Scout [50] is an embedded DSL in C++ that supports forall-style parallel loops. Scout uses LLVM for code generation for CPUs and GPUs and leverages Legion for distributed execution.

Delite [23] is a compiler framework and runtime for embedded DSLs on heterogeneous architectures for which a number of DSLs have been implemented [40, 66]. The Delite compiler framework provides support for parallel patterns that can implemented efficiently in hardware, in addition to general-purpose and domain-specific optimizations on those parallel patterns. The Delite backend compiler and runtime provide support for code generation and execution on heterogeneous machines, respectively.

9.2 Explicit Parallelism

Explicitly parallel programming models are ones that provide explicitly parallel program semantics. In general, such systems expose the user to the various and well-known pitfalls of traditional parallel programming such as data races and deadlocks, and frequently suffer from inferior ease of use compared to implicitly parallel programming models. The trade-off justifying this cost is that often these programming models are designed to offer as close to bare-metal performance as possible. The system implementation offers no help to the user for finding parallelism in the program, but this also means that the system cannot fail to find parallelism (e.g. due to an overly conservative compiler optimization) because the parallelism is explicitly specified by the user. It is a goal of Regent to provide superior ease of use (via sequential semantics) while maintaining performance comparable to these systems at least for key classes of codes of interest in high-performance scientific computing.

9.2.1 Message Passing

Message passing, particularly via MPI [64], is the dominant programming paradigm on supercomputers today. MPI is a SPMD programming model where multiple copies, or *ranks*, of a program execute simultaneously (usually one rank per physical processor). The address space in message passing models is *not* shared between ranks, and thus the user must explicitly distribute data between the various local memories of the ranks. Somewhat ironically, this distribution—though it must be explicit in the user's mind when designing the program—is not made explicit in the text of the program source, thus compounding issues already inherent in explicit parallel and distributed programming. The user is responsible for avoiding well-known pitfalls of explicitly parallel and distributed programming such as deadlocks, mismatched message sends and receives, non-determinism in control flow, etc.

Achieving overlap between communication and computation can be a challenge in message passing models such as MPI. The asynchronous APIs provided by MPI require the user to explicitly find other code to execute while waiting for the asynchronous operation to complete. By definition, this code must be unrelated to asynchronous operation, otherwise it would depend on the result of that operation. As a result, the proper use of asynchronous constructs in MPI requires that the user apply contortions to the code that can be particularly damaging to the readability and maintainability of the code. In many cases, production applications select to prefer maintainability over extracting the last ounce of performance from the code, and thus may leave performance on the table [14]. Regent provides sequential semantics, but the underlying execution proceeds by constructing a dependence graph of tasks which can be executed in a deferred fashion, thus providing both better ease of use and potentially superior performance.

When used in heterogeneous supercomputers, MPI is typically augmented to produce various MPI+X programming models. For example, MPI+OpenMP might be used in machines with multiple cores per node and MPI+CUDA might be used in machines with NVIDIA GPUs. These hybrid models introduce additional complications. Data movement can be especially challenging. For performance, it is desirable to perform data movement asynchronously and to overlap it with useful

computation. By definition, the only computations that can be overlapped must be unrelated to the data transfers, thus causing contortions to the code when this optimization is applied by hand. In an MPI+X model, this data transfer may need to be performed in two parts: for example when using MPI+CUDA for GPUs separate APIs are required for asynchronous copies via MPI and CUDA and their progress must be monitored separately. Furthermore, the node-level programming model may introduce additional synchronization points, such as OpenMP does at the end of each parallel loop, potentially negating any gains made. As a result, it can be challenging to properly exploit the performance benefits of heterogeneous supercomputers with MPI+X. By targeting Legion, Regent is able to leverage a runtime system that fully manages data transfers to and from any heterogeneous processors, improving the performance portability of codes to such systems.

9.2.2 Partitioned Global Address Space

Partitioned Global Address Space (PGAS) languages such as Split-C [29], UPC [5] and Titanium [3] address some of the pitfalls of explicit message passing. Unlike message passing models, PGAS languages support a global address space. Thus pointers to data are valid anywhere in the machine, though the data might not be present in local memory. Attempts to access non-local data typically result in a message to the remote machine; if these accesses are to individual elements, the accesses may be very inefficient. Regent is similar to PGAS models in that the names of regions are valid anywhere in the machine, and pointers to elements inside regions are effectively indices that are also valid anywhere. Unlike PGAS models, Regent focuses heavily on asynchronous execution and bulk data transfers to achieve efficient execution on modern supercomputers. Regent also provides sequential semantics by default and avoid pitfalls of explicitly parallel programming present in PGAS models.

9.2.3 Explicit Task Parallelism

Realm [69] and OCR [6] are runtime systems for explicit task-based parallelism on heterogeneous distributed-memory machines. StarPU [10] and PaRSEC [22] also

provide an explicitly task-parallel layer, though most end users are expected to use the implicitly task-parallel interfaces to these systems. Tasks in these models take events as preconditions and produce events as postconditions. These events may be used by the application to construct a dependence graph over tasks. An advantage of this approach is that the overhead of launching tasks is reduced as there is no built-in dynamic analysis to determine the dependence graph. However, due to the nature of the task abstraction, the runtime system itself has no knowledge of the desired behavior of the underlying application and thus has no way to ensure correct execution of an application written for its interface. The programmer is thus exposed to most of the traditional hazards of explicitly parallel programming. In practice, these systems are not designed to be targeted directly by the end user but by the designer of a higher-level system such as Regent which will typically take responsibility for the discovery of dependencies between tasks.

Concurrent Collections (CnC) [24] is a graph specification language for explicit task parallelism. Unlike the systems above, where tasks and dependencies are created dynamically via runtime calls, in CnC the dependence graph is specified statically in a graph specification language. To capture certain forms of dynamic behavior, the CnC graph describes not only data dependencies but also control dependencies. Tasks themselves are specified separately and are not able to be checked by the system for adherence to the CnC graph specification.

Uintah [51] is a domain-specific runtime system for programs operating on structured meshes. Uintah supports tasks that operate patches of a mesh, and manages asynchronous execution and data movement between tasks. Dependencies between tasks are specified explicitly and are the responsibility of the user, though certain classes of bugs such as cycles in the dependence graph can be caught automatically during program execution.

9.2.4 Places

X10 [28] is an explicitly parallel programming language with places and hierarchical tasks. Places identify distinct local memories and the compute resources associated

with them. As with PGAS models, pointers to data are valid anywhere in the machine; however unlike PGAS models only local data may be accessed directly, and remote data must be accessed by performing a task launch on the remote node. Places support the launching of asynchronous tasks that may explicitly move computation and data around the machine. As a result, programming with places is explicit parallel programming.

Flat X10 [17] is a subset of this language that restricts programs to a two-level task hierarchy where the top level consists of forall-style parallel loops. A compiler for Flat X10 is able to transform the program into a SPMD-style X10 program with explicit synchronization between tasks. However, as the original Flat X10 program already contains explicit communication (in the form of remote task launches), the compiler need not make changes to the structure of communication in the program. In contrast, control replication is able to automatically generate efficient explicit communication for an implicitly parallel program with implicit data movement.

9.2.5 Actors

Charm++ [44] is an actor-based programming model for high-performance computing. Actors represent units of locally-addressable data and methods that may be invoked on that data. Actors may be moved dynamically between nodes for load-balancing purposes. Actors are similar to tasks in that they lead to a natural expression of task parallelism in the application, and the chain of method calls between actors can be seen as forming a dependence graph of sorts. Actor-based programming is explicitly parallel programming and subject to the normal pitfalls therein.

Chapter 10

Conclusion

This thesis has presented Regent, a programming language for task-based implicit (and explicit) parallelism inspired by the Legion programming model. Regent uses a combination of static and dynamic analysis, compile-time program transformation and runtime scheduling to achieve performance on modern supercomputers. For several classes of programs of interest in high-performance scientific computing, such as unstructured mesh codes, Regent is the first (to the best of our knowledge) to demonstrate practical levels of scalability (up to 1024 nodes and 12288 cores) for implicitly parallel versions of these codes. Essential to these results are a number of static optimizations performed by the Regent compiler, and in particular a novel control replication optimization that automatically transforms implicitly parallel programs into scalable SPMD-style codes.

Regent has been used to develop several mini-applications, as the backend for a DSL for grid and particle based codes, and has even served as the basis of a Stanford course in parallel programming. In this time, we have validated that Regent works for the use cases for which it was intended. We have also identified a number of use cases which require additional thought and investigation.

Today, Regent users still need to write custom mappers in C++. This is something that affects many first-time users as they begin performance runs of Regent applications. The C++ mapping interface is verbose, which is unsurprising, given the power it exposes, and that it is expressed in C++. For a while now, we have thought it would

make sense to design a complementary language to Regent for writing mappers. Such a language could radically reduce the burden of tuning a Regent application for different architectures.

Regent's choice to target Legion has allowed very rapid development of the language. However, Legion's dynamic analysis also imposes a cost that has at times been limiting—thus the focus on control replication to make Regent applications scale. Having demonstrated that control replication works effectively in Regent, we are now investigating whether a similar optimization can be provided directly by the Legion runtime. A difference between Regent and Legion is that in order for Legion to effectively apply control replication, changes will need to be made to the program source. For example, it is only reasonable to dynamically control replicate programs already phrased in terms of index launches. There will likely be other restrictions, and in general these will be properties that Legion cannot check on its own, but will rely on the programmer to maintain. In this sense, while we hope to provide some of the same performance benefits in Legion proper, the soundness benefits can only come from a compiled language such as Regent.

Another possible avenue of investigation would be to attempt to generate a static dataflow program, bypassing Legion's dynamic analysis (at least within the context of a task) and generating code directly to Realm, Legion's underlying execution layer. Regent's control replication optimization, as a by-product, produces what can be seen as a static representation of the program's dataflow. While control replication itself only uses this to generate explicit copies and synchronization, a similar optimization could potentially generate direct calls into a lower-level API such as Realm. The upside would be much lower runtime overhead, enabling Regent to perform efficiently with much more fine-grained tasks.

A compelling property of Regent, and other task-based models, is how they focus on application structure (trees of tasks and regions, and their dependencies or relationships) rather than execution on a particular machine. This structure makes it easier to target complex hardware. Ultimately, by making the hardware easier to use, task-based models such as Regent enable hardware architects to be more aggressive in their designs. In an era where Moore's Law may soon end, this may be turn out to be

critical to continued improvements in performance.

Bibliography

[1] High Performance Computing Center at Stanford University. http://hpcc.stanford.edu/.

[2] Cilk 5.4.6 reference manual. http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf, 1998.

[3] Titanium language reference manual. http://titanium.cs.berkeley.edu/doc/lang-ref.pdf, 2006.

[4] OpenMP application program interface. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, 2013.

[5] UPC language specifications, version 1.3. http://upc.lbl.gov/publications/upc-spec-1.3.pdf, 2013.

[6] The Open Community Runtime interface. https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;f=ocr/spec/ocr-1.1.0.pdf, 2014.

[7] Piz Daint & Piz Dora - CSCS. http://www.cscs.ch/computers/piz_daint, 2016.

[8] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. Technical report, Inria, 2016.

[9] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting*, pages 298–299. Springer, 2012.

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

[11] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Programming distributed memory sytems using OpenMP. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[12] Michael Bauer, John Clark, Eric Schkufza, and Alex Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in Sequoia. In *PPoPP*, pages 13–24, 2011.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Supercomputing (SC)*, 2012.

[14] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Structure slicing: Extending logical regions with fields. In *Supercomputing (SC)*, 2014.

[15] Janine Bennett, Robert Clay, Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke, Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin, Ryan Grant, Si Hammond, Stephen Olivier, Laxmikant Kale, Nikhil Jain, Eric Mikida, Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler, Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland, Pat McCormick, Samuel Gutierrez, Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, and Todd Gamblin. ASC ATDM Level 2 milestone

#5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms.

[16] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Transactions on Graphics (TOG)*, 35(2):21, 2016.

[17] Ganesh Bikshandi, Jose G. Castanos, Sreedhar B. Kodali, V. Krishna Nandivada, Igor Peshansky, Vijay A. Saraswat, Sayantan Sur, Pradeep Varma, and Tong Wen. Efficient, portable implementation of asynchronous multi-place programs. In *PPoPP*, pages 271–282. ACM, 2009.

[18] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *International Conference on Functional Programming (ICFP)*, pages 213–225, 1996.

[19] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, et al. Effective automatic parallelization with Polaris. In *International Journal of Parallel Programming*. Citeseer, 1995.

[20] 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 *OOPSLA*, 2009.

[21] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *Supercomputing (SC)*, page 33. ACM, 2013.

[22] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[23] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 89–100. IEEE, 2011.

[24] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent Collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[25] Donald E. Burton. Consistent finite-volume discretization of hydrodynamics conservation laws for unstructured grids. Technical Report UCRL-JC-118788, Lawrence Livermore National Laboratory, Livermore, CA, 1994.

[26] Ümit Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011.

[27] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *Int'l Journal of HPC Apps.*, 2007.

[28] 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 *OOPSLA*, 2005.

[29] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing (SC)*, pages 262–273, 1993.

[30] Ron Cytron, Jim Lipkis, and Edith Schonberg. A compiler-assisted approach to SPMD execution. In *Supercomputing (SC)*, pages 398–406. IEEE Computer Society Press, 1990.

[31] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 1998.

[32] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Opearting Systems Design & Implementation (OSDI)*, pages 10–10, 2004.

[33] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. PLDI, 2013.

[34] H. Carter Edwards and Christian R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *Extreme Scaling Workshop (XSW), 2013*, pages 18–24, Aug 2013.

[35] 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 *Supercomputing*, November 2006.

[36] Charles R. Ferenbaugh. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience*, 2014.

[37] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[38] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[39] Jay P. Hoeflinger. Extending OpenMP to clusters. *White Paper, Intel Corporation*, 2006.

[40] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 349–362, 2012.

[41] Mike Houston, Ji-Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 143–152, 2008.

[42] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua - an extensible extension language. *Softw., Pract. Exper.*, 1996.

[43] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Supercomputing (SC)*, 1991.

[44] L.V. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *OOPSLA*, pages 91–108, 1993.

[45] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 7–1. ACM, 2007.

[46] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 226–236, 2007.

[47] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. A hybrid approach of OpenMP for clusters. PPoPP, pages 75–84. ACM, 2012.

[48] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[49] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[50] Patrick McCormick, Christine Sweeney, Nick Moss, Dean Prichard, Samuel K. Gutierrez, Kei Davis, and Jamaludin Mohd-Yusof. Exploring the construction of a domain-aware toolchain for high-performance computing. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 1–10. IEEE Press, 2014.

[51] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin Berzins. Investigating applications portability with the Uintah DAG-based runtime system on petascale supercomputers. In *Supercomputing (SC)*, pages 1–12, 2013.

[52] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical report, Stanford, 1997.

[53] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Distributed memory code generation for mixed irregular/regular computations. PPoPP, pages 65–75. ACM, 2015.

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

[55] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally.

[56] Harvey Richardson. High Performance Fortran: history, overview and current developments. *Thinking Machines Corporation*, 14:17, 1996.

[57] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 1998.

[58] Hitoshi Sakagami, Hitoshi Murai, Yoshiki Seo, and Mitsuo Yokokawa. 14.9 TFLOPS three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 51–51. IEEE, 2002.

[59] Mitsuhisa Sato, Hiroshi Harada, Atsushi Hasegawa, and Yutaka Ishikawa. Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *Scientific Programming*, 9(2, 3):123–130, 2001.

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

[61] Yoshiki Seo, Hidetoshi Iwashita, Hiroshi Ohta, and Hitoshi Sakagami. HPF/JA: extensions of High Performance Fortran for accelerating real-world applications. *Concurrency and Computation: Practice and Experience*, 14(8-9):555–573, 2002.

[62] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for HPC with logical regions. In *Supercomputing (SC)*, 2015.

[63] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. Control Replication: Compiling implicit parallelism to efficient SPMD with logical regions. In *Supercomputing (SC)*, 2017.

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

[65] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. ISCA, pages 1–12, 2000.

[66] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *International Conference on Machine Learning (ICML)*, pages 609–616, 2011.

[67] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 2000.

[68] Sean Treichler, Michael Bauer, and Alex Aiken. Language support for dynamic, hierarchical data partitioning. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[69] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

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

[71] Rob F. Van der Wijngaart, Abdullah Kayi, Jeff R. Hammond, Gabriele Jost, Tom St. John, Srinivas Sridharan, Timothy G. Mattson, John Abercrombie, and Jacob Nelson. Comparing runtime systems with exascale ambitions using the parallel research kernels. In *International Conference on High Performance Computing*, pages 321–339. Springer, 2016.

[72] Rob F. Van der Wijngaart and Timothy G. Mattson. The parallel research kernels. In *HPEC*, pages 1–6, 2014.

[73] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO*, pages 24–32, 2007.

[74] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.