A HYBRID APPROACH TO AUTOMATIC PROGRAM PARALLELIZATION VIA EFFICIENT TASKING WITH COMPOSABLE DATA PARTITIONING

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

Wonchan Lee
December 2019

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

Alex Aiken, Primary Adviser

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

John Mitchell

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

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Despite the decades of research, distributed programming is still a painful task and programming systems designed to improve productivity fall short in practice. Auto-parallelizing compilers simplify distributed programming by parallelizing sequential programs automatically for distributed execution. However, their applicability is severely limited due to the fundamental undecidability of their static analysis problem. Runtime systems for implicit parallelism can handle a broader class of programs via an expressive programming model, but their runtime overhead often becomes a performance bottleneck. To design a practical system for productive distributed programming, one must combine the strengths of different parallelization paradigms to overcome their weaknesses when used in isolation.

This dissertation presents a *hybrid approach* to automatic program parallelization, which combines an auto-parallelizing compiler with an implicitly parallel tasking system. Our approach parallelizes programs in two steps. First, the auto-parallelizer materializes data parallelism in a program into task parallelism. Next, the tasking system dynamically analyzes dependencies between tasks and executes independent tasks in parallel. This two-stage process gives programmers a second chance when the auto-parallelizer "fails": When a part of a program is not amenable to the compiler auto-parallelization, the programmer can gracefully fall back to the runtime parallelization by writing that part directly with task parallelism. Furthermore, hand-written tasks can be seamlessly integrated with the auto-parallelized part via *composable data partitioning* enabled by our auto-parallelizer, which allows them to share the partitioning strategy and thereby avoid excessive communication.

Key to the success of this hybrid approach is to minimize the overhead of the tasking system. To achieve this goal, we introduce *dynamic tracing*, a runtime mechanism for efficient tasking. The most expensive component in the tasking system is dynamic dependence analysis. Although this dynamic analysis is necessary when applications exhibit true dynamic behavior, the analysis is redundant for common cases where dependencies are (mostly) unchanging. Dynamic tracing eliminates this redundancy in dynamic dependence analysis by recording the dependence analysis of an execution trace and then replaying the recording for the subsequent occurrences of the same trace. To guarantee that a recording of a trace correctly replaces the trace's original analysis, dynamic tracing also records memory locations that hold valid data when it records a trace and replays the recording only when those locations are still valid. Dynamic tracing significantly improves the efficiency of tasking, and thereby brings the strong scalability of explicit parallelism to implicit task parallelism.

Acknowledgments

My advisor Alex Aiken has been always my first line of defense, and often the last one as well. Alex went through all the rough drafts that I produced and turned them into a human readable form. He also had patiently waited for me to find my own ideas despite the lack of progress during my initial years at Stanford. With his guidance, I was able to eventually find those ideas, develop them, and turn them into this dissertation. I am deeply grateful for everything that he has done for me throughout my time at Stanford.

This dissertation would have been literally impossible without the Legion project. My work is built on the great foundation that has been perfected over the past few years: Regent (by Elliott Slaughter), Legion (by Mike Bauer), and Realm and DPL (by Sean Treichler). Without them paving the way I would have taken much longer to achieve much less. Fantastic members of the Legion team, Mike Bauer, Sean Treichler, Elliott Slaughter, Manolis Papadakis, Karthik Murthy, Todd Warszawski, Zhihao Jia and the others, have been always available whenever I am in dire need of help. I feel fortunate to be a part of this amazing team and I am glad that I managed to contribute to their already fascinating project.

Helping Alex start the CS315B course about Legion was a rare opportunity for me to disseminate ideas from a research project in the form of classroom education. Working with students from various departments, I learned a lot about what development tools must provide to make them accessible to a broad audience, a valuable lesson that could not be learned elsewhere. Some of the CS315B students, such as Akshay Subramaniam, Jaehwan Choi, Grace Johnson, and Ellis Hoag, developed their final projects into research projects even after class, and collaborating with them on

those projects has been both inspiring and rewarding.

Part of my dissertation work came out of my collaboration with the members of the PSAAP II project, Gianluca Iaccarino, Thomas Economon, Hilario Torres, Lluis Jofre, Heather Pacella, and the others. I would like to thank all of them for being consistent supporters of the Legion project and brave testers that have had to bump into a lot of troubles trying out the most experimental features developed over the past few years.

All my accomplishments would have been definitely impossible without support from my wife Kyungjin. Living in a country other than one's own is always tough and I could not imagine how difficult it would have been for her to decide to join me in the United States. I cannot be more thankful that I can share every moment with her throughout this journey. I would like to thank my parents and family as well for being supportive of me pursuing a Ph.D. abroad. I sincerely hope one day I will be able to reciprocate their love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed programming is hard. Each node in a distributed memory machine has direct access only to its local memory. Thus, programs for such a machine must explicitly move data between memories when the data is partitioned across multiple nodes. Besides this burden, programs must also avoid well-known parallel programming hazards, such as deadlocks and data races, which are in and of themselves notoriously difficult to fix. Writing a correct program, let alone an efficient one, is a challenging task.

The first generation of programming systems for high performance computing tried to tackle this programmability issue with *auto-parallelizers* [7, 25, 42, 47, 49, 74], a compiler that transforms sequential programs directly into parallel versions running on distributed memory machines. As the parallelizability of a program cannot be determined statically in general, only data parallel loops, which can be checked as parallelizable at compile time, were targeted. Nevertheless, these auto-parallelizers perform a great deal of non-trivial code transformations to generate the code necessary for distributed execution, such as data partitioning and communication and synchronization between distributed processes. When applicable, auto-parallelization is an accessible form of distributed programming as it guarantees *sequential semantics* for the output parallel program, making parallel programs almost as easy to write as sequential programs.

Despite years of research and standardization efforts [47], auto-parallelizers for

distributed memory machines have not been widely adopted. A major hindrance to adoption is that there has been no good solution when the auto-parallelizers "fail": Auto-parallelizing the entire program is often infeasible due to the fundamental limitations of static analyses. In such cases, programmers would want to apply auto-parallelization selectively to portions of the program and combine the result with the rest of the program that would be manually parallelized. Unfortunately, this *composability* problem has never been fully addressed for auto-parallelizers, limiting their applicability for applications that are not auto-parallelizable in their entirety. Furthermore, programmers sometimes find auto-parallelized code has unsatisfying performance, because many of the performance-related decisions cannot be made accurately in the absence of runtime information. In particular, indirect accesses, which have been a major challenge for auto-parallelizers, cannot be handled optimally without knowing the index values, which are only available at runtime. Even worse is that auto-parallelizers provide only limited ways to fix performance issues and cannot be tuned to their full extent.

These issues stem from the fact that programming models in which auto-parallelizers generate distributed programs lack high-level abstractions that express concepts involved in parallelization. For example, when composing multiple data-parallel programs, they must share data partitions whenever they can to avoid unnecessary communication. However, almost all code generation approaches for distributed memory systems, even those proposed recently [21, 60, 61], depend only on explicit, low-level communication, and they synthesize the whole data partitioning code. As a result, output programs embed an opaque program component implementing a particular partitioning strategy, which makes them difficult to compose with other programs. Although programming systems such as High Performance Fortran (HPF) [47] and a family of related Fortran versions [25, 42] provide *data distributions*, program annotations describing primary data partitions, they cannot be freely exchanged between programs because "data distributions were not themselves data objects" [47]. Other decisions that are critical to performance, such as planning of data movement and allocation and management of non-local data, all belong to the internals of auto-parallelizers and there is no programmable interface to change them because they are

not directly expressible but encoded in the programming model.

On the other hand, there has been recent progress on implicitly parallel tasking models [10, 17, 30, 43, 64], which overcome the main limitations of compiler-based approaches. These programming models are built on abstractions for describing the decomposition of computation and data: A program uses *tasks* to sub-divide its computation; a task is an opaque function that occupies a processing unit for the duration of its execution. When the program executes, it submits tasks to the runtime system, which then discovers parallelism by dynamically analyzing dependencies between the tasks. In many cases, each of the tasks accesses only a sub-collection of the data and tasking models provide first-class data partitions [17, 64] to make a decomposition of data easily expressed and shared by tasks. First-class data partitions also enable the runtime system to handle necessary synchronization and communication between dependent tasks as they give the runtime system a precise description of the data shared by tasks. With tasks and first-class data partitions, programs can express complex, irregular patterns of parallelism, which are beyond the capability of past auto-parallelizers, and yet enjoy sequential semantics guaranteed by the runtime system. Furthermore, these abstractions facilitate a programmable interface for performance tuning [17], achieving portable and transparent performance for programs that are automatically parallelized at runtime.

However, implicit parallel tasking models have drawbacks that sometimes make them unattractive. First of all, unlike auto-parallelizers, these tasking systems perform dependence analysis dynamically, which can become a performance bottleneck, especially for *strong scaling* cases, i.e., cases where the total problem size is held fixed while the size of parallel machine is increased. Although programs of highly dynamic nature would benefit from the flexibility of dynamic dependence analysis, the full analysis is often redundant for cases where the dependence structure rarely or never changes. Nevertheless, identifying such cases and safely and selectively avoiding the full dependence analysis for them is a non-trivial problem that is yet to be solved. Secondly, even though the runtime system automatically discovers potential opportunities for parallel execution, programmers must still expose parallelism by issuing a sufficient number of parallel tasks in their programs. Writing such programs with

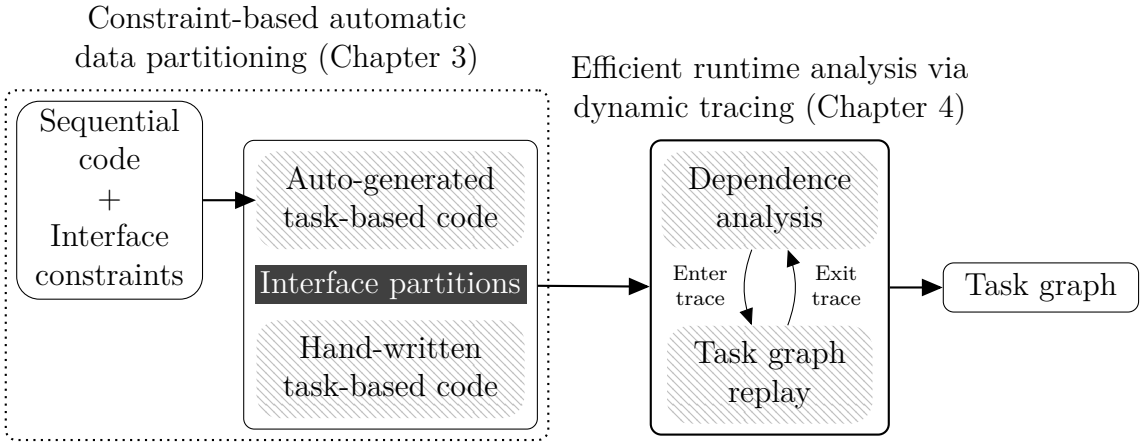


Figure 1.1: A hybrid approach to automatic program parallelization

good performance is often far from straightforward even for data parallel problems, because they require a careful choice of data partitions to minimize the induced communication between different data parallel sections of code.

This dissertation presents a hybrid approach to automatic program parallelization combining the benefits of compiler-based and runtime-based approaches. In this approach, the compiler auto-parallelization uses an implicitly parallel tasking model as the target programming model to leverage the flexibility of runtime-based approaches. Abstractions for data partitions enable composability for auto-parallelized output programs, making them easily integrated with the hand-written task-based code. The runtime system is then optimized to have efficiency close to that of compiler-optimized or hand-written parallel programs, via a just-in-time (JIT) specialization of the runtime analysis for the parts that do not require a full analysis.

Figure 1.1 shows the overall structure of this hybrid approach. First, the portion of a program that is amenable to compiler auto-parallelization is parallelized by the compiler. Instead of emitting low-level code with explicit synchronization and communication, this compiler transformation produces a task-based program using first-class data partitions as the output. The transformation ensures that the output task-based program exposes all data parallelism available in the input program. *Interface constraints* in the input program capture input/output relationships between the program to be parallelized and the surrounding code, and they guide the compiler

to generate efficient code integrated with that surrounding code by reusing interface partitions that the constraints specify. Next, the composed program is parallelized by the runtime system for distributed execution. The runtime system performs a dynamic dependence analysis on tasks in the composed program and constructs a *task graph*, a DAG of tasks whose edges encode task dependencies, which executes on a distributed memory machine. Key to efficient runtime analysis is *dynamic tracing*, a runtime mechanism to memoize dependence analysis results, which consists of the following components:

- The *recorder* first records the task graph from dependence analysis of a *trace*, a sequence of recurrent tasks, along with the *precondition* upon which the recorded graph can safely replace the normal dependence analysis.
- The *replayer* then identifies subsequent occurrences of the same trace and replays the recording whenever the precondition is satisfied.

The tracing eliminates the overhead of dynamic dependence analysis for traces whenever it can, thereby bringing the efficiency of manually parallelized programs to implicitly parallel programs having traces.

In the following sections, we provide an overview of the two components enabling this hybrid approach: automatic data partitioning and dynamic tracing.

1.1 Automatic Data Partitioning

Data partitioning is an essential step to exploit data parallelism in implicitly parallel tasking models. For a data parallel loop, parallelism can be realized by partitioning the data into sub-collections and running parallel tasks each of which executes a subset of loop iterations using its sub-collection argument. To preserve the semantics of the original loop, the data partition must be *legal*; i.e., the sub-collection argument to each task must contain all the data accessed by the task's iterations. As programs generally have multiple data access patterns in different loops, the possible legal partitions are those satisfying all the constraints of all loops, while the performant partitions are

```
1 for p in Particles:
2   c = Particles[p].cell
3   Particles[p].pos += f(Cells[c].vel, Cells[h(c)].vel)
4
5 for c in Cells:
6   Cells[c].vel += g(Cells[c].acc, Cells[h(c)].acc)
```

Figure 1.2: Parallelizable loops

a subset of the legal partitions. Therefore, the goal of the auto-parallelizer is to automatically find performant legal partitions for a given program.

We tackle this data partitioning problem with a *constraint-based* approach. The crux is characterizing all possible legal partitions using *partitioning constraints*, a set of constraints on first-class data partitions. Partitioning constraints are inferred automatically from data accesses in programs, and they serve as a specification for implementations of data partitioning. To find implementations that match the specification, we employ a constraint solver that synthesizes partitioning code in DPL, the Dependent Partitioning Language [71], a domain-specific language for data partitioning. Among all partitioning implementations matching the specification, the solver finds the one with minimal communication, which is often the most performant implementation.

Our constraint-based approach enables composability in auto-parallelized programs by allowing programmers to use constraints to guide the constraint solving process. For example, when the code to be parallelized must accept input from another program component with fixed data partitions and the programmer wants to reuse the partitions in the parallelized code, he can provide interface constraints that capture invariants on those partitions. Our constraint solver then exploits these external constraints to discharge some or all partitioning constraints and synthesizes partitioning code that reuses some or all of the data partitions that they specify; those external constraints serve as an interface conveying information about existing partitions to our automated data partitioning process.

We illustrate our constraint-based approach using the program in Figure 1.2,

```
1  task T1(Particles, Cells1, Cells2):
2    for p in Particles:
3      c = Particles[p].cell
4      Particles[p].pos += f(Cells1[c].vel, Cells2[h(c)].vel)
5
6  task T2(Cells3, Cells4):
7    for c in Cells3:
8      Cells3[c].vel += g(Cells3[c].acc, Cells4[h(c)].acc)
9
10 parallel for i in P1:
11   T1(P1[i], P2[i], P3[i])
12
13 parallel for j in P4:
14   T2(P4[j], P5[j])
```

Figure 1.3: Loops parallelized with tasks

which showcases a common pattern of using indirect accesses to establish relationships between different physical entities; the program is written in Regent [64], the high-level task-based programming language that we use in this dissertation.

The program in Figure 1.2 stores properties of particles and cells in *regions* `Particles` and `Cells`. A region is a collection of values which can be partitioned into *subregions* (sub-collections of the original values). All elements of a region have the same type, and every element has a unique index. The values in a region may have *fields*, such as the `cell`, `vel`, and `acc` fields used in Figure 1.2. The first loop iterates over `Particles` to update the position of each particle `p`. The index `c` of the cell where each particle resides is stored in `Particles[p].cell` (line 2). The change in each particle's position is then computed using the velocity of the cell at `c` and its neighbor `h(c)` (line 3). The second loop updates the velocity of each cell similarly (line 6).

The program in Figure 1.3 parallelizes the loops in Figure 1.2 using partitions of `Particles` and `Cells`. Each **`parallel for`** loop launches tasks for subregions in the partition, each of which runs a subset of the original loop iterations.

The partitions $P_1, \ldots, P_5$ in Figure 1.3 are legal only when they make the following

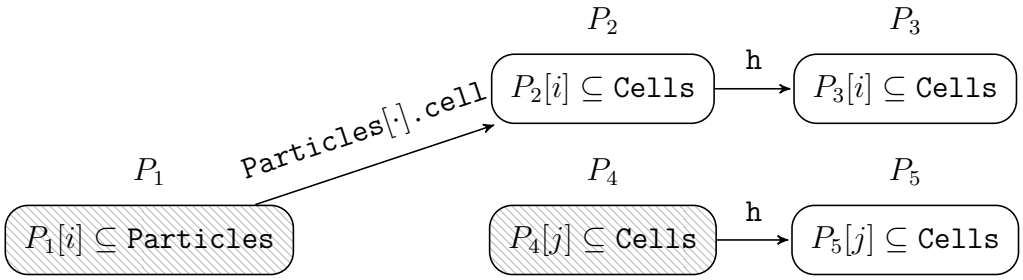


Figure 1.4: Partitioning constraints

indirect accesses safe:

- `Cells1[c].vel` at line 4;
- `Cells2[h(c)].vel` at line 4; and
- `Cells4[h(c)].acc` at line 8.

Figure 1.4 shows the partitioning constraints that capture the conditions under which $P_1, \ldots, P_5$ in Figure 1.3 are legal. Each node in the graph corresponds to a partition. The node labeled with P_1 denotes a partition of `Particles`, whereas the others are (potentially different) partitions of `Cells`. Shaded nodes represent partitions that must be *complete*; a partition is complete when its subregions include all elements of the region. Nodes for partitions P_1 and P_4 are shaded because they must cover the iteration space of the loops at lines 1 and 4. Edges between nodes specify constraints on partitions. The edge from P_2 to P_3, labeled with the function `h`, requires that each subregion $P_3[j]$ contain the image of $P_2[j]$ under `h`, that is, $\forall (k, v) \in P_2[j].\ \exists v'.\ (\texttt{h}(k), v') \in P_3[j]$. The edge from P_4 to P_5 describes the same constraint but on P_4 and P_5. The other edge between P_1 and P_2 is interpreted similarly:

$$\forall (k, v) \in P_1[i].\ \exists v'.\ \left(\texttt{Particles}[k]\texttt{.cell}, v'\right) \in P_2[i]$$

Figure 1.5 gives two partitioning strategies satisfying the constraints in Figure 1.4, expressed as DPL programs that construct partitions using the high-level partitioning operators **`equal`**, **`image`**, and **`preimage`**. DPL is the partitioning sub-language of

```
1 P1 = equal(Particles, N)
2 P2 = image(P1, Particles[·].cell, Cells)
3 P3 = image(P2, h, Cells)
4 P4 = equal(Cells, N)
5 P5 = image(P4, h, Cells)
```

(a) Program A

```
1 P2 = P4 = equal(Cells, N)
2 P1 = preimage(Particles, Particles[·].cell, P2)
3 P3 = P5 = image(P2, h, Cells)
```

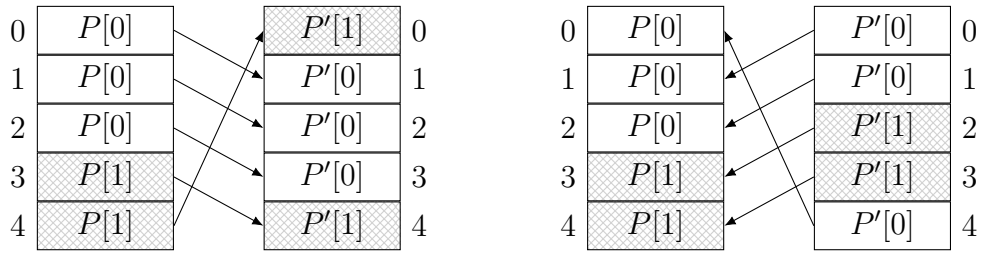
(b) Program B

Figure 1.5: DPL programs solving the constraints in Figure 1.4

Regent that computes partitions of regions at runtime. The main idea in DPL is that some partitioning operators, such as **equal**, create partitions of regions directly, while others compute a new partition as a function of an existing partition (thus the name *dependent* partitioning language). Sophisticated data partitions can be constructed by composing the small set of primitive DPL operators.

In Figure 1.5, program A derives P_2, P_3, and P_5 from **equal** partitions of P_1 and P_4; the **equal** operator creates a complete partition of a region with (approximately) equal size subregions. Partitions P_2, P_3 and P_5 use **image** partitions to satisfy the partitioning constraints. The **image** operator uses an existing partition and a function

$$\texttt{f}(i) = (i+1)\%5$$



(a) $P' =$ **image**$(P,\texttt{f},-)$ (b) $P' =$ **preimage**$(-,\texttt{f},P)$

Figure 1.6: Image and preimage operators

to define a compatible partition of a region. For example, if $P_2 = \langle r_1, \ldots, r_n \rangle$, then the statement $P_3 = \texttt{image}(P_2, \texttt{h}, \texttt{Cells})$ creates $P_3 = \langle \texttt{h}(r_1), \ldots, \texttt{h}(r_n) \rangle$, where $\texttt{h}(r_i) \subseteq \texttt{Cells}$. Figure 1.6a gives a visual representation of the **image** operator: The region on the left is already partitioned into two subregions, indicated by the sets of light and dark elements. The image of function `f` mapping elements of the left-hand region to elements of the righthand region then defines two subregions of the righthand region.

Program B implements a different strategy, first creating an **equal** partition of `Cells` for both P_2 and P_4. Note that P_2 is assigned a complete partition even though the partitioning constraint does not require it to be complete; as long as P_2 contains the image of $\texttt{Particles}[\cdot]\texttt{.cell}$, it can have extra elements. To construct the partition P_1 from the already defined P_2, program B uses the **preimage** operator. As illustrated in Figure 1.6b, **preimage** takes an existing partition of the region on the righthand side and constructs a compatible partition using the preimage of the function; i.e., if the provided partition is $\langle r_1, \ldots, r_n \rangle$ and the function is `h`, then the computed partition is $\langle \texttt{h}^{-1}(r_1), \ldots, \texttt{h}^{-1}(r_n) \rangle$. Finally, P_3 and P_5 are computed using the image of P_2 under `h`.

Without any prior knowledge about `Cells` and `Particles`, it is not clear whether the partitioning strategy in Figure 1.5a or Figure 1.5b is better. Program A has an additional pair of partitions of `Cells`, which is not necessarily worse than program B if communication due to the extra partitions is justified; in a scenario where spatial distribution of the particles is significantly skewed, program B can suffer from load imbalance in the first loop in Figure 1.3, whereas program A is immune to this issue because the subregions of P_1 have equal size. On the other hand, if the particles in each subregion of P_1 are spread throughout the domain, each subregion of P_2 can be as big as `Cells`, leading to excessive communication.

Given that we cannot identify an optimal DPL program that satisfies the constraints at compile-time, we use heuristics to guide the constraint resolution process. For example, when a given set of partitioning constraints admit multiple DPL programs, our constraint solver chooses the one with the fewest partitions (program B in this example). This approach does not always produce satisfactory solutions when important information about the execution context is missing. Programs also often

```
1  parallel for i in pParticles:
2    for p in pParticles[i]:
3      new_cell = locate(pParticles[i][p].pos)
4      if pParticles[i][p].cell != new_cell:
5        pParticles[i][p].cell = new_cell
6        find j such that new_cell ∈ pCells[j]
7        if i != j:
8          send pParticles[i][p] to pParticles[j]
9
10 assert(image(pParticles,Particles[·].cell,Cells) ⊆ pCells)
```

Figure 1.7: Pseudo-code example with a user-provided constraint

have parts that are hard to auto-parallelize well, or may not be possible to auto-parallelize at all. Our approach can gracefully handle these situations by allowing programmers to provide additional constraints encoding knowledge of which strategies are best and/or existing partitions used outside the scope of auto-parallelization. For example, if the loops in Figure 1.3 were embedded in an outer loop where particles' pointers to cells are updated every iteration, the DPL program in Figure 1.5b would need to repartition `Particles` every iteration to reflect the updates. If only a few particles change cells on each iteration, then repartitioning the entire `Particles` region is wasteful. A simple way to mitigate this inefficiency is to exchange particles manually; the pseudo-code in Figure 1.7 sends a particle to the right "owner" whenever the cell to which the particle moves belongs to a subregion different from the current one. The most important part in this pseudo code is the assertion at line 10 specifying the invariant on `pParticles` and `pCells`, i.e., that the subregion `pCells`$[i]$ contains all the cells pointed to by the particles in `pParticles`$[i]$. The constraint solver uses this assertion to discharge all partitioning constraints in Figure 1.4 except those on P_3 and P_5, for which the solver emits the following DPL program using `pCells` to derive P_3 and P_5:

$$P_3 = P_5 = \texttt{image(pCells, h, Cells)}$$

This example demonstrates the key benefit of constraint-based approaches that separate specification from implementation [8]; as long as the manual particle exchange

```
1 task F(R) reads(R),writes(R)
2 task G(R) reads(R),writes(R)
3
4 while *:
5   for i = 0, 2: F(A[i])
6   for i = 0, 2: G(A[h(i)])
```

Figure 1.8: Example program

code maintains the invariant, the entire program mixing parts that are parallelized by different means is correct.

1.2 Dynamic Tracing

Task-based programs, whether hand-written or auto-generated by the constraint-based approach in Section 1.1, require a dependence analysis to be parallelized because dependencies between tasks are implicit in those programs. A dependence analysis is a process that converts a task-based program into a *task graph*, where all task dependencies are materialized. A task graph is a DAG of tasks whose edges encode task dependencies; the graph is used by runtime systems for concurrent task execution on distributed memory machines. A constructed task graph can be executed by continuously exhausting "ready" tasks in the graph, i.e., by repeatedly removing and executing a set of tasks with no predecessors until the graph becomes empty.

Runtime systems perform this dependence analysis dynamically to get accurate dependencies. For example, the program in Figure 1.8 issues four tasks `F(A[0])`, `F(A[1])`, `G(A[h(0)])`, `G(A[h(1)])` for every iteration of the while loop, where the opaque function `h` hampers any attempt to statically analyze dependencies between these tasks. In contrast, precise dynamic dependence analysis is straightforward. For example, if `h(0)` = 1 and `h(1)` = 0, dynamic dependence analysis shows there are dependencies between `F(A[0])` and `G(A[h(1)])`, and between `F(A[1])` and `G(A[h(0)])`. Note that task declarations at lines 1-2 are annotated with *permissions* describing how tasks `F` and `G` access the region argument `x`, which allows the dynamic dependence analysis to infer

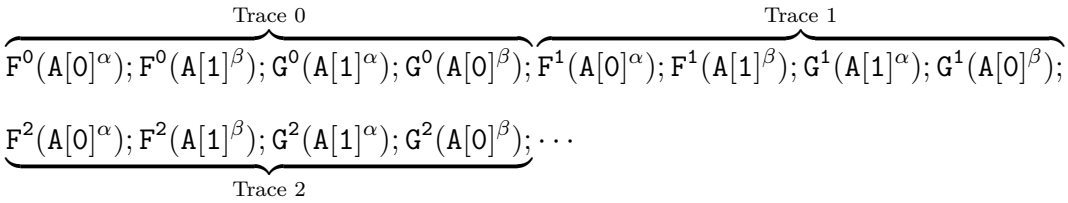


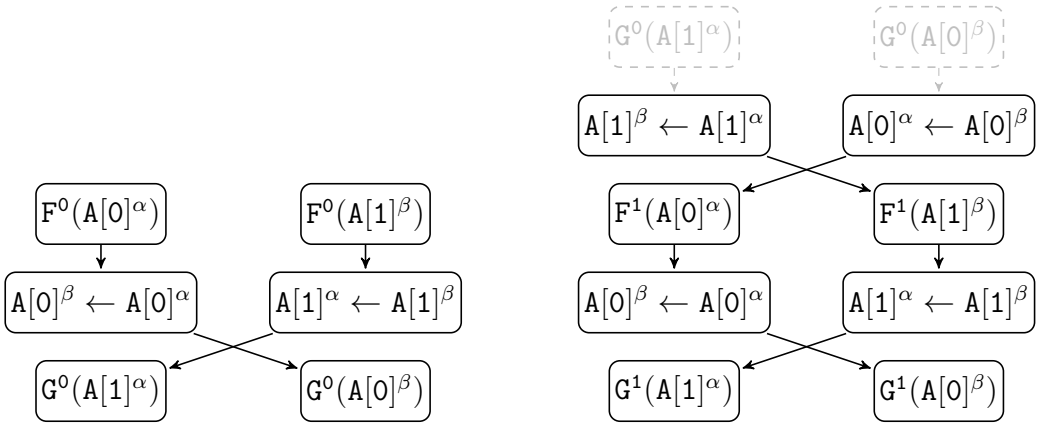
Figure 1.9: Traces of the program in Figure 1.8

dependencies without introspecting task definitions. Many implicitly parallel tasking paradigms use permissions for this purpose [10, 11, 17, 30, 43, 56, 64].

Unfortunately, this dynamic dependence analysis often becomes a performance bottleneck. The cost of dynamic dependence analysis might be hidden by running the analysis in parallel with the application, but only when the cost of analyzing a task is on average less than the task's execution time [17]. The cost of dynamic dependence analysis therefore places a lower bound on the granularity of tasks that can be handled efficiently and how well applications strong scale. To avoid limiting performance, dynamic dependence analysis must be as efficient as possible.

Dynamic tracing reduces the overhead of dynamic dependence analysis by avoiding redundant analyses for *traces* of tasks, which are simply sequences of tasks issued by the program. Programs often have a trace of tasks that occurs repeatedly. For example, the while loop in Figure 1.8 issues a trace of two `F` tasks and two `G` tasks in each iteration. For the same trace of tasks the dynamic dependence analysis often, but not always, produces the same task graph. The key idea in dynamic tracing is to capture the task graph for such traces and reuse it to replace the trace's dependence analysis whenever possible.

However, care must be taken when reusing a captured task graph because data dependencies on distributed memory machines may require data movement. Figure 1.9 shows an example sequence of tasks issued by the program where traces are annotated. We use node identifiers α, β, etc. as superscripts to regions to distinguish different *instances* of the same region on different nodes. In Figure 1.9, for example, the task $\texttt{F}^0(\texttt{A}[0]^\alpha)$ uses an instance of the region $\texttt{A}[0]$ on node α, whereas $\texttt{G}^0(\texttt{A}[0]^\beta)$



(a) Trace 0 (b) Trace 1

Figure 1.10: Task graphs for traces in Figure 1.9

uses an instance of that region on node β. Tasks execute on the node where their arguments are placed and we use superscripts on their names to differentiate multiple invocations of the same task. Because for each value of `i`, the tasks `F(A[i])` and `G(A[i])` execute on different nodes, and since `F(A[i])` writes to `A[i]`, the updated value of `A[i]` must be copied to the node where `G(A[i])` will run; for example, the task graphs for traces 0 and 1 in Figure 1.10 have such copies. (Note that misaligned region accesses in this example rarely occur in realistic scenarios and are presented only to discuss salient issues in dynamic tracing.) Furthermore, unlike the task graph for trace 0, the task graph for trace 1 has additional copies prepended to tasks `F(A[0])` and `F(A[1])` because of their data dependencies on `G(A[1])` and `G(A[0])` in the previous trace. This difference demonstrates that task graphs for the same trace can be sometimes different when the set of *valid* instances, i.e., instances that contain the last changes made to regions, are different. Therefore, to safely replay a captured task graph, dynamic tracing must also record valid instances at the point where that graph is captured as the *precondition* and check the validity of instances in this precondition before replaying the graph. Having the task graph from trace 1 recorded, dynamic tracing can replay this graph from trace 2 onward as all of those traces satisfy the precondition.

Dynamic tracing must also cope with changes in the *mapping* between regions and instances. Suppose now that in trace k the choices of nodes for the regions of tasks

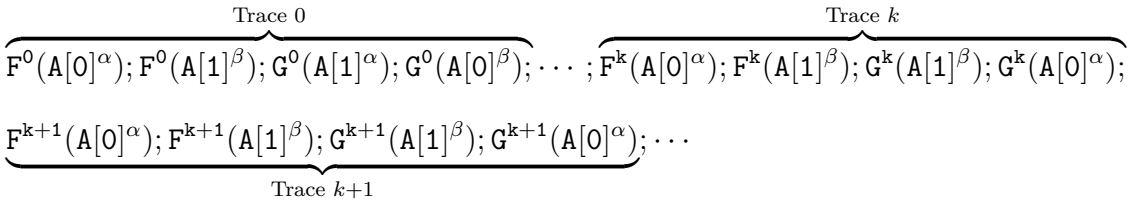
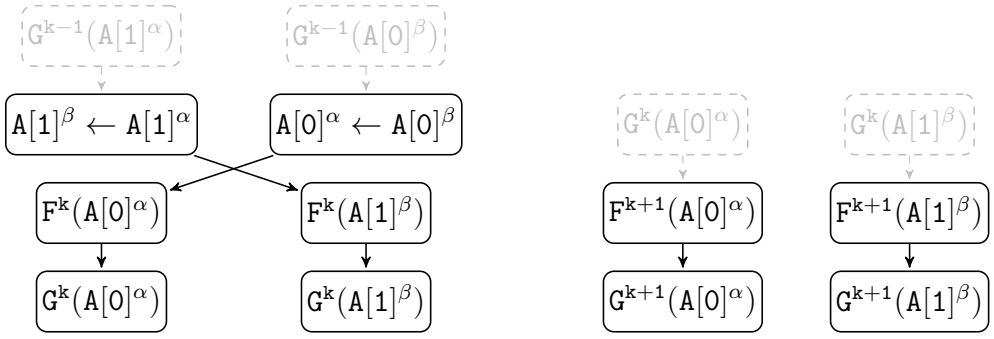


Figure 1.11: Traces after mapping change

$\mathtt{G}(\mathtt{A}[\mathtt{h}(0)])$ and $\mathtt{G}(\mathtt{A}[\mathtt{h}(1)])$ are swapped as shown in Figure 1.11. Trace k then looks different from the first $k-1$ traces because the mapping of the instances has changed, leading dynamic tracing to reject replaying the capture of trace 1 and instead capture a new trace. Figure 1.12a shows the task graph for trace k. However, dynamic tracing cannot replay the graph from trace k for trace $k+1$ either because trace $k+1$ does not pass the precondition check; the capture from trace k requires instances $\mathtt{A}[1]^\alpha$ and $\mathtt{A}[0]^\beta$ to be valid while trace $k+1$ sees $\mathtt{A}[1]^\beta$ and $\mathtt{A}[0]^\alpha$ as valid instances. (Figure 1.12b shows the task graph for trace $k+1$.) Dynamic tracing will also need to capture the graph for trace $k+1$ and will be able to replay it starting with trace $k+2$, assuming the precondition remains satisfied.



(a) Trace k (b) Trace $k+1$

Figure 1.12: Task graphs for traces k and $k+1$

1.3 Contributions

This work makes two key contributions:

- Chapter 3 presents a constraint-based approach to automatic data partitioning. To the best of our knowledge, our constraint-based auto-parallelizer provides precise control over the auto-parallelization process in the way that no existing work does, which is key to both composability and tunability for auto-parallelized programs. We evaluate the implementation of our constraint-based approach using the Regent compiler [64] (Section 3.5). For a set of Regent programs that are already hand-optimized for distributed memory execution, their sequential counterparts auto-parallelized in our approach achieved comparable performance (within 5%). We also demonstrate with a case study on Soleil-X [2,68], a multi-physics solver for a particle-laden turbulent flow problem, that the composability of auto-parallelized programs enabled by our approach provides a path to embracing auto-parallelizers in practice (Section 3.6).

- Chapter 4 describes dynamic tracing. To the best of our knowledge, dynamic tracing is the first technique to just-in-time specialize task graphs in distributed task-based runtimes with dynamic dependence analysis. We present a complete design of dynamic tracing with several key optimizations (Sections 4.1-4.3.2). For five already optimized Legion applications, we demonstrate that our implementation of dynamic tracing for the Legion runtime [17] improves strong scaling performance up to 7.0×, and by 4.9× on average, when running on up to 256 nodes (Section 4.4). We also demonstrate that dynamic tracing improves strong scaling performance of S3D-Legion [70], an exascale software for turbulent combustion simulation using the Legion runtime (Section 4.4.3).

Chapter 2 provides a background on the implicitly parallel tasking model used in this dissertation. We discuss the related work in Chapter 5 and conclude in Chapter 6.

1.4 Publications

Materials in Chapters 3 and 4 are modified and revised from their original publications [51, 52], in collaboration with Elliott Slaughter, Michael Bauer, Sean Treichler, Manolis Papadakis, Todd Warszawski, Michael Garland and Alex Aiken.

Chapter 2

Programming Model

In this chapter, we describe the implicitly parallel tasking model used in this dissertation. Our tasking model is based on Legion [17], a data-centric programming model for implicit task parallelism, and therefore borrows most of the concepts from it. Example programs in this and the following chapters are written in a variant of Regent [64], a high-level programming language for the Legion programming model. Many of the abstractions in our programming model can be also found in other implicitly parallel tasking models [10, 30, 43].

2.1 Tasks and Regions

In our programming model, a program is decomposed into *tasks*. A task is a unit of computation, and, as is standard, tasks are distinguished functions in a program (using the keyword `task`). Figrue 2.1 shows an example program with three tasks `T1`, `T2` and `T3`.

Tasks store data in *regions*, which are collections of values. All elements of a region have the same type, and every element has a unique *index*. The set of indices for which a region has elements is the *index space* of that region. We write $\mathsf{ispace}(R)$ to denote the index space of the region R. For example, the program in Figrue 2.1 uses regions `A`, `B`, `C`, `L` and `R`, with the usual constructs to access the elements (e.g., the expression `A[i]` at line 5) and to loop over the indices (e.g., the statement **`for i in A:`** at line

```
fun r(x): x + 1

task T1(A):
  for i in A:
    A[i] = f(i)

task T2(B, A, L):
  for i in B:
    left = L[i]; right = r(i)
    B[i] = g(A[left], A[right])

task T3(C, B, R):
  for i in C:
    range = R[i]; sum = 0
    for j in range:
      sum += B[j]
    C[i] = sum
```

Figure 2.1: Example program

4). Only the assignments to regions in a task have side effects visible to other tasks; all the other task arguments are passed as values.

2.2 Partitions

Regions can be *partitioned* in our programming model. A partition of a region is an indexed collection of *subregions*, sub-collections of that region. Partitions are first-class objects and they can be passed freely between tasks. One region can have multiple partitions and they all serve as different views of the same data stored in the region; any updates made with one partition must be visible to all the other partitions. The index of a subregion is called the *color* and the set of all colors in a partition is the *color space* of that partition.

Programs construct partitions at runtime. Partitioning code is described in the Dependent Partitioning Language (DPL) [71], a domain-specific language for data

partitioning. Figure 2.2 shows the syntax for a subset of DPL used in this dissertation. The **equal** operator creates partitions without using any other partitions; the expression $\texttt{equal}(R, N)$ creates a partition of R with N subregions having approximately equal size. The union, intersection, and difference operators on partitions are applied subregion-wise; for example, a union of two partitions results in a partition whose ith subregion is a union of the ith subregions of the operands:

$$(E_1 \diamond E_2)[i] \triangleq E_1[i] \diamond E_2[i] \qquad \text{where } \diamond \in \{\cup, \cap, -\}$$

The **image** operator creates a partition of a function's range from an existing partition of the function's domain; the expression $\texttt{image}(E, f, R)$ is a partition of R derived from E using f as follows:

$$\texttt{image}(E, f, R)[i] \triangleq \{\big(f(k), v'\big) \in R \mid (k, v) \in E[i]\}.$$

(Note that the function f takes the indices of each subregion as arguments, not its values.) We treat the regions used in indirect accesses (e.g., the region `L` in Figure 2.1) as functions; such a region `R` is written `R`$[\cdot]$ in the **image** expressions. We use a symbol f_{ID} for the identity function, i.e., $f_{\mathsf{ID}}(x) = x$. The **preimage** operator is an inverse of **image**, i.e., deriving a partition of a function's domain from an existing partition of the function's range; the expression $\texttt{preimage}(R, f, E)$ is a partition of f's domain R derived from E as follows:

$$\texttt{preimage}(R, f, E)[i] \triangleq \{(k, v') \in R \mid \big(f(k), v\big) \in E[i]\}.$$

Figure 1.6 visualizes the **image** and **preimage** operators for an example function `f`. Programs often use functions that map a single index to a set of indices to express a one-to-many relationship. For example, the region `R` in Figure 2.1 maps each index in `C` to a set of indices in `B`. We define *generalized* image and preimage operators

$$\begin{array}{llll} \text{Regions} & R & \text{Partitions} \quad P & \text{Functions} \quad f, F \\ \text{Expressions} & E & ::= \quad P \mid \texttt{equal}(R, N) \mid E \cup E \mid E \cap E \mid E - E \\ & & \mid \quad \texttt{image}(E, f, R) \mid \texttt{preimage}(R, f, E) \\ & & \mid \quad \texttt{IMAGE}(E, F, R) \mid \texttt{PREIMAGE}(R, F, E) \end{array}$$

Figure 2.2: DPL syntax

```
1  pA1  = equal(A, N1)
2
3  pB2  = equal(B, N2)
4  pA2L = image(pB2, L[·], A)
5  pA2R = image(pB2, r, A)
6  pA2  = pA2L ∪ pA2R
7  pL2  = image(pB2, f_ID, L)
8
9  pC3  = equal(C, N3)
10 pB3  = IMAGE(pC3, R[·], B)
11 pR3  = image(pC3, f_ID, R)
```

Figure 2.3: Example DPL program

IMAGE and **PREIMAGE** that derive partitions using such functions (ranged over by the variable F):

$$\textbf{IMAGE}(E, F, R)[i] \triangleq \{(l, v') \in R \mid (k, v) \in E[i] \wedge l \in F(k)\}$$
$$\textbf{PREIMAGE}(R, F, E)[i] \triangleq \{(l, v') \in R \mid (k, v) \in E[i] \wedge k \in F(l)\}$$

The **image** and **preimage** operators are indeed a special case of these operators; for example, with a lifting $f_\uparrow$ of a function f such that $f_\uparrow(x) = \{f(x)\}$, we have

$$\textbf{image}(E, f, R) = \textbf{IMAGE}(E, f_\uparrow, R).$$

Figure 2.3 shows a DPL program that creates partitions for the regions `A`, `B`, and `C` in Figure 2.1, whose example run is depicted in Figure 2.4. Rectangles in Figure 2.4 correspond to elements in regions. The rectangles on the right side are filled with colors of the subregions to which their element belongs; rectangles with multiple colors represent elements included in more than one subregion.

An example task that uses the partitions in Figure 2.3 to launch the tasks in Figure 2.1 is shown in Figure 2.5; the `main` task launches tasks for all colors in the color spaces using **`parallel for`** loops and accesses subregions of the partitions with index expressions.

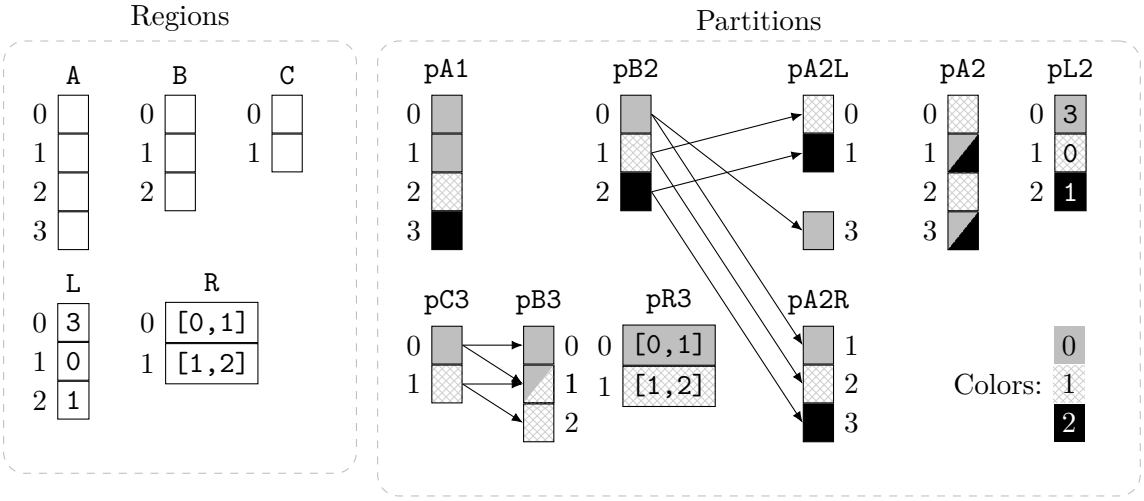


Figure 2.4: Example partitions from the program in Figure 2.3. Values of the regions `A`, `B`, and `C` are elided as they are not used in partitioning.

```
1 task main():
2   -- Region creations
3   -- The DPL code in Figure 2.3
4   parallel for c in pA1:
5     T1(pA1[c])
6   parallel for c in pB2:
7     T2(pB2[c], pA2[c], pL2[c])
8   parallel for c in pC3:
9     T3(pC3[c], pB3[c], pR3[c])
```

Figure 2.5: Example of parallel task launches

2.3 Execution Semantics

In our programming model, a program has *sequential semantics*; i.e., a program yields the result as if all tasks in that program executed sequentially in program order. For example, for the partitions in Figure 2.4 any (potentially parallel) execution of the program in Figure 2.5 must be equivalent to running the tasks in the following order:

`T1(pA1[0])`; `T1(pA1[1])`; `T1(pA1[2])`; `T2(pB2[0], pA2[0], pL2[0])`; `T2(pB2[1], pA2[1], pL2[1])`;
`T2(pB2[2], pA2[2], pL2[2])`; `T3(pC3[0], pB3[0], pR3[0])`; `T3(pC3[1], pB3[1], pR3[1])`;

```
1 task T(R, S):
2   R[1] = S[0]
3   R[2] = S[1]
4
5 task main():
6   A = region(int, 3)
7   T(A, A)
```

Figure 2.6: Example of ill-behaved task instances

Any system that implements our programming model must guarantee sequential semantics for programs.

Before a task can run, all of the task's region arguments must be *mapped* to *region instances*. A region instance is a physical allocation of a region. One region can be mapped to multiple instances, most commonly when tasks sharing the same region run on different nodes, and one instance can serve multiple subregions of a region. An invocation of a task whose regions are mapped is a *task instance*. The mapping does not change during execution of a task instance, but can be different in different task instances of the same task. Every program has a *mapper*, a program component that makes mapping decisions for tasks according to some (possibly dynamic) policy.

Any access to a region in a task must be *coherent* regardless of the mapping decision. If a task instance updates a region instance of a region R, any subsequent task instances reading region instances of a region that intersects R must see the update. For example, all task instances of `T2` at line 7 in Figure 2.5 must see all the updates to the region `A` made by the task instances of task `T1` at line 5, regardless of the mapping for the partitions `pA1` and `pA2`, which partition the same region `A`. In our programming model maintaining coherence is the responsibility of the runtime system. The program specifies what data is to be used, and the runtime system manages coherence by generating copies and inserting synchronization to ensure the data is current when and where it is needed.

To avoid fine-grained management of coherence during execution of task instances, we require every task instance to be *well-behaved*. A task instance is well-behaved unless it uses multiple overlapping region instances and updates at least one of them.

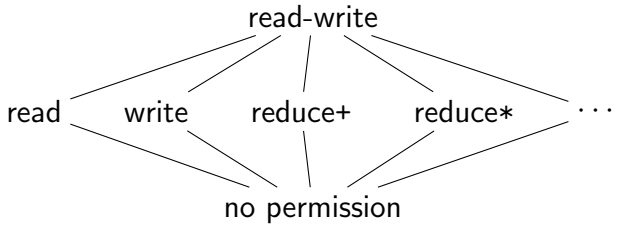


Figure 2.7: Permission lattice

The program in Figure 2.6 shows an example of ill-behaved task instance; if the task instance at line 7 uses two different instances $\mathtt{A}^{\alpha}$ and $\mathtt{A}^{\beta}$ for its region arguments (i.e., $\{\mathtt{R} \mapsto \mathtt{A}^{\alpha}, \mathtt{S} \mapsto \mathtt{A}^{\beta}\}$), the runtime system must transfer the update to $\mathtt{A}^{\alpha}$ at line 2 to $\mathtt{A}^{\beta}$ before the assignment at line 3 executes. This assumption simplifies the management of coherence by avoiding the need to deal with multiple aliased regions within a single task.

2.4 Dependence Analysis

A parallel schedule of task instances has sequential semantics if it respects all *dependencies* between those task instances. Task instances have a (data) dependence if they access regions that overlap with each other and at least one of them updates the region. A *dependence analysis* of a program is a process to identify all dependencies between task instances.

The runtime system that implements our programming model performs this dependence analysis dynamically with a pipeline of phases. First, a program launches a sequence of task instances and submits them to the runtime system. The runtime system then analyzes dependencies between task instances and constructs a *task graph*. A task graph is a DAG of task instances whose edges encode dependencies. Lastly, task instances are scheduled on processors according to their topological order in the constructed task graph.

To facilitate dependence analysis, tasks are annotated with *permissions* on regions, which describe how tasks access data. With permissions the system can analyze dependencies between task instances without introspecting task definitions, assuming

that the tasks respect their permission annotations. A task can have *read*, *write*, *read-write*, or *reduction* permissions on a region. Reduction permissions are used when tasks update regions only using commutative and associative operators. When a task has multiple permissions on a region, those permissions are *joined*; a set of all possible permissions on a region forms a lattice, shown in Figure 2.7, where the top element is the read-write permission and reduction permissions with different operators are different elements. For example, if a task has reduction permissions on a region R with the `+` and `*` operators, the joined permission on R is read-write permission. Task instances of a task inherit the task's permissions on regions to their region instances. Figure 2.8 shows permission annotations for the tasks in Figure 2.1.

Commutativity and associativity of reduction operators can be exploited by the runtime system to extract additional parallelism. A reduction is semantically equivalent to a read access followed by a write access; i.e., a reduction statement `x += y` has the same meaning as a statement `x = x + y`. However, the *reduction contribution* `y` can be computed without reading the value of `x`. Hence, if there are multiple reductions to the same element in a region, their contributions can be independently calculated in parallel and later applied atomically to that element. This extra parallelism is useful when individual reduction contributions are expensive to compute and thus desirable to parallelize. Figure 2.9 shows how this parallelism can be exploited by keeping buffers of reduction contributions. Note that the commutativity and associativity of the `+` operator allows the reduction contributions from the tasks `T` and `S` to be applied out of order and still produce the same result as the program order execution of those tasks. Many distributed memory runtime systems handle reductions in this way [17, 30, 60]. We assume that if a task has reduction permission on R, all of its task instances use a fresh instance I for R as a reduction buffer, initialized with the identity value of the reduction operator.

```
1 task T1(A) writes(A)
2 task T2(B, A, L) writes(B), reads(A), reads(L)
3 task T3(C, B, R) writes(C), reads(B), reads(R)
```

Figure 2.8: Permission annotations for the tasks in Figure 2.1

```
1 task T(R) reduces+(R): for e in R: R[e] += f(e)
2 task S(R) reduces+(R): for e in R: R[e] += g(e)
3 task main():
4   T(R); S(R)
```

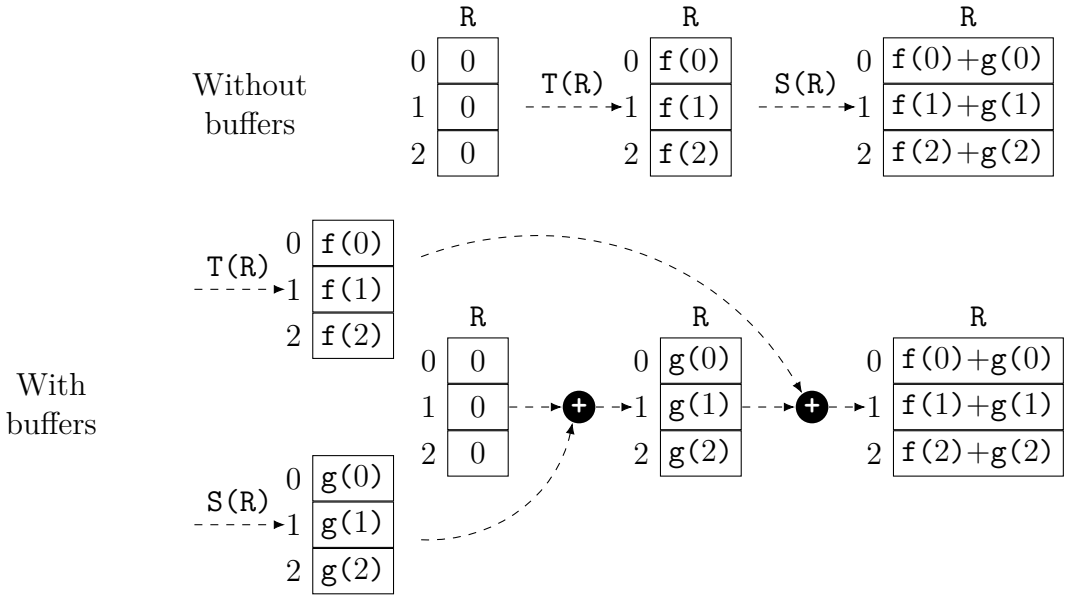


Figure 2.9: Handling reduction using reduction buffers

Another responsibility of the dependence analysis is to maintain coherence of region instances. The runtime system must keep track of *validity* of region instances (i.e., whether the region instance contains any of the last updates to the region) and introduce copies between region instances whenever necessary. Because of a potential many-to-many relationship between subregions and region instances, the validity must be tracked at the granularity of individual indices in the index space. We use a simple but inefficient algorithm that updates the validity of instances by applying the following sequence of rules to each task instance:

R When the task instance reads a region R using an instance I,

$\mathbf{R_1}$ If I is already valid for the entire $\mathsf{ispace}(R)$, no data movement is introduced.

$\mathbf{R_2}$ If I is not yet valid for some subset X of $\mathsf{ispace}(R)$, then I is made valid by issuing copies from existing instances that are valid for some indices in X to I. Each of the issued copies is treated as a task instance that reads from the source instance and writes to I, except it does not invalidate valid instances. The instance I is marked valid for the entire $\mathsf{ispace}(R)$.

$\mathbf{Red_1}$ If the task instance reads a region R using an instance I and there are outstanding reduction instances for some subset X of $\mathsf{ispace}(R)$, those reduction instances are applied to I. Each reduction application is treated as a task instance that reads from the reduction instance and writes to I; i.e., it makes I the sole valid instance for all indices in X and invalidates other instances, including outstanding reduction instances, for any indices in X.

W If the task writes to a region R using an instance I, then I becomes the sole valid instance for the entire $\mathsf{ispace}(R)$. All the other instances, including outstanding reduction instances, are invalidated for any indices in $\mathsf{ispace}(R)$.

$\mathbf{Red_2}$ If the task reduces to a region R using an instance I, then I is simply recorded as an outstanding reduction instance for all indices in $\mathsf{ispace}(R)$.

Figure 2.10 demonstrates the dynamic dependence analysis using a simple example. In addition to task instances, the task graphs in Figure 2.10 have nodes for copies and reduction applications:

- A copy $S^\beta \leftarrow R^\alpha$ copies elements from the source instance R^α to the target instance S^β for indices in $\mathsf{ispace}(R) \cap \mathsf{ispace}(S)$. For example, a copy $\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha$ is introduced by the task instance $\mathtt{T_b}(\mathtt{A}^\beta, \mathtt{B}^\beta)$ because the region instance $\mathtt{B}^\beta$ is not yet valid for the region `B`.

- A reduction application $S^\beta \leftarrow S^\beta \circ R^\alpha$ applies reduction contributions in the reduction instance R^α to the target instance S^β using the $\circ$ operator for indices in $\mathsf{ispace}(R) \cap \mathsf{ispace}(S)$. For example, a reduction application $\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha \mathtt{+} \mathtt{A}^\beta$ is necessary for the task instance $\mathtt{T_c}(\mathtt{A}^\alpha, \mathtt{B}^\alpha)$ to make coherent read access to the region `A` with the region instance A^α.

Tasks: **task** $\mathtt{T_a}(\mathtt{R},\mathtt{S})$ **reads**(R),**writes**(R),**writes**(S)
task $\mathtt{T_b}(\mathtt{R},\mathtt{S})$ **reduces+**(R),**reads**(S)
task $\mathtt{T_c}(\mathtt{R},\mathtt{S})$ **reads**(R),**reads**(S),**writes**(S)

Task Instance	Valid Instances / Outstanding reductions	Task Graph
$\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$	$\mathtt{A} \mapsto \mathtt{A}^\alpha$ (rule $\mathbf{R_1}$,$\mathbf{W}$) $\mathtt{B} \mapsto \mathtt{B}^\alpha$ (rule $\mathbf{W}$)	$\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$
$\mathtt{T_b}(\mathtt{A}^\beta,\mathtt{B}^\beta)$	$\mathtt{A} \mapsto \mathtt{A}^\alpha$ $\mathtt{B} \mapsto \{\mathtt{B}^\beta,\mathtt{B}^\alpha\}$ (rule $\mathbf{R_2}$) $\mathtt{A} \mapsto \mathtt{A}^\beta$ (rule $\mathbf{Red_2}$)	$\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$ → $\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha$ → $\mathtt{T_b}(\mathtt{A}^\beta,\mathtt{B}^\beta)$
$\mathtt{T_c}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$	$\mathtt{A} \mapsto \mathtt{A}^\alpha$ (rule $\mathbf{R_1}$) $\mathtt{B} \mapsto \mathtt{B}^\alpha$ (rule $\mathbf{R_1}$,$\mathbf{Red_1}$)	$\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$ → $\mathtt{T_c}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$; $\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha$; $\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha\mathtt{+}\mathtt{A}^\beta$; $\mathtt{T_b}(\mathtt{A}^\beta,\mathtt{B}^\beta)$

Figure 2.10: Example dependence analysis. Changes after the analysis of each task instance are highlighted with shading. The regions A and B in this example have no overlap and different region instances of the same region are distinguished by their superscripts.

Note that the last task graph has edges for *transitive* dependences, i.e, dependences that are transitively expressed by other dependencies. For example, the edge between the task instance $\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$ and the reduction application $\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha\mathtt{+}\mathtt{A}^\beta$ represents a transitive dependence as they are already connected by another path that goes through the task instance $\mathtt{T_b}(\mathtt{A}^\beta,\mathtt{B}^\beta)$. Transitive dependences are not harmful for parallelism, because they impose no additional constraints, and dependence analysis algorithms often include them in the result as additional analysis would be required to remove them. We assume that task graphs may have edges for transitive dependencies.

Figure 2.11 shows a task graph of the program in Figure 2.5 for an example run

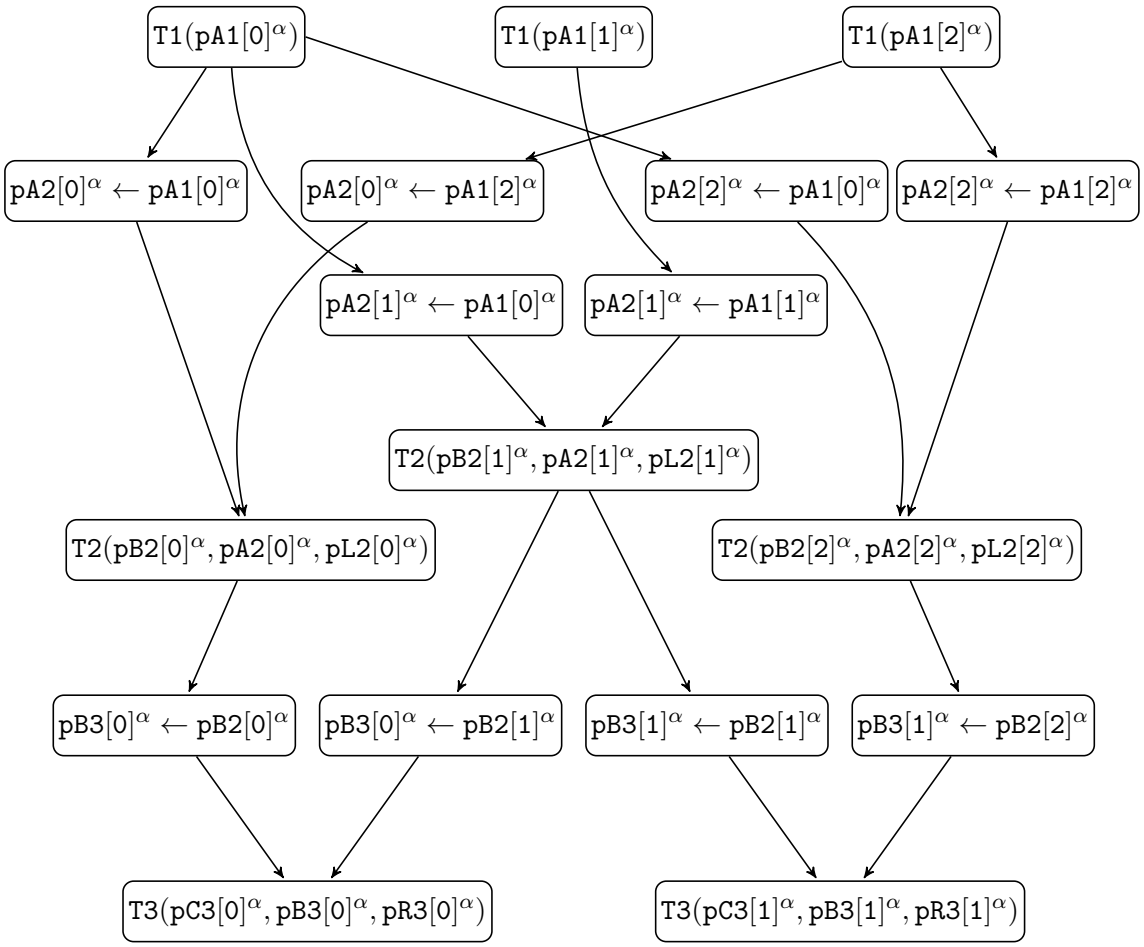


Figure 2.11: Task graph of the program in Figure 2.5

using the partitions in Figure 2.4. In the task graph, each subregion argument R to a task is mapped to a region instance R^α on some node α. Note that the task graph has a copy for each pair of subregions of different partitions of the same region whenever those subregions overlap with each other. For example, the subregion `pA1[0]` overlaps with all three subregions `pA2[0]`, `pA2[1]` and `pA2[2]` of the partition `pA2`, hence three copies originating from the task instance `T1`($\texttt{pA1[0]}^\alpha$). Note also that the tasks launched at each **`parallel for`** loop are all independent as those tasks write to disjoint subregions without reading others' updates.

Chapter 3

Automatic Data Partitioning

Data partitioning is an essential step in extracting data parallelism implicit in sequential programs. Once we have legal data partitions for a data parallel loop, the rest of the parallelization is simply to convert the loop into a form that launches parallel tasks for subregions of those partitions, each of which runs a subset of the loop iterations using its subregion arguments. The parallelism exposed by those tasks is later discovered and realized by the runtime system of our implicitly parallel tasking model. The goal of the auto-parallelizer then boils down to finding performant data partitions that are legal for the whole program to be parallelized.

In this chapter, we present a constraint-based approach to automatic data partitioning. Our approach finds performant legal data partitions for a program in two steps. First, we soundly capture a set of possible legal partitions for a program using *partitioning constraints*. Partitioning constraints are inferred automatically by a static analysis of region accesses in parallelizable loops (Section 3.1). Second, we run a constraint solver that synthesizes DPL programs as solutions to the inferred partitioning constraints (Section 3.2). The constraint solver finds a performant solution that satisfies a give system of partitioning constraints with fewest data partitions. We also perform optimizations on DPL programs to minimize data movement for reductions (Section 3.3). The implementation of our approach in the Regent compiler [64] (Section 3.4) demonstrates that programs auto-parallelized in our approach have scalability comparable to hand-optimized counterparts (Section 3.5). Furthermore, the

$$
\begin{array}{llll}
\text{Regions} & R & \text{Partitions} \quad P & \text{Functions} \quad f, F \\
\text{Constraints} & C & ::= & \phi \mid E \subseteq E \mid C \wedge C \\
\text{Predicates} & \phi & ::= & \mathbf{PART}(E, R) \mid \mathbf{DISJ}(E) \mid \mathbf{COMP}(E, R) \\
\text{Expressions} & E & ::= & P \mid \texttt{equal}(R) \mid E \cup E \mid E \cap E \mid E - E \\
 & & \mid & \texttt{image}(E, f, R) \mid \texttt{preimage}(R, f, E) \\
 & & \mid & \texttt{IMAGE}(E, F, R) \mid \texttt{PREIMAGE}(R, F, E)
\end{array}
$$

Figure 3.1: Constraint language syntax

application of our approach to Soleil-X [2,68] showcases that the composability of auto-parallelized programs enabled by our approach is key to embracing auto-parallelizers in practice (Section 3.6).

3.1 Constraint Inference

We first define a constraint language for describing partitioning constraints. Figure 3.1 shows the syntax of the partitioning constraint language. Ground terms are regions (using symbol R) and partitions (using symbol P). A partitioning constraint is a conjunction of subset constraints and predicates on partitions. A predicate $\mathbf{PART}(E, R)$ means that E is a partition of the region R; i.e., each subregion $E[i]$ must be a subset of the region R:

$$\mathbf{PART}(E, R) \triangleq \forall i. E[i] \subseteq R.$$

A predicate $\mathbf{DISJ}(E)$ requires E to be a disjoint partition and a $\mathbf{COMP}(E, R)$ requires E to be a complete partition of R:

$$\mathbf{DISJ}(E) \triangleq E[i] \cap E[j] = \varnothing \text{ when } i \neq j \qquad \mathbf{COMP}(E, R) \triangleq \bigcup_i E[i] = R$$

A subset constraint $E_1 \subseteq E_2$ denotes that each subregion $E_2[i]$ contains the corresponding subregion $E_1[i]$:

$$E_1 \subseteq E_2 \triangleq \forall i.\ E_1[i] \subseteq E_2[i]$$

This relation implicitly requires that the set of indices of E_2 subsume that of E_1. The subset constraint is anti-symmetric, i.e.,

Algorithm 1: Constraint inference algorithm

Procedure `Infer(`*loop*`):`
  // Assume *loop* has the form **`for i in R:`** *body*
  // Assume *body* is normalized
  $Env \leftarrow \{\texttt{i} \mapsto \lambda r.\textbf{image}(P_{\texttt{R}}, f_{\mathsf{ID}}, r)\}$ ($P_{\texttt{R}}$ is fresh)
  $Rgn \leftarrow \{\texttt{i} \mapsto \texttt{R}\}$
  $C \leftarrow \textbf{PART}(P_{\texttt{R}}, \texttt{R}) \wedge \textbf{COMP}(P_{\texttt{R}}, \texttt{R})$
  for *each statement* $s \in$ *body* **do**
    // Region accesses appear only in statements of these forms:
    if s **is** `y = S[x]` **or** `S[x] = y` **or** `S[x] += y :`
      $E \leftarrow Env(\texttt{x})(\texttt{S})$
      $C \leftarrow C \wedge \textbf{PART}(P, \texttt{S}) \wedge E \subseteq P$ (P is fresh)
      if s **is** `y = S[x] :`
        $Env \leftarrow Env \cup \{\texttt{y} \mapsto \lambda r.\textbf{image}(P, \texttt{S}[\cdot], r)\}$
        $Rgn \leftarrow Rgn \cup \{\texttt{y} \mapsto \texttt{S}\}$
      elseif s **is** `S[x] += y` **and** $Rgn(\texttt{x}) \neq \texttt{R}$ `:`
        $C \leftarrow C \wedge \textbf{DISJ}(P_{\texttt{R}})$
    elseif s **is** `y = f(x)` **or** `y = F(x) :`
      $R_{\texttt{x}} \leftarrow Rgn(\texttt{x})$
      $E \leftarrow Env(\texttt{x})(R_{\texttt{x}})$
      $C \leftarrow C \wedge \textbf{PART}(P, R_{\texttt{x}}) \wedge E \subseteq P$ (P is fresh)
      if s **is** `y = f(x) :`
        $Env \leftarrow Env \cup \{\texttt{y} \mapsto \lambda r.\textbf{image}(P, \texttt{f}, r)\}$
      elseif s **is** `y = F(x) :`
        $Env \leftarrow Env \cup \{\texttt{y} \mapsto \lambda r.\textbf{IMAGE}(P, \texttt{F}, r)\}$
      $Rgn \leftarrow Rgn \cup \{\texttt{y} \mapsto R_{\texttt{x}}\}$
    elseif s **is** `y = x :`
      $Env \leftarrow Env \cup \{\texttt{y} \mapsto Env(\texttt{x})\}$
      $Rgn \leftarrow Rgn \cup \{\texttt{y} \mapsto Rgn(\texttt{x})\}$

$$E_1 = E_2 \triangleq E_1 \subseteq E_2 \wedge E_2 \subseteq E_1.$$

Expressions are DPL operators from Figure 2.2. Integer arguments denoting the number of subregions are elided in partitioning constraints because they do not affect constraint solving. Note that our constraint language can express DPL programs; a DPL statement $P = E$ is expressible with antisymmetry and a DPL program is just a sequence of DPL statements.

Algorithm 1 shows the constraint inference algorithm, which takes a loop and

produces a system of partitioning constraints. The algorithm is concerned only with *parallelizable* loops. A loop is parallelizable when values defined in one loop iteration are never consumed by other iterations of the same loop. For brevity, we characterize parallelizable loops syntactically as follows.

- Region accesses are either *centered* or *uncentered*. A region access `R[e]` is centered when `e` is the loop variable (or an alias), and is uncentered otherwise.
- An uncentered access is *admissible* only when it has an index expression derived from another region access (e.g., `R[S[e]]`) or it has the form `R[f(i)]` where `i` is the loop variable.
- A parallelizable loop is an outermost loop of the form

  ```
  for i in R: ...
  ```

  for some region `R` (the *iteration space* of the loop), which satisfies these conditions:

  - All write accesses to regions are centered. (A centered reduction is considered a centered read access followed by a centered write access.)
  - A region with an uncentered reduction (e.g., `R[S[e]] += ...`) does not have any other read access or a reduction with a different operator.
  - A region with an uncentered read (e.g., `... = R[S[e]]`) does not have any other write access.

This syntactic definition is sound but incomplete; i.e., there are loops that a more sophisticated analysis, such as polyhedral analysis [21], can prove parallelizable but our syntactic check cannot.

At the highest level, our method for constraint-based partitioning has three components:

- Initially a separate partition (represented by a unique partition variable) is assigned to every region access in a parallelizable loop. Another unique partition

Program	Constraints
`for i in R:`	$\mathbf{PART}(P_1, \mathtt{R}) \wedge \mathbf{COMP}(P_1, \mathtt{R}) \wedge$
`  j = R[i]`	$\mathbf{PART}(P_2, \mathtt{R}) \wedge P_1 \subseteq P_2 \wedge$
`  v = f(S[j])`	$\mathbf{PART}(P_3, \mathtt{S}) \wedge \mathbf{image}(P_2, \mathtt{R}[\cdot], \mathtt{S}) \subseteq P_3 \wedge$
`  R[i] = v`	$\mathbf{PART}(P_4, \mathtt{R}) \wedge P_1 \subseteq P_4$

Figure 3.2: Example of constraint inference

variable is associated with the loop index. For each of these variables, we generate constraints that guarantee the partition will have all the elements needed to execute correctly.

- We solve the constraints by rewriting them into an equivalent form where each remaining constraint corresponds to a concrete dependent partitioning operation; the partitioning code can be read directly from the resolved form of the constraints.

- Allowing a separate partition for every region access admits the widest possible range of partitioning strategies, but can result in solutions with multiple equivalent partitions. We unify partition variables with isomorphic constraints to reduce the final number of partitions that need to be created.

The following example illustrates the constraint inference steps for the example loop in Figure 3.2.

Example 1. *Algorithm 1 first conjoins the following predicates on a partition symbol* P_1 *for the iteration space* `R` *(line 6):*

$$\mathbf{PART}(P_1, \mathtt{R}) \wedge \mathbf{COMP}(P_1, \mathtt{R})$$

The algorithm also maintains two environments: Env that maps each variable to a lambda function that returns an **`image`** *expression of a region argument and Rgn that maps each variable to the region from which the variable's value was drawn. The initial environments at lines 4-5 have mappings of the loop variable* `i` *to a function* $\lambda r.\mathbf{image}(P_1, f_{\mathsf{ID}}, r)$ *and to the iteration space* `R`*, respectively. For the read access* `R[i]`*,*

Program	Constraints
`for i in R:`	$\mathbf{PART}(P_1, \mathtt{R}) \wedge \mathbf{COMP}(P_1, \mathtt{R}) \wedge$
`j = R[i]`	$\mathbf{PART}(P_2, \mathtt{R}) \wedge P_1 \subseteq P_2 \wedge$
`S[j] += c`	$\mathbf{PART}(P_3, \mathtt{S}) \wedge \mathtt{image}(P_2, \mathtt{g}, \mathtt{S}) \subseteq P_3 \wedge \mathbf{DISJ}(P_1)$

Figure 3.3: Example with a disjointness predicate on the iteration space

the algorithm introduces a partition symbol P_2 and generates the following constraints (lines 9-11):

$$\mathbf{PART}(P_2, \mathtt{R}) \wedge P_1 \subseteq P_2$$

Note that the expression $\mathtt{image}(P_1, f_{\mathsf{ID}}, \mathtt{R})$ *is simplified to* P_1*. Since the value of this region access is assigned to the variable* `j`*, the algorithm updates the environments (lines 13-14), which then become the following:*

$$Env = \{\mathtt{i} \mapsto \lambda r.\mathtt{image}(P_1, f_{\mathsf{ID}}, r), \mathtt{j} \mapsto \lambda s.\mathtt{image}(P_2, \mathtt{R}[\cdot], s)\} \quad Rgn = \{\mathtt{i} \mapsto \mathtt{R}, \mathtt{j} \mapsto \mathtt{R}\}$$

For the uncentered read access `S[j]`*, the algorithm infers the following constraints on a new partition symbol* P_3 *(lines 9-11), where the subset constraint has an* `image` *expression in the lower bound:*

$$\mathbf{PART}(P_3, \mathtt{S}) \wedge \mathtt{image}(P_2, \mathtt{R}[\cdot], \mathtt{S}) \subseteq P_3$$

Finally, the write access `R[i] = ...` *is handled similarly to other centered accesses, resulting in the partitioning constraint in Figure 3.2.*

Note that the partitioning constraint in Figure 3.2 does not have a disjointness predicate on the partition of the iteration space. If the final solution uses a non-disjoint partition of the iteration space, there is redundant computation because some loop iterations are executed multiple times. This redundancy is useful in cases (as demonstrated by Zhou et al. [76]) when recomputing loop iterations on separate nodes is cheaper then the internode communication the redundant computation replaces. In Section 3.3, we discuss how we can optimize communication from uncentered reductions using an *aliased* (non-disjoint) partition of the iteration space.

However, we do need a disjoint partition of the iteration space when the loop has an uncentered reduction access (lines 15-16 in Algorithm 1). Figure 3.3 shows an example where an uncentered reduction on the region `S` imposes a disjointness constraint on the partition P_1 of the iteration space `R`. To see why disjointness is mandatory in this case, we need to revisit how uncentered reductions are handled in our programming model. Unlike centered reductions, which can be applied immediately, uncentered reductions require two steps. First, each distributed task allocates a temporary instance to keep the reduction contribution from each iteration that it owns. Then, temporary instances are merged, either eagerly or lazily, back to the partitions that the subsequent read accesses use. Because this merge step aggregates all contributions in temporary instances, each contribution must be counted exactly once to preserve the original semantics. Therefore, the iteration space must be partitioned disjointly in this case.

Algorithm 1 runs in linear time in the size of the program and produces partitioning constraints sound with respect to the semantics of parallelizable loops. These partitioning constraints always have at least one trivial solution, obtained by replacing each subset constraint with an equality.

3.2 Constraint Solver

In this section, we describe a constraint solver that takes partitioning constraints as input and produces DPL programs as solutions. Our constraint solver transforms the input partitioning constraint into a *resolved* form, the constraint conjoined with exactly one equality $P_i = E_i$ for each partition symbol P_i. Once the partitioning constraint is solved, the added equalities form one solution program. In the rest of this section, we explain the algorithm to resolve partitioning constraints and the heuristics to minimize the number of partitions constructed by the output program.

3.2.1 Resolution

Conceptually, a partitioning constraint C can be resolved by the following procedure:

L1 $\mathbf{PART}(\texttt{equal}(R), R) \wedge \mathbf{DISJ}(\texttt{equal}(R)) \wedge \mathbf{COMP}(\texttt{equal}(R), R)$
L2 $\mathbf{PART}(P_1, R) \wedge \mathbf{PART}(P_2, R) \Longrightarrow \mathbf{PART}(P_1 \diamond P_2, R) \quad (\diamond \in \{\cup, \cap, -\})$
L3 $\mathbf{PART}(\texttt{image}(E, f, R), R)$ L4 $\mathbf{PART}(\texttt{preimage}(R, f, E), R)$
L5 $\mathbf{PART}(\texttt{IMAGE}(E, F, R), R)$ L6 $\mathbf{PART}(\texttt{PREIMAGE}(R, F, E), R)$
L7 $E_1 \subseteq E_2 \wedge \mathbf{COMP}(E_1, R) \wedge \mathbf{PART}(E_2, R) \Longrightarrow \mathbf{COMP}(E_2, R)$
L8 $\mathbf{COMP}(E_1, R) \vee \mathbf{COMP}(E_2, R) \Longrightarrow \mathbf{COMP}(E_1 \cup E_2, R)$
L9 $\mathbf{COMP}(E_1, R_1) \Longrightarrow \mathbf{COMP}(\texttt{preimage}(R_2, f, E_1), R_2)$
L10 $\mathbf{DISJ}(E_2) \wedge E_1 \subseteq E_2 \Longrightarrow \mathbf{DISJ}(E_1)$
L11 $\mathbf{DISJ}(E_1) \vee \mathbf{DISJ}(E_2) \Longrightarrow \mathbf{DISJ}(E_1 \cap E_2)$
L12 $\mathbf{DISJ}(E_1) \Longrightarrow \mathbf{DISJ}(E_1 - E_2)$
L13 $\mathbf{DISJ}(E_1 \cup E_2) \implies \mathbf{DISJ}(E_1) \wedge \mathbf{DISJ}(E_2)$
L14 $\mathbf{DISJ}(E_1) \Longrightarrow \mathbf{DISJ}(\texttt{preimage}(R, f, E_1))$
L15 $E_1 \subseteq E_3 \wedge E_2 \subseteq E_3 \Longrightarrow E_1 \cup E_2 \subseteq E_3$
L16 $E_1 \subseteq \texttt{preimage}(R_1, f, E_2) \wedge \mathbf{PART}(E_2, R_2) \Longrightarrow \texttt{image}(E_1, f, R_2) \subseteq E_2$

Figure 3.4: DPL lemmas for resolution

1. Synthesize expressions $E_1, \ldots, E_n$ for all partition symbols $P_1, \ldots, P_n$ in C.

2. Check consistency of the strengthened constraint

$$C \wedge P_1 = E_1 \wedge \ldots \wedge P_n = E_n.$$

3. If the consistency check fails, go to (1) and synthesize different expressions.

The consistency check in step (2) verifies that each predicate in the constraint is entailed by other predicates or known lemmas of DPL operators, shown in Figure 3.4. Any set of expressions that pass this check is a solution that satisfies the input constraint.

All lemmas in Figure 3.4 are direct consequences from definitions of the DPL operators and properties of sets. The first six lemmas enumerate all possible cases where partitions of a region R can be constructed. Lemmas L7-9 (resp. lemmas L10-14) state when the completeness (resp. disjointness) of a partition is propagated to others.

Algorithm 2 shows the constraint solving algorithm tailored to partitioning constraints inferred by Algorithm 1. This algorithm tries to minimize backtracking due to adding an equation that causes the constraint system to become inconsistent (have no

Algorithm 2: Constraint solving algorithm

Procedure `Solve`(C, S)**:**
  // C is a partitioning constraint to solve.
  // S is a partial solution found so far.

  // Each call to `Solve()` picks one remaining constraint, adds
  // an equality to attempt to solve it, and calls `Solve()`
  // recursively to solve the rest of the system.
  for *each* $P = E \in S$ **do**
    // Replace P by E, eliminating P from the constraint C
    $C \leftarrow C[P \mapsto E]$
  Remove all tautologies $E \subseteq E$ from C
  for *each* $\texttt{image}(P, f, R) \subseteq E \in C$ *for a closed* E **do**
    // Assume $\exists R'.\mathbf{PART}(P, R') \in C$
    $S_{next} \leftarrow \texttt{Solve}(C, S \wedge P = \texttt{preimage}(R', f, E))$
    // If the system is not inconsistent ($\varnothing$), return the solution **if** $S_{next} \neq \varnothing$ **: return** S_{next}
  for *each* P *with subset constraints* $E_i \subseteq P$ *for closed* $E_i s$ **do**
    $S_{next} \leftarrow \texttt{Solve}(C, S \wedge P = \bigcup_i E_i))$
    if $S_{next} \neq \varnothing$ **: return** S_{next}
  // $\mathsf{depth}(P) \triangleq k$ when $E_1 \subseteq \cdots \subseteq E_k \subseteq P$
  for $d = max(\{\mathsf{depth}(P_i) \mid P_i \in C\}), 1$ **do**
    for *each* $\mathbf{PART}(P, R) \wedge \mathbf{DISJ}(P) \in C$ *s.t.* $\mathsf{depth}(P) = k$ **do**
      $S_{next} \leftarrow \texttt{Solve}(C, S \wedge P = \texttt{equal}(R))$
      if $S_{next} \neq \varnothing$ **: return** S_{next}
  for *each* P *with a subset constraint* $P \subseteq E$ *for closed* E **do**
    $S_{next} \leftarrow \texttt{Solve}(C, S \wedge P = E)$
    if $S_{next} \neq \varnothing$ **: return** S_{next}
  for *each* $\mathbf{COMP}(P, R) \in C$ **do**
    $S_{next} \leftarrow \texttt{Solve}(C, S \wedge P = \texttt{equal}(R))$
    if $S_{next} \neq \varnothing$ **: return** S_{next}
  // Lemmas in Figure 3.4 are used for this resolution
  if $\forall C_{sub} \in C.C - C_{sub} \Longrightarrow C_{sub}$ **: return** S
  else: return $\varnothing$

solutions). The solver picks promising candidates using the following insights based on the lemmas in Figure 3.4:

1. If a partition symbol P has subset constraints $E_1 \subseteq P, \ldots, E_k \subseteq P$ where each E_i is *closed*, i.e., contains no partition symbol, the union $E_1 \cup \cdots \cup E_k$ of these expressions is a good candidate for P (lemma L15).

2. The only way to create a fresh disjoint partition is the **`equal`** operator (lemma L1) and only intersection, difference, and **`preimage`** operators preserve the disjointness of operands (lemmas L11, L12, and L13). Therefore, a partition symbol with a **DISJ** predicate must be created using only these operators. Likewise, complete partitions can be expressed only by **`equal`** (lemma L1), union (lemma L8), and **`preimage`** (lemma L9), or combinations of these operators.

3. For a subset constraint $E_1 \subseteq E_2$, disjointness "flows" from right to left (lemma L10). When both sides of a subset predicate $E_1 \subseteq E_2$ must be disjoint, the solver resolves all symbols in the expression E_2 and then derives E_1.

4. The **`preimage`** operator can produce partitions that satisfy subset constraints containing **`image`** (lemma L16). Combined with observation (2), these lemmas imply that the solver must use a **`preimage`** partition to discharge a subset constraint of the form **`image`**$(E_1, ...) \subseteq E_2$ when both E_1 and E_2 must be disjoint.

Algorithm 2 can always solve partitioning constraints generated by Algorithm 1: Because Algorithm 1 introduces a fresh partition symbol for the RHS of each added subset constraint, the subset constraints never form a cycle. Thus, the solver can always find a trivial solution that uses **`equal`** partitions for iteration spaces and has equalities strengthened from all subset constraints. However, this naïve solution is inefficient because it does not reuse partitions from one parallelizable loop in the others. To maximize the partition reuse in the solution, the constraint solver performs *unification* of partition symbols, which is the topic of the next subsection.

The following examples demonstrate how Algorithm 2 resolves partitioning constraints.

Example 2. *Suppose we have this partitioning constraint from Figure 3.3:*

$$\begin{aligned}&\mathbf{PART}(P_1, \mathtt{R}) \wedge \mathbf{COMP}(P_1, \mathtt{R}) \wedge \mathbf{DISJ}(P_1) \wedge \mathbf{PART}(P_2, \mathtt{R}) \wedge P_1 \subseteq P_2 \\ \wedge\ &\mathbf{PART}(P_3, \mathtt{S}) \wedge \mathtt{image}(P_2, \mathtt{g}, \mathtt{S}) \subseteq P_3\end{aligned}$$

Because P_1 has a **DISJ** *predicate, the solver uses an* `equal` *partition for P_1 (line 22). After substituting P_1 with* `equal`$(\mathtt{R})$*, the original constraint simplifies to:*

$$\mathbf{PART}(P_2, \mathtt{R}) \wedge \mathtt{equal}(\mathtt{R}) \subseteq P_2 \wedge \mathbf{PART}(P_3, \mathtt{S}) \wedge \mathtt{image}(P_2, \mathtt{g}, \mathtt{S}) \subseteq P_3.$$

Since P_2 has a closed expression on the LHS of its subset constraint, the solver strengthens it to a equality (line 17), yielding the following constraint after simplification.

$$\mathbf{PART}(P_3, \mathtt{S}) \wedge \mathtt{image}(\mathtt{equal}(\mathtt{R}), \mathtt{g}, \mathtt{S}) \subseteq P_3.$$

Again, P_3 has a closed expression on the LHS of its subset constraints and the solver resolves it similarly. Finally, the solver produces the following solution (after performing common subexpression elimination):

$$P_1 = \mathtt{equal}(\mathtt{R}) \quad P_2 = P_1 \quad P_3 = \mathtt{image}(P_2, \mathtt{g}, \mathtt{S})$$

Example 3. *Suppose now we have an extra predicate* $\mathbf{DISJ}(P_3)$ *in the partitioning constraint as follows:*

$$\begin{aligned}&\mathbf{PART}(P_1, \mathtt{R}) \wedge \mathbf{COMP}(P_1, \mathtt{R}) \wedge \mathbf{DISJ}(P_1) \wedge \mathbf{PART}(P_2, \mathtt{R}) \wedge P_1 \subseteq P_2 \\ \wedge\ &\mathbf{PART}(P_3, \mathtt{S}) \wedge \mathtt{image}(P_2, \mathtt{g}, \mathtt{S}) \subseteq P_3 \wedge \mathbf{DISJ}(P_3)\end{aligned}$$

The solver notices that P_3, the RHS of the subset constraint $\mathtt{image}(P_2, \mathtt{g}, \mathtt{S}) \subseteq P_3$*, must be disjoint, and creates an* `equal` *partition for P_3 (line 22) and a* `preimage` *partition for P_2 (line 14):*

$$P_3 = \mathtt{equal}(\mathtt{S}) \quad P_2 = \mathtt{preimage}(\mathtt{R}, \mathtt{g}, \mathtt{equal}(\mathtt{S})).$$

In the remaining constraint

$$\mathbf{PART}(P_1, \mathtt{R}) \wedge \mathbf{COMP}(P_1, \mathtt{R}) \wedge \mathbf{DISJ}(P_1) \wedge P_1 \subseteq \mathtt{preimage}(\mathtt{R}, \mathtt{g}, \mathtt{equal}(\mathtt{S})),$$

P_1 has a closed expression on the RHS of its subset constraint, and the solver strengthens it to an equality (line 24). The final solution is as follows:

$$P_3 = \mathtt{equal}(\mathtt{S}) \quad P_2 = \mathtt{preimage}(\mathtt{R}, \mathtt{g}, P_3) \quad P_1 = P_2$$

3.2.2 Unification

A single unification step strengthens the original constraint by conjoining an equality between *unifiable* partition symbols:

$$\begin{aligned}&\mathbf{PART}(P_1, R) \wedge \mathbf{PART}(P_2, R) \wedge \ldots \\ &\quad \Longleftarrow \mathbf{PART}(P_1, R) \wedge \mathbf{PART}(P_2, R) \wedge P_1 = P_2 \wedge \ldots\end{aligned}$$

Partition symbols are unifiable only when they represent partitions of the same region. The constraint after unification can be further simplified by replacing one of the unified symbols with the other.

Example 4. *The partition symbols P_1, P_2, and P_4 in Figure 3.2 can be unified as follows:*

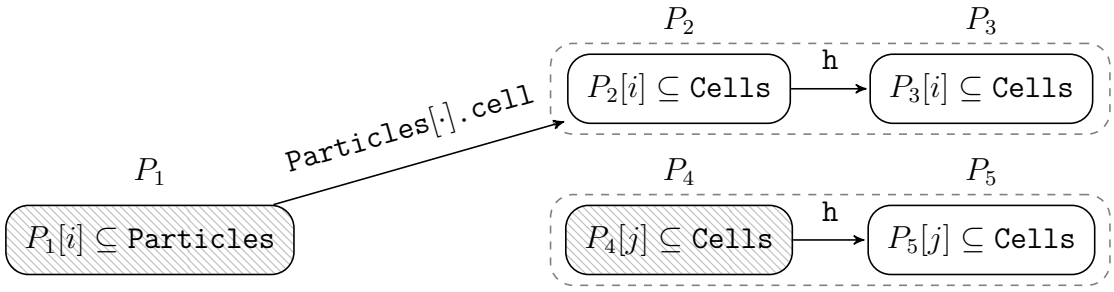
$$\begin{aligned}&\mathbf{PART}(P_1, \texttt{Particles}) \wedge \mathbf{COMP}(P_1, \texttt{Particles}) \\ \wedge\ &\mathbf{PART}(P_3, \texttt{Cells}) \wedge \texttt{image}(P_1, f_1, \texttt{Cells}) \subseteq P_3.\end{aligned}$$

Because unification can introduce equalities inconsistent with the original constraint, the constraint after unification might not have any solution. For example, unification can make some subset constraints recursive as follows:

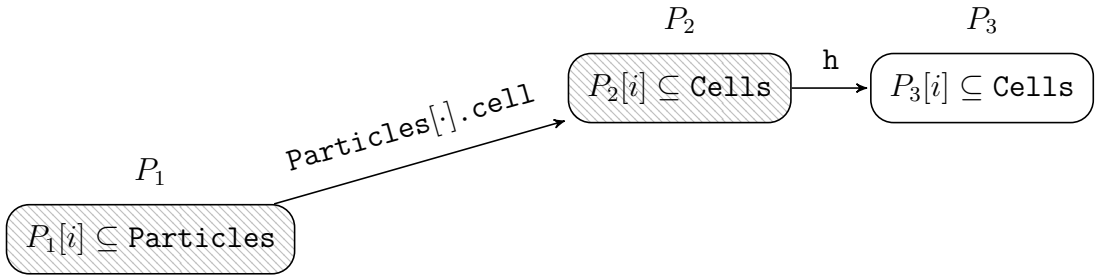
$$\begin{aligned}&\mathbf{PART}(P_1, R) \wedge \mathbf{PART}(P_2, R) \wedge \texttt{image}(P_1, f, R) \subseteq P_2 \Longleftarrow \\ &\mathbf{PART}(P_1, R) \wedge \texttt{image}(P_1, f, R) \subseteq P_1 \wedge P_1 = P_2.\end{aligned}$$

This recursive constraint can be satisfied only by constructing a fixpoint of the function f, which is not expressible in our constraint language. Therefore, the goal of unification is to find a maximal set of unifications that preserves consistency of the partitioning constraint.

Finding all viable unifications requires an exhaustive search in the general case. To make the search efficient, we focus on unifications that reduce the number of subset constraints; intuitively, if unification between partition symbols eliminates some subset constraints, the constraint after unification is no more difficult to resolve than the original one. Such unifications manifest as isomorphic subgraphs in a graph that represents a partitioning constraint. In this *constraint graph* each node corresponds to a partition symbol, an unlabeled edge from P_1 to P_2 represents the subset constraint $P_1 \subseteq P_2$, and an edge labeled with a function symbol f encodes the subset



(a) Common subgraph



(b) After unification ($P_2 = P_4 \wedge P_3 = P_5$)

Figure 3.5: Unification as a common subgraph problem

constraint $\texttt{image}(P_1, f, R) \subseteq P_2$. (Other cases need not be expressed by this graph, because the inference algorithm only generates subset constraints of the two forms.) Isomorphic subgraphs in this graph correspond to partition symbols connected by the same subset constraints (after renaming symbols). Thus, unifying partition symbols in these isomorphic subgraphs also merges multiple subset constraints, one from each subgraph, into one.

Example 5. *Let us revisit the example in Section 1.1: Figure 3.5a shows a constraint graph for the following constraint (predicates are elided):*

$$\cdots \wedge \texttt{image}(P_1, \texttt{Particles}[\cdot].\texttt{cells}, \texttt{Cells}) \subseteq P_2$$
$$\wedge\ \texttt{image}(P_2, \texttt{h}, \texttt{Cells}) \subseteq P_3 \wedge \texttt{image}(P_4, \texttt{h}, \texttt{Cells}) \subseteq P_5.$$

In Figure 3.5a, the subgraph of P_2 and P_3 is isomorphic to that of P_4 and P_5. The solver unifies P_2 and P_4 and P_3 and P_5, with the result shown in Figure 3.5b.

Algorithm 3 shows the constraint solver algorithm with unification. The algorithm uses Algorithm 2 to check if the system of constraints after unification is still solvable

Algorithm 3: Constraint solver algorithm with unification

```
1  Procedure UnifyAndSolve(C_1 ∧ ... ∧ C_N):
2      // Each C_i is represented by a set of conjuncts
3      Sort C_1, ..., C_N in descending order of |C_i|
4      C ← C_1
5      for i = 2, N do
6          C' ← C_i
7          while C' ≠ ∅ do
8              G ← the next biggest common subgraph in C and C'
9              if G = ∅ :
10                 C ← C ∧ C'
11                 C' ← ∅
12             else:
13                 U ← P'_1 = P_1 ∧ ... ∧ P'_K = P_K induced by G
14                 if Solve(C ∧ C', U) ≠ ∅ :
15                     // Filter out unified terms
16                     C' ← C'[P'_1 ↦ P_1] ··· [P'_K ↦ P_K] − C
17     return Solve(C, ∅)
```

(line 13). Although finding the largest common subgraph in a constraint graph (line 7) is known to be NP-complete [40], in practice unification is not a significant cost as constraint graphs are small and we do not attempt to find the absolutely maximal common subgraph. Furthermore, the algorithm greedily tries to unify the first few largest subgraphs in a constraint graph (line 3), based on the observation that these subgraphs often contain other smaller subgraphs. In the average case, common subgraphs can be identified simply by constructing a product graph of constraint graphs. Assuming the unification succeeds in a constant number of trials, the asymptotic time complexity of this greedy algorithm is $O(NM^2)$, where N is the number of constraints to unify and M is the number of graph nodes.

3.2.3 External Constraints

As seen in Section 1.1, programmers often have invariants on existing partitions used in manually parallelized parts. The constraint solver can exploit these invariants by adding them to the partitioning constraint for a program and holding their partition

symbols fixed (no expressions are synthesized for external constraints).

Example 6. *The program in Figure 1.7 specifies an invariant on partitions* `pCells` *and* `pParticles`*, which can be added to the partitioning constraint from Example 5 as follows:*

$$\cdots \wedge \texttt{image}(P_1, \texttt{Particles}[\cdot]\texttt{.cells}, \texttt{Cells}) \subseteq P_2$$
$$\wedge\ \texttt{image}(P_2, \texttt{h}, \texttt{Cells}) \subseteq P_3 \wedge \texttt{image}(P_4, \texttt{h}, \texttt{Cells}) \subseteq P_5$$
$$\wedge\ \texttt{image}(\texttt{pParticles}, \texttt{Particles}[\cdot]\texttt{.cells}, \texttt{Cells}) \subseteq \texttt{pCells}$$

The solver finds unifications between P_1 *and* `pParticles`*;* P_2*,* `pCells`*, and* P_4*; and* P_3 *and* P_5*, yielding the following constraint:*

$$\cdots \wedge \texttt{image}(\texttt{pParticles}, \texttt{Particles}[\cdot]\texttt{.cells}, \texttt{Cells}) \subseteq \texttt{pCells}$$
$$\wedge\ \texttt{image}(\texttt{pCells}, \texttt{h}, \texttt{Cells}) \subseteq P_3.$$

Since the LHS of the subset constraint on P_3 *is closed, the solver strengthens it to an equality and eventually produces this solution:*

$$P_1 = \texttt{pParticles} \qquad P_2 = P_4 = \texttt{pCells}$$
$$P_3 = P_5 = \texttt{image}(\texttt{pCells}, \texttt{h}, \texttt{Cells}).$$

3.2.4 Generalized Image and Preimage

Some programs have loops where the iteration space is determined by values of a region, typically arising in sparse matrix algorithms. The SpMV code using Compressed Sparse Row (CSR) format in Figure 3.6 is one such example. In this code, the matrix is represented by the region `Mat` where the field `val` contiguously stores the non-zero values of the matrix and the field `ind` stores the column indices of those non-zeros. The inner loop at line 3 then iterates over columns of the `i`th row in the matrix using the value `Ranges[i]`, a pair of lower and upper bounds of indices in `Mat`.

These loops with *data dependent* iteration spaces require **`IMAGE`** and **`PREIMAGE`** partitions that derive partitions using functions from indices to *sets* of indices. In Figure 3.6a, the region `Ranges` maps each iteration of the outer loop to a set of iterations of the inner loop, and thus partitions for regions accessed in this inner loop,

```
1 for i in Y:
2   range = Ranges[i]
3   for k in range:
4     Y[i] += Mat[k].val * X[Mat[k].ind]
```

(a) SpMV code

```
1 P1 = equal(Y, N)
2 P2 = image(P1, fID, Ranges)
3 P3 = IMAGE(P2, Ranges[·], Mat)
4 P4 = image(P3, Mat[·].ind, X)
```

(b) Synthesized DPL code

Figure 3.6: SpMV example

such as `Mat` and `X`, must be constructed by collecting (and flattening) the image of this map. The DPL code synthesized for the SpMV code is shown in Figure 3.6b.

Note that the partitioning strategy in Figure 3.6b can lead to suboptimal performance when the the number of non-zeros in each row is uneven, because the partition of the matrix is derived from an **`equal`** partition of `Ranges`. In this case the user can construct a balanced partition of `Ranges` using, for example, a graph partitioning heuristic, such as the one proposed by Ravishankar et al. [61], and provide it as an external constraint.

3.3 Optimizations

As described in Section 3.1, uncentered reductions on distributed memory systems are implemented using temporary buffers, because different tasks can make changes to the same element, and these changes must be reconciled to ensure the result is correct. In our programming model, tasks must specify which partitions need these buffers using reduction permissions, as described in Section 2.4. However, using a reduction buffer of the size of the whole partition is often inefficient because the buffering is required only on the part that is accessed by multiple parallel tasks. Furthermore, if

the partition for uncentered reductions is disjoint, which means each location in the region is updated only by one task, no reduction buffer is necessary. In the rest of this section, we describe two optimizations in the solver to minimize the size of reduction buffers.

3.3.1 Relaxing Disjointness Requirements

One strategy to synthesize a disjoint partition for uncentered reductions (thereby eliminating the reduction buffer) is to use an **`equal`** partition for these reductions and derive a **`preimage`** partition for the iteration space as in Example 3: The solver requires P_2 (the partition symbol for the uncentered reduction in Figure 3.3) to be a disjoint partition by introducing an extra predicate $\mathbf{DISJ}(P_2)$, and the resolution algorithm produces a solution where P_2 is assigned to $\texttt{equal}(\texttt{S})$ and P_1 to a **`preimage`** partition derived from P_2. With this solution, the loop in Figure 3.3 need not request a reduction buffer to parallelize its uncentered reductions.

This strategy does not work when a loop has multiple uncentered reductions using different functions. If the solver uses an **`equal`** partition for these uncentered reductions, then the partition of the iteration space, which must be disjoint because of the uncentered reductions, must contain all preimages of those different functions and the solver cannot prove it to be disjoint using the resolution lemmas. The following example illustrates this issue with multiple uncentered reductions.

Example 7. *Figure 3.7 shows the partitioning constraint for a loop with two uncentered reductions. Using an* **`equal`** *partition for both P_2 and P_3 would lead the solver to an assignment of P_1 to a union of preimages* $\texttt{preimage}(\texttt{R}, \texttt{f}, ..)$ *and* $\texttt{preimage}(\texttt{R}, \texttt{g}, ..)$*, which cannot satisfy the predicate* $\mathbf{DISJ}(P_1)$*.*

Program	Constraints
`for i in R:`	$\mathbf{PART}(P_1, \texttt{R}) \ \wedge \mathbf{COMP}(P_1, \texttt{R}) \wedge \mathbf{DISJ}(P_1)$
`S[f(i)] += R[i]`	$\wedge\ \mathbf{PART}(P_2, \texttt{S}) \wedge \texttt{image}(P_1, \texttt{f}, \texttt{S}) \subseteq P_2$
`S[g(i)] += R[i]`	$\wedge\ \mathbf{PART}(P_3, \texttt{S}) \wedge \texttt{image}(P_1, \texttt{g}, \texttt{S}) \subseteq P_3$

Figure 3.7: Example loop with multiple uncentered reductions

Program	Constraints
`for i in R:`	$\mathbf{PART}(P_1, \mathtt{R})$
`  if f(i) in S: S[f(i)] += R[i]`	$\wedge\ \mathbf{PART}(P_2, \mathtt{S}) \wedge \mathbf{image}(P_1, \mathtt{f}, \mathtt{S}) \subseteq P_2$
`  if g(i) in S: S[g(i)] += R[i]`	$\wedge\ \mathbf{PART}(P_3, \mathtt{S}) \wedge \mathbf{image}(P_1, \mathtt{g}, \mathtt{S}) \subseteq P_3$

(a) Relaxed loop

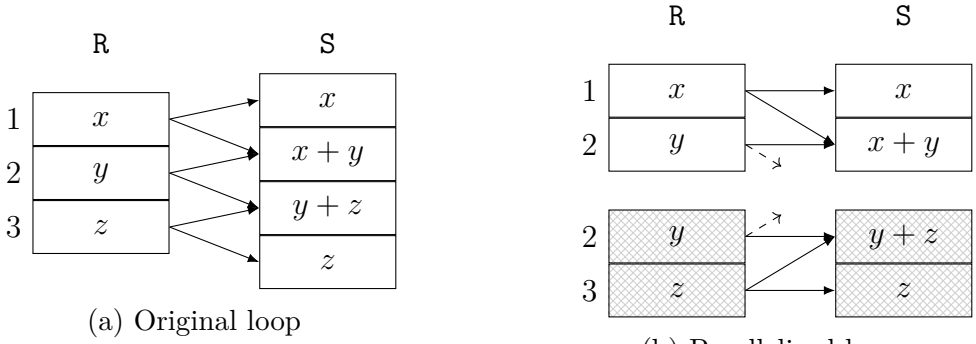
```
parallel for p in P1:
  R = P1[p]
  S = P2[p]
  for i in R:
    if f(i) in S: S[f(i)] += R[i]
    if g(i) in S: S[g(i)] += R[i]                    (P2 = P3)
```

(b) Parallelized loop

Figure 3.8: Relaxing disjointness constraint from uncentered reductions

The obvious alternative of assigning a disjoint partition to only one of the uncentered reductions would still require a reduction buffer for the other uncentered reduction. However, the disjointness requirement can be lifted completely by rewriting the loop into a *relaxed* form, shown in Figure 3.8a. This loop has a guard for each uncentered reduction. In a serial execution these guards are trivial (always true), but when regions used in these guards are replaced by partitions (shown in Figure 3.8b), the guards prevent contributions in the original loop from being applied multiple times. Therefore, the solver no longer needs a **DISJ** predicate on the iteration space partition and can use the union of preimages, which was not viable before the relaxation. Figure 3.9 shows how guard conditions work; even though some iteration space elements appear in more than one subregion of the iteration space partition, each iteration contributes to each reduction only once. (Note that the partitions P_2 and P_3 in the constraints were unified in the parallelized code.)

This relaxation is not always beneficial, because it introduces redundant computation and extra communication due to overlap among subregions of the iteration space partition, and is not always applicable. We heuristically relax loops only when all loops using the same region as the iteration space can be relaxed.



(a) Original loop

(b) Parallelized loop

Figure 3.9: Example execution of loops in Figure 3.8

3.3.2 Finding Private Sub-Partitions

In cases when the relaxation is not applied, the optimizer tries to subtract a *private sub-partition* from the reduction partition. A private sub-partition of a partition P is a disjoint partition P_p that satisfies $P_p \subseteq P$. Since the private sub-partition is disjoint, the program need not request a reduction buffer. However, the parallel loop must be modified to account for the fact that now the reduction partition is divided into two parts; if the original reduction partition P is divided into a private sub-partition P_p and the rest $P_s = P - P_p$, then the original parallel loop:

```
parallel for j in P':
  for i in P'[j]:
    P[j][g(i)] += P'[j][i]
```

must be rewritten to:

```
parallel for j in P':
  for i in P'[j]:
    if g(i) in P_p[j]: P_p[j][g(i)] += P'[j][i]
    else:  P_s[j][g(i)] += P'[j][i]
```

Although there is no general construction of private sub-partitions for a partition, we can use the following theorem when the partition is derived by the **image** operator from another disjoint partition.

Theorem 1. *Let $f_R(P)$ and $f_R^{-1}(P)$ be defined as follows:*

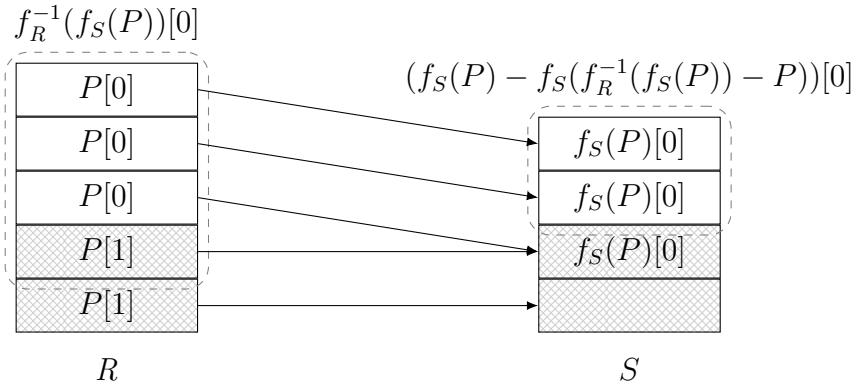


Figure 3.10: Private sub-partition theorem

$$f_R(P) \triangleq \texttt{image}(P, f, R) \qquad f_R^{-1}(P) \triangleq \texttt{preimage}(R, f, P)$$

For a disjoint partition P of a region R, the following expression constructs a private sub-partition of $f_S(P)$ for any f and S:

$$f_S(P) - f_S(f_R^{-1}(f_S(P)) - P).$$

Proof. (Sketch) Each image subregion $f_S(P)[i]$ contains all elements pointed to by those in $P[i]$. Then, the sub-expression $f_R^{-1}(f_S(P))$ extends each subregion $P[i]$ with the elements from the other subregions $P[j]$ $(j \neq i)$ that also point to the subregion $f_S(P)[i]$. Subtracting P from this expanded partition leaves each subregion with only the elements originally from other subregions. Therefore, its image (i.e. $f_S(f_R^{-1}(f_S(P)) - P)$) represents the shared part in the original image partition $f_S(P)$, and thus its complement is a private sub-partition. □

Figure 3.10 illustrates the private sub-partition construction in Theorem 1.

Once the solver identifies a private sub-partition from a partition, a reduction buffer needs to be allocated only for the shared part. This construction can be generalized to cases when the reduction partition consists of multiple image partitions, for which the solver simply takes an intersection of all private sub-partitions in individual image partitions.

3.4 Implementation

We have implemented our constraint-based approach in Regent [64]. Regent provides both first-class support for data partitions and all DPL operators in Figure 2.2 [71], which make it a suitable base system for our approach. Regent uses Legion [17], a distributed runtime system for implicit task parallelism, which detects and enforces data dependencies between tasks and also resolves data movement between data partitions. The constraint inference algorithm and solver are implemented as an optimization pass in the Regent compiler.

The inference algorithm examines parallelizable loops in tasks. When parallelizable loops are nested, the outermost loop is chosen as the target of parallelization. The final stage of auto-parallelization is a source-to-source transformation that converts the original loop into a form that launches parallel tasks using synthesized data partitions. All parallelizable loops are also amenable to CUDA code generation supported by the Regent compiler.

In the rest of this section we describe additional optimizations specific to the Regent implementation.

3.4.1 Optimizing Uncentered Reads

In many cases, some of the elements accessed by uncentered read access are also accessed by centered access in the same loop. For example, a task that computes a 3-point stencil `A[i-1]+A[i]+A[i+1]` on a window of 1D space can find the stencil elements in a subregion containing elements for the centered access `A[i]` except when the task visits either end of its window (in which case at least one of the stencils requires an element from another window). Keeping such elements in multiple partitions is redundant and inefficient in terms of communication.

One way to eliminate this redundancy is to make elements accessed by uncentered reads co-located with those for centered accesses by unifying their partitions. In fact this is the constraint solver's default strategy; for example, for the program in Figure 3.11, the constraint solver would synthesize a union partition $P_{\mathtt{S}}$ of two partitions $P_{\mathtt{S[i]}}$ and $P_{\mathtt{S[f(i)]}}$ for the region `S`, where $P_{\mathtt{S[i]}}$ and $P_{\mathtt{S[f(i)]}}$ contain the elements for the

```
1 for i in R:
2   v = S[i]
3   w = S[f(i)]
4   R[i] = v + w
```

Figure 3.11: Example loop with an uncentered read

```
1 parallel for p in P_R:
2   R = P_R[p]
3   S_priv = P_priv[p]
4   S_ghst = P_ghst[p]
5   for i in R:
6     v = S_priv[i]
7     if f(i) in S_ghst: w = S_ghst[f(i)]
8     else: w = S_priv[f(i)]
9     R[i] = v + w
```

Figure 3.12: Modified loop using ghost region for the uncentered read

centered access at line 2 and the uncentered access at line 3, respectively. However, such union partitions have sparse subregions in general, for which Regent's runtime system currently constructs dense instances for the whole bounding volume; for example, even when a subregion has elements for only two indices 1 and 100, the runtime system allocates an instance containing elements for the whole interval $[1, 100]$. This bloating can limit the scalability of parallelized programs. Our implementation creates union partitions only for the uncentered accesses via affine functions, which incur only a limited amount of bloating. Alternatively, we could change the runtime system to allocate compressed instances for sparse regions, which however would make time complexity of the access no longer asymptotically constant.

Another strategy is to isolate "ghost" elements, i.e., elements exclusive to uncentered accesses, in a partition. For the example program in Figure 3.11 we use the following sub-partitions P_{priv} and P_{ghst} for the uncentered access:

$$P_{priv} = P_{\mathtt{S[i]}} \text{ and } P_{ghst} = P_{\mathtt{S[f(i)]}} - P_{priv}.$$

The original program must be also modified as in Figure 3.12 to account for the

```
1 parallel for p in P_R:
2   R = P_R[p]
3   S_priv = P_priv[p]
4   S_ghst = P_ghst[p]
5   for i in R:
6     v = S_priv[i]
7     if P_C[p][i]: w = S_ghst[f(i)]
8     else: w = S_priv[f(i)]
9     R[i] = v + w
```

Figure 3.13: Example of a cache replacing the guard in Figure 3.12

fact that the original uncentered access is now served by two subregions (lines 7-8). Note that the sub-partition P_{ghst} contains only the ghost elements (and its subregions are often called ghost regions). Only the elements in P_{ghst} require data movement, assuming that the **for** loop runs on a node where each subregion of P_{priv} is allocated.

3.4.2 Caching Inclusion Checks

Guards introduced by the optimizations in Sections 3.3 and 3.4.1 can be expensive to check when the subregions are sparse. To amortize the cost of those checks our implementation replaces them with a cache that stores values of guard conditions. The cache is implemented with a region of boolean values and can be initialized using a **preimage** partition under the function used in the uncentered access. For example, the guard at line 7 in Figure 3.12 is replaced with the cache $P_{\texttt{C}}$`[p]` in Figure 3.13; the cache $P_{\texttt{C}}$ is constructed by the code in Figure 3.14, where **fill**`(R, c)` is a Regent operator that assigns the constant value `c` to every element in the region `R`.

Alternatively, we could *split* the loop to statically disambiguate accesses to multiple regions, as Koelbel and Mehrotra [49] and Adve and Mellor-Crummey [7] distinguished accesses to local data from those to non-local data. We decided not to use this transformation because of the potential combinatorial explosion of cases in the output program.

```
1  C = region(bool, N)                          -- N is the size of R
2  -- Initialize the cache to false
3  fill(C, false)
4
5  -- Construct a partition that contains only the elements for
6  -- which the guard evaluates to true, i.e., a partition P_true
7  -- such that ∀p ∈ P_true. ∀i. f(i) ∈ P_ghst[p] ⟹ i ∈ P_true[p]
8  P_true = preimage(C, f, P_ghst)
9
10 -- Set the cache to true for the elements in P_true
11 for p in P_true:
12   fill(P_true[p], true)
13
14 P_C = image(P_R, f_ID, C)                    -- P_C is isomorphic to P_R
```

Figure 3.14: Initialization code for the cache $P_{\mathtt{C}}$ in Figure 3.13

3.5 Evaluation

We evaluate our implementation using the SpMV code in Figure 3.6 as well as four larger Regent programs: Stencil [72], MiniAero [41], Circuit [65], and PENNANT [38]. All programs have a "main" loop where they spend most of the execution time, and this main loop consists only of parallelizable loops. We measure *weak scaling* performance of the benchmark programs; in weak scaling, the problem size per node is held fixed while the number of nodes in parallel machine is increased. For the last four Regent programs (Stencil, MiniAero, Circuit, and PENNANT) we also compare them with hand-optimized counterparts that are already optimized for weak scaling performance in the previous work [65]; those hand-optimized counterparts achieved near-perfect weak scaling because they programs need only a constant amount of per-node communication for a fixed problem size per node. Therefore, any acute degradation of weak scaling performance is in large part attributable to an inefficient data partitioning strategy.

All experiments were performed on Piz Daint [4], a Cray X50 system; each compute node is equipped with one Intel Xeon E5-2690 CPU with 12 physical cores, one NVIDIA Tesla P100, and 64GB of system memory.

Table 3.1 presents a breakdown of compilation time for benchmark programs. The table also shows the size of each program in terms of the number of auto-parallelized loops and total compilation times of hand-optimized counterparts as a baseline. The constraint inference and solver algorithms and the rewriting to parallel code constitute less than 10 percent of the total compilation time, and the binary code generation is a dominant component. Note that the baseline does not strictly match the time for generating a binary from the auto-parallelized code, because the auto-parallelizer produces a program that is different from the hand-optimized counterpart.

The weak scaling performance of benchmark programs was were measured once the programs reached a steady state. All computation tasks running within the measurement window used only GPUs.

3.5.1 SpMV Microbenchmark

Figure 3.15 shows weak scaling performance of the SpMV code in Figure 3.6. In the experiments, we use a diagonal matrix where each row has a fixed number of non-zeros. With this balanced synthetic matrix the auto-parallelized SpMV code achieved 99% parallel efficiency on 256 nodes.

3.5.2 Stencil

Stencil is a 9-point stencil program for a 2D grid. The stencil consists of a center and eight neighbor points, two for each direction in 2D space. The uncentered access

	SpMV	Stencil	Circuit	MiniAero	PENNANT
Constraint inference	1.7ms	5.0ms	28.4ms	58.5ms	110.7ms
Constraint solver	1.7ms	4.0ms	4.3ms	5.8ms	13.1ms
Code rewrite	49ms	0.3s	0.3s	1.6s	1.9s
Binary generation	2.3s	6.5s	7.2s	22.8s	31.4s
Total	2.4s	6.8s	7.5s	24.4s	33.4s
Number of parallel loops	1	2	3	26	37
Baseline	N/A	8.7s	8.3s	22.7s	27.6s

Table 3.1: Compilation time breakdown

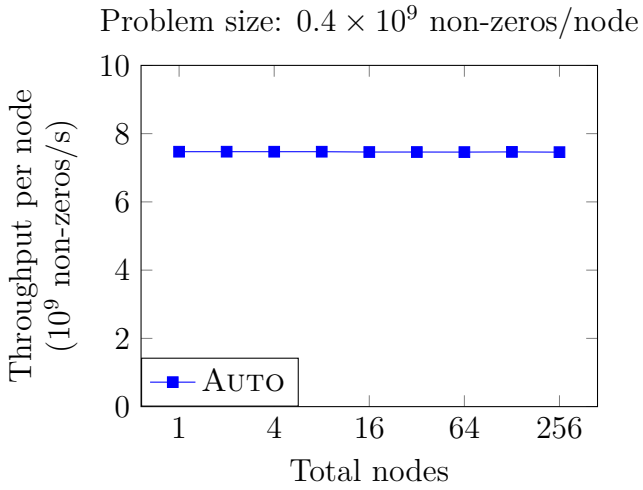


Figure 3.15: Weak scaling performance of SpMV

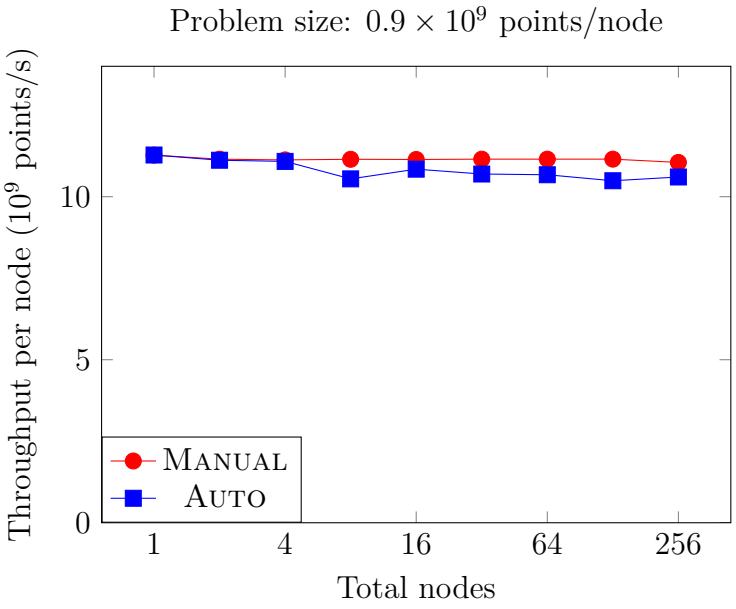


Figure 3.16: Weak scaling performance of Stencil

for each neighbor point corresponds to a distinct subset constraint, for which the constraint solver synthesizes an **`image`** partition of an affine function.

Figure 3.16 shows performance of the hand-optimized code and the auto-parallelized code. The auto-parallelized version achieves 93% parallel efficiency on 256 nodes,

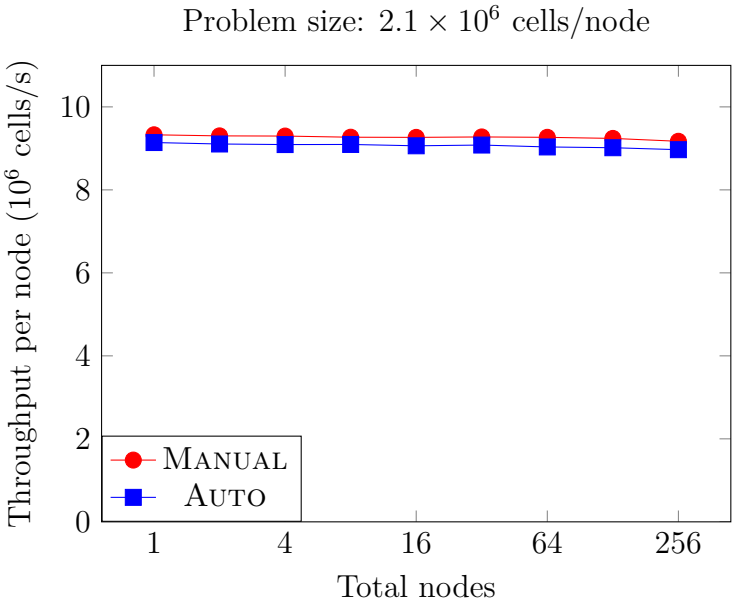


Figure 3.17: Weak scaling performance of MiniAero

whereas the parallel efficiency of the hand-optimized version is 98%. In terms of absolute performance, the auto-parallelized version is slower than the hand-optimized version by 3% on average. The discrepancy is due to an optimization for communication manually applied to the hand-optimized version: The code maintains a copy of the halo part in its own region, which consolidates inter-node data movement for halo exchanges in each direction into a single transfer, while the eight partitions used by the auto-parallelized version require two data transfers per direction. A similar consolidation optimization would be possible in our approach given that uncentered accesses use affine functions, but not pursued.

3.5.3 MiniAero

MiniAero is a proxy application that solves the Navier-Stokes equation for compressible flows. MiniAero uses a 3D hexahedron mesh with faces shared between neighboring hexahedron cells. The simulation calculates flux between cells pointed to by each face. All tasks in the simulation loop read face properties and update cell properties via uncentered reductions using pointers in each face, similar to Figure 3.7; the

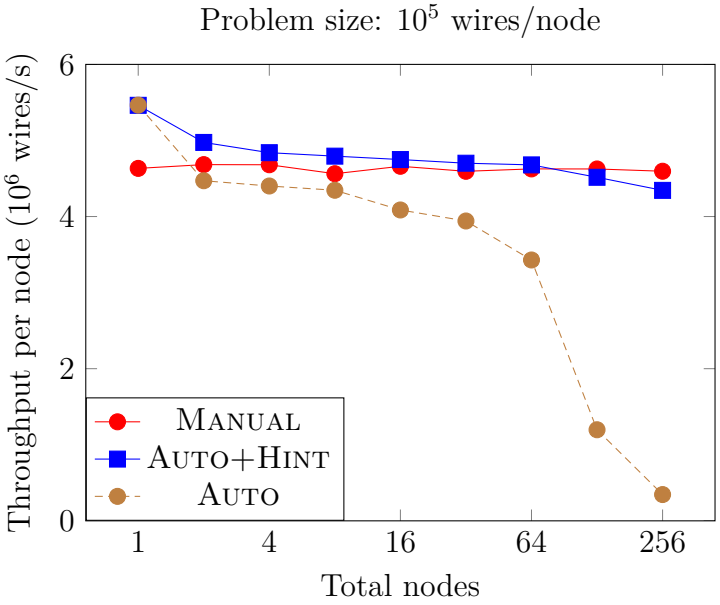


Figure 3.18: Weak scaling performance of Circuit

optimizer applies the optimization in Section 3.3.1 to these reductions to eliminate reduction buffers completely.

Figure 3.17 shows performance of hand-optimized and auto-parallelized versions of MiniAero. Both achieve 98% parallel efficiency on 256 nodes, but the auto-parallelized version is 2% slower on average. This difference is explained by different mesh generators used in the two versions: The mesh generator in the hand-optimized code duplicates faces when they point to cells from two different subregions so that faces surrounding cells in each subregion can be contiguously indexed. On the other hand, because the auto-parallelized code uses a mesh generated for sequential execution, faces in each face subregion can be non-contiguously indexed, leading to a small performance degradation in CUDA kernels generated by the Regent compiler.

3.5.4 Circuit

Circuit simulates electric currents along wires in an unstructured circuit graph. Each wire has pointers to incoming and outgoing nodes, which are used for uncentered read accesses and reductions to the region of nodes. Circuit graphs are randomly generated

in a way that circuit nodes form clusters; a maximum of 20% of wires connect nodes in two different clusters.

To showcase the ability to exploit external constraints, we use the existing parallel circuit graph generator that produces inputs to Circuit and auto-parallelize computation tasks with and without a user constraint describing the initial partition of nodes produced by the generator. Figure 3.18 compares these configurations (Auto+Hint and Auto) with the hand-optimized code (Manual). Without the user constraint, the auto-parallelized code uses an equal partition of circuit nodes, which makes the code match the hand-optimized one within 5% only up to eight nodes. The circuit generator is designed to simulate sparsely connected components, and thus assigns the first 1% of entries in the region of circuit nodes to those connected to nodes in other clusters. As a result, the equal partition of the region of nodes puts all these "shared" nodes in one subregion, making the task using this subregion a communication bottleneck.

To fix this performance issue, we give the solver an interface constraint describing the externally computed circuit partition. The parallel circuit generator uses two partitions of the region `rn` of circuit nodes, `pn_private` for private nodes and `pn_shared` for shared nodes, and the union of these partitions is a disjoint, complete partition of `rn`, as expressed by the following user constraint:

$$\mathbf{DISJ}(\texttt{pn\_private} \cup \texttt{pn\_shared}) \wedge \mathbf{COMP}(\texttt{pn\_private} \cup \texttt{pn\_shared}, \texttt{rn})$$

With this user constraint, the performance of the auto-parallelized code stays within 5% of the hand-optimized code on 256 nodes and shows better performance up to 64 nodes. The latter is due to the fact that the hand-optimized code always requests reduction buffers for the entire subset reserved for shared circuit nodes even when only a few nodes in this subset are shared, whereas the auto-parallelized code computes tight private sub-partitions to reduce the size of reduction buffers for uncentered reductions.

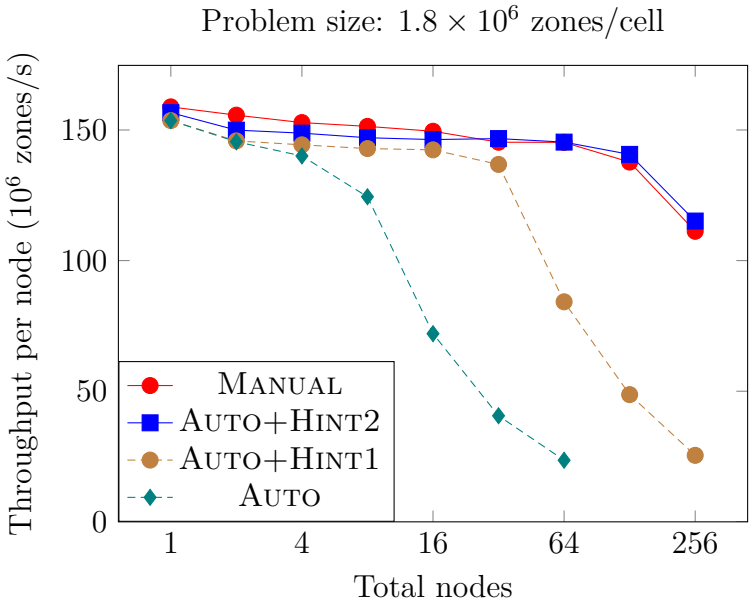


Figure 3.19: Weak scaling performance of PENNANT

3.5.5 PENNANT

PENNANT is a proxy application for Lagrangian hydrodynamics on 2D meshes. Each polygonal *zone* in the mesh consists of triangular *sides*; each pair of sides share two *points*. Each side has five pointers used in uncentered accesses: two pointers to the previous and next neighbor sides in the same zone, one to the zone, and the last two to points at the vertices of the zone.

Similar to the random circuit generator in Circuit, PENNANT's mesh generator separates points shared by sides in different subregions from those owned by a single subregion of sides. Shared points reside in the initial entries in the region of points. Because of this separation, performance of the auto-parallelized code without any user constraint (Auto in Figure 3.19) keeps up with the hand-optimized one (Manual) only up to four nodes and then drops due to the communication bottleneck.

After adding an external constraint describing the partitioning of points, the auto-parallelized code matches the hand-optimized one within 6% up to 32 nodes (Auto+Hint1). The auto-parallelized code still struggles to scale beyond 64 nodes, but for a different reason: the partitions constructed by the synthesized DPL code

exhibit sparsity patterns inefficiently handled by the underlying runtime system, even though they are equivalent to those used in the hand-optimized one in terms of induced inter-node communication. We circumvent this issue by providing additional constraints to guide the solver to synthesize simpler DPL code as follows:

- We reused the existing disjoint, complete partitions `rs_p` and `rz_p` of sides and zones, respectively. The parallel mesh generator guarantees that zones pointed to by the sides in the ith subregion of `rs_p` are all contained in the ith subregion of `rz_p` (i.e.,$\texttt{image}(\texttt{rs\_p}, \texttt{rs}[\cdot]\texttt{.mapsz}, \texttt{rz}) \subseteq \texttt{rz\_p}$).

- Additionally, each side s has all its neighbor sides accessed via $\texttt{rs}[s]\texttt{.mapss3}$ and $\texttt{rs}[s]\texttt{.mapss4}$ in the same subregion:

$$\begin{aligned} &\texttt{image}(\texttt{rs\_p}, \texttt{rs}[\cdot]\texttt{.mapss3}, \texttt{rs}) \subseteq \texttt{rs\_p} \\ \wedge\ &\texttt{image}(\texttt{rs\_p}, \texttt{rs}[\cdot]\texttt{.mapss4}, \texttt{rs}) \subseteq \texttt{rs\_p} \end{aligned}$$

  Although these constraints are recursive, the solver can still check the consistency as long as a satisfying partition (`rs_p`) is provided.

- Finally, the mesh generator creates a partition `rp_p_private` of private points, which can be used as a private sub-partition for uncentered reductions using $\texttt{rs}[\cdot]\texttt{.mapsp1}$:

$$\texttt{preimage}(\texttt{rs}, \texttt{rs}[\cdot]\texttt{.mapsp1}, \texttt{rp\_p\_private}) \subseteq \texttt{rs\_p}$$

With these additional user constraints there is no noticeable difference between the auto-parallelized and hand-optimized versions (Auto+Hint2). This example shows the constraint interfaces' ability to provide extra information to gracefully deal with cases where the auto-parallelizers' heuristics do not quite match reality. Writing the additional constraints is still much easier than parallelizing the code by hand, and preserves the option of parallelizing the code in a different way in a different context or after further improvements in the underlying runtime system.

3.6 Case Study: Soleil-X

In this section, we discuss our experience in applying the constraint-based approach to Soleil-X [68]. Soleil-X is a multi-physics solver for a particle-laden turbulent flow problem developed for the Predictive Science Academic Alliance Program (PSAAP) II program [2] at Stanford. The solver aims at high-fidelity simulations of a concentrated solar energy receiver (whose governing equations are beyond the scope of this dissertation and can be found in other papers [45, 57]), which involves simulations of three physical phenomena: Eulerian fluid flows, Lagrangian particle dynamics, and thermal radiation. Another goal of the solver is the productive development of simulation components using alternative programming technologies to the traditional MPI software stack.

Soleil-X was initially written in Liszt [20], a domain-specific language (DSL) for mesh-based PDE solvers. A Liszt program consists of a set of kernels, which concisely describe per-element computations, and a control plane that determines the sequential control flow. The DSL compiler auto-parallelizes the program for distributed execution by exploiting parallelism implicit in computation kernels. As the hard task of parallelization is handled by the DSL compiler, domain scientists can focus on implementing a domain logic. The initial result was promising; the fluid solver fit completely into Liszt's programming abstractions and the prototype implementation scaled well to a small number of nodes.

However, extrapolating this initial result to the whole simulation was quickly impeded by the DSL's limited capability. At the time Liszt provided distributed code generation only for for-all style parallel computations over mesh elements using affine stencils and anything beyond this pattern required a serious extension to the DSL design and implementation. Unfortunately, the two remaining components, Lagrangian particles and thermal radiation with the discrete ordinate method (DOM) [54], turned out to be difficult to implement given this limited expressivity of Liszt.

We eventually decided to change the implementation language to Regent. Regent is a superset of Liszt in terms of language expressivity and has constructs that are directly mappable from the Liszt counterparts, making it easy to build a translator

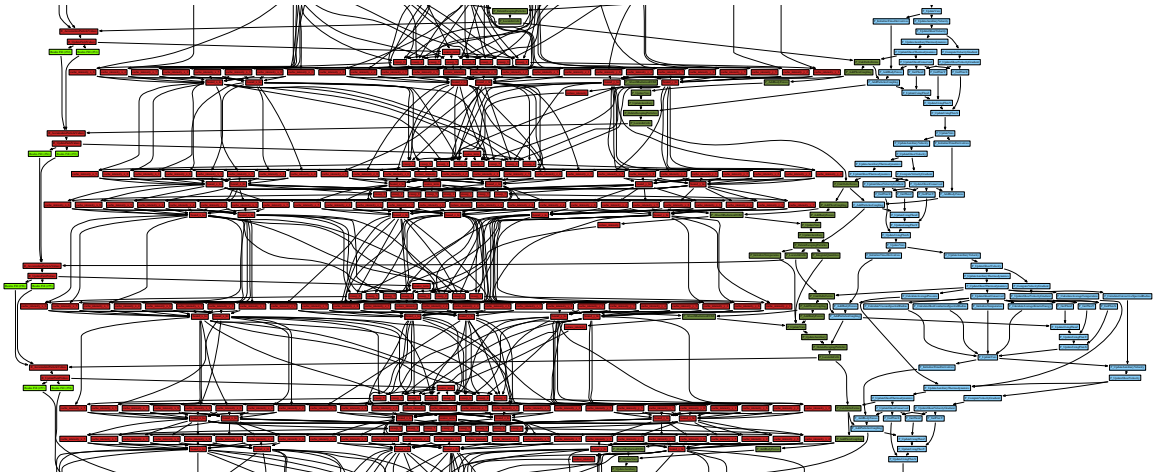


Figure 3.20: Task graph of Soleil-X for a single time step on a single processor. Fluid, particle, radiation tasks are blue, green, and red colored, respectively.

from Liszt to Regent. Furthermore, unlike Liszt, which is domain specific, Regent is a general purpose language that was not only able to express the remaining components but also capable of handling other future, unanticipated extensions to the application.

Our constraint-based approach was conceived in the midst of this transition. From lessons we learned from our experience with Liszt, we aimed at designing an auto-parallelizer that enables the composition of program components parallelized by different means; we wanted the auto-parallelizer to handle all the existing kernels in the Soleil-X code base, which are known to be parallelizable, and yet allow us to manually write components that are not amenable to auto-parallelization and to integrate them seamlessly with the rest of the program. The constraint-based nature of our auto-parallelizer was key to achieving this goal; the coupled fluid and particle simulation was entirely auto-parallelized by the compiler except for the particle transfer, for which we were able to swap the compiler generated implementation with a more efficient, hand-written one (similar to the example in Figure 1.7) satisfying the partitioning constraints; the DOM solver for radiation was also manually parallelized and seamlessly mixed with the rest of the program by making it reuse data partitions used in the auto-parallelized components.

Figure 3.20 shows the task graph of Soleil-X for a single time step on a single

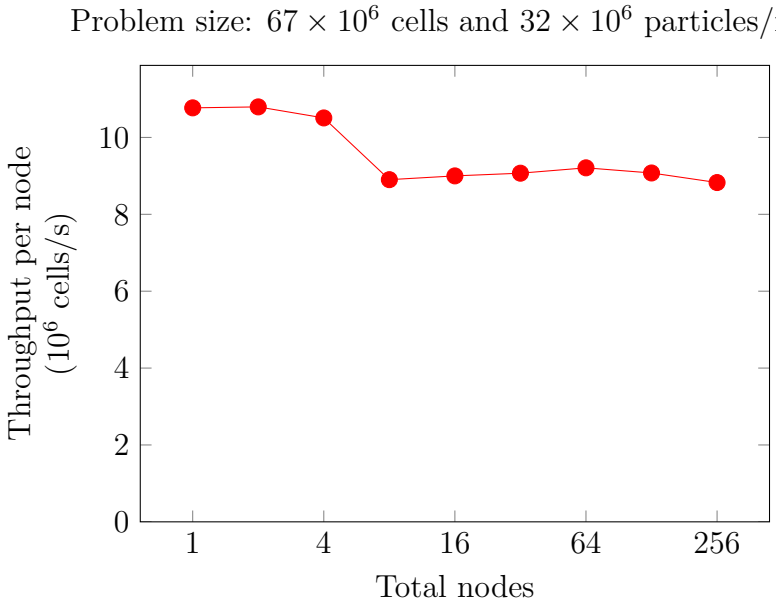


Figure 3.21: Weak scaling performance of Soleil-X

processor. Blue, green, and red colors represent that the task is part of the fluid, particle, and radiation simulation, respectively. Four horizontal clusters of nodes in the graph correspond to the fact that Soleil-X uses a fourth-order Runge-Kutta marching scheme. Another interesting observation is that the graph has edges that connect nodes with different colors, which represent data dependencies between tasks in different simulation components. This fine-grained composition of simulation components at the granularity of individual tasks is enabled by the auto-parallelizer facilitating such composition via partitioning constraints.

We measured weak scaling performance of Soleil-X on Sierra supercomputer [5]; each compute node has 4 NVIDIA Tesla V100 GPUs and an IBM Power9 CPU with 44 cores. We ran computation tasks only on GPUs and used the Power9 processor for running the runtime system. Figure 3.21 plots the weak scaling performance of Soleil-X on up to 256 nodes. The plot shows bi-modal performance, which is due to an increase in data movement; the fluid solver in Soleil-X leaves some of the dimensions unpartitioned, for which no communication is necessary, on up to 4 nodes, and every node starts to communicate with the maximum number of neighbors only

starting with 8 nodes. The performance is stable (within 1%) from 8 nodes onward, demonstrating that the auto-parallelized code maintains communication efficiency once it reaches the maximum number of per-node neighbors.

Chapter 4

Dynamic Tracing

In our implicitly parallel tasking model, the runtime system is responsible for discovering parallelism implicit in programs. To serve this need, the runtime system constructs a task graph that captures all dependencies between task instances. The constructed task graph then encodes available parallelism in a set of mutually unreachable nodes, representing a set of tasks that can potentially run in parallel.

However, the runtime system's dependence analysis is an expensive way to build task graphs as it is a generic "interpreter" of task instances agnostic to the program structure; this interpreter takes each task instance one at a time, analyzes the task instance's dependencies on previous task instances, and records those dependencies to the task graph. If instead our goal is to build a specific graph for a set of task instances, we can specialize the interpreter's analysis to a process that constructs just that one graph. This *explicit* graph construction is much more efficient than the dynamic dependence analysis as it has no interpretive overhead. Furthermore, because programs often have a *trace* of task instances launched repeatedly, specializing the dependence analysis for such traces can greatly reduce runtime overhead, thereby improving the strong scalability of parallel execution.

Drawing from this interpreter analogy, dynamic tracing specializes the dependence analysis of a trace of repetitive task instances (hence the name "tracing") to a *graph calculus* program (Section 4.1); graph calculus is a simple imperative language with

commands that directly construct task graphs. The graph calculus program is associated with a *precondition* that must be satisfied for the program to correctly replace the original dependence analysis of the trace, and a *postcondition* that must be applied to make the dependence analysis state consistent with the task graph constructed by the program. Whenever a previously specialized trace appears during program execution, the specialized program is executed to replace the dependence analysis as long as the precondition is satisfied (Section 4.2).

Our empirical evaluation with the five benchmark programs and S3D-Legion [70], an exascale software for turbulent combustion simulation, demonstrates that our implementation of dynamic tracing in the Legion runtime [17] significantly improves strong scaling performance (Section 4.4).

4.1 Recording Dependence Analysis

Dynamic tracing starts with the *recorder* recording the dependence analysis of a trace. A recording for a trace is initiated in two cases: when a trace has appeared for the first time, or when no recording of a trace passes the precondition check described in Section 4.2.

We assume that traces are already delimited in a program. A trace is a sequence of task instances that are issued between a **begin_trace** and a matching **end_trace** statement. At least some of the places that tracing can be beneficial are obvious, such as around important loops. The following example from Figure 1.8 delimits all traces in the program:

```
1 while *:
2   begin_trace
3   for i = 0, 2: F(A[i])
4   for i = 0, 2: G(A[h(i)])
5   end_trace
```

The recorder uses *graph calculus*, whose syntax is shown in Figure 4.1, to express task graphs. Graph calculus uses *events* that signal the termination of operations.

$$
\begin{array}{l}
T \in \mathit{TaskId} \quad I \in \mathit{RegionInstance} \quad e \in \mathit{Event} \quad \circ \in \mathit{ReductionOp} = \{\texttt{+}, \texttt{*}, \ldots\} \\
\mathit{op} \in \mathit{Operation} \quad c \in \mathit{Command} \\
\quad c \;\; ::= \;\; e := \texttt{op}(\mathit{op}, e) \mid e := \texttt{merge}(\overline{e}) \mid e := \texttt{fence} \mid c; c \\
\mathit{op} \;\; ::= \;\; T(\overline{I}) \mid I \leftarrow I \mid I \leftarrow I \circ I \cdots
\end{array}
$$

Figure 4.1: Syntax of graph calculus

An `op` command has the form $e_2 := \texttt{op}(op, e_1)$. The operation *op* begins execution after the event e_1 triggers, and the event e_2 triggers when *op* terminates. Possible operations include tasks (of the form $T(\overline{I})$), copies (of the form $I \leftarrow I$), and reduction applications (of the form $I \leftarrow I \circ I$). To express multiple predecessors for an operation, the `merge` command merges a set of events into an event that is triggered when the events being merged are all triggered. A `fence` command creates a *fence*, an operation that finishes only after all preceding operations terminate. Fences allow graph calculus commands to work correctly with earlier untraced parts of the execution, as the previous dependent operations potentially include operations not in the trace. Finally, the calculus has command sequencing.

The recorder generates graph calculus commands from a dependence analysis of a trace as follows. Each trace operation *op* has a corresponding command $e_2 := \texttt{op}(op, e_1)$. The termination event e_2 is unique (is not used on the left-hand side of any other `op` command). The event e_1 is the merge (using a `merge` command) of the termination events of *o*'s dependence predecessors in the trace. For example, in Figure 4.2, task instance $\texttt{T}_\texttt{c}(\texttt{A}^\alpha, \texttt{B}^\alpha)$ has two predecessors $\texttt{T}_\texttt{a}(\texttt{A}^\alpha, \texttt{B}^\alpha)$ and $\texttt{A}^\alpha \leftarrow \texttt{A}^\alpha\texttt{+}\texttt{A}^\beta$, whose events $\texttt{e}_2$ and $\texttt{e}_6$ are merged into $\texttt{e}_7$. If there is no predecessor (e.g., because this is the first operation of the trace), a fence is introduced to safely capture any dependencies on those operations that are not recorded. Task instance $\texttt{T}_\texttt{a}(\texttt{A}^\alpha, \texttt{B}^\alpha)$ in Figure 4.2 uses fence $\texttt{e}_1$ as it has no predecessor in the trace.

When the recorder reaches the end of the trace, the recorder inserts an `op` statement for a *summary* operation, a task instance that writes to all region instances used in the trace. The key difference between a fence and a summary operation is that a fence waits on all the preceding operations, both within and out of the current trace, whereas the summary operation has dependencies only on operations within the trace.

Any subsequent operation that has dependencies on any of the replayed operations can safely catch the dependencies transitively through the summary operation.

The recorder also computes the *precondition* and *postcondition* of recorded commands, which are used in the replaying stage; the precondition is a set of region instances that must be valid for recorded commands to replay the same subgraph as the original dependence analysis; the postcondition is a set of region instances that become valid after recorded commands replay a subgraph. The precondition and postcondition are computed by processing trace operations in order, beginning with empty pre and postconditions, and applying the following rules:

- If rule $\mathbf{R_1}$ was applied to the region instance and the region instance is not in the postcondition, that region instance is added to the pre and postcondition. For example, in Figure 4.2, the region instance $\mathtt{A}^{\alpha}$ is added to the pre and postcondition when the task instance $\mathtt{T_a}(\mathtt{A}^{\alpha}, \mathtt{B}^{\alpha})$ is recorded.

- If rule $\mathbf{R_2}$ was applied to the region instance and the source instance of the copy is not in the postcondition, that source instance is added to the pre and postcondition. The target instance of the copy is added to the postcondition. For example, the target instance $\mathtt{B}^{\beta}$ of the copy $\mathtt{B}^{\beta} \leftarrow \mathtt{B}^{\alpha}$ in Figure 4.2 is added to the postcondition.

- If rule $\mathbf{Red_1}$ was applied to the region instance, the postcondition of that region is cleared for the set of indices where the reduction was applied, and that instance is added to the postcondition. Each applied outstanding reduction is then handled by the following rules:

  - If the reduction instance is not in the postcondition, that reduction instance is added to the precondition.
  - Otherwise, the reduction instance is removed from the postcondition as it is no longer outstanding.

  For example, when the reduction application $\mathtt{A}^{\alpha} \leftarrow \mathtt{A}^{\alpha}\mathtt{+}\mathtt{A}^{\beta}$ in Figure 4.2 is recorded, the reduction instance $\mathtt{A}^{\beta}$ is removed from the postcondition and the target instance $\mathtt{A}^{\alpha}$ becomes a sole instance in the postcondition for the region $\mathtt{A}$.

Tasks: **task** $\mathtt{T_a}(\mathtt{R},\mathtt{S})$ **reads**(R),**writes**(R),**writes**(S) **task** $\mathtt{T_b}(\mathtt{R},\mathtt{S})$ **reduces+**(R),**reads**(S)
task $\mathtt{T_c}(\mathtt{R},\mathtt{S})$ **reads**(R),**reads**(S),**writes**(S)

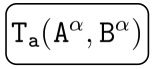
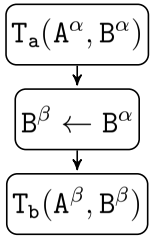
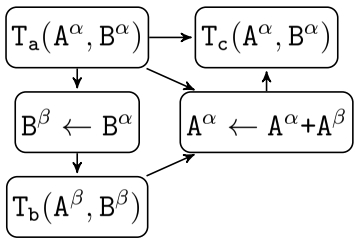
Task Instance	Task Graph	Recorder State	Recorded Commands
$\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$		Events: $\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha) \mapsto \mathtt{e}_2$ Preconditions: $\mathtt{A} \mapsto \mathtt{A}^\alpha$ Postconditions: $\mathtt{A} \mapsto \mathtt{A}^\alpha, \mathtt{B} \mapsto \mathtt{B}^\alpha$ Region Instances: $\mathtt{A}^\alpha, \mathtt{B}^\alpha$	$\mathtt{e}_1 := \mathtt{fence};$ $\mathtt{e}_2 := \mathtt{op}(\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha), \mathtt{e}_1);$
$\mathtt{T_b}(\mathtt{A}^\beta,\mathtt{B}^\beta)$		Events: $\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha) \mapsto \mathtt{e}_2$ $\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha \mapsto \mathtt{e}_3$ $\mathtt{T_b}(\mathtt{A}^\beta,\mathtt{B}^\beta) \mapsto \mathtt{e}_4$ Preconditions: $\mathtt{A} \mapsto \mathtt{A}^\alpha$ Postconditions: $\dfrac{\mathtt{A} \mapsto \mathtt{A}^\alpha}{\mathtt{A} \mapsto \mathtt{A}^\beta}$, $\mathtt{B} \mapsto \{\mathtt{B}^\beta, \mathtt{B}^\alpha\}$ Region Instances: $\mathtt{A}^\beta, \mathtt{A}^\alpha, \mathtt{B}^\beta, \mathtt{B}^\alpha$	$\mathtt{e}_3 := \mathtt{op}(\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha, \mathtt{e}_2);$ $\mathtt{e}_4 := \mathtt{op}(\mathtt{T_b}(\mathtt{A}^\beta,\mathtt{B}^\beta), \mathtt{e}_3);$
$\mathtt{T_c}(\mathtt{A}^\alpha,\mathtt{B}^\alpha)$		Events: $\mathtt{T_a}(\mathtt{A}^\alpha,\mathtt{B}^\alpha) \mapsto \mathtt{e}_2$ $\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha \mapsto \mathtt{e}_3$ $\mathtt{T_b}(\mathtt{A}^\beta,\mathtt{B}^\beta) \mapsto \mathtt{e}_4$ $\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha{+}\mathtt{A}^\beta \mapsto \mathtt{e}_6$ $\mathtt{T_c}(\mathtt{A}^\alpha,\mathtt{B}^\alpha) \mapsto \mathtt{e}_8$ Preconditions: $\mathtt{A} \mapsto \mathtt{A}^\alpha$ Postconditions: $\mathtt{A} \mapsto \mathtt{A}^\alpha, \mathtt{B} \mapsto \mathtt{B}^\alpha$ Region Instances: $\mathtt{A}^\beta, \mathtt{A}^\alpha, \mathtt{B}^\beta, \mathtt{B}^\alpha$	$\mathtt{e}_5 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_4);$ $\mathtt{e}_6 := \mathtt{op}(\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha{+}\mathtt{A}^\beta, \mathtt{e}_5);$ $\mathtt{e}_7 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_6);$ $\mathtt{e}_8 := \mathtt{op}(\mathtt{T_c}(\mathtt{A}^\alpha,\mathtt{B}^\alpha), \mathtt{e}_7);$
(End of the trace)		Insert a summary operation:	$\mathtt{e}_9 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_6, \mathtt{e}_8);$ $\mathtt{e}_{10} := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}(\mathtt{A}^\beta, \mathtt{A}^\alpha, \mathtt{B}^\beta, \mathtt{B}^\alpha), \mathtt{e}_9);$

Figure 4.2: Recording of the dependence analysis in Figure 2.10

Original	Dataflow Analysis	Transitive Reduction
$\mathtt{e}_1 := \mathtt{fence};$		$\mathtt{e}_1 := \mathtt{fence};$
$\mathtt{e}_2 := \mathtt{op}(\mathtt{T}_\mathtt{a}, \mathtt{e}_1);$	$\mathtt{e}_2 \mapsto \mathtt{e}_1$	$\mathtt{e}_2 := \mathtt{op}(\mathtt{T}_\mathtt{a}, \mathtt{e}_1);$
$\mathtt{e}_3 := \mathtt{op}(\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha, \mathtt{e}_2);$	$\mathtt{e}_3 \mapsto \mathtt{e}_1, \mathtt{e}_2$	$\mathtt{e}_3 := \mathtt{op}(\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha, \mathtt{e}_2);$
$\mathtt{e}_4 := \mathtt{op}(\mathtt{T}_\mathtt{b}, \mathtt{e}_3);$	$\mathtt{e}_4 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3$	$\mathtt{e}_4 := \mathtt{op}(\mathtt{T}_\mathtt{b}, \mathtt{e}_3);$
$\mathtt{e}_5 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_4);$	$\mathtt{e}_5 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4$	$\mathtt{e}_5 := \mathtt{merge}(\mathtt{e}_4);$
$\mathtt{e}_6 := \mathtt{op}(\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha{+}\mathtt{A}^\beta, \mathtt{e}_5);$	$\mathtt{e}_6 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_5$	$\mathtt{e}_6 := \mathtt{op}(\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha{+}\mathtt{A}^\beta, \mathtt{e}_5);$
$\mathtt{e}_7 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_6);$	$\mathtt{e}_7 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_5$ $\mathtt{e}_6$	$\mathtt{e}_7 := \mathtt{merge}(\mathtt{e}_6);$
$\mathtt{e}_8 := \mathtt{op}(\mathtt{T}_\mathtt{c}(\mathtt{A}^\alpha, \mathtt{B}^\alpha), \mathtt{e}_7);$	$\mathtt{e}_8 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_5$ $\mathtt{e}_6, \mathtt{e}_7$	$\mathtt{e}_8 := \mathtt{op}(\mathtt{T}_\mathtt{c}(\mathtt{A}^\alpha, \mathtt{B}^\alpha), \mathtt{e}_7);$
$\mathtt{e}_9 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_6, \mathtt{e}_8);$	$\mathtt{e}_9 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_5$ $\mathtt{e}_6, \mathtt{e}_7, \mathtt{e}_8$	$\mathtt{e}_9 := \mathtt{merge}(\mathtt{e}_8);$
$\mathtt{e}_{10} := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}, \mathtt{e}_9);$	$\mathtt{e}_{10} \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_5$ $\mathtt{e}_6, \mathtt{e}_7, \mathtt{e}_8, \mathtt{e}_9$	$\mathtt{e}_{10} := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}, \mathtt{e}_9);$

Figure 4.3: Transitive reduction on the commands in Figure 4.2 (region instances in task instances are elided.)

- If rule **W** was applied to the region instance, the postcondition of that region is cleared and that region instance is added to the postcondition. For example, the task instance $\mathtt{T}_\mathtt{c}(\mathtt{A}^\alpha, \mathtt{B}^\alpha)$ in Figure 4.2 removes the region instance $\mathtt{B}^\beta$ from the postcondition.

- If rule $\mathbf{Red_2}$ was applied to the region instance, the reduction instance is added to the postcondition (until it is applied partially or completely by a subsequent task instance that reads from that region). For example, in Figure 4.2, the reduction instance $\mathtt{A}^\beta$ is added to the postcondition when the task instance $\mathtt{T}_\mathtt{b}(\mathtt{A}^\beta, \mathtt{B}^\beta)$ is recorded.

After a trace is recorded, and before it can be used, we apply two standard compiler passes to optimize the trace: transitive reduction and copy propagation.

Transitive reduction optimizes graph calculus commands by removing transitive dependencies. We run a dataflow analysis that discovers all transitive predecessors for each event and then, among the events being merged by each merge command, we remove those that are transitive predecessors of any other event. In Figure 4.3,

Before Copy Reduction	After Copy Propagation
$\mathtt{e}_1 := \mathtt{fence};$	$\mathtt{e}_1 := \mathtt{fence};$
$\mathtt{e}_2 := \mathtt{op}(\mathtt{T}_\mathtt{a}, \mathtt{e}_1);$	$\mathtt{e}_2 := \mathtt{op}(\mathtt{T}_\mathtt{a}, \mathtt{e}_1);$
$\mathtt{e}_3 := \mathtt{op}(\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha, \mathtt{e}_2);$	$\mathtt{e}_3 := \mathtt{op}(\mathtt{B}^\beta \leftarrow \mathtt{B}^\alpha, \mathtt{e}_2);$
$\mathtt{e}_4 := \mathtt{op}(\mathtt{T}_\mathtt{b}, \mathtt{e}_3);$	$\mathtt{e}_4 := \mathtt{op}(\mathtt{T}_\mathtt{b}, \mathtt{e}_3);$
$\mathtt{e}_5 := \mathtt{merge}(\mathtt{e}_4);$	
$\mathtt{e}_6 := \mathtt{op}(\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha + \mathtt{A}^\beta, \mathtt{e}_5);$	$\mathtt{e}_6 := \mathtt{op}(\mathtt{A}^\alpha \leftarrow \mathtt{A}^\alpha + \mathtt{A}^\beta, \mathtt{e}_4);$
$\mathtt{e}_7 := \mathtt{merge}(\mathtt{e}_6);$	
$\mathtt{e}_8 := \mathtt{op}(\mathtt{T}_\mathtt{c}(\mathtt{A}^\alpha, \mathtt{B}^\alpha), \mathtt{e}_7);$	$\mathtt{e}_8 := \mathtt{op}(\mathtt{T}_\mathtt{c}(\mathtt{A}^\alpha, \mathtt{B}^\alpha), \mathtt{e}_6);$
$\mathtt{e}_9 := \mathtt{merge}(\mathtt{e}_8);$	
$\mathtt{e}_{10} := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}, \mathtt{e}_9);$	$\mathtt{e}_{10} := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}, \mathtt{e}_8);$

Figure 4.4: Copy Propagation on the commands in Figure 4.3 (region instances in task instances are elided.)

event $\mathtt{e}_2$ is removed in the first and second merge commands because it is a transitive predecessor of event $\mathtt{e}_4$, and $\mathtt{e}_6$. Removing transitive dependencies reduces the cost of replaying the graph.

Transitive reductions sometimes leave only a single event in a merge command, which is equivalent to a copy assignment. We run copy propagation to eliminate those unnecessary copies. For example, in Figure 4.4, the merge command $\mathtt{e}_5 := \mathtt{merge}(\mathtt{e}_4)$ is removed and all occurrences of event $\mathtt{e}_5$ are replaced by $\mathtt{e}_4$.

4.2 Replaying Dependence Analysis

The next component of dynamic tracing is to replay dependence analysis for a trace. Figure 4.5 illustrates how the *replayer* replays dependence analysis for the second appearance of trace $\mathtt{T}_\mathtt{a}(\mathtt{A}^\alpha, \mathtt{B}^\alpha); \mathtt{T}_\mathtt{b}(\mathtt{A}^\beta, \mathtt{B}^\beta); \mathtt{T}_\mathtt{c}(\mathtt{A}^\alpha, \mathtt{B}^\alpha)$ using a recording from the first appearance of the trace. First, the replayer checks that each region instance in the precondition is currently valid (Step 1). If any region instance in the precondition is not valid, the replayer cannot reuse recorded commands, because the original dependence analysis of the trace would issue a copy to make that region instance valid, which is not replayed by the commands. If all recordings fail to pass the precondition check, the replayer stops the current replay and the recorder starts a new recording

session. Otherwise, the replayer proceeds with a recording whose precondition is satisfied. In Figure 4.5, the set of valid instances after task instance $\texttt{T}_\texttt{c}(\texttt{A}^\gamma, \texttt{B}^\alpha)$ is analyzed subsumes the precondition and therefore the recording can be replayed.

Next, the replayer runs recorded commands to reconstruct a subgraph (Step 2). Any explicitly parallel runtime system that supports a synchronization primitive such as an event or stream that can be used to express dependencies between tasks and data movement operations can implement graph calculus. Many common runtime APIs support the requirements for graph calculus. For example, both CUDA [1] and OpenCL [48] can support graph calculus via their use of streams and events respectively to mediate dependencies between kernels and copy operations. Furthermore, for distributed memory cases, systems like Realm [69] and OCR [3] have event primitives that can be used on any node to handle distributed execution of graph calculus commands for computation and data movement.

When replaying a trace, graph calculus commands execute sequentially to construct a subgraph equivalent to the one produced by the original dependence analysis. The semantics of graph calculus commands is straightforward, except for the fence command. A fence command creates a new fence with dependencies on all operations that use any region instance used by commands in the trace. However, the fence is not connected to operations that do not access any region instances used in the trace. This is to prevent those operations, which are independent of the replayed subgraph, from being unnecessarily blocked by that fence. In Figure 4.5, all users of region instances $\texttt{A}^\alpha$, $\texttt{A}^\beta$, $\texttt{B}^\alpha$, and $\texttt{B}^\beta$, which are the ones used in the recorded commands, are connected to the new fence `fence`. Note that the replayed subgraph does not contain transitive dependencies between $\texttt{T}_\texttt{a}(\texttt{A}^\alpha, \texttt{B}^\alpha)$ and $\texttt{A}^\alpha \leftarrow \texttt{A}^\alpha\texttt{+}\texttt{A}^\beta$, and between $\texttt{T}_\texttt{a}(\texttt{A}^\alpha, \texttt{B}^\alpha)$ and $\texttt{T}_\texttt{c}(\texttt{A}^\alpha, \texttt{B}^\alpha)$, unlike the subgraph for the first trace, due to the optimizations in Section 4.1.

Finally, the replayer updates the list of valid instances using the postcondition (Step 3). The known valid instances after a replay of a subgraph may be incorrect because the replayed commands are not analyzed again by dependence analysis. The replayer ensures the system has the correct set of valid instances after replay by tagging region instances in the postcondition as valid and invalidating all other instances.

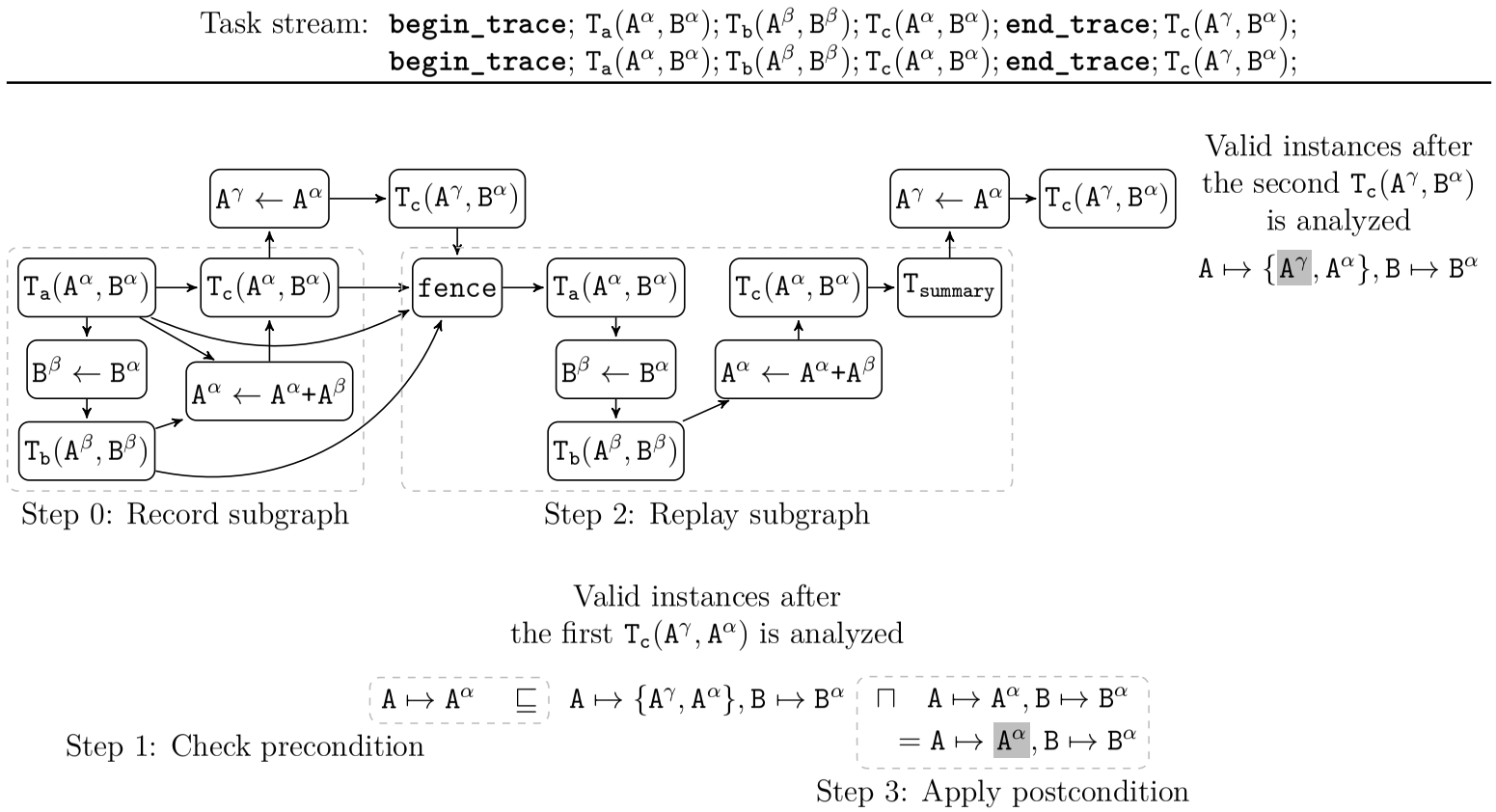


Figure 4.5: Replay of dependence analysis using the recording in Figure 4.2. Region instances in the summary operation are elided.

In Figure 4.5, region instance $\mathtt{A}^{\gamma}$ is invalidated after the replay.

Before restarting dependence analysis, the replayer reinitializes the dependence analysis state using the summary operation. This makes the dependence analysis aware of the net effect of the replayed operations; any subsequent operation can catch its dependencies on any of the replayed operations transitively through this summary operation. For example, the dependence between the reduction application $\mathtt{A}^{\alpha} \leftarrow \mathtt{A}^{\alpha}\mathtt{+}\mathtt{A}^{\beta}$ in the replayed graph and the subsequent copy $\mathtt{R}^{\gamma} \leftarrow \mathtt{R}^{\alpha}$ is captured by those between $\mathtt{A}^{\alpha} \leftarrow \mathtt{A}^{\alpha}\mathtt{+}\mathtt{A}^{\beta}$ and the summary operation $\mathtt{T}_{\mathtt{summary}}$, and between $\mathtt{T}_{\mathtt{summary}}$ and $\mathtt{R}^{\gamma} \leftarrow \mathtt{R}^{\alpha}$.

Algorithm 4 shows the complete dynamic tracing algorithm. The algorithm has two modes: analysis mode (DEP) and tracing mode (TRACE). If it is in analysis

Algorithm 4: Dynamic tracing algorithm

```
   Data: A tracing state ST ∈ {DEP, TRACE}, initially DEP
   Data: A current trace TR, initially ∅
 1 Procedure DynamicTracing(call):
 2 |   if call is a task :
 3 |   |   T ← Map(call)
 4 |   |   if ST is DEP :
 5 |   |   |   AnalyzeDependence(T)
 6 |   |   elseif ST is TRACE :
 7 |   |   |   TR ← TR;T
 8 |   elseif call is begin_trace :
 9 |   |   ST ← TRACE
10 |   |   TR ← ∅
11 |   elseif call is end_trace :
12 |   |   RecordOrReplay()
13 |   |   ST ← DEP
14 Procedure RecordOrReplay():
15 |   if ∃ recording R for TR that passes precondition check :
16 |   |   Replay(R)
17 |   |   ApplyPostcondition(R)
18 |   else:
19 |   |   R ← Record(TR)
20 |   |   register R to the runtime system
```

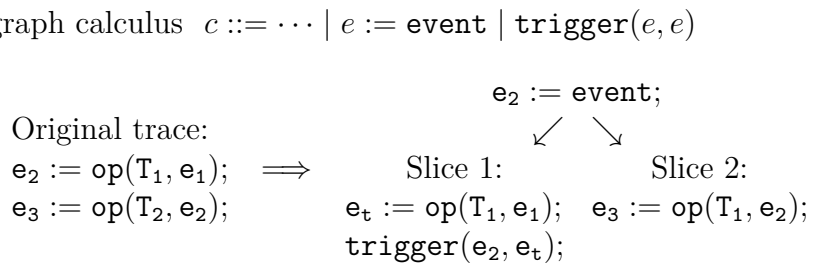


Figure 4.6: Transformation for parallel trace replay

mode, the algorithm maps each task call to a task instance that goes through the normal dependence analysis. Otherwise, the algorithm builds a trace of task instances until it hits the end of that trace (line 11), and it either records or replays the trace (`RecordOrReplay`), based on the criteria described in this section. The algorithm changes from analysis mode to tracing mode when it sees the beginning of a trace (line 9), and from tracing mode to analysis mode once it finishes either a recording or a replay (line 13).

4.2.1 Parallel Trace Replay

A recorded graph calculus program can be further optimized so that they can be replayed in parallel. Figure 4.6 illustrates the key transformation for parallel replay. This transformation splits a graph calculus program for a trace into *slices*. Commands appear in slices in the same order as the original program and any events that are created in one slice and referenced in other slices are connected using the graph calculus extension shown in the figure. A command $\texttt{e} := \texttt{event}$ creates a new untriggered event and assigns it to an event variable $\texttt{e}$. A command $\texttt{trigger}(\texttt{e}_1, \texttt{e}_2)$ registers an event dependence such that event $\texttt{e}_1$ is notified as soon as $\texttt{e}_2$ is triggered, which simply corresponds to adding an edge between the operations represented by $\texttt{e}_2$ and $\texttt{e}_1$. Slices generated from a trace can be replayed in parallel.

Minimizing events that "cross" the slice boundary is important for reducing the number of intermediate events for parallel replay, for which we exploit the implicit knowledge encoded in an application's task mappings: We put tasks mapped to

the same processor in the same slice as much as possible because in a well-mapped program they are more likely to have dependencies on one another.

4.3 Optimizations for Idempotent Recordings

Recognizing *idempotent recordings* is crucial to providing an optimized implementation of dynamic tracing. A recording of a trace is idempotent when its postcondition implies its precondition. For example, the recording in Figure 4.2 is idempotent as its postcondition $\mathtt{A} \mapsto \mathtt{A}^{\alpha}, \mathtt{B} \mapsto \mathtt{B}^{\alpha}$ contains its precondition $\mathtt{A} \mapsto \mathtt{A}^{\alpha}$. In this section, we present two optimizations for idempotent recordings.

4.3.1 Eliding Precondition Check and Postcondition Application

The most important property of idempotent recordings is that once an idempotent recording is replayed for a trace, it becomes replayable without having to apply its postcondition and check its precondition again for another replay that immediately follows. In other words, a list of valid instances that satisfies the precondition of an idempotent recording once will still satisfy that precondition no matter how many times the recording is replayed. This allows two further optimizations:

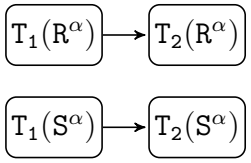
- Once the precondition of an idempotent recording passes, the algorithm never checks the precondition for future consecutive replays of the same trace.
- The algorithm can delay applying the postcondition of an idempotent tracing until it gets a different trace or a task that is not in any trace.

Algorithm 5 shows a modified algorithm to incorporate these optimizations. There are several differences in Algorithm 5 from Algorithm 4. First, Algorithm 5 keeps the previous trace and recording to check that the same trace is repeatedly replayed (line 30–31). Next, it replays a recording without any check when it realizes it is replaying an idempotent recording repeatedly (line 18–19). Finally, it applies the

Algorithm 5: Optimized dynamic tracing algorithm

```
Data: A tracing state ST ∈ {DEP, TRACE}, initially DEP
Data: A current trace TR, initially ∅
Data: A previous trace TR′, initially ∅
Data: A previous recording R′, initially ∅
Procedure DynamicTracing(call):
    if call is a task :
        T ← Map(call)
        if ST is DEP :
            if R′ is idempotent :
                ApplyPostcondition(R′)
                R′ ← ∅
            AnalyzeDependence(T)
        elseif ST is TRACE :
            TR ← TR;T
    elseif call is begin_trace :
        ST ← TRACE
        TR ← ∅
    elseif call is end_trace :
        RecordOrReplay()
        ST ← DEP
Procedure RecordOrReplay():
    if TR = TR′ ∧ R′ is idempotent :
        Replay(R′)
    else:
        if R′ is idempotent :
            ApplyPostcondition(R′)
        if ∃ recording R for TR that passes precondition check :
            Replay(R)
            if R is not idempotent :
                ApplyPostcondition(R)
        else:
            R ← Record(TR)
            register R to the runtime system
        R′ ← R
        TR′ ← TR
```

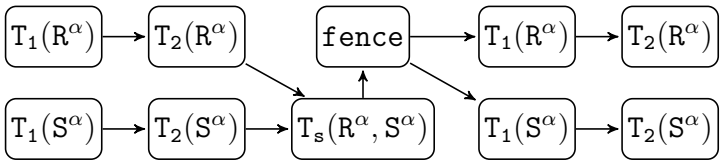
Tasks: **task** $\mathtt{T}_1(\mathtt{R})$ **reads**(R),**writes**(R) **task** $\mathtt{T}_2(\mathtt{R})$ **reads**(R)
Trace: $\mathtt{T}_1(\mathtt{R}^\alpha); \mathtt{T}_1(\mathtt{S}^\alpha); \mathtt{T}_2(\mathtt{R}^\alpha); \mathtt{T}_2(\mathtt{S}^\alpha);$



(a) Task graph of the trace

$$C = \begin{cases} \mathtt{e}_1 := \mathtt{fence}; \\ \mathtt{e}_2 := \mathtt{op}(\mathtt{T}_1(\mathtt{R}^\alpha), \mathtt{e}_1); \\ \mathtt{e}_3 := \mathtt{op}(\mathtt{T}_1(\mathtt{S}^\alpha), \mathtt{e}_1); \\ \mathtt{e}_4 := \mathtt{op}(\mathtt{T}_2(\mathtt{R}^\alpha), \mathtt{e}_2); \\ \mathtt{e}_5 := \mathtt{op}(\mathtt{T}_2(\mathtt{S}^\alpha), \mathtt{e}_3); \\ \mathtt{e}_6 := \mathtt{merge}(\mathtt{e}_4, \mathtt{e}_5); \\ \mathtt{e}_7 := \mathtt{op}(\mathtt{T}_\mathtt{s}(\mathtt{R}^\alpha, \mathtt{S}^\alpha), \mathtt{e}_6); \end{cases}$$

(b) Recorded commands



(c) Task graph from two replays

Figure 4.7: Example of spurious dependencies in trace replays

pending postcondition in cases when the current trace is different from the previous one (line 21–22) or when the task does not belong to any trace (line 5–6).

4.3.2 Fence Elision

Another important optimization that idempotent recordings allow is *fence elision*. Although the fence and the summary operation safely connect a subgraph replayed by graph calculus commands to that generated by dependence analysis and vice versa, they may introduce spurious dependencies between operations because they are a join point in the task graph. For example, in Figure 4.7c, the summary operation $\mathtt{T}_\mathtt{s}(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$ and the fence `fence` add spurious dependencies between the first $\mathtt{T}_2(\mathtt{R}^\alpha)$ and the second $\mathtt{T}_1(\mathtt{S}^\alpha)$, and between the first $\mathtt{T}_2(\mathtt{S}^\alpha)$ and the second $\mathtt{T}_1(\mathtt{R}^\alpha)$. In case of repeatedly replaying the same trace with an idempotent recording, the replayer can keep appending the subgraph from each replay without needing to issue a fence and register the summary operation as these replays do not require precondition checks.

Figure 4.8 illustrates fence elision. First, we "extend" the trace by unrolling the

Region Instance	Readers	Writers
$\mathtt{R}^\alpha$	$\mathtt{e}_2, \mathtt{e}_4$	$\mathtt{e}_2$
$\mathtt{S}^\alpha$	$\mathtt{e}_3, \mathtt{e}_5$	$\mathtt{e}_3$

(a) Readers and writers of region instances in C

$\mathtt{e}'_1 := \mathtt{fence};$
$\mathtt{e}'_2 := \mathtt{op}(\mathtt{T}_1(\mathtt{R}^\alpha), \mathtt{e}'_1);$
$\mathtt{e}'_3 := \mathtt{op}(\mathtt{T}_1(\mathtt{S}^\alpha), \mathtt{e}'_1);$
$\mathtt{e}'_4 := \mathtt{op}(\mathtt{T}_2(\mathtt{R}^\alpha), \mathtt{e}'_2);$
$\mathtt{e}'_5 := \mathtt{op}(\mathtt{T}_2(\mathtt{S}^\alpha), \mathtt{e}'_3);$
$\mathtt{e}'_6 := \mathtt{merge}(\mathtt{e}'_4, \mathtt{e}'_5);$
$\mathtt{e}'_7 := \mathtt{op}(\mathtt{T}_\mathtt{s}(\mathtt{R}^\alpha, \mathtt{S}^\alpha), \mathtt{e}'_6);$

(b) Unroll trace once

$\mathtt{e}_{\mathtt{A1}} := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_4);$
$\mathtt{e}'_2 := \mathtt{op}(\mathtt{T}_1(\mathtt{R}^\alpha), \mathtt{e}_{\mathtt{A1}});$
$\mathtt{e}_{\mathtt{A2}} := \mathtt{merge}(\mathtt{e}_3, \mathtt{e}_5);$
$\mathtt{e}'_3 := \mathtt{op}(\mathtt{T}_1(\mathtt{S}^\alpha), \mathtt{e}_{\mathtt{A2}});$
$\mathtt{e}_{\mathtt{B1}} := \mathtt{merge}(\mathtt{e}'_2, \mathtt{e}_2);$
$\mathtt{e}'_4 := \mathtt{op}(\mathtt{T}_2(\mathtt{R}^\alpha), \mathtt{e}_{\mathtt{B1}});$
$\mathtt{e}_{\mathtt{B2}} := \mathtt{merge}(\mathtt{e}'_3, \mathtt{e}_3);$
$\mathtt{e}'_5 := \mathtt{op}(\mathtt{T}_2(\mathtt{S}^\alpha), \mathtt{e}_{\mathtt{B2}});$
$\mathtt{e}'_6 := \mathtt{merge}(\mathtt{e}'_4, \mathtt{e}'_5);$
$\mathtt{e}'_7 := \mathtt{op}(\mathtt{T}_\mathtt{s}(\mathtt{R}^\alpha, \mathtt{S}^\alpha), \mathtt{e}'_6);$

(c) Replace fence $\mathtt{e}'_1$ with readers and writers in C

$\mathtt{e}'_2 := \mathtt{op}(\mathtt{T}_1(\mathtt{R}^\alpha), \mathtt{e}_4);$
$\mathtt{e}'_3 := \mathtt{op}(\mathtt{T}_1(\mathtt{S}^\alpha), \mathtt{e}_5);$
$\mathtt{e}'_4 := \mathtt{op}(\mathtt{T}_2(\mathtt{R}^\alpha), \mathtt{e}'_2);$
$\mathtt{e}'_5 := \mathtt{op}(\mathtt{T}_2(\mathtt{S}^\alpha), \mathtt{e}'_3);$
$\mathtt{e}'_6 := \mathtt{merge}(\mathtt{e}'_4, \mathtt{e}'_5);$
$\mathtt{e}'_7 := \mathtt{op}(\mathtt{T}_\mathtt{s}(\mathtt{R}^\alpha, \mathtt{S}^\alpha), \mathtt{e}'_6);$

(d) After optimization

```
// Only in the first replay:
e4 := fence;
e5 := e4;

e'2 := op(T1(R^α), e4);
e'3 := op(T1(S^α), e5);
e'4 := op(T2(R^α), e'2);
e'5 := op(T2(S^α), e'3);
e4 := e'4;
e5 := e'5;

// Only in the last replay:
e'6 := merge(e'4, e'5);
e'7 := op(Ts(R^α, S^α), e'6);
```

(e) Final commands for repeated replays

Figure 4.8: Fence elision for the trace in Figure 4.7

recorded commands in Figure 4.7b once, as in Figure 4.8b. Events that belong to the second trace are renamed to those with a prime, to distinguish them from those in the first trace. Second, dependencies on the fence in the second trace are replaced with

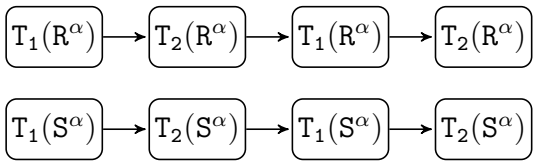


Figure 4.9: Task graph with fence elision

the actual dependencies on operations in the first trace. In the unrolled commands, each operation that belongs to the second trace either immediately or transitively depends on fence $\mathtt{e}_1'$ that blocks operations in the first trace. After we remove that fence, each operation individually waits for dependent operations in the first trace. For each region instance of a task instance, the predecessors from the first trace are identified as follows:

- If the task instance can write to the region instance r, all readers and writers of r are added to the predecessors.
- If the task instance only reads from r, only the writers of r are added to the predecessors.

For example, the original predecessor event $\mathtt{e}_2'$ of task instance $\mathtt{T}_2(\mathtt{R}^\alpha)$ is merged with event $\mathtt{e}_2$, which is the writer of $\mathtt{R}^\alpha$ in the first trace, to get a new predecessor event $\mathtt{e}_{\mathtt{B1}}$ in Figure 4.8c. Once all uses of the fence are replaced with individual events, transitive reduction and copy propagation are applied to the commands. (The result is in 4.8d.) Finally, we generalize the optimized commands to get the final commands in Figure 4.8e for repeated replays. Note that the first two commands and the last two commands are used only in the first and the last replay, respectively. Figure 4.9 shows a task graph from two replays with fence elision.

We can also concatenate two different traces in a similar way when one's postcondition subsumes the precondition of another. However, concatenating two different traces is of less use than unrolling the same trace as the latter appears more frequently in real applications.

4.4 Evaluation

We have implemented dynamic tracing in Legion, a data-centric runtime system for implicit task parallelism [17]. The Legion runtime is a good testbed for dynamic tracing as it faithfully implements the implicitly parallel tasking model described in Chapter 2. Legion has a dependence analysis pipeline similar to the one described in Section 2.4 and builds a task graph using Realm [69], a low-level system for building and executing distributed task graphs. We augment Legion's existing dependence analysis to generate graph calculus programs for traces. Graph calculus is implemented as a set of commands that internally call the Realm API to construct task graphs.

We evaluate dynamic tracing on the five Regent programs used in Chapter 3, which range from small and regular benchmarks to complex irregular applications: Stencil, a 9-point stencil benchmark on 2D grids; Circuit, a circuit simulator for unstructured circuit graphs; PENNANT and MiniAero, proxy applications for unstructured meshes; and Soleil-X, a compressible fluid solver on 3D grids developed to study turbulent fluid flow in channels. All programs were run with *control replication* [65], an optimization that is orthogonal to dynamic tracing. These programs have competitive or better weak scaling performance than reference implementations [65], where reference implementations are available. Table 4.1 shows benchmark metrics, the number of the tasks and copies each node must analyze per iteration. Programs using unstructured meshes have indirect indexing on regions, which require dependencies to be resolved dynamically. For three programs (Stencil, PENNANT, and MiniAero), we compare with publicly available reference MPI versions.

Due to their iterative nature, all five programs have a "main" loop where they spend most of their execution time. For Stencil, Circuit, and PENNANT, we annotate

	Stencil	Circuit	PENNANT	MiniAero	Soleil-X
Number of tasks	16	27	67	288	448
Number of copies	31	49	54	552	928

Table 4.1: Number of tasks and copies per iteration

the body of this main loop. For MiniAero and Soleil-X, which implement a fourth-order Runge-Kutta time marching scheme, we set the annotation on the body of this time marching loop nested within the main loop. Each application has only one trace because there is no change in the task mapping, and dynamic tracing can find one idempotent recording of the trace. Identifying loops that merit annotation was trivial for these programs and could easily be automated.

We use GCC 5.3 to compile the Legion runtime and the MPI reference implementations. Regent uses LLVM for code generation; we use LLVM 3.8. We report performance for each application on up to 256 nodes of the Piz Daint supercomputer [4], a Cray XC50 system; nodes are connected by an Aries interconnect and each node has 64 GB of memory and one Intel Xeon E5-2690 CPU with 12 physical cores.

4.4.1 Runtime Overhead

Before we evaluate dynamic tracing on actual performance of the benchmark programs, we first study the benefit of dynamic tracing on runtime overhead. Runtime overhead places a lower bound on the granularity of tasks that can be handled efficiently, and thus it puts an uppper bound to which programs strong scale; i.e., beyond certain scale, the execution is completely dominated by runtime overhead and the speed-up saturates.

In a first study, we use the synthetic benchmark program in Figure 4.10, which has two desirable properties. First, the program performs no actual computation so we can count all execution time as runtime overhead. Second, the program exhibits a simple pattern of task dependencies, which allows us to compute a bound on the possible improvement from dynamic tracing. Each iteration of the outermost loop launches `N` parallel tasks `S` times where `N` is the number of CPUs remaining after allocating some for the runtime. The tasks form `N` chains of dependent tasks, where the `i`th chain consists of `S` tasks that read and write region `A[i]`. Figure 4.11 illustrates the task graph of the synthetic benchmark program.

We place the tracing annotation on the outer `for` loop (lines 4 and 10) and vary

```
1 task F(x) reads(x),writes(x)
2
3 while *:
4   begin_trace
5   for s =  0, S:
6     for i = 0, N:
7       F(A[i])
8   end_trace
```

Figure 4.10: Synthetic benchmark program

the value of S to study the effect of trace size (S · N) on the reduction of runtime overhead. We also run the program with different numbers of runtime threads to measure the benefit of parallel replay.

Figure 4.12a shows the improvement in the runtime overhead for four configurations of parallel replay. The legend shows the number of runtime threads being allocated for parallel dynamic dependence analysis and trace replay, and also the corresponding value of N. In all four plots, a longer trace leads to a greater improvement in the runtime overhead as it better amortizes the constant overhead of initializing every trace replay.

The plots also show that increasing the number of runtime threads has diminishing returns, which occurs for two reasons. First, dynamic tracing only reduces the runtime overhead for dependence analysis and there are several other steps in Legion's task

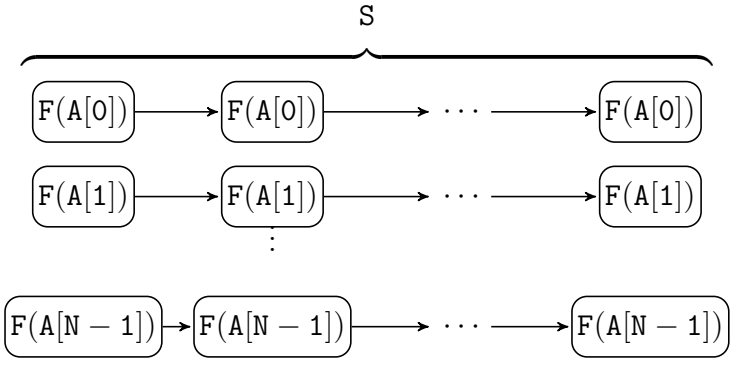
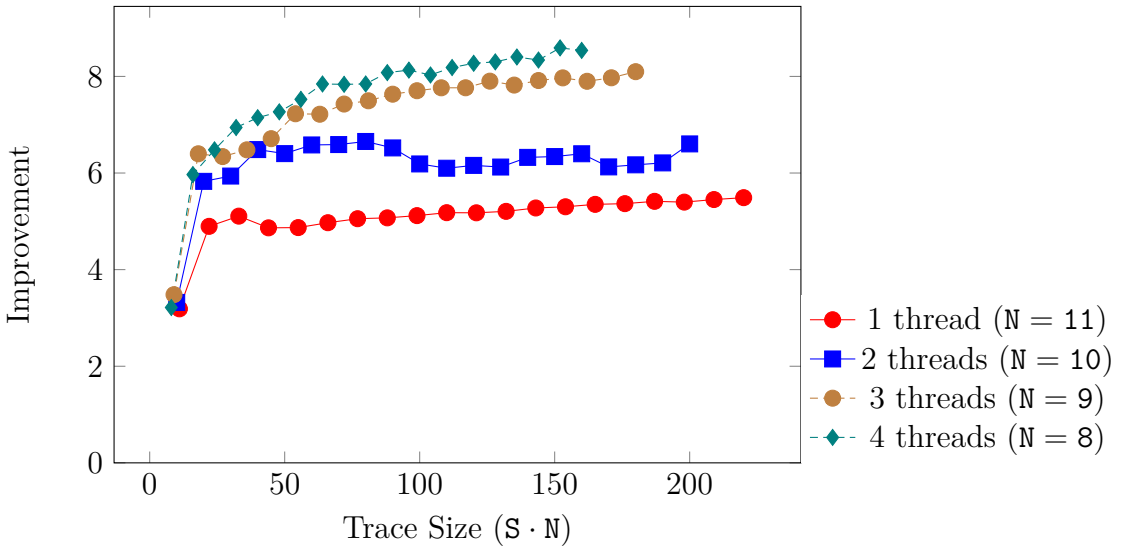
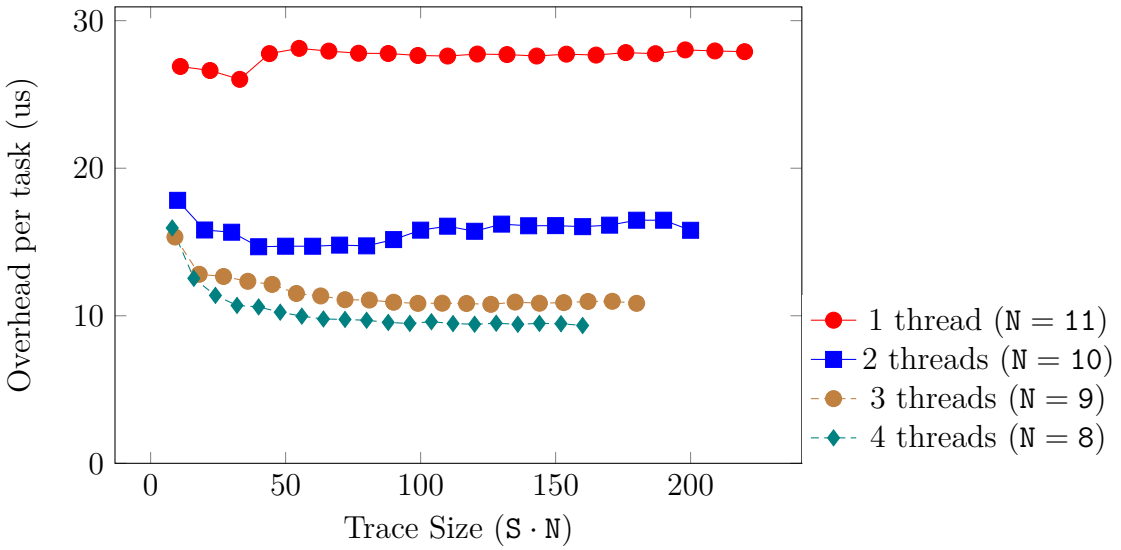


Figure 4.11: Task graph of the program in Figure 4.10



(a) Improvement in the runtime overhead



(b) Average overhead per task with dynamic tracing

Figure 4.12: Runtime overhead of the synthetic benchmark program

processing pipeline. Second, the performance of parallel dependence analysis and trace replay scale sub-linearly in the number of runtime threads, because both parallel dependence analysis and trace replay have portions that run sequentially; Legion performs a sequential preliminary analysis on tasks for parallelizing the subsequent

dependence analysis and dynamic tracing sequentially initializes crossing events for parallel trace replay. To better understand how these two factors incur diminishing returns, we use the following model $O_{\mathsf{dep}}(T)$ of runtime overhead when the number of runtime threads is T:

$$O_{\mathsf{dep}}(T) = C_{\mathsf{dep}} \cdot s(T) + \frac{C_{\mathsf{pipe}}}{T},$$

where C_{dep} denotes the dependence analysis overhead with one runtime thread, C_{pipe} is all the cost of Legion's task processing pipeline except for dependence analysis, and $s(T)$ models the sub-linear speedup governed by Amdahl's law; i.e.,

$$s(T) = \frac{1}{(1-p) + p/T},$$

where p is the proportion of dependence analysis that is parallelized ($0 < p < 1$). In the model, we assume the cost C_{pipe} of Legion's task pipeline except for dependence analysis can be perfectly parallelized across T threads as they are embarrassingly parallel. The model $O_{\mathsf{replay}}(T)$ of the trace replay overhead when the number of runtime threads is T is the same as $O_{\mathsf{dep}}(T)$ except that the dependence analysis overhead is replaced with the parallel trace replay overhead $C_{\mathsf{replay}} \cdot s(T)$:

$$O_{\mathsf{replay}}(T) = C_{\mathsf{replay}} \cdot s(T) + \frac{C_{\mathsf{pipe}}}{T},$$

where C_{replay} denotes the trace replay overhead with one runtime thread. (We use the same $s(T)$ to model the sub-linearity of both parallel dependence analysis and trace replay, to simplify the analysis, though using two different models does not change the result.) The improvement $I(T)$ in runtime overhead is a ratio of $O_{\mathsf{dep}}(T)$ to $O_{\mathsf{replay}}(T)$:

$$I(T) = \frac{O_{\mathsf{dep}}(T)}{O_{\mathsf{replay}}(T)} = \frac{C_{\mathsf{dep}} + C_{\mathsf{pipe}}/(s(T) \cdot T)}{C_{\mathsf{replay}} + C_{\mathsf{pipe}}/(s(T) \cdot T)}.$$

Note that as T increases, $I(T)$ approaches asymptote $I = C_{\mathsf{dep}}/C_{\mathsf{replay}}$; this means that the improvement in the dependence analysis overhead becomes a dominant component in $I(T)$. Finally, the return $R(T) = I(T+1) - I(T)$ of using an additional runtime

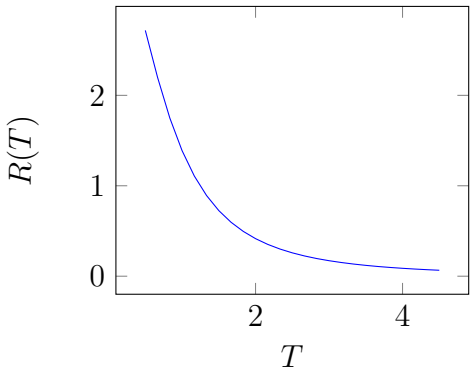


Figure 4.13: Diminishing return function $R(T)$

thread when there are T threads reaches 0 as T goes to infinity (i.e., $\lim_{T\to\infty} R(T) = 0$), which implies that $R(T)$ is diminishing as T increases. The plot of $R(T)$ in Figure 4.13 also clearly shows the trend of diminishing returns. (For the plot, we fit our model to the experimental results by assuming that dependence analysis is $10\times$ heavier than the rest of analysis pipeline, that 90% of parallel dependence analysis and trace replay is perfectly parallelized, and that dynamic tracing eliminates 85% of the dependence analysis overhead; i.e., $10C_{\mathsf{pipe}} = C_{\mathsf{dep}}$, $C_{\mathsf{replay}} = 0.15C_{\mathsf{dep}}$, and $p = 0.9$.)

Figure 4.12b shows the average runtime overhead per task with dynamic tracing. Average overhead per task decreases as trace size increases and eventually saturates once the overhead for initializing trace replay is sufficiently amortized. The plots exhibit a similar trend of diminishing returns as those in Figure 4.12a, but because of Amdahl's law; the $C_{\mathsf{replay}} \cdot s(T)$ term becomes dominant in $O_{\mathsf{replay}}(T)$ as T increases.

Next, we measure the extent to which dynamic tracing reduces the runtime overhead for five benchmark programs. Again, the improvement in runtime overhead only gives an upper bound on the possible improvement in strong scaling performance; the actual strong scaling improvement is influenced by many factors (such as inter-node communication) of which runtime overhead is just one, though it is often the most important one. To isolate the runtime overhead from application work or communication, we apply the same methodology used for the synthetic benchmark: We modify applications to only launch tasks and run no actual computations, and we count their

	Stencil	Circuit	PENNANT	MiniAero	Soleil-X
No Tracing	2.23	10.29	10.47	4.99	19.41
Tracing	0.29	0.53	0.86	0.68	2.26
Improvement	7.6×	19.5×	12.2×	7.4×	8.6×
Trace size	47	76	121	210	344
Trace optimization	0.72	1.70	3.90	1.75	5.86

Table 4.2: Runtime overhead per trace (all in milliseconds)

execution time as runtime overhead. We allocate three runtime threads as we believe this configuration is most cost effective according to our first study with the synthetic benchmark; the same configuration is also used in the strong scaling runs in the next section. Table 4.2 summarizes the measured runtime overhead per trace. In all five programs, dynamic tracing reduces the runtime overhead by more than 7×. Circuit and PENNANT enjoy noticeably greater improvement than the others because they have reduction tasks and copies that make dynamic dependence analysis more expensive. Table 4.2 also shows the one-time cost for trace optimization, which is just a few milliseconds even for the longest trace.

4.4.2 Strong Scaling Performance

In this section, we evaluate dynamic tracing on strong strong scaling performance of the benchmark programs. We demonstrate that with dynamic tracing the benchmark programs strong scale well up to 256 nodes for problem sizes for which they stop scaling at 32 or fewer nodes otherwise. We measured performance when the program reached steady state; i.e., the state where the program starts replaying a recording repeatedly. In all experiments the Legion runtime is configured to use 3 CPUs (out of 12) per node. To study the effect of optimizations for idempotent recordings on performance, we also measure the performance of runs where dynamic tracing is used without those optimizations.

All tables in this section (Tables 4.3-4.8) have three columns: column No Tracing for the baseline performance without tracing, column Tracing for the performance with tracing, and column Tracing(no opt.) showing performance where dynamic tracing is

Nodes	Problem size: 0.4×10^9 cells				
	No Tracing	Tracing(no opt.)	Tracing	MPI	
				9 ranks	12 ranks
1	1.0	1.0	1.0	0.9	1.0
2	2.0	2.0	2.0	1.9	2.1
4	4.1	*4.0*	4.1	3.7	4.1
8	8.1	*8.0*	8.2	7.4	8.2
16	16.0	*15.7*	16.2	14.8	16.3
32	31.3	*30.1*	32.0	29.2	32.4
64	50.1	54.3	61.9	57.9	63.3
128	68.3	108.0	126.3	116.6	134.3
256	**77.2**	**269.7**	**320.0**	**259.8**	**387.5**
$\frac{\textbf{max}(\text{Tracing})}{\textbf{max}(\text{No Tracing})}$	4.2				

Table 4.3: Strong scaling performance of Stencil

used without optimizations. Some tables (Tables 4.3-4.4) have additional columns showing performance of the MPI reference. The tables show throughputs normalized by the baseline performance on a single node (i.e., the 1-node number in No Tracing). Numbers in bold face show the maximum throughput achieved in each configuration; the improvement in strong scaling performance is calculated by dividing this maximum throughput with dynamic tracing to that in the baseline configuration. Underlined numbers in column Tracing(no opt.) mean that the runs performed worse than those without dynamic tracing.

Stencil Table 4.3 shows the strong scaling performance of Stencil. Dynamic tracing improves the speedup of Stencil by 4.2×. Turning off optimizations for idempotent recordings is detrimental to the achievable improvement; it degrades the maximum speedup by 16% on 256 nodes. Furthermore, some runs without the optimizations are slightly slower than the baseline, because jitter in the execution is magnified by spurious task dependencies from fences between replayed traces, which do no exist in the baseline execution.

The MPI version of Stencil 21% is faster than the Legion version (column 12 ranks). This difference is due to the fact that Legion requires resources for its runtime system

Nodes	Problem size: 10^6 cells			
	No Tracing	Tracing(no opt.)	Tracing	MPI 8 ranks
1	1.0	*0.9*	1.0	0.3
2	2.1	*1.8*	2.1	0.6
4	4.2	*3.8*	4.4	1.3
8	8.1	8.7	9.0	3.0
16	14.9	17.3	16.8	7.3
32	21.7	31.3	32.1	16.0
64	**24.0**	54.0	55.0	30.0
128	23.7	90.9	94.8	51.0
256	22.9	**121.4**	**123.4**	**58.2**
$\frac{\textbf{max}(\text{Tracing})}{\textbf{max}(\text{No Tracing})}$	5.1			

Table 4.4: Strong scaling performance of MiniAero

to make dynamic decisions (e.g., about tracing). When the MPI version uses the same number of application processors as Legion (column 9 ranks), it performs worse than the Regent version (by 19%).

MiniAero Table 4.4 shows the strong scaling performance of MiniAero. MiniAero enjoys greater improvement (5.1×) than Stencil as it has a longer trace (210 vs. 47). Turning off optimizations for idempotent recordings degrades the speedup by an average of 6% and a maximum of 15%. Those optimizations are important especially at small node counts because the imbalance in workload of MiniAero's tasks becomes greater on fewer nodes. The MPI reference of MiniAero, which only allows the number of ranks to be a power of 2, starts 3× slower than the Regent version, which is

	Tracing	No Tracing
Number of tasks per processor	36	
Average number of regions per task	3.1	
Average time per iteration	6.6ms	34ms
Average task granularity	183us	940us

Table 4.5: Average task granularity for MiniAero

Nodes	Problem size: 29×10^6 zones				
	No Tracing	Tracing(no opt.)	Tracing	MPI	
				9 ranks	12 ranks
1	1.0	1.0	1.0	0.9	1.2
2	2.2	2.2	2.2	1.9	2.3
4	4.5	*4.3*	4.6	3.8	4.6
8	8.6	*8.4*	9.1	7.6	9.3
16	16.6	*15.3*	16.8	14.9	18.3
32	29.6	*28.4*	30.3	29.2	36.4
64	54.0	55.8	60.8	56.7	71.8
128	**71.4**	106.2	117.6	115.6	139.6
256	68.8	**185.0**	**198.5**	**211.4**	**250.1**
$\frac{\textbf{max}(\text{Tracing})}{\textbf{max}(\text{No Tracing})}$	2.8				

Table 4.6: Strong scaling performance of PENNANT

consistent with [65], and loses scalability earlier.

We can calculate the average task granularity supported by dynamic tracing, as tasks in MiniAero are almost completely overlapped with the runtime overhead and copies. Table 4.5 shows the minimum time per iteration and the number of tasks each processor runs, from which we derive the average task granularity. The average task granularity is 183 microseconds with dynamic tracing, whereas without dynamic tracing the runtime can sustain only tasks that are on average 940 microseconds long.

PENNANT Table 4.6 shows the strong scaling performance of PENNANT. The improvement in strong scaling performance is 2.8×, which is smaller than that of Stencil despite the fact that PENNANT has a longer trace. Unlike the other programs, the main loop in PENNANT is guarded by a convergence predicate that in turn prevents a replay of the trace until the condition is resolved. A trace replay overlaps with tasks only for 25% or less of the time per iteration, which explains an improvement that is (approximately) 4× off of the improvement in the runtime overhead. The use of idempotent recordings improves performance by an average of 6% and a maximum of 11%.

The MPI version of PENNANT is 26% faster than the Legion version (column 12

Nodes	Problem size: 74×10^3 wires		
	No Tracing	Tracing(no opt.)	Tracing
1	1.0	1.0	1.0
2	2.0	2.0	2.0
4	3.9	4.0	3.9
8	7.3	7.5	7.6
16	13.3	13.6	14.0
32	**24.7**	25.5	25.9
64	18.0	48.1	49.7
128	8.3	87.4	90.2
256	4.2	**133.7**	**131.4**
$\frac{\textbf{max}(\text{Tracing})}{\textbf{max}(\text{No Tracing})}$	5.3		

Table 4.7: Strong scaling performance of Circuit

ranks). Again, this difference is due to Legion allocating 25% of the resource (i.e., 3 CPUs out of 12) for its runtime system. When the MPI version uses the same number of CPUs as Legion for application tasks (column 9 ranks), MPI PENNANT is slower than the Regent version up to 128 nodes and becomes 6% better on 256 nodes.

Circuit Table 4.7 shows the strong scaling performance of Circuit. The improvement in Circuit is greater (5.3×) than in MiniAero even though MiniAero's trace is longer, because Circuit has greater runtime overhead than MiniAero (Table 4.2) due to its temporary reduction buffers, which are reused in replayed traces. Circuit also shows the biggest discrepancy between the improvement in the runtime overhead and strong scaling simply because the runs did not reach a point where they are limited by the replay overhead. Finally, Circuit is immune to the absence of optimizations for idempotent traces, because Circuit has all-to-all dependencies between tasks on each node, which results in sightly longer sequences of graph calculus commands after fence elision.

Soleil-X Table 4.8 shows the strong scaling performance of Soleil-X. The improvement is greatest (7.0×) among all five benchmark programs, because Soleil-X has the longest trace. The use of idempotent recordings brings additional improvement of an

Nodes	Problem size: 8.4×10^6 cells		
	No Tracing	Tracing(no opt.)	Tracing
1	1.0	1.0	1.0
2	2.0	2.0	2.0
4	3.3	3.8	3.9
8	5.3	6.9	7.2
16	7.6	11.9	12.7
32	9.0	17.9	18.7
64	10.2	31.5	32.4
128	**10.6**	53.0	54.8
256	5.7	**69.5**	**74.0**
$\frac{\textbf{max}(\text{Tracing})}{\textbf{max}(\text{No Tracing})}$	7.0		

Table 4.8: Strong scaling performance of Soleil-X

average of 4% and a maximum of 7%.

Like MiniAero, tasks in Soleil-X are almost completely overlapped with the runtime overhead and copies, and thus we can calculate the average task granularity supported by dynamic tracing. Table 4.9 shows the average task granularity of Soleil-X; dynamic tracing can support tasks of 413 microseconds on average. Note that the task granularity for Soleil-X is twice that for MiniAero because Soleil-X has roughly twice as many regions per task, leading to twice as many copies on average to replay per task (5.4 regions per task on average vs. 3.1).

	Soleil-X	
	Tracing	No Tracing
Number of tasks per processor	56	
Average number of regions per task	5.4	
Average time per iteration	23ms	161ms
Avergae task granularity	413us	2,879us

Table 4.9: Average task granularity for Soleil-X

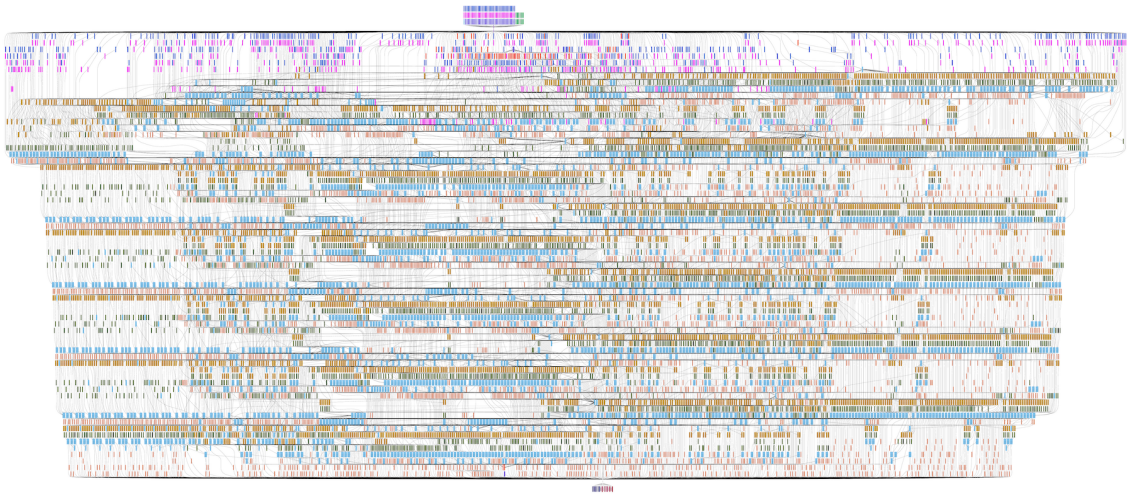
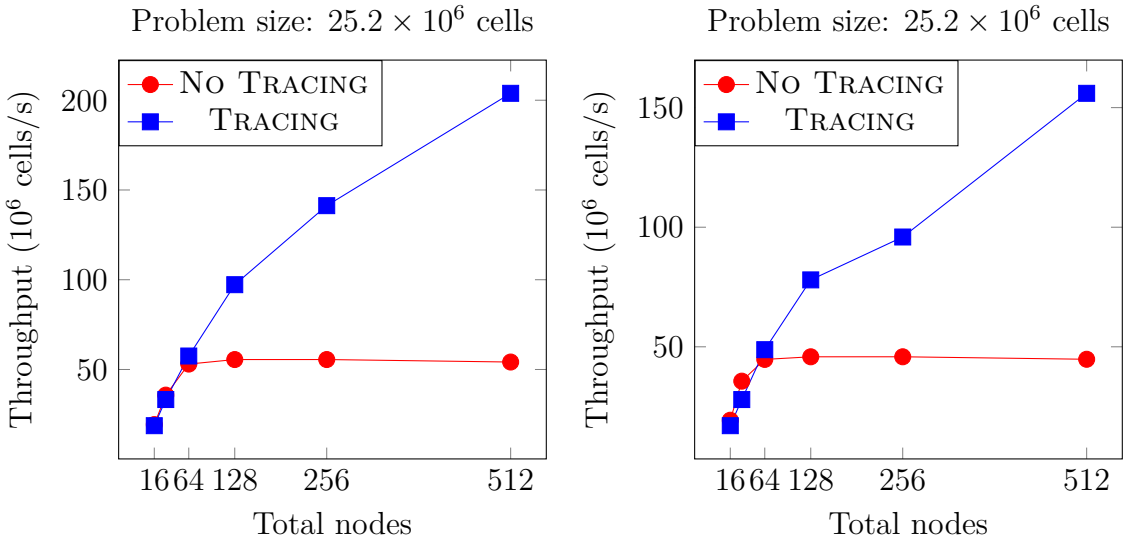


Figure 4.14: Task graph of S3D-Legion simulating n-dodecane for a single time step

4.4.3 S3D-Legion

We also evaluate dynamic tracing on S3D-Legion [70], an exascale combustion simulation package using the Legion runtime system. S3D-Legion is designed to perform high-fidelity reactive flow direct numerical simulation (DNS) of turbulent combustion. As the DNS problem that S3D-Legion solves is computationally intensive, S3D-Legion makes heavy use of GPUs and employs a custom built DSL compiler (Singe [16]) for generating optimized GPU kernels for different chemical species. Legion plays an essential role in identifying and enforcing complex dependencies between those GPU tasks and overlapping the execution with necessary data transfers to maximize the throughput.

S3D-Legion is a real-world example illustrating why we need specialization within a dynamic runtime system. Figure 4.14 shows the task graph of S3D-Legion for a single time step execution of the n-dodecane (a partially pre-mixed diesel fuel surrogate) simulation. The task graph has six horizontal clusters of nodes, which is consistent with the fact that S3D-Legion uses a six-stage Runge-Kutta marching scheme [46]. The dependence pattern in this graph is highly complex and irregular, and therefore Legion's dynamic dependence analysis is key to capturing all dependencies both soundly and precisely. However, despite the complexity and irregularity, tasks in



(a) Periodic boundary condition (b) Non-periodic boundary condition

Figure 4.15: Strong scaling performance of S3D-Legion

S3D-Legion form only three distinct traces of tasks: two traces that include tasks for periodic checkpointing, and another one that runs for the majority of execution time. Hence, dynamic tracing still can reduce the runtime overhead involved in identifying the complex dependence pattern by specializing those traces.

We performed strong scaling experiments on Piz Daint. We ran an n-dodecane simulation for the problem size of 25.2×10^6 cells in total. The performance was measured once S3D-Legion reached a steady state. Figure 4.15 shows the strong scaling performance of S3D-Legion from 16 nodes to 512 nodes. For the problem size we chose, the runs are limited by the runtime overhead at 64 nodes without dynamic tracing. With dynamic tracing, the performance scales further up to 512 nodes and the strong scaling performance is improved by 3.7× for a periodic boundary condition and 3.4× for a non-periodic boundary condition. The average task granularity that the runtime can support becomes as short as 200 microseconds. Note that performance is slightly worse with the non-periodic boundary condition due to some reductions that cannot be overlapped with the computation.

Chapter 5

Related Work

Research on automatic program parallelization has a rich history. We discuss prior efforts that are closely related to our hybrid approach.

5.1 Composable and Configurable Parallelization

High Performance Fortran (HPF) [47] and a family of related systems [25, 42] are early efforts that recognized the problem of composability and configurability in auto-parallelization. These systems address the problem with a solution similar to ours: they provide control over data partitioning via *data distributions*, an annotation language for describing primary data partitions. A data distribution determines each *rank's* ownership of the program data. (The coarsest level of parallelization in a program is often referred to as a *rank*. In most applications a program is parallelized so that there is one rank per machine node.) The compiler then infers non-local data accesses in each rank (i.e., accesses to the data that is not owned by the rank), based on the data distribution, and inserts communication and synchronization to preserve sequential semantics of the program. When programs have subroutines that share data with their caller, those subroutines can declare what data distribution they expect. In cases where the caller's distribution is different from the callee's, the compiler is responsible for the data transposition.

The biggest drawback of these systems is that data distributions are "not themselves data objects" [47], which limit composability. Unlike the partitioning constraints in our work, which are *descriptive*, data distributions are *prescriptive*; i.e., they are compiler directives requesting data partitions of a particular shape. Because of this nature, they cannot specify a subroutine parametric in the caller's data distribution. A special kind of data distribution that *inherits* the caller's data distribution was proposed to addressed this issue, but no compiler has ever fully supported it because of the implementation complexity [47].

Another drawback is that data distributions provide little control over data partitioning. Data distributions only describe the ownership of data in each rank and the implementation of non-local data accesses is internal to the compiler. Hence, even when the programmer knows that his performance issue is due to inefficient non-local data accesses, he cannot fix the issue because the fix cannot be expressed by data distributions, the only tunable knob available to the programmer.

The key insight of our work is that first-class data partitions, which were then considered infeasible due to the runtime overhead, can elegantly solve issues with which the compiler-based systems like HPF would struggle. In our performance evaluation, we demonstrate that fine-grained control over the data partitioning process via partitioning constraints facilitates composability and performance tuning.

Halide [32, 59] and Tiramisu [12] are DSL compilers with a programmable interface to configure performance-sensitive parallelization parameters. Instead of automatically choosing a parallelization strategy for a given program, these compilers use a sub-language of *schedules*, with which the program can describe its parallelization strategy. Scheduling commands include directives for code transformations, such as tiling and reordering, which are choices orthogonal to data partitioning but could be incorporated in our auto-parallelizer. By design, the parallelization in these DSL compilers is semi-automatic because programs must provide a schedule (although automatic scheduling has been presented for shared-memory Halide programs [55]), whereas interface constraints are optional in our data partitioning approach that is otherwise fully automated. One issue with Tiramisu's design is that scheduling commands in Tiramisu also specify data movement and synchronization (e.g., `send` and

`receive` for inter-node communication), which makes them no longer a strictly performance decision but can also affect correctness; programmers must guarantee that their scheduling commands are semantically correct. On the other hand, external constraints in our approach cannot compromise the correctness of data movement and synchronization inserted by the runtime system.

Another example of a compiler with tunable performance is Sequoia [36, 44, 62]. Similar to our programming model, Sequoia has a *mapping* phase where the compiler statically assigns tasks to processors and data collections to memories. Programmers can control this mapping phase by providing mapping decisions for their programs. Besides mapping decisions, programmers can also specify *tunable parameters*, such as a tile size of a loop, which affect the performance of their programs. Mapping decisions and tunable parameters in Sequoia can be changed independently of programs and even tuned automatically by some search process [62]. Although Legion, the tasking system used in our evaluation, inherits this capability from Sequoia, performance optimization through mapping decisions and tunable parameters is not pursued in our work as it is an optimization orthogonal to the program auto-parallelization.

5.2 Distributed Code Generation for Affine Programs

Because of their ubiquity and importance in high performance computing, affine programs have been one of the major targets for the auto-parallelization on distributed memory machines; distributed code generation for affine programs indeed has been studied in great depth over the past decades [7, 9, 21, 26, 29, 37, 47, 60]. An important property of affine programs is that the working sets of their data accesses can be expressed by *polyhedra*, which makes dependence analysis and communication inference feasible at compile time, using well known methods [28, 58].

A major challenge in distributed code generation for affine programs is to generate efficient communication code. Previous approaches [7, 9, 21, 26, 29] aim at minimizing the volume of inter-processor communication, but they are still sub-optimal because

the exact volume cannot be computed without knowing the processor count, the dimension of data, and the exact mapping of tasks to processors, all of which are unavailable at compile time. Furthermore, an optimal placement of communication that maximizes the overlap between communication and computation is even harder to determine at compile time, because the execution time of both is unknown. Our hybrid approach is free of these issues because the communication is resolved by the runtime system; the runtime system can accurately compute the volume of communication and identify exactly when and to where the data must be sent.

Although we use an optimization specific to affine data accesses in Section 3.4.1, our constraint-based framework is agnostic to whether data accesses in the program are affine or irregular. Further optimizations based on the convexity of affine data accesses could be easily incorporated in the implementation of data partitions and DPL operators with no fundamental changes to our framework.

5.3 Inspector/Executor Frameworks

Irregular accesses are a major challenge for auto-parallelizing compilers because their working sets cannot be resolved at compile time. The state-of-the-art technique for handling irregular accesses is the Inspector/Executor (I/E) framework [15, 50, 60, 61, 74]. The I/E framework defers the analysis of irregular accesses to a runtime *inspector*, which records working sets of those irregular accesses during execution and fetches non-local data from remote nodes if necessary. The recorded working sets are consumed by an *executor*, the original code modified to operate on those working sets.

Although the I/E techniques can be effective, they often produce the code that is hard to understand and compose; these techniques commonly meta-program the inspector code that tracks working sets in some custom data structure, and the decisions and heuristics made in this meta-programmed code are neither transparent to nor configurable by programmers. An important observation in this dissertation is that a programming language can natively express inspectors using first-class data partitions and partitioning operators, which can address the issues of the I/E framework. In our approach, a data partition expresses a working set of an irregular access

and a synthesized DPL program constructs this working set at runtime, with the help from runtime system to fetch remote data for the partition As we demonstrated in Section 3.5, this native support enables configurable and composable auto-parallelization even for programs with irregular accesses.

The sparse polyhedral framework [67], which is used as a foundation for the I/E framework, is similar in spirit to our approach; using sparse polyhedrons as a high-level abstraction, the framework facilitates the composition of auto-generated inspector code. Sparse polyhedrons are useful for applying compiler transformations to the inspector code as they precisely capture the shape of working sets. However, sparse polyhedrons are still internal to the compiler and thus cannot be used as an interface for mixing the inspector code with the manually parallelized code, whereas partitions and constraints in our approach are a user-facing interface for configuring the auto-parallelization process. We believe that the two approaches are complementary to each other.

5.4 Languages with Data Parallelism

ZPL [66] is an array-based programming language for implicit parallelism. ZPL programs have a global view of arrays and each statement accesses arrays in a vector form, which naturally reveals data parallelism within the statement. To express diverse access patterns in practice, ZPL provides *regions*, an abstraction that describes a subset of a global array, and operators to define regions. With regions and vector statements, one can succinctly write a complex array-based program for distributed memory systems with no explicit parallelization. However, irregular applications with dynamic behavior, which can be easily expressed and efficiently executed in our hybrid approach via task parallelism, are not amenable to ZPL's array-based programming style. Chapel [23] overcomes this limitation by taking a multi-resolution approach providing a suite of multiple parallelization paradigms, which is in spirit similar to our hybrid approach. However, Chapel's task parallelism falls short as it requires explicit parallelization with manual synchronization and communication that is not efficiently composed with its support for implicit data parallelism.

As we described in Section 3.6, our automatic data partitioning is motivated by the shortcomings of Liszt [20, 33], a DSL for mesh-based PDE solvers. The syntactic definition of parallelizable loops is similar to the *single-phase* rule in Lizst, which requires kernels to have only one *phase* of access to each data field. Liszt achieves an end-to-end auto-parallelization on heterogeneous, distributed memory machines [33], but provides no mechanism for configuration or composition.

5.5 Constraint-Based Program Analysis

Many program analysis problems can be reduced to constraint solving problems for which off-the-shelf solvers exist [14,31,35,73]. The theory of first-class data partitions, however, is not a standard theory, nor is it obvious how to convert it into one, because the solutions of our constraints are functional programs with a special set of function primitives (the DPL operators). One of our contributions is the formalization of automatic parallelization as a space of possible data partitionings captured by a system of constraints, together with an algorithm for resolving those constraints.

5.6 Efficient Task Graph Representations

Dynamic tracing is a first step towards efficient tasking in implicitly parallel tasking systems; any tasking systems [10, 17, 30, 43] that depend on dynamic dependence analysis to convert implicit parallelism to a task graph, an explicit representation of task parallelism, can benefit from our dynamic tracing technique. Hoque et al. [43] have also reported that implicitly parallel tasking systems require a larger granularity of tasks than explicitly parallel programs to be efficient because of runtime overhead, which implies that dynamic tracing could greatly reduce this gap between the two paradigms.

Traces that are specialized in dynamic tracing can be further optimized once a tasking system provides a more efficient representation of a task graph; the current design of dynamic tracing requires task graphs to be reconstructed by executing graph calculus commands, but this overhead can be further reduced by compiling

those commands to a direct task graph representation. Execution templates [53] are one such example. An execution template is an explicitly parallel representation for which the runtime system provides efficient distributed execution via a constant time instantiation of the whole template on each node (it still requires messages linear in the number of nodes). Execution templates require each command to specify a *before set*, i.e., a set of previous commands for which the command must wait, the information that a specialized trace already possesses. We believe that dynamic tracing and execution template are complementary to each other.

PTG (Parameterized Task Graph) [22,27] is another task graph representation for efficient tasking. A PTG expresses multiple task graphs with a single parameterized task graph; each value of the parameter leads to a distinct task graph instance of the PTG. Instances of a PTG explicitly enumerate each task's dependencies on other tasks. To express irregular task dependencies, PTGs provide dependencies predicated by conditions on parameter values. Because task dependencies are already materialized in PTGs, instances of PTGs require no additional runtime analysis and are ready for direct execution. PTGs are appealing as they are a constant-size description of multiple task graphs, and summarizing similar subgraphs in a trace's task graph into a PTG would be beneficial for the space usage. However, the fact that PTGs cannot express task dependencies that are determined by program data limits their applicability to irregular applications, for which dynamic dependence analysis in implicitly parallel tasking systems is most useful.

TensorFlow [6] takes an interesting approach to tasking that is slightly different from other tasking systems for implicit parallelism; task graphs in TensorFlow contain nodes for describing control flow [75], such as while loops and switch statements, whereas other tasking systems piggy back on the host language's constructs to resolve a program's control flow and then construct a DAG of tasks. Embedding control flow in task graphs allows a single task graph having a while loop to concisely represent task graphs from all the loop iterations, which eliminates the overhead of repeated instantiations for those graphs. Currently, dynamic tracing accepts only sequences of tasks with no control flow, and extending it to accept CFGs of tasks would enjoy similar benefits.

5.7 JIT Compilers

The idea of JIT specialization was first popularized for interpreted implementations of languages. Some languages use an interpreter to achieve portability in the low-level program representation, such as the Java byte code, and other languages, such as Python and JavaScript, have a dynamic type system, which in general prevents an ahead-of-time compilation to efficient binary code due to the absence of type information. JIT compilers for these languages eliminate overhead of the interpreter by specializing traces of code to platform specific binaries, similar to dynamic tracing that reduces the dependence analysis overhead in a tasking system by specializing traces of tasks.

Work on JIT compilers [13, 18, 19, 24, 34, 39] devotes considerable effort to extracting traces of a meaningful size during program execution. Traces are found incrementally across different control blocks and stitched together to comprise a longer trace that potentially has more optimization opportunities. Once traces are discovered, compiling them and running their binary code in a shared memory environment is well understood. In contrast, dynamic tracing relies on traces being specified, and instead focuses on describing how to capture the resulting task graphs and soundly replay them. A major difference is that dynamic tracing also must deal with the additional complexity of operating correctly in distributed memory environments, ensuring that data is correctly copied and placed in the machine.

5.8 Memoization for Stateful Algorithms

Avoiding an expensive re-computation using memoization is a well-known practice. Although memoization for pure functions can be done trivially by tabularizing input-output pairs, one must track changes in the state to correctly memoize stateful algorithms. To soundly memoize the dynamic dependence analysis algorithm, dynamic tracing checks the precondition for safe replay and applies the postcondition, a summary of the changes in valid region instances.

FastSim [63], a simulator for an out-of-order processor, employs a memoization

similar to dynamic tracing; the simulator records the simulator actions, such as measuring cycles for a given instruction or estimating cycles for a memory operation, for a given microarchitecture configuration and replays the recorded actions, called *fast forwarding*, when the same configuration reappears. To simplify memoization, the fast-forwarding does not check the simulator's cache state, which is highly variable, and it instead aborts the replay and fall backs to a detailed simulation whenever a memory operation in a replay exhibits different behavior due to a cache inconsistency. However, this approach would not be profitable for dynamic tracing because purging a (partially) constructed task graph is much more expensive than precisely tracking and altering the validity of region instances for a correct memoization.

Chapter 6

Conclusion

This dissertation has presented a hybrid approach to automatic program parallelization. In the first stage of our approach, an auto-parallelizer converts data parallelism in a program into task parallelism. To enable the seamless composition of an auto-parallelized part and the rest of the code parallelized manually, the auto-parallelizer allows programmers to guide the automated process by specifying constraints that describe the interface between those parts. In the next stage, task parallelism in the program from the previous stage is fulfilled by a tasking system that automatically identifies and realizes opportunities for parallel task execution. The efficiency of this tasking system is improved by dynamic tracing, a technique that reduces runtime overhead by eliminating redundancy in the runtime analysis for recurring traces of tasks. In sum, our hybrid approach maximizes the benefit of automatic parallelization and yet allows implicitly parallel programs to have performance comparable to those that are parallelized with explicit synchronization and communication.

The auto-parallelizer designed in this dissertation only handles data parallel loops that have independent iterations. One possible extension is support for wavefront parallelism, i.e., loops whose iterations have dependencies only on some of the other iterations. An obvious strategy would be to fall back to a full-blown polyhedral analysis, but a more intriguing approach is to cast it as a language design problem: How can we design language primitives so that the dependence structure is surfaced syntactically? We believe that well-designed abstractions would greatly simplify the

auto-parallelizer's work to "guess" the programmer's intention and also allow the dependencies to be captured easily by partitioning constraints.

The cost model of the constraint solver is another avenue of improvement. The current constraint solver simply uses the number of data partitions constructed by the synthesized DPL code as its cost model, which it attempts to minimize. However, minimizing the number of partitions is not necessarily an optimal strategy and there are cases where having more partitions is beneficial. An interesting question is then how we can quantify this extra benefit and whether we can capture it in the form of constraints so that it is compatible with the rest of the reasoning process. Some of the reasoning might need runtime information for precise results, in which case the compiler's role would be to minimize the amount of such deferred reasoning to runtime.

Eliminating the overhead of dependence analysis is an important use of recurrent traces but not the only one. The current implementation of dynamic tracing still repeatedly constructs task graphs for traces using Realm API calls whose overhead could be further reduced. One possible optimization is to construct task graphs at a larger granularity; instead of running graph calculus commands in an interpretive manner, the commands can be compiled to an efficient internal representation for task graphs, which would be instantiated in bulk upon requests from dynamic tracing. This internal representation would better amortize the overhead of instantiation and can employ a more efficient mechanism to enforce task dependencies than the default one used in the general case.

An important message in this dissertation is that program optimization techniques must be studied in the context of realistic programming systems and not simply pursued in isolation. Components in software become diverse as the software grows and one cannot expect a particular optimization to apply to an entire program. Finding the right interface between components to which an optimization is applicable and those components that are out of scope for the optimization becomes paramount to making that optimization usable in practice. In this dissertation, we have addressed two instances of this interface problem: A constraint-based approach to composable

data partitioning, which enables the composition of programs parallelized and optimized by different means, and a tracing technique that correctly mixes two modes of task graph construction, one that efficiently builds the graph by running a graph constructing program and another that incrementally discovers dependence edges for a node via runtime analysis. The problem of designing such composable interfaces for program optimization is, we believe, one of the most important for the design of practical compilation-based systems.

Bibliography

[1] CUDA programming guide 9.1. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Sept. 2013.

[2] Predictive science academic alliance program (psaap) ii. https://exascale.stanford.edu/, 2013.

[3] The Open Community Runtime interface. https://xstackwiki.modelado.org/images/1/13/Ocr-v0.9-spec.pdf, 2014.

[4] Piz Daint & Piz Dora - CSCS. http://www.cscs.ch/computers/piz_daint, 2018.

[5] Sierra Supercomputer - LLNL. https://computing.llnl.gov/computers/sierra, 2018.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 265–283, 2016.

[7] Vikram S. Adve and John M. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 186–198, 1998.

[8] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2):79–111, 1999.

[9] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 126–138, 1993.

[10] C. Augonnet et al. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, February 2011.

[11] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[12] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 193–205, 2019.

[13] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, 2000.

[14] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, 2011.

[15] Ayon Basumallik and Rudolf Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 119–128. ACM, 2006.

[16] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: leveraging warp specialization for high performance on gpus. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 119–130, 2014.

[17] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 66, 2012.

[18] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: A trace-based jit compiler for cil. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, 2010.

[19] Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, 2010.

[20] 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 Trans. Graph.*, 35(2):21:1–21:12, 2016.

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

[22] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011.

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

[24] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: Trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, 2009.

[25] Barbara M. Chapman, Piyush Mehrotra, and Hans P. Zima. Programming in vienna fortran. *Scientific Programming*, 1(1):31–50, 1992.

[26] Michael Classen and Martin Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, 2006.

[27] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. Ptg: An abstraction for unhindered parallelism. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 21–30, Piscataway, NJ, USA, 2014. IEEE Press.

[28] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual with application to integer programming. In B. Roy, editor, *Combinatorial Programming: Methods and Applications*, pages 93–102, 1975.

[29] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 375–386, 2013.

[30] J. Davison de St.Germain, J. McCorquodale, S.G. Parker, and C.R. Johnson. Uintah: a massively parallel problem solving environment. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 33–41, 2000.

[31] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

[32] Tyler Denniston, Shoaib Kamil, and Saman P. Amarasinghe. Distributed halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 5:1–5:12, 2016.

[33] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, 2011.

[34] Stefano Dissegna, Francesco Logozzo, and Francesco Ranzato. Tracing compilation by abstract interpretation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, 2014.

[35] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*.

[36] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*, page 83, 2006.

[37] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, 1996.

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

[39] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[40] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[41] 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.

[42] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling fortran D for MIMD distributed memory machines. *Commun. ACM*, 35(8):66–80, 1992.

[43] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, 2017.

[44] Mike Houston, Ji Young Park, Manman Ren, Timothy J. Knight, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 143–152, 2008.

[45] Lluís Jofre, Gianluca Geraci, Hillary Fairbanks, Alireza Doostan, and Gianluca Iaccarino. Multi-fidelity uncertainty quantification of irradiated particle-laden turbulence. *CTR Annual Research Briefs*, pages 21–34, 11 2017.

[46] Christopher A. Kennedy, Mark H. Carpenter, and R. Michael Lewis. Low-storage, explicit runge-kutta schemes for the compressible navier-stokes equations. *Appl. Numer. Math.*, 35(3):177–219, November 2000.

[47] 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.

[48] The OpenCL Specification, Version 1.0. The Khronos OpenCL Working Group, December 2008.

[49] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):440–451, 1991.

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

[51] Wonchan Lee, Manolis Papadakis, Elliott Slaughter, and Alex Aiken. A constraint-based approach to automatic data partitioning for distributed memory execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019 (to appear)*, 2019.

[52] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. Dynamic tracing: Memoization of task graphs for dynamic task-based runtimes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2018*, 2018.

[53] 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.

[54] Michael F Modest. Chapter 17 - the method of discrete ordinates (sn-approximation). In Michael F Modest, editor, *Radiative Heat Transfer (Third Edition)*, pages 541 – 584. Academic Press, Boston, third edition edition, 2013.

[55] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, 2016.

[56] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with starss. *IJHPCA*, 23(3):284–299, 2009.

[57] Hadi Pouransari and Ali Mani. Effects of Preferential Concentration on Heat Transfer in Particle-Based Solar Receivers. *Journal of Solar Energy Engineering*, 139(2), 11 2016. 021008.

[58] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, pages 4–13, 1991.

[59] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, 2013.

[60] 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.

[61] 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.

[62] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, pages 280–291, 2008.

[63] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998.*, pages 283–294, 1998.

[64] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015*, pages 81:1–81:12, 2015.

[65] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. Control replication: Compiling implicit parallelism to efficient spmd with logical regions. In *Proceedings*

of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, 2017.

[66] Lawrence Snyder. The design and development of zpl. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 8–1–8–37, 2007.

[67] Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018.

[68] Hilario Torres, Manolis Papadakis, Lluís Jofre, Wonchan Lee, Alex Aiken, and Gianluca Iaccarin. Soleil-x: Turbulence, particles, and radiation in the regent programming language. In *Proceedings of the IEEE/ACM Parallel Applications Workshop, Alternatives To MPI, PAW-ATM 2019 (to appear)*, 2019.

[69] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: an event-based low-level runtime for distributed memory architectures. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 263–276, 2014.

[70] Sean Treichler, Michael Bauer, Ankit Bhagatwala, Giulio Borghesi, Ramanan Sankaran, Hemanth Kolla, Patrick S McCormick, Elliott Slaughter, Wonchan Lee, Alex Aiken, et al. S3d-legion: An exascale software for direct numerical simulation of turbulent combustion with complex multicomponent chemistry. In *Exascale Scientific Applications*, pages 257–278. Chapman and Hall/CRC, 2017.

[71] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 344–358, 2016.

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

[73] Sven Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, 2010.

[74] Reinhard von Hanxleden, Ken Kennedy, Charles Koelbel, Raja Das, and Joel H. Saltz. Compiler analysis for irregular problems in fortran D. In *Languages and Compilers for Parallel Computing, 5th International Workshop, New Haven, Connecticut, USA, August 3-5, 1992, Proceedings*, pages 97–111, 1992.

[75] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Gordon Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 18:1–18:15, 2018.

[76] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, 2012.