Copyright
by
Dimitrios Prountzos
2015

The Dissertation Committee for Dimitrios Prountzos
certifies that this is the approved version of the following dissertation:

Elixir: Synthesis of Parallel Irregular Algorithms

Committee:

Keshav Pingali, Supervisor

Jayadev Misra

Mooly Sagiv

Sumit Gulwani

William Cook

Don Batory

Elixir: Synthesis of Parallel Irregular Algorithms

by

Dimitrios Prountzos, B.E.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2015

Acknowledgments

I wish to thank my advisor Keshav Pingali for his support and mentorship. His relentless enthusiasm for research has always been a great source of inspiration. His method of work, his insistence on excellence, and his passion for precision and clarity have tremendously helped me grow as a scientist and will guide me in the rest of my professional career.

I would also like to thank Jayadev Misra, Don Batory, William Cook, Mooly Sagiv, and Sumit Gulwani for serving on my committee. Their feedback greatly improved the quality of my work. I am grateful to Mooly Sagiv for the opportunity to visit his group. I am also thankful to Kathryn McKinley for her mentorship during my first few years at UT.

A special thanks goes to my friend and long-term collaborator Roman Manevich. Roman has taught me a lot about formal methods and has given me great all-around advice over the years. Additionally, I am grateful to Roman and Hamotal for their hospitality both in Austin and during my visit to Israel.

The Galois group has been a home throughout the years to a number of unique characters sharing a passion and common vision for advancing parallel programming. I would like to thank Milind Kulkarni, Mario Mendez-Lojo, Rashid Kaleem, Xin Sui, Muhammad Amber Hassaan, Donald Nguyen, Andrew Lenharth, Sreepathi Pai, as well as all the other past and present

members of the lab, for all the intellectually stimulating discussions and all the great work we achieved together. I would also like to thank Julie Heiland for her help and guidance.

I would like to thank my friends Xin Sui, Vasilis Liaskovitis, Hemant Mohapatra, Dimitris Kaseridis, Thekla Boutsika, Yannis Rouselakis, Christos Argyropoulos, Khubaib, and Dimitris Sounas for the good times we had and for making graduate school so much more enjoyable. A very special thanks to my friends Kostas Menychtas and Kostas Karantasis for their support and for endless long-distance calls and philosophical discussions.

I would also like to thank my family for always being there for me and for encouraging me to follow my dreams. Finally, I would like to thank Riitta-Ilona for her unwavering love and companionship throughout the graduate school adventure and for all the adventures that are yet to come.

Elixir: Synthesis of Parallel Irregular Algorithms

Publication No. _____

Dimitrios Prountzos, Ph.D.
The University of Texas at Austin, 2015

Supervisor: Keshav Pingali

Algorithms in new application areas like machine learning and data analytics usually operate on unstructured sparse graphs. Writing efficient parallel code to implement these algorithms is very challenging for a number of reasons.

First, there may be many algorithms to solve a problem and each algorithm may have many implementations. Second, synchronization, which is necessary for correct parallel execution, introduces potential problems such as data-races and deadlocks. These issues interact in subtle ways, making the best solution dependent both on the parallel platform and on properties of the input graph. Consequently, implementing and selecting the best parallel solution can be a daunting task for non-experts, since we have few performance models for predicting the performance of parallel sparse graph programs on parallel hardware.

This dissertation presents a synthesis methodology and a system, Elixir, that addresses these problems by (i) allowing programmers to specify solutions at a high level of abstraction, and (ii) generating many parallel implementations automatically and using search to find the best one. An Elixir specification consists of a set of operators capturing the main algorithm logic and a schedule specifying how to efficiently apply the operators. Elixir employs sophisticated automated reasoning to merge these two components, and uses techniques based on automated planning to insert synchronization and synthesize efficient parallel code.

Experimental evaluation of our approach demonstrates that the performance of the Elixir generated code is competitive to, and can even outperform, hand-optimized code written by expert programmers for many interesting graph benchmarks.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Irregular Algorithms: A Challenge for Parallel Programming

Our world relies increasingly on computers to handle our communications, organize our economy, and enable advances in many scientific disciplines. Parallelism is one of the key ingredients that make computers effective tools that drive innovation in all these sectors. Consequently, the problem of efficient parallel computation has been the subject of extensive study. Yet, despite various notable advances, the problem of how to exploit parallelism easily is far from settled. In fact, due to the growing need for more widespread use of parallelism, the field of parallel computing is evolving across a number of different dimensions. These include *the problems* that we focus on parallelizing, *the people* involved with parallel programming, and *the architectures* that parallel computing is performed on.

Three decades ago parallel programming was primarily performed on dedicated supercomputers, and the focus was on problems derived from various computational science disciplines. These domains include problems like dense linear algebra, FFT and finite-differences, in which the key data structures are vectors and dense matrices accessed in statically known patterns, such as

by rows or blocks. The programmers involved in this process were primarily computational scientists, who were experts on both the domain in question and parallel programming, and whose goal was to extract every last bit of performance out of these complex parallel supercomputers. Consequently, the parallel programming community has acquired a deep understanding of the patterns of parallelism and locality for the above *regular algorithms*. These insights have led to new languages and tools that make it easier to develop parallel implementations of such algorithms.

However, outside of computational science most algorithms are *irregular*. Irregular algorithms access complex data structures such as *unstructured sparse graphs* and *sets* in ways that are not predictable at compile-time. These sparse graph computations play a central role in important emerging application areas like web search and machine learning on big data. For example, web search is based on an algorithm called page-rank computation that operates on the web graph, which is a graph in which nodes represent web-pages and edges represent hyper-links between web-pages [84]. Many recommender systems are based on inference on a sparse bipartite graph in which one set of nodes represents consumers and the other set of nodes represents items for sale [155, 19]. Because of the enormous size of the data sets and the need for rapid responses to queries, these kinds of sparse graph computations must be performed in parallel. In addition to the new problem domains, we also have a much more diverse set of parallel architectures than before. From high-end servers that support cloud computing in data-centers to our laptops, effectively

all machines today are parallel computers. In fact, even our cell phones employ multicore and heterogeneous processors, such as GPUs, in order to improve performance. This proliferation of diverse parallel architectures introduces a much more diverse group of self-trained programmers and data-scientists to the problem of parallelism. Since such programmers do not necessarily have the training nor the time to become parallel programming experts, we need solutions that provide both high productivity and effective utilization of parallel architectures. The main challenge today is what kind of programming methodologies can we provide to this group of programmers in order to deal with all these irregular applications on this diverse set of parallel architectures?

1.2 Abstractions for Parallelism in Irregular Algorithms

How can one reason about and exploit parallelism in irregular algorithms? The standard abstraction for reasoning about parallelism in regular algorithms is the *dependence graph* [76]. However, dependences in most irregular algorithms are functions of runtime values such as the values on the nodes and edges of a graph, so the static dependence graph is not a useful abstraction for these algorithms. It is more useful take a data-centric view that we call the *operator formulation*, in which an algorithm is viewed as the iterated application of an *operator* to a data structure such as a graph [115]. This abstraction is illustrated here using two algorithms for solving the single source shortest path (SSSP) problem – Dijkstra's algorithm [34] and chaotic relaxation [28] – which repeatedly update estimates of the shortest distance to

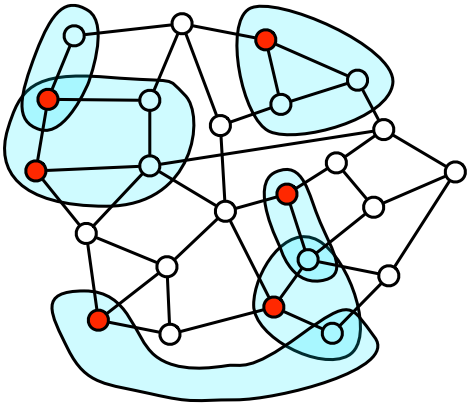


Figure 1.1: Operator Formulation

each node from a source until a fixpoint is reached. The key concepts in the operator formulation are the following:

- *Active elements* are sites in the graph (nodes or edges) where computation must be performed. For the SSSP algorithms, active nodes are nodes whose distance estimates have been updated; initially, the only active node is the source. In Figure 1.1, active nodes are shown in red.

- The *operator* is a description of the work that must be done at an active node. For SSSP, this is the relaxation operator [34], which updates the distance estimates of the immediate neighbors of the active node, if needed. The application of an operator to an active element is an *activity*.

- The region of the graph read or written by an activity is called its *neighborhood*. Operators may add or remove nodes and edges in the neighborhood. Some nodes in the neighborhood may become active themselves.

Neighborhoods are shaded blue in Figure 1.1. Note that in general, the neighborhood of an active node is distinct from its neighbors in the graph.

In general, there may be many active elements in a graph during the execution of an algorithm. We can categorize algorithms in two classes, depending on whether they place constraints on the processing of active elements. In *unordered* algorithms, active elements can be processed in any order, although some scheduling orders may be more efficient than others. Unordered algorithms are usually implemented using worklists organized for a particular scheduling heuristic. Chaotic relaxation [28] is an unordered algorithm, since it does not enforce any particular order on the application of relaxations. Other examples of unordered algorithms are Delaunay mesh generation and refinement, and preflow-push maxflow computation.

In *ordered* algorithms, there is some algorithm-specified order in which active elements must appear to have been processed. This order is usually implemented using a priority queue. Dijkstra's SSSP algorithm, Prim's minimum spanning tree algorithm are two such examples.

Amorphous data-parallelism (ADP) is the parallelism that arises from simultaneously processing active nodes, subject to neighborhood and ordering constraints. ADP is a generalization of conventional data-parallelism [65] in which (i) concurrently executing operations may conflict with each other, (ii) activities can be created dynamically, and (iii) activities may modify the

underlying data structure [115].

1.3 Exploiting Amorphous Data Parallelism

How can one exploit amorphous data parallelism? As we already mentioned traditional compile-time parallelization is not a very effective approach, because dependences between computations are not known statically.

Instead, parallelization methodologies for this domain rely on the notion of *optimistic parallelization* [115]. In this approach, most of the work required to parallelize an irregular algorithm is performed at runtime. Each thread executes a parallel task T_i "optimistically", that is, under the assumption that other threads will not concurrently execute tasks with conflicting accesses to the data accessed by T_i. The execution of T_i is performed *speculatively* [57, 62, 126]. In speculative execution, the effects of each task T_i are made visible to other threads only upon the successful completion of T_i. In case of conflict between two tasks one of them is aborted and its effects on the shared state are undone.

The Galois system [81] and systems based on transactional memory [62] are based on this parallelization strategy. In order to further simplify programming in the above methodology the Galois system also exploits the separation between algorithms and data structures that has become the norm in object-oriented programming: the application programmer writes sequential programs in Java or C++ but uses a library of concurrent data structures that perform all the necessary concurrency management in co-ordination

with a sophisticated speculation-based runtime system. Since the library data structures are written by a small number of expert parallel programmers, this approach simplifies the task of most application programmers.

However, there is a number of significant problems that arise in practice in performance programming of multicore and manycore processors, which are not addressed by the above approach. Some of the major complications are the following.

Choice of algorithm: Programmers usually have a choice of many algorithms for solving a given problem: even a relatively simple problem like `SSSP` can be solved using Dijkstra's algorithm [34], the Bellman-Ford algorithm [34], the label-correcting algorithm [104], and delta-stepping [104], among others. Manually implementing and optimizing a number of different solutions in order to find the best one is a tedious and time-consuming process.

Architecture trade-offs: There are complicated trade-offs between parallelism and work-efficiency in these algorithms. For example, Dijkstra's algorithm orchestrates the application of relexations very carefully and achieves very good work-efficiency, but it has relatively little parallelism. Conversely, the Bellman-Ford and label-correcting algorithms are less strict about the order of applying relaxations and can exhibit a lot of parallelism but may be less work-efficient. Therefore, the best algorithmic choice may depend on the number of cores that are available to solve

the problem. The best algorithmic choice may depend on the core architecture. If Single-Instruction Multiple-Data (SIMD) style execution is supported efficiently by the cores, as is the case with Graphics Processing Units (GPUs), the Bellman-Ford algorithm may be preferable to the label-correcting or delta-stepping algorithms. Conversely, for Multiple-Instruction Multiple-Data (MIMD) style execution, label-correcting or delta-stepping may be preferable.

Input sensitivity: The amount of parallelism in irregular graph algorithms is usually dependent on the structure of the input graph. For example, regardless of which algorithm is used, there is little parallelism in the `SSSP` problem if the graph is a long chain (more generally, if the diameter of the graph is large); conversely, for graphs that have a small diameter such as those that arise in social network applications, there may be a lot of parallelism that can be exploited by the Bellman-Ford and label-correcting algorithms. Therefore, the best algorithmic choice may depend on the size and structure of the input graph.

Scheduling and synchronization choices: Even for a given algorithm, there are usually a large number of implementation choices that must be made by the performance programmer. Each of the `SSSP` algorithms listed above has a host of implementations; for example, label corrections in the label correcting algorithm can be scheduled in FIFO, LIFO, and other orders, and the scheduling policy can make a big difference in the overall

performance of the algorithm. Similarly, delta-stepping has a parameter that can be tuned to increase parallelism at the cost of performing extra computation. As in other parallel programs, synchronization can be implemented using spin-locks, abstract locks, or Compare-and-Swap (CAS) operations. These choices can affect performance substantially, but even expert programmers cannot always make the right choices.

1.4 Elixir: Synthesis of Irregular Algorithms

One promising approach for addressing the above problems is *program synthesis* [98, 5, 51]. Instead of writing programs in a high-level language like C++ or Java, the programmer writes a higher level specification of *what* needs to be computed, leaving it to an automatic system to synthesize efficient parallel code for a particular platform from that specification. This approach has been used successfully in domains like signal processing where algorithms have a very concise mathematical expression [124].

In the case of irregular algorithms, however, we currently do not have a synthesis methodology that (i) allows for a concise, high-level expression of problems, and (ii) allows one to derive efficient parallel implementations that are competitive with hand-optimized code. Moreover, we currently have few insights into the structure of parallelism and locality in irregular algorithms, which would allow development of techniques and tools that transform irregular programs to execute efficiently on parallel computers.

To address these problems, this dissertation develops a methodology

and a system, Elixir, for synthesis of parallel programs from irregular algorithm specifications based on the operator formulation. Elixir [120, 121], starts from a high-level non-deterministic algorithm specification and produces parallel C++ implementations for multicore shared memory machines. The Elixir methodology can be summarized by the motto:

Algorithm = Operators + Schedule

An Elixir specification consists of two components: (i) a set of operators over irregular data-structures, which are actions that capture the main algorithm logic (i.e., what needs to be done to solve a problem), and (ii) a declarative specification of the parallel schedule (i.e., how to perform these actions to get an efficient solution). Different algorithm variants for irregular problems correspond to different schedules for a given set of operators.

Internally, Elixir performs sophisticated compiler analyses to merge and optimize the two components and inserts automatically synchronization necessary for correct parallelization. The Elixir approach has two advantages. First, it improves productivity by allowing programmers to express high-level solutions and avoid writing explicitly parallel code, which introduces problems such as race conditions and deadlocks. Second, it enables a systematic exploration of the implementation space, simply by describing different scheduling strategies in the Elixir domain-specific language. Such explorations are necessary to optimize programs for complex parallel platforms, where analytical

performance modeling is usually intractable and where the best solution is often input and platform dependent. Below, we present briefly the main features of Elixir.

1.4.1 Elixir Language Abstractions and Automated Reasoning

An Elixir specification is an implicitly parallel algorithm description. The operators, which can be applied non deterministically, capture all the latent concurrency for the family of algorithms using them as their base logic. Expressing these different variants easily, in order to find the best performing one, calls for a language that allows to express schedules separately from the operators, and which permits schedules that can express competitive algorithm variants. Unfortunately, contemporary languages like Java or C++ only support lower-level descriptions of computations and do not provide a satisfactory solution to this problem. Elixir address this problem by providing a language that expresses a rich class of schedules and a novel methodology for knitting the operator and schedule specifications together to produce efficient parallel programs.

A significant innovation in Elixir is its refined view of the schedule. One aspect of the schedule is concerned with *ordering* the processing of different operator instances that are enabled during the algorithm execution. Broadly, one can implement ordering via *dynamic* mechanisms that bind operator scheduling at runtime or *static* mechanisms that fix the scheduling at compile time. Typically, dynamic mechanisms are implemented as data-structures used by

the algorithm, e.g. priority queues and sets, that store handles to tasks and order them based on the values of algorithm-specific ranking functions. Static mechanisms, on the other hand, manifest as code in the program that corresponds to combinations of simpler operators that either perform more complex updates in the same portion of the graph or update larger portions of the graph. A different aspect of the schedule is concerned with finding the *delta* of each operator, that is, the new tasks generated as a result of the operator execution. Hence, the implementation must also contain logic to discover and schedule the delta. Usually, algorithm variants differ in their choices of static mechanisms, dynamic mechanisms, as well as the logic for scheduling the operator delta.

The Elixir language allows users to specify both static and dynamic parts of the schedule. Additionally, Elixir infers the operator delta automatically. Inferring the delta requires performing sophisticated static analyses in order to reason about the effects of each operator and identify the new work that it generates as a side-effect. Once Elixir infers the operator deltas, it then fuses the operators and the different ingredients of the schedule together to create an algorithm variant, and it also adds synchronization code to execute operators atomically for correct parallel execution. This provides a significant advantage over existing parallel programming frameworks. In existing frameworks the schedule is fused together with the operators into a single software artifact. Consequently, the programmer has to disentangle and re-assemble these elements manually and reason about the correctness of the new solution.

This makes it very hard to experiment with different algorithm variants. In Elixir, on the contrary, the programmer generates different variants by simply changing elements of the high-level specification and letting the tool take care of the implementation details.

Furthermore, Elixir can automatically customize the synchronization code to the operator and schedule, which is not possible in existing programming frameworks.

1.4.2 Synthesis via Automated Planning

Another innovation of the Elixir methodology relates to the synthesis techniques it uses to automatically explore various implementation strategies for irregular problems. As a first step, Elixir fuses the operators and the schedule and creates a high-level imperative program that is implemented in terms of statements in a high-level intermediate representation (HIR), as typically found in compilers. Turning such high-level solutions into efficient parallel programs in a language like C++ requires applying certain key transformations: (i) inserting synchronization to ensure atomic operator execution, (ii) selecting efficient implementations of HIR statements, and (iii) finding a good ordering of HIR statements.

The space of possible programs to consider is usually large, since there are multiple valid statement execution orders and candidate implementations for each statement, as well as multiple synchronization policies to consider.

Since different policies for each transformation interact in subtle ways,

and since the best solution is usually input and platform dependent, selecting a priori the combination that leads to the best parallel implementation is intractable. Consequently, it is necessary to have a method that generates multiple efficient programs automatically so that we can find the best program through search-based techniques.

One solution is to approach the issue via a traditional compilation strategy, where transformations are applied as different phases, in some order, to produce the final program. This solution suffers from the well-known phase ordering problem and may not lead to the highest quality code for many problems. Elixir addresses this problem by using a novel synthesis approach based on *automated planning* [121]. The Elixir synthesizer encodes each of the above transformations declaratively via constraints as a planning problem, and then combines their respective constraints to create a composite planning problem. Elixir uses off-the-shelf planners to find solutions to the composite problem. Each individual solution corresponds to a correctly synchronized program in which HIR statements have been reordered and lowered to specific implementations. The key advantage of this approach is that searching for solutions that *simultaneously* solve all constraints avoids the phase-ordering problem and produces better code. This methodology leads to the first *integrated* compilation approach involving tasks such as scheduling and synchronization.

1.5 Contributions and Organization

This thesis makes several contributions in the areas of programming languages, program synthesis, and parallel programming.

- **Synthesis for irregular graph problems**. This thesis presents the first approach that is able to synthesize efficient parallel programs for the very challenging domain of sparse graph problems. Chapter 2 and Chapter 3 present the design of Elixir, which is the first system that achieves high productivity and high performance for this domain. Chapter 4 discusses a number of case studies that demonstrate the ability of Elixir to synthesize parallel implementations that perform competitively with, and in many cases even outperform highly optimized hand-written implementations.

- **Language Abstractions for concisely expressing parallel irregular problems**. Chapter 2 presents a language that enables programmers to express irregular programs abstractly as operators plus schedules, along with a scheduling language that can express various interesting algorithmic variants for each problem. Chapter 5 presents an extended case study on the problem of betweenness centrality. This study demonstrates how the Elixir language can be used as a conceptual tool both for understanding existing algorithms for irregular problems and for designing new efficient parallel algorithms.

- **Automated reasoning for synthesizing incremental computations**. Chapter 2 describes a static analysis that can infer the new work that gets generated as a side effect of applying a specific operator. In the context of Elixir this is useful in order to synthesize work-efficient algorithms that incrementally update the graph in order to compute the solution efficiently. In a more general context, this technique can be use for inferring incremental computations from a non-deterministic specification.

- **An integrated compilation approach for synthesis of efficient parallel graph programs**. Chapter 3 presents the first integrated compilation approach for the very challenging domain of sparse graph problems. Our approach uses automated planning as a mechanism to declaratively specify and compose a number of program transformations that are necessary for synthesizing efficient parallel graph programs. Our formulation of integrated compilation via planning is generic and can be applied to contexts that are different from the synthesis of graph programs.

- **Static analysis techniques that mitigate the overheads of speculative parallelization**. In Chapter 6 we focus on the optimization of irregular algorithms in a more conventional programming setting where solutions are expressed in languages like C++ or Java and are parallelized using speculation-based approaches. Examples of speculative systems are the Galois system [81] and transactional memory variants. In

such programming environments we show how static analysis can be used to optimize the overheads of speculative parallelization of irregular algorithms. Given an input program containing regions with transactional semantics, the analysis calculates a per program point over-approximation of the set of acquired locks. Using this information it detects cases where synchronization is redundant (an object is already locked). More importantly, this analysis identifies fail-safe points – points after which transactions are guaranteed to commit and speculation (locking and recording undo actions) is no longer necessary. This analysis also identifies irregular programs called cautious operators, which read all shared state before modifying any shared data. Such programs do not require storing rollback information. Reasoning about irregular graph problems automatically to identify such patterns is particularly challenging, since the space of possible program configurations is infinite. Our analysis uses a novel abstraction scheme to record sets of locked objects residing deeply in the heap, in a way that deals with the so-called "state-space explosion" phenomenon, which occurs when a static analysis explores a prohibitively large number of "abstract states", exhausting space or time resources.

In Chapter 7 we review the literature of related work and in Chapter 8 we conclude with a summary of our contributions and discussion of future work. Parts of the work presented in this thesis have appeared in the following publications [120, 119, 121, 122].

Chapter 2

Elixir: A System for Synthesizing Irregular Graph Algorithms

In this chapter [1], we describe a system called Elixir that synthesizes parallel programs for shared-memory multicore processors, starting from irregular algorithm specifications based on the *operator formulation of algorithms* [115]. The operator formulation is a data-centric description of algorithms in which algorithms are expressed in terms of their action on data structures rather than in terms of program-centric constructs like loops. There are three key concepts: *active elements*, *operator*, and *ordering*.

Active elements are sites in the graph where there is computation to be done. For example, in `SSSP` algorithms, each node has a label that is the length of the shortest known path from the source to that node; if the label of a node is updated, it becomes an active node since its immediate neighbors must be examined to see if their labels can be updated as well.

[1] Part of the work presented in this chapter has appeared in *"Dimitrios Prountzos, Roman Manevich, Keshav Pingali. 'Elixir: A System for Synthesizing Concurrent Graph Programs'. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) 2012".* The first author is responsible for the conception and the implementation of the ideas presented in this publication. Additional authors provided assistance with the presentation of the material.

The operator is a description of the computation that is done at an active element. Applying the operator to an active element creates an *activity*. In general, an operator reads and writes graph elements in some small region containing the active element. These elements are said to constitute the *neighborhood* of this activity.

The ordering specifies constraints on the processing of active elements. In *unordered* algorithms, it is semantically correct to process active elements in any order, although different orders may have different work-efficiency and parallelism. A parallel implementation may process active elements in parallel provided the neighborhoods do not overlap. The non-overlapping criterion can be relaxed by using commutativity conditions, but we do not consider these in this thesis. The preflow-push algorithm for maxflow computation, Boruvka's minimal spanning tree algorithm, and Delaunay mesh refinement are examples of unordered algorithms. In *ordered* algorithms on the other hand, there may be application-specific constraints on the order in which active elements are processed. Discrete-event simulation is an example: any node with an incoming message is an active element, and messages must be processed in time order.

The specification language described in this chapter permits application programmers to specify (i) the operator, and (ii) the schedule for processing active elements; Elixir takes care of the rest of the process of generating parallel implementations. Elixir addresses the following major challenges.

- How do we design a language that permits operators and scheduling policies to be defined concisely by application programmers?
- The execution of an activity can create new active elements in general. How can newly created active elements be discovered incrementally without having to re-scan the entire graph?
- How should synchronization be introduced to make activities atomic?

The rest of this chapter is organized as follows. Section 6.2 presents the key ideas and challenges, using a number of algorithms for the SSSP problem. Section 2.2 formally presents the Elixir graph programming language and its semantics for the simple case of dynamic scheduling alone. Section 2.3 formalizes the semantics of statements combining both static and dynamic scheduling. Section 2.4 describes the automated reasoning techniques that Elixir uses to merge the operators with the schedule and create a high-level incremental algorithm.

2.1 An Informal Introduction to the Elixir Approach

In this section, we present the main ideas behind Elixir, using the SSSP problem as a running example. In Section 2.1.1 we discuss the issues that arise when SSSP algorithms are written in a conventional programming language. In Section 2.1.2, we describe how operators can be specified in Elixir independently of the schedule. and how a large number of different scheduling policies can be specified abstractly by the programmer. In particular, we show how

the Dijsktra, Bellman-Ford, label-correcting and delta-stepping algorithms can be specified in Elixir just by changing the scheduling specification. Finally, in Section 2.1.3 we sketch how Elixir addresses the two most important synthesis challenges: how to synthesize efficient implementations, and how to insert synchronization.

2.1.1 SSSP Algorithms and the Relaxation Operator

Given a weighted graph and a node called the source, the SSSP problem is to compute the distance of the shortest path from the source node to every other node (we assume the absence of negative weight cycles). As mentioned in Section 1.3, there are many sequential and parallel algorithms for solving the SSSP problem such as Dijkstra's algorithm [34], the Bellman-Ford algorithm [34], the label-correcting algorithm [104], and delta-stepping [104]. In all algorithms, each node a has an integer attribute $a.dist$ that holds the length of the shortest known path to that node from the source. This attribute is initialized to ∞ for all nodes other than the source where it is set to 0, and is then lowered to its final value using iterative *edge relaxation*: if $a.dist$ is lowered and (i) there is an edge (a, b) of length w_{ab}, and (ii) $b.dist > a.dist + w_{ab}$, the value of $b.dist$ is lowered to $a.dist + w_{ab}$. However, the order in which edge relaxations are performed can be different for different algorithms, as we discuss next.

In Figure 2.1 we present pseudocode for the sequential label-correcting and the Bellman-Ford SSSP algorithms with the edge relaxation highlighted

in grey. Although both algorithms use relaxations, they may perform them in different orders and different numbers of times. We call this order the *schedule* for the relaxations. The label-correcting algorithm maintains a worklist of edges for relaxation. Initially, all edges connected to the source are placed on this worklist. At each step, an edge (a, b) is removed from the worklist and relaxed; if there are several edges on the worklist, the edge to be relaxed is chosen heuristically. If the value of $b.dist$ is lowered, all edges connected to b are placed on the worklist for relaxation. The algorithm terminates when the worklist is empty. The Bellman-Ford algorithm performs edge relaxations in rounds. In each round, all the graph edges are relaxed in some order. A total of $|V| - 1$ rounds are performed, where $|V|$ is the number of graph nodes. Although both algorithms are built using the same basic ingredient, as Figure 2.1 shows, it is not easy to change from one to another. This is because *the code for the relaxation operator is intertwined intimately with the code for maintaining worklists, which is an artifact of how the relaxations are scheduled by a particular algorithm.* In a concurrent setting, the code for synchronization makes the programs even more complicated.

2.1.2 `SSSP` in Elixir

Figure 2.2 shows several `SSSP` algorithms written in Elixir. The major components of the specification are the following.

```
1   Label Correcting

3  INITIALIZATION:
4  for each node a in V {
5    if (a == Src) a.dist = 0;
6    else a.dist = ∞;
7  }
8  RELAXATION:
9  Wl = new worklist();
10 // Initialize worklist.
11 for each e = (Src, _, _) {
12  Wl.add(e);
13 }
14 while (!Wl.empty()) {
15  (a,b,w) = Wl.remove();
16   if (a.dist + w < b.dist) {
17    b.dist = a.dist + w;
18    for each e = (b, c, w')
19      Wl.add(e);
20  }
21 }
```

```
1   BellmanFord

3  INITIALIZATION:
4  for each node a in V {
5    if (a == Src) a.dist = 0;
6    else a.dist = ∞;
7  }
8  RELAXATION:
9  for i = 1..|V| − 1 {
10   for each e = (a, b, w) {
11    if (a.dist + w < b.dist) {
12     b.dist = a.dist + w;
13    }
14 }
```

Figure 2.1: Pseudocode for label-correcting and Bellman-Ford `SSSP` algorithms.

2.1.2.1 Operator Specification

Lines 1–2 define the graph. Nodes and edges are represented abstractly by relations that have certain attributes. Each node has a unique ID attribute (**node**) and an integer attribute `dist`; during the execution of the algorithm, the `dist` attribute of a node keeps track of the shortest known path to that node from the source. Edges have a source node, a destination node, and an integer attribute `wt`, which is the length of that edge. Line 4 defines the source node. `SSSP` algorithms use two operators, one called `initDist` to initialize the `dist` attribute of all nodes (lines 6–7), and another called `relaxEdge` to perform edge relaxations (lines 9–13).

Operators are described by rewrite rules in which the left-hand side is a

```
1  Graph [ nodes(node : Node, dist : int)
2          edges(src : Node, dst : Node, wt : int) ]

4  source : Node

6  initDist = [ nodes(node a, dist d) ] →
7               [ d = if (a == source) 0 else ∞]

9  relaxEdge = [ nodes(node a, dist ad)
10               nodes(node b, dist bd)
11               edges(src a, dst b, wt w)
12               ad + w < bd ] →
13              [ bd = ad + w ]

15 init = foreach initDist
16 sssp = iterate relaxEdge ≫ sched
17 main = init; sssp
```

Algorithm	Schedule specification
Dijkstra	1 sched = **metric** ad ≫ **group** b
Label-correcting	1 sched = **group** b ≫ **unroll** 2 ≫ **approx metric** ad
Δ-stepping-style	1 DELTA : **unsigned int** 2 sched = **metric** (ad + w) / DELTA
Bellman-Ford	1 NUM_NODES : **unsigned int** 2 // override sssp 3 sssp = **for** i=1..(NUM_NODES −1) 4 step 5 step = **foreach** relaxEdge

Figure 2.2: Elixir programs for `SSSP` algorithms.

predicated subgraph pattern, called the redex pattern, and the right-hand side is an *update*.

A predicated subgraph pattern has two parts, a *shape constraint* and a *value constraint*. A subgraph G' of the graph is said to *satisfy* the shape constraint of an operator if there is a bijection between the nodes in the shape constraint and the nodes in G' that preserves the edge relation. The shape constraint in the `initDist` operator is satisfied by every node in the graph, while the one in the `relaxEdge` operator is satisfied by every pair of nodes connected by an edge. A value constraint filters out some of the subgraphs that satisfy the shape constraint by imposing restrictions on the values of attributes; in the case of the `relaxEdge` operator, the conjunction of the shape and value constraints restricts attention to pairs of nodes (a, b) which have an edge from a to b, and whose `dist` attributes satisfy the constraint $a.dist + w_{ab} < b.dist$. A subgraph that satisfies both the shape and value constraints of an operator is said to match the predicated sub-graph pattern of that operator, and will be referred to as a *redex* of that operator.

The right-hand side of a rewrite rule specifies updates to some of the value attributes of nodes and edges in a subgraph matching the predicated subgraph pattern on the left-hand side of that rule. To simplify exposition, we restrict attention to *local computation* algorithms [115] that are not allowed to morph the graph structure by adding or removing nodes and edges. Additionally, we assume that our operators work over a bounded number of nodes and

edges. Elixir allows disjunctive operators of the form:

$$op_1 \textbf{ or } \ldots \textbf{ or } op_k$$

Problems like betweenness centrality [20], connected components, and preflow-push use disjunctive operators with multiple disjuncts.

Statements define how operators are applied to the graph. A looping statement has one the forms '`foreach op`', '`for i=low..high op`', or '`iterate op`' where `op` is an operator. A `foreach` statement finds all matches of the given operator and applies the operator once to each matched subgraph in some unspecified order. Line 15 defines the initialization statement to be the application of `initDist` once to each node. A `for` statement applies an operator once for each value of `i` between `low` and `high`. An `iterate` statement applies an operator ad infinitum by repeatedly finding some redex and applying the operator to it. This statement terminates when no redexes exist. Line 16 expresses the essence of the `SSSP` computation as the repeated application of the `relaxEdge` operator (for now, ignore the text "`>> sched`"). It is the responsibility of the user to guarantee that `iterate` arrives to a fixed-point after a finite number of steps by specifying meaningful value constraints. Finally, line 17 defines the entire computation to be the initialization followed by the distances computation.

Elixir programs can be executed sequentially by repeatedly searching the graph until a redex is found, and then applying the operator there. Three optimizations are needed to make this baseline, non-deterministic interpreter,

efficient.

1. Even in a sequential implementation, the order in which redexes are executed can be important for work-efficiency and locality. The best order may be problem-dependent, so it is necessary to give the application programmer control over scheduling. Section 5.3 gives an overview of the scheduling constructs in Elixir.
2. To avoid scanning the graph repeatedly to find redexes, it is desirable to maintain a worklist of potential redexes in the graph. The application of an operator may enable and disable redexes, so the worklist needs to be updated incrementally whenever an operator is applied to the graph. The worklist can be allowed to contain a *superset* of the set of actual redexes in the graph, provided an item is tested when it is taken off the worklist for execution. Section 2.1.3.1 gives a high-level description of how Elixir maintains worklists.
3. In a parallel implementation, each activity should appear to have executed atomically. Therefore, Elixir must insert appropriate synchronization. Section 2.1.3.2 describes some of the main issues in doing this.

2.1.2.2 Scheduling Constructs

Elixir provides a compositional language for specifying commonly used scheduling strategies declaratively and automatically synthesizes efficient implementations of them.

We use Dijkstra-style `SSSP` computation to present the key ideas of

our language. This algorithm maintains nodes in a priority queue, ordered by the distance attribute of the nodes. In each iteration, a node of minimal priority is removed from the priority queue, and relaxations are performed on all outgoing edges of this node. This is described by the composition of two basic scheduling policies:

1. Given a choice between relaxing edge $e_1 = (a_1, b_1)$ and edge $e_2 = (a_2, b_2)$ where $a_1.dist < a_2.dist$, give e_1 a higher priority for execution. In Elixir, this is expressed by the specification `metric ad` (recall that `ad` is the distance of the edge source `a`).

2. To improve spatial and temporal locality, it is desirable to co-schedule active edges that have the same source node a, in preference to interleaving the execution of edges of the same priority from different nodes. In Elixir this is expressed by the specification `group b`, which groups together `relaxEdge` applications on all edges outgoing from a to a successor b. This can be viewed as a *refinement* of the `metric ad` specification, and the composition of these policies is expressed as `metric ad >> group b`.

These two policies exemplify two general scheduling schemes: dynamic and static scheduling. Scheduling strategies that bind the scheduling of redexes at runtime are called *dynamic scheduling strategies*, since they determine the priority of a redex using values known only at runtime. Typically, they are implemented via a dynamic worklist data-structure that prioritizes its contents

based on the specific policy. In contrast, *static scheduling strategies*, such as grouping, bind scheduling decisions at compile-time and are reflected in the structure of the source code that implements composite operators out of combinations of basic ones. One of the contributions of our work is the combination of static and dynamic scheduling strategies in a single system. The main scheduling strategies supported by Elixir are the following.

metric e The arithmetic expression `e` over the variables of the redex pattern is the priority function. In practice, many algorithms use priorities heuristically so they can tolerate some amount of priority inversion in scheduling. Exploiting this fact can lead to more efficient implementations, so Elixir supports a variant called `approx metric e`.

group V specifies that every redex pattern node `v` $\in$ `V` should be matched in all possible ways.

unroll k Some implementations of `SSSP` perform two-level relaxations: when an edge `(a,b)` is relaxed, the outgoing edges of `b` are co-scheduled for relaxation if they are active, since this improves spatial and temporal locality. This can be viewed as a form of loop unrolling. Elixir supports `k`-level unrolling, where `k` is under the control of the application programmer.

(*op*$_1$ or *op*$_2$) $\gg$ fuse specifies that instances of op_1, op_2 working on the same redex should create a new composite operator applying the operators in se-

quence (in left-to-right order of appearance). Fusing improves locality and amortizes the cost of acquiring and releasing locks necessary to guarantee atomic operator execution.

The `group, unroll`, and `fuse` operations define static scheduling strategies. We use the language of Nguyen *et al.* [108] to define dynamic scheduling policies that combine `metric` with `LIFO` and `FIFO` policies, and use implementations of these worklists from the Galois framework [2].

Figure 2.2 shows the use of Elixir scheduling constructs to define a number of `SSSP` implementations. The label-correcting variant [104] is an unordered algorithm, which, on each step, starts from a node and performs relaxations on all incident edges, up to two "hops" away. The delta-stepping variant [104] operates on single edges and uses a Δ parameter to partition redexes into equivalence classes. This heuristic achieves work-efficiency by processing nodes in order of increasing distance from the source, while also exposing parallelism by allowing redexes in the same equivalence class to be processed in parallel. Finally Bellman-Ford [34] works in a SIMD style by performing a series of rounds in which it processes all edges in the graph.

2.1.3 Synthesis Challenges

We finish this section with a brief description of the main challenges that Elixir addresses. First, we discuss how Elixir optimizes worklist maintenance and second how it synchronizes code to ensure atomic operator execution.

2.1.3.1 Synthesizing Work-efficient Implementations

To avoid scanning the graph repeatedly for redexes, it is necessary to maintain a worklist of redexes, and update this worklist incrementally when a redex is executed since this might enable or disable other redexes. To understand the issues, consider the label-correcting implementation in Figure 2.1, which iterates over all outgoing edges of b and inserts them into the worklist. Since the worklist can be allowed to contain a superset of the set of the redexes (as long as items are checked when they are taken from the worklist), another correct but less efficient solution is to insert all edges incident to either a or b into the worklist. However, the programmer manually reasoned that the only place where new "useful" work can be performed is at the outgoing edges of b, since only $b.dist$ is updated. Additionally, the programmer could experiment with different heuristics to improve efficiency. For example, before inserting an edge (b, c) into the worklist, the programmer could eagerly check whether $b.dist + w_{bc} \geq c.dist$.

In a general setting with disjunctive operators, different disjuncts may become enabled on different parts of the graph after an operator application. Manually reasoning about where to apply such incremental algorithmic steps can be daunting. Elixir frees the programmer from this task. In Figure 2.2 there is no code dealing with that aspect of the computation; Elixir automatically synthesizes the worklist updates and also allows the programmer to easily experiment with heuristics like the above without having to write much code.

Another means of achieving work-efficiency is by using a good prior-

ity function to schedule operator applications. In certain implementations of algorithms such as betweenness centrality and breadth first search, the algorithm transitions through different priority levels in a very structured manner. Elixir can automatically identify such cases and synthesize customized dynamic schedulers that are optimized for the particular iteration patterns.

2.1.3.2 Synchronizing Operator Execution

To guarantee correctness in the context of concurrent execution, the programmer must make sure that operators execute atomically. Although it is not hard to insert synchronization code into the basic `SSSP` relaxation step, the problem becomes more complex once scheduling strategies like unroll and group are used since the resulting "super-operator" code can be quite complex. There are also many synchronization strategies that could be used such as abstract locks, concrete locks, and lock-free constructs like CAS instructions, and the trade-offs between them are not always clear even to expert programmers.

Elixir frees the programmer from having to worry about these issues because it automatically introduces appropriate fine grained locking. This allows the programmer to focus on the creative parts of problem solving and still get the performance benefits of parallelism.

2.2 The Elixir Graph Programming Language

In this section, we formalize our language whose grammar is shown in Figure 2.3. Technically, a graph program defines graph transformations, or

attid		Graph attributes
acid		Action identifiers
opid		Operation identifiers
var		Operator variables and global variables
ctype		C++ type
program	::=	graphDef global$^+$ opDef$^+$ actionDef$^+$
graphDef	::=	**Graph** [**nodes**(attDef$^+$ **edges**(attDef$^+$)]
attDef	::=	attid : ctype | attid : **set**[ctype]
global	::=	var **:** ctype
opDef	::=	opid **=** opExp
opExp	::=	[tuple* (boolExp)] $\rightarrow$ [assign*]
tuple	::=	**nodes**(att*) | **edges**(att*)
boolExp	::=	**!**boolExp | boolExp & boolExp | arithExp < arithExp
	|	arithExp **==** arithExp | var **in** setExp
arithExp	::=	number | var | arithExp + arithExp | arithExp **-** arithExp
	|	**if** (boolExp) arithExp **else** arithExp
setExp	::=	**empty** | {var} | setExp + setExp | setExp **-** setExp
assign	::=	var = arithExp | var = setExp | var = boolExp
att	::=	attid var
actionDef	::=	acid = stmt
stmt	::=	**iterate** schedExp | **foreach** schedExp
	|	**for** var = arithExp **..** arithExp stmt | acid
	|	invariant? stmt invariant?
	|	stmt; stmt
schedExp	::=	ordered | unordered
unordered	::=	disjuncts
ordered	::=	opsExp fuseTerm? groupTerm? metricTerm
disjuncts	::=	disjunctExp | disjunctExp **or** disjuncts
disjunctExp	::=	opsExp statSched dynSched
opsExp	::=	opid | opid **or** opsExp
statSched	::=	fuseTerm? groupTerm? unrollTerm?
dynSched	::=	approxMetricTerm? timeTerm?
fuseTerm	::=	**>> fuse**
groupTerm	::=	**>> group** var*
unrollTerm	::=	**>> unroll** number
metricTerm	::=	**>> metric** arithExp
approxMetricTerm	::=	**>> approx metric** arithExp
timeTerm	::=	**>> LIFO** | **>> FIFO**

Figure 2.3: Elixir language grammar (EBNF).

actions, that may be used within an application. It first defines a graph type by listing the data attributes associated with its nodes and edges. Next, a program defines global variables that actions may access for reading. Global variables may be accessed for reading and writing outside of actions by the larger application. The graph program then defines operators and actions. Operators define unit transformations that may be applied to a given subgraph. They are used as building blocks in statements that apply operators iteratively. An important limitation of operators is that they may only update data attributes, but not morph the graph (add or remove nodes and edges). Actions compose statements and name them. They compile to C++ functions that take a single graph reference argument.

2.2.1 Graphs and Patterns

Let $Attr$ denote a finite set of *attributes*. An attribute denotes a subtype of one of the following types: the set of numeric values Num (integers and reals), graph nodes $Node$ and sets of graph nodes $\wp(Node)$. Let $Val \stackrel{\text{def}}{=} Num \cup Node \cup \wp(Node)$ stand for the union of those types.

Definition 2.2.1 (Graph)**.** [2] *A graph* $G = (V^G, E^G, Att^G)$ *is a triple where* $V^G \subset Node$ *the graph nodes,* $E^G \subseteq V^G \times V^G$ *are the graph edges, and* $Att^G : ((Attr \times V^G) \to Val) \cup ((Attr \times V^G \times V^G) \to Val)$ *associates values with nodes and edges. We denote the set of all graphs by Graph.*

[2] Our formalization naturally extends to graphs with several node and edge relations, but for simplicity of the presentation we have just one of each.

Definition 2.2.2 (Pattern)**.** *A pattern* $P = (V^P, E^P, Att^P)$ *is a connected graph over variables. Specifically,* $V^P \subset Var$ *are the pattern nodes,* $E^P \subseteq V^P \times V^P$ *are the pattern edges, and* $Att^P : (Attr \times V^P) \to Var \cup (Attr \times V^P \times V^P) \to Var$ *associates a distinct variable (not in* V^P*) with each node and edge. We call the latter set of variables* attribute variables*. We refer to* (V^P, E^P) *as the* shape *of the pattern.*

In the sequel, when no confusion is likely, we may drop superscripts denoting the association between a component and its containing compound type instance, e.g., $G = (V, E)$.

Definition 2.2.3 (Matching)**.** *Let* G *be a graph and* P *be a pattern. We say that* $\mu : V^P \to V^G$ *is a* matching *(of* P *in* G*), written* $(G, \mu) \models P$*, if it is one-to-one, and for every edge* $(x, y) \in E^P$ *there exists an edge* $(\mu(x), \mu(y)) \in E^G$*. We denote the set of all matchings by* $Match : Var \to Node$*.*

We extend a matching $\mu : V^P \to V^G$ to evaluate attribute variables $\mu : (Graph \times Var) \to Val$ as follows. For every attribute a, pattern nodes $y, z \in V^P$, and attribute variable x, we define:

$$\mu(G, x) = Att^G(a, \mu(y)) \quad \text{if} \quad Att^P(a, y) = x \ ,$$
$$\mu(G, x) = Att^G(a, \mu(y), \mu(z)) \quad \text{if} \quad Att^P(a, y, z) = x \ .$$

Lastly, we let μ extend to evaluate expressions over the variables of a pattern by structural induction over the natural definitions of the sub-expression types defined in Figure 2.3.

2.2.2 Operators

In the sequel, we will denote syntactic expressions by referring to their respective non-terminals in Figure 2.3.

Definition 2.2.4 (Operator)**.** *An operator is a triple denoted by* $op = [R^{op}, Gd^{op}] \to [Upd^{op}]$ *where* R^{op} *is called the* redex pattern*;* Gd^{op} *is a Boolean-valued expression over the variables of the redex pattern, called the* guard*; and* $Upd^{op} : V^R \to Expr$ *contains an assignment per attribute variable in the redex pattern, in terms of the variables of the redex pattern (for brevity, we omit identity assignments).*

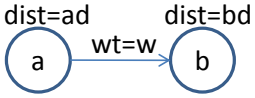
We now define the semantics of an operator as a function that transforms a graph for a given matching $[\![\cdot]\!] : opExp \to (Graph \times Match) \to Graph$. Let $op = [R, Gd] \to [Upd]$ be an operator and let $\mu : V^R \to V^G$ be a matching (of R in G). We say that μ satisfies the *shape constraint* of op if $(G, \mu) \models R$. We say that μ satisfies the *value constraint* of op (and shape constraint), written $(G, \mu) \models R, Gd$, if $\mu(G, Gd) = \mathsf{True}$. In such a case, μ induces the subgraph D denoted by $\mu(R)$, which we call a *redex* and define by:

$$\begin{array}{rcl} V^D & \stackrel{\text{def}}{=} & \{\mu(x) \mid x \in V^R\} \ , \\ E^D & \stackrel{\text{def}}{=} & E^G \cap (V^D \times V^D) \ , \\ Att^D & \stackrel{\text{def}}{=} & Att^G|_{(Att \times V^D) \cup (Att \times V^D \times V^D)} \ . \end{array}$$

We define an operator application by:

$$[\![op]\!](G, \mu) = \begin{cases} G' = (V^G, E^G, Att'), & (G, \mu) \models R, Gd; \\ G, & \text{else} \end{cases}$$

(a) A redex pattern.



(b) A graph.

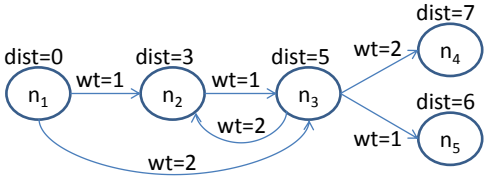


Figure 2.4: A redex pattern and a graph where the pattern can be matched.

We have that $D = \mu(R)$ and the node and edge attributes in D are updated using the expressions in *Upd*, respectively:

$$Att'(a,v) = \begin{cases} \mu(G, Upd(y)), & v \in V^D, v = \mu(x_v) \\ & \text{and } Att^R(a, x_v) = y; \\ Att(a,v) & \text{else.} \end{cases}$$

$$Att'(a,u,v) = \begin{cases} & (u,v) \in E^D, \\ \mu(G, Upd(y)), & u = \mu(x_u), v = \mu(x_v) \\ & \text{and } Att^R(a, x_u, x_v) = y; \\ Att(a,u,v) & \text{else.} \end{cases}$$

Example 2.2.5. *Figure 2.4a shows the redex pattern of the relaxEdge operator and Figure 2.4b shows a graph instance. The matching* $\mu_{1,2} = \{a \mapsto n_1, b \mapsto n_2\}$ *yields the following evaluations:*

$$\begin{aligned} \mu_{1,2}(G, ad) &= 0 \\ \mu_{1,2}(G, w) &= 1 \\ \mu_{1,2}(G, bd) &= 3 \\ \mu_{1,2}(G, ad + w < bd) &= \mathit{True}\ . \end{aligned}$$

Therefore, the subgraph induced by the nodes n_1 *and* n_2 *is a redex.*

The operator application $[\![relaxEdge]\!](G, \mu_{1,2}) = G'$ *establishes the fol-*

lowing evaluations where differences are shown in boxes:

$$\begin{aligned} \mu_{1,2}(G', ad) &= 0 \\ \mu_{1,2}(G', w) &= 1 \\ \mu_{1,2}(G', bd) &= \boxed{1} \\ \mu_{1,2}(G', ad + w < bd) &= \boxed{\mathit{False}} \ . \end{aligned}$$

The remainder of this section defines the semantics of statements. `iterate` and `foreach` statements have two distinct flavors: *unordered iteration* and *ordered iteration*. We define them in that order. We do not define `for` statements as their semantics is quite standard in all imperative languages.

2.2.3 Semantics of Unordered Statements

Unordered statements have the form

$$\texttt{iterate}\ \mathit{schedExp}_1 \mathbf{or}\ \ldots \mathbf{or}\ \mathit{schedExp}_k$$

or

$$\texttt{foreach}\ \mathit{schedExp}_1 \mathbf{or}\ \ldots \mathbf{or}\ \mathit{schedExp}_k$$

where each $\mathit{schedExp}_i$ has the form

$$\mathit{opsExp} \gg \mathit{statSched} \gg \mathit{dynSched}\ .$$

The expression *opsExp* is either a single operator *op* or a disjunction of operators $op_1 \mathbf{or}\ \ldots \mathbf{or}\ op_k$ having the same shape ($R^{op_1} = \ldots = R^{op_k}$). Intuitively, a disjunction represents alternative graph transformations. We define the shorthand $op_{i..j} = op_i \mathbf{or}\ \ldots \mathbf{or}\ op_j$.

The expression *statSched*, called a *static schedule*, is a possibly empty sequence of *static scheduling terms*, which may include `fuse`, `group`, and `unroll`. If *opsExp* is a disjunction then it must be followed by a `fuse` term. An expression of the form $opsExp \gg statSched$ defines a composite operator by grouping together operator applications in a statically-defined (i.e., determined at compile-time) way. We refer to such an expression as a *static operator*.

The expression *dynSched*, called a *dynamic schedule*, is a possibly empty sequence of *dynamic scheduling terms*, which may include `approx metric`, `LIFO`, and `FIFO`. A dynamic schedule determines the order by which static operators are selected for execution by associating a dynamic priority with each redex.

To simplify the exposition, in this section we present the semantics under the simplifying assumption that *statSched* is empty. Section 2.3 contains the details for the general case.

2.2.3.1 Preliminaries

Definition 2.2.6 (Active Element)**.** *An active element, denoted by* $elem\langle op, \mu\rangle$*, pairs an operator* op *with a matching* $\mu \in Match$*. It stands for the potential application of* op *to* (G, μ)*. We denote the set of all active elements by* $\mathcal{A}$*.*

We define the new set of redexes for an operator and for a disjunction of operators, respectively by

$$\begin{aligned} \textsc{Rdx}[\![op]\!]G &\stackrel{\text{def}}{=} \{\mu \in Match \mid (G,\mu) \models R^{op}, Gd^{op}\} \ , \\ \textsc{Rdx}[\![op_{1..k}]\!]G &\stackrel{\text{def}}{=} \textsc{Rdx}[\![op_1]\!]G \cup \ldots \cup \textsc{Rdx}[\![op_k]\!]G \ . \end{aligned}$$

We define the set of redexes of an operator op' created by an application of an operator op by

$$\textsc{Delta}[\![op, op']\!]\,(G,\mu) \stackrel{\text{def}}{=} \begin{array}{ll} \textbf{let} & G' = [\![op]\!](G,\mu) \\ \textbf{in} & \textsc{Rdx}[\![op']\!]G' \setminus \textsc{Rdx}[\![op']\!]G \ . \end{array}$$

We lift the operation to disjunctions:

$$\textsc{Delta}[\![op_a, op_{c..d}]\!]\,(G,\mu) \stackrel{\text{def}}{=} \bigcup_{c \le i \le d} \textsc{Delta}[\![op_a, op_i]\!]\,(G,\mu) \ .$$

2.2.3.2 Defining Dynamic Schedulers

Let `iterate` *exp* be a statement and let $op_{1..k}$ be the operators belonging to *exp*. An `iterate` statement executes by repeatedly finding a redex for an operator and applying that operator to the redex. An execution of `iterate` yields a (possibly infinite) sequence $G = G_0, \ldots, G_k, \ldots$ where $G_{i+1} = [\![op]\!](G_i, \mu_i)$.

We now define a scalar priority $pr(t, e, G_i)$ and partial order $\le_t$ for a scheduling term $t \in \{\texttt{metric}, \texttt{approx metric}, \texttt{LIFO}, \texttt{FIFO}\}$, an active element e, and a graph G_i:

$$\begin{array}{lcl} pr(\texttt{metric}\,a, e, G_i) & \stackrel{\text{def}}{=} & \mu_i(G_i, a) \\ pr(\texttt{approx metric}\,a, e, G_i) & \stackrel{\text{def}}{=} & \mathit{fuzz}(\mu_i(G_i, a)) \\ pr(\texttt{LIFO}, e, G_i) = pr(\texttt{FIFO}, e, G_i) & \stackrel{\text{def}}{=} & i \\ p \le_{\texttt{metric}\,a} p' & \Leftrightarrow & p \le p' \\ p \le_{\texttt{approx metric}\,a} p' & \Leftrightarrow & p \le p' \\ p \le_{\texttt{LIFO}} p' & \Leftrightarrow & p \ge p' \\ p \le_{\texttt{FIFO}} p' & \Leftrightarrow & p \le p' \end{array}$$

where $\mathit{fuzz}(x)$ is some approximation of x.

For an expression $d = t_1 >> \ldots >> t_k \in \mathit{dynSched}$, we define $pr(d, e, G_i) \stackrel{\text{def}}{=} \langle pr(t_1, e, G_i), \ldots, pr(t_k, e, G_i) \rangle$ and $p \leq_d p'$ by the lexicographic quasi order $\leq_{lex}$, which is also defined for vectors of different lengths. A prioritized active element $\mathit{elem}\langle op, \mu, p \rangle$ is an active element associated with a priority. We denote the set of all prioritized active elements by $\mathcal{A}^{\mathcal{P}}$. For two prioritized active elements $v = (op_v, \mu_v, p_v)$ and $w = (op_w, \mu_w, p_w)$, we define $v \leq w$ if $p_v \leq_{lex} p_w$

We define the type of prioritized worklists by $\mathcal{W}^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{A}^{\mathcal{P}*}$. We say that a prioritized worklist $\omega = e_1 \cdot \ldots \cdot e_k \in \mathcal{W}^{\mathcal{P}}$ is ordered according to a dynamic scheduling expression $d \in \mathit{dynSched}$, if $e_i \leq_d e_j$ implies $i \leq j$ for every $1 \leq i \leq j \leq k$. We write $\textsc{Priority}[\![d]\!]\,\omega = \omega'$ if ω' is a permutation of ω preserving the quasi order $\leq_d$. We define the following worklist operations for a dynamic scheduling expression d:

$$
\begin{array}{lcl}
\textsc{Empty} & \stackrel{\text{def}}{=} & \epsilon \ , \\
\textsc{Pop}\ \omega & \stackrel{\text{def}}{=} & (\mathit{head}(\omega), \mathit{tail}(\omega)) \ , \\
\textsc{Merge}[\![d]\!]\,(\omega, \delta) & \stackrel{\text{def}}{=} & \textsc{Priority}[\![d]\!]\,(\omega \cdot \delta) \ , \\
\textsc{Init}[\![d]\!]\,G & \stackrel{\text{def}}{=} & \textsc{Merge}[\![d]\!]\,(\epsilon, \textsc{Rdx}[\![op_{1..k}]\!]G)
\end{array}
$$

where $\mathit{head}(\omega)$ and $\mathit{tail}(\omega)$ refer to the first element and (possibly empty) suffix of a sequence, respectively.

2.2.3.3 Iteratively Executing Operators

Consider the set of program states $\Sigma \stackrel{\text{def}}{=} \mathit{Graph} \cup \mathit{Graph} \times \mathcal{W}^{\mathcal{P}}$. We will write $G + \mathit{Wl}$ to denote a graph-worklist pair. The meaning of statements is given in terms of a transition relation with two forms of transitions:

`iterate`$^{\text{init}}$ starts executing `iterate` e by initializing a worklist with the set of redexes found in G
$\langle \texttt{iterate}\ exp, G\rangle \Longrightarrow \langle \texttt{iterate}\ exp, G + Wl\rangle$ if
$Wl = \textsc{Init}[\![exp]\!]\, G$

`iterate`$^{\text{step}}$ executes an operator op_i in exp
$\langle \texttt{iterate}\ exp, G + Wl\rangle \Longrightarrow \langle \texttt{iterate}\ exp, G' + Wl''\rangle$ if

$$\begin{array}{lcl} (elem\langle op_i, \mu, p\rangle, Wl') & = & \textsc{Pop}\ Wl \\ G' & = & [\![op_i]\!](G, \mu) \\ \Delta & = & \textsc{Delta}[\![op_i, op_{1..k}]\!]\,(G, \mu) \\ Wl'' & = & \textsc{Merge}[\![exp]\!]\,(Wl', \Delta) \end{array}$$

`iterate`$^{\text{done}}$ returns the graph when no more operators can be scheduled
$\langle \texttt{iterate}\ exp, G + \textsc{Empty}\rangle \Longrightarrow G$

`foreach`$^{\text{init}}$, `foreach`$^{\text{done}}$ same rules as for `iterate`

`foreach`$^{\text{step}}$ executes an operator op_i in exp
$\langle \texttt{foreach}\ exp, G + Wl\rangle \Longrightarrow \langle \texttt{foreach}\ exp, G' + Wl'\rangle$ if

$$\begin{array}{lcl} (elem\langle op_i, \mu, p\rangle, Wl') & = & \textsc{Pop}\ Wl \\ G' & = & [\![op_i]\!](G, \mu) \end{array}$$

Figure 2.5: An operational semantics for Elixir statements.

1. $\langle S, \sigma\rangle \Longrightarrow \sigma'$, means that the statement S transforms the state σ into σ' and finishes executing;

2. $\langle S, \sigma\rangle \Longrightarrow \langle S', \sigma'\rangle$, means that the statement S transform the state σ into σ' to which the remaining statement S' should be applied.

The definition of $\Longrightarrow$ is given by the rules in Figure 2.5. The semantics induced by the transition relation yields (possibly infinite) sequences of states $\sigma_1, \ldots, \sigma_k, \ldots$. A correct parallel implementation gives the illusion that each transition occurs atomically, even though the executions of different transitions may interleave.

2.2.4 Semantics of Ordered Statements

Ordered statements have the form

`iterate` *opsExp* `>>` `metric` *exp* `>>` *statSched* .

The static scheduling expression *statSched* is the same as in the unordered case, except that we do not allow `unroll`. The expression *opsExp* is either a single operator *op* or a disjunction of operators $op_{1..k}$ having the same shape. If *opsExp* is a disjunction then it is followed by a `fuse` term.

Prioritized active elements are dynamically partitioned into equivalence classes C_i based on the value of *exp*. The execution then proceeds as follows: We start by processing active elements from the equivalence class C_0, which has the lowest priority. Applying an operator to active elements from C_i can produce new active elements at other priority levels, e.g., C_j. Once the work at priority level i is done we start processing work at the next level. We will restrict our attention to the class of algorithms where the priority of new active elements is greater than or equal to the priority of existing active elements ($i \leq j$). Under this restriction, we are guaranteed to never miss work as we process successive priority levels. The execution terminates when all work (at the highest priority level) is done. All the algorithms that we studied belong to this class. The above execution strategy admits a straightforward and efficient parallelization strategy: associate with each C_i a bucket B_i and process in parallel all work in bucket B_i before moving to B_{i+1}. This implements a generic "level-by-level" parallel execution strategy. Instances of this scheme

have been used to parallelize algorithms like breadth-first-search (e.g. [9]).

2.2.5 Using Strong Guards for Fixed-Point Detection

Our language allows defining `iterate` actions that do not terminate for all inputs. It is the responsibility of the programmer to avoid defining such actions. When an action does terminate for a given input, it is the responsibility of the compiler to ensure that the emitted code detects the fixed-point and stops.

Let μ be a matching and $D = \mu(G)$ be the matched subgraph. Further, let $G' = [\![op]\!](G, \mu)$. One way to check whether an operator application leads to a fixed-point is to check whether an operator has had an any effect, i.e., $Att'^D \stackrel{?}{=} Att^D$. This requires comparing the result of the operator application to a backup copy of D, created prior to its application. However, this approach is rather expensive. We opt for a more efficient alternative by placing a requirement on the guards of operators, as explained next.

Definition 2.2.7 (Strong Guard)**.** *We say that an operator op has a* strong guard *if for every matching μ, applying the operator disables the guard. That is, if $G' = [\![op]\!](G, \mu)$ then $(G', \mu) \not\models Gd^{op}$.*

A strong guard allows to check $(G, \mu) \not\models Gd^{op}$, which involves just reading the attributes of D and evaluating a Boolean expression.

Further, strong guards help us improve the precision of our incremental worklist maintenance by supplying more information to the automatic reason-

ing procedure, as explained in Section 2.4.3. Our compiler checks for strong guards at compile-time and signals an error to the programmer otherwise (see details in Section 2.4.3). In our experience, strong guards do not limit expressiveness. For efficiency, operators are usually written to act on a graph region in a single step, which leads to disabling their guard.

2.3 Combining Static and Dynamic Scheduling — the General Case

In this section, we present the semantics of static scheduling expressions and explain how to combine static and dynamic scheduling to execute statements.

2.3.1 Semantics of Static Operators

Let $statOp ::= opsExp >> statSched$ denote the language of static operator expressions.

Our next goal is to define the semantics of a static operator $\textsc{Static}[\![\cdot]\!] : statOp \to (Graph \times Match) \to Graph$. Intuitively, a static operator can be thought of as a small program where each scheduling term represents a nested loop (nesting level goes from right to left). A fused disjunction represents a loop executing its inner operators from left to right. Evaluation of a static operator on a given graph and matching is done by step-wise execution of these loops. Some of the steps create new redexes, which themselves require more steps, until no more redexes appear. We use two concepts to represent

these execution steps: *partially evaluated expressions*, or *pe-exprs* for short, and *extended active elements*. A pe-expr contains two types of *cursors*: $\bullet$, which appears to the right of a scheduling term or an operator and acts as a program counter; and $\circ$, which appears to the right of an `unroll` term to denote an unfinished execution of the corresponding loop. We define extended active elements, which generalize active elements, below.

Definition 2.3.1 (Extended Active Element)**.** *An extended active element, denoted by* $\mathit{elem}\langle e, \mu\rangle$*, pairs a pe-expr* $e \in \mathit{statOp}$ *with a matching* $\mu \in \mathit{Match}$*. Intuitively, it means that* e *is currently evaluating at* μ*. We denote the set of all extended active elements by* $\mathcal{E}$*.*

For a sequence of matchings $\overline{\mu} = \mu_1 \cdot \ldots \cdot \mu_k$ and expression $e \in \mathit{statOp}$, we define a constructor for a sequence of extended active elements as follows: $\mathit{elems}\langle e, \overline{\mu}\rangle \stackrel{\text{def}}{=} \mathit{elem}\langle e, \mu_1\rangle \cdot \ldots \cdot \mathit{elem}\langle e, \mu_k\rangle$.[3] We write $\mathit{Pat}(\mathit{opsExp})$ to denote the shape of the patterns appearing in the operators in opsExp (the shape should be equal for all of them).

We define the following helper operation. Let $R = R^{op}$ be the pattern of an operator op and $\overline{v} \subseteq V^R$ be a subset of the pattern nodes. We require $V^R \setminus \overline{v}$ be non-empty and induce a connected subgraph of R. We define the set of matchings $V^R \to G$ identifying with μ on the node variables $\overline{v}$ by

$$\begin{aligned} \textsc{Expand}[\![R, \overline{v}]\!](G, \mu) &\stackrel{\text{def}}{=} \{\mu' \in V^R \to V^G \mid \mu|_{\overline{v}} = \mu'|_{\overline{v}}\} \\ \textsc{Expand}[\![op, \overline{v}]\!](G, \mu) &\stackrel{\text{def}}{=} \textsc{Expand}[\![\mathit{Pat}(op), \overline{v}]\!](G, \mu) \ . \end{aligned}$$

[3] We write $\alpha \cdot \beta$ to denote the concatenation of two sequences α and β.

We lift the DELTA operation to disjunctions of general expressions:

$$\textsc{Delta}[\![op_{a..b}, op_{c..d}]\!]\,(G,\mu) \stackrel{\text{def}}{=} \bigcup_{a\le i\le b, c\le j\le d} \textsc{Delta}[\![op_i, op_j]\!]\,(G,\mu)\ .$$

The type $\mathcal{SW} \stackrel{\text{def}}{=} \mathcal{E}^*$, which we refer to as a *static worklist*, denotes all sequences of extended active elements. For a non-empty sequence $e \cdot s$ where s is possibly the empty sequence, we define $\mathit{head}(e \cdot s) = e$ and $\mathit{tail}(e \cdot s) = s$. With slight abuse of notation, we will often view a multiset as an arbitrary sequence containing the same elements, allowing the use of operations such as union. For a sequence ω and a multiset of extended active elements X, write $\omega \cup X$ to denote any sequence ω' containing all elements of ω and X.

We define the semantics of an extended active element for a graph $\mathcal{E}[\![\cdot]\!] : \mathcal{E} \to \mathit{Graph} \to (\mathit{Graph} \times \mathcal{SW})$ in Figure 2.6. An application of the function to a graph, $\mathcal{E}[\![e, \mu]\!]G = (G', \omega)$, returns the mutated graph G' along with any new extended active elements required for completing the execution of e.

We define the helper function *ExhaustStatic*, which accepts a graph and a static worklist, and iteratively applies each extended active element in the worklist to the graph until no new work is discovered. This process terminates as unrolling is performed for a fixed number of times. A static operator executes by starting with a static worklist containing a single element, which is constructed from the input expression and matching, and executing

$\mathcal{E}[\![elem\langle op\ \bullet \texttt{>>}s, \mu\rangle]\!]G \stackrel{\text{def}}{=}$
let $G' = [\![op]\!](G, \mu)$
in (G', ϵ)

$\mathcal{E}[\![elem\langle op\ \bullet \texttt{>>}b \circ \texttt{>>}s, \mu\rangle]\!]G \stackrel{\text{def}}{=}$
let $G' = [\![op]\!](G, \mu)$
$\overline{\delta} = \textsc{Delta}[\![op, op]\!]\,(G, \mu)$
$\overline{\omega} = elems\langle op\ \texttt{>>}b \bullet \texttt{>>}s, \overline{\delta}\rangle$
in $(G', \overline{\omega})$

$\mathcal{E}[\![elem\langle op_{1..k}\ \texttt{>>fuse}\ \bullet\ \texttt{>>}s, \mu\rangle]\!]G \stackrel{\text{def}}{=}$
let $\omega = elem\langle op_1 \bullet\ \textbf{or}\ op_{2..k}\ \texttt{>>fuse>>}s, \mu\rangle$
in (G, ω)

$\mathcal{E}[\![elem\langle op_{1..j} \bullet \textbf{or}\ op_{j+1..k}\ \texttt{>>fuse>>}s, \mu\rangle]\!]G \stackrel{\text{def}}{=}$
let $G' = [\![op_j]\!](G, \mu)$
$\omega = elem\langle op_{1..j+1} \bullet \textbf{or}\ op_{j+2..k}\ \texttt{>>fuse>>}s, \mu\rangle$
in (G', ω)

$\mathcal{E}[\![elem\langle op_{1..k} \bullet\ \texttt{>>fuse>>}s, \mu\rangle]\!]G \stackrel{\text{def}}{=}$
let $G' = [\![op_k]\!](G, \mu)$
in (G', ϵ)

$\mathcal{E}[\![elem\langle op_{1..k} \bullet\ \texttt{>>fuse>>}b \circ \texttt{>>}s, \mu\rangle]\!]G \stackrel{\text{def}}{=}$
let $G' = [\![op_k]\!](G, \mu)$
$\overline{\delta} = \textsc{Delta}[\![op_{1..k}, op_{1..k}]\!]\,(G, \mu)$
$\overline{\omega} = elems\langle op_{1..k}\ \texttt{>>fuse>>}b \bullet \texttt{>>}s, \overline{\delta}\rangle$
in $(G', \overline{\omega})$

$\mathcal{E}[\![elem\langle p\ \texttt{>>group}\ \overline{v} \bullet \texttt{>>}s, \mu\rangle]\!]G \stackrel{\text{def}}{=}$
let $R = Pat(p)$
$\overline{\alpha} = \textsc{Expand}[\![R, \overline{v}]\!](G, \mu)$
$\overline{\omega} = elems\langle p \bullet \texttt{>>group}\ \overline{v}\texttt{>>}s, \overline{\alpha}\rangle$
in $(G, \overline{\omega})$

$\mathcal{E}[\![elem\langle p\ \texttt{>>unroll}\ 0 \bullet\ \texttt{>>}s, \mu\rangle]\!]G \stackrel{\text{def}}{=} \mathcal{E}[\![elem\langle p \bullet\ \texttt{>>}s, \mu\rangle]\!]G$

For $k > 0$ and $k' = k - 1$
$\mathcal{E}[\![elem\langle p\ \texttt{>>unroll}\ k\bullet, \mu\rangle]\!]G \stackrel{\text{def}}{=}$
let $\omega = elem\langle p \bullet \texttt{>>unroll}\ k'\ \circ \texttt{>>unroll}\ 1, \mu\rangle$
in (G, ω)

Figure 2.6: Rules for evaluating extended active elements. The expression '$\texttt{>>}\,s$' is a possibly empty sequence of static scheduling terms and '$b \circ$' captures the leftmost appearance of a $\circ$.

ExhaustStatic to completion:

$$
\begin{array}{l}
\mathit{ExhaustStatic}(G, \omega) \stackrel{\text{def}}{=} \\
\quad \textbf{let} \quad e = \mathit{head}(\omega) \\
\qquad\quad (G', \Delta) = \mathcal{E}[\![e]\!]\, G \\
\qquad\quad \omega' = \mathit{tail}(\omega) \cup \Delta \\
\quad \textbf{in} \quad \textbf{if } \omega = \epsilon \textbf{ then } (G, \epsilon) \textbf{ else } \mathit{ExhaustStatic}(G', \omega') \ .
\end{array}
$$

$$
\begin{array}{l}
\textsc{Static}[\![e]\!](G, \mu) \stackrel{\text{def}}{=} \\
\quad \textbf{let} \quad (G', \epsilon) = \mathit{ExhaustStatic}(G, \mathit{elem}\langle e\bullet, \mu\rangle) \\
\quad \textbf{in} \quad G' \ .
\end{array}
$$

Example 2.3.2 (Static Operator Execution)**.** *We show an example execution of the static operator*

$$\mathit{relaxEdge} \texttt{ >> group } b \texttt{ >> unroll } 1$$

on the graph G shown in Figure 2.4b and the matching $\mu_{1,2} = \{a \mapsto n_1, b \mapsto n_2\}$ using the shorthand notation $\mu_{i,j}$ for the matching $\{a \mapsto n_i, b \mapsto n_j\}$.

The procedure Static *starts by passing the input graph G and the initial extended active element shown on the first line of Figure 2.7 to the ExhaustStatic procedure, which proceeds to evaluate the graph-worklist pairs until the worklist becomes empty. Figure 2.7 shows graph and static worklist after each step involving the evaluation of a single extended active element. We use a LIFO order for the union of a delta and the current worklist. Finally, the graph G_2 is returned as a result.*

2.3.2 Executing Statements with Static and Dynamic Scheduling

We now explain how to extend program states to support interleaved execution of static operators and multiple disjuncts (having separate static

Graph	**Static Worklist**
G	$elem\langle$relaxEdge >> `group` b >> `unroll` $1\bullet, \mu_{1,2}\rangle$
G	$elem\langle$relaxEdge >> `group` $b \bullet$ >> `unroll` $0 \circ$ >>`unroll` $1, \mu_{1,2}\rangle$
G	$elem\langle$relaxEdge $\bullet$ >> `group` b >> `unroll` $0 \circ$ >> `unroll` $1, \mu_{1,2}\rangle$
	$elem\langle$relaxEdge $\bullet$ >> `group` b >> `unroll` $0 \circ$ >> `unroll` $1, \mu_{1,3}\rangle$
$G_1 = G[n_2.dist = 1]$	$elem\langle$relaxEdge >> `group` b>> `unroll` $0 \bullet$ >> `unroll` $1, \mu_{2,3}\rangle$
	$elem\langle$relaxEdge $\bullet$ >> `group` b >> `unroll` $0 \circ$ >> `unroll` $1, \mu_{1,3}\rangle$
G_1	$elem\langle$relaxEdge >> `group` $b \bullet$ >> `unroll` $1, \mu_{2,3}\rangle$
	$elem\langle$relaxEdge $\bullet$ >> `group` b >> `unroll` $0 \circ$ >> `unroll` $1, \mu_{1,3}\rangle$
G_1	$elem\langle$relaxEdge $\bullet$ >> `group` b >> `unroll` $1, \mu_{2,3}\rangle$
	$elem\langle$relaxEdge $\bullet$ >> `group` b >> `unroll` $0 \circ$ >> `unroll` $1, \mu_{1,3}\rangle$
$G_2 = G_1[n_3.dist = 2]$	$elem\langle$relaxEdge $\bullet$ >> `group` b >> `unroll` $0 \circ$ >> `unroll` $1, \mu_{1,3}\rangle$
G_2	ϵ

Figure 2.7: A static operator evaluation example.

and dynamic scheduling expressions).

Intuitively, an `iterate` statement executes an unbounded number of static operator instances (i.e., a static operator together with a given matching) in parallel. This requires the ability to "pause" the execution of one static operator instance, in order to switch to (partially) executing another static operator instance, and then resume the execution of the paused static operator. The semantics of static operator is already geared towards this pause-resume mode of execution — a static worklist captures every intermediate state of the static operator execution. To express the interleaved execution of a set of static operators we maintain a multiset of "paused" static worklists of the type $Statics \stackrel{\text{def}}{=} \mathcal{SW}^*$.

To support multiple disjuncts we maintain a separate prioritized work-

list per disjunct via a map $\mathcal{DW}^{\mathcal{P}}$: *disjunctExp* $\rightarrow \mathcal{W}^{\mathcal{P}}$. We extend the operations defined in the previous sections as follows: EMPTY returns the empty map; POP arbitrarily chooses a disjunct d and pops the corresponding prioritized worklist $\mathcal{DW}^{\mathcal{P}}(d)$; MERGE merges elements, each belong to a given disjunct, to their corresponding prioritized worklist; and INIT initializes each prioritized worklist with the elements belonging to the corresponding disjunct.

We combine the two components defined above into a scheduler type: $\Theta : \mathcal{DW}^{\mathcal{P}} \times \mathit{Statics}$. We also define the operation ADD, which merges extended active elements into a given static worklist. The order of elements in the resulting static worklist is arbitrary.

We extend the DELTA operation to general expressions containing by removing the static scheduling sub-expressions.

We generalize program states by replacing worklists with schedulers: $\Sigma \stackrel{\text{def}}{=} \mathit{Graph} \cup \mathit{Graph} \times \Theta$. We denote such states by $G + \mathit{DWl} + \mathit{statics}$.

Figure 2.8 shows the semantics of `iterate` and `foreach` statements for general expressions.p

2.4 Synthesis

Elixir fuses together the operators and schedule and generates a program expressed in an imperative, high-level intermediate representation (HIR). Subsequently, it lowers the HIR code to parallel C++ code.

In this section, we explain how to emit HIR code to implement Elixir

$\texttt{iterate}^{\text{init}}$ starts executing $\texttt{iterate}\ e$ by initializing a scheduler with the set of redexes found in G

$\langle \texttt{iterate}\ exp, G\rangle \Longrightarrow \langle \texttt{iterate}\ exp, G + DWl + \epsilon\rangle$ if

$DWl = \textsc{Init}[\![exp]\!]\, G$

$\texttt{iterate}^{\text{schedule}}$ schedules a new static operator for execution

$\langle \texttt{iterate}\ exp, G + DWl + statics\rangle \Longrightarrow \langle \texttt{iterate}\ exp, G + DWl' + statics'\rangle$ if

$(DWl', elem\langle e, \mu\rangle) = \textsc{Pop}\ DWl$

$statics' = \textsc{Add}(statics, elem\langle e\bullet, \mu\rangle)$

$\texttt{iterate}^{\text{step}}$ advances an already executing static operator one step

$\langle \texttt{iterate}\ exp, G + DWl + statics\rangle \Longrightarrow \langle \texttt{iterate}\ exp, G' + DWl' + statics''\rangle$ if

$(statics', elem\langle e, \mu\rangle) = \textsc{Pop}\ statics$

$(G', \overline{\omega}) = \mathcal{E}[\![elem\langle e, \mu\rangle]\!]G$

$statics'' = \textsc{Add}(statics', \omega)$

$\Delta = \textsc{Delta}[\![e, exp]\!]\,(G, \mu)$

$DWl' = \textsc{Merge}[\![exp]\!]\,(DWl, \Delta)$

$\texttt{iterate}^{\text{done}}$ returns the graph when no more operators can be scheduled and all executing static operators have finished

$\langle \texttt{iterate}\ exp, G + \textsc{Empty}[\![exp]\!]\rangle \Longrightarrow G$

$\texttt{foreach}^{\text{init}}$, $\texttt{foreach}^{\text{done}}$, $\texttt{foreach}^{\text{schedule}}$ same rules as for $\texttt{iterate}$

$\texttt{iterate}^{\text{step}}$ advances a static operator one step

$\langle \texttt{foreach}\ exp, G + DWl + statics\rangle \Longrightarrow \langle \texttt{foreach}\ exp, G' + DWl + statics''\rangle$ if

$(statics', elem\langle e, \mu\rangle) = \textsc{Pop}\ statics$

$(G', \overline{\omega}) = \mathcal{E}[\![elem\langle e, \mu\rangle]\!]G$

$statics'' = \textsc{Add}(statics', \omega)$

Figure 2.8: An operational semantics for Elixir statements — the general case.

statements. In Chapter 3 we discuss how to synthesize parallel code from this HIR.

We use the notation $\mathsf{Code}(e)$ for the code fragment implementing the mathematical expression e in our HIR imperative language.

This section is organized as follows. First, we discuss our assumptions regarding the HIR language. Section 2.4.1 describes the synthesis of operator-related procedures. Section 2.4.2 describes the synthesis of the EXPAND operation, which is used to synthesize RDX and as a building block in synthesizing DELTA. Section 2.4.3 describes the synthesis of DELTA via automatic reasoning. Section 2.4.4 puts together the elements needed to synthesize unordered statements.

Implementation Language and Notational Conventions

We assume an HIR language containing standard constructs for sequencing, conditions, looping, and evaluation of arithmetic and Boolean expressions such as the ones used in Elixir. Operations on sets are realized via methods on set data structures. We assume that the language allows static typing by the notation $v : t$, meaning that variable v has type t. To promote succinctness, variables do not require declaration and come into scope upon initialization. We write $v_{i..j}$ to denote the sequence of variables $v_i, \ldots, v_j$. Record types are written as $\textbf{record}[f_{1..k}]$, meaning that an instance r of the record allows accessing the values of fields $f_{1..k}$, written as $r[f_i]$. We use static loops (loops preceded by the **static** keyword) to concisely denote loops over

a statically-known range, which the compiler unrolls, instantiating the induction variables in the loop body as needed. We similarly use static conditions. In defining procedures, we will use the notation $f[statArgs](dynArgs)$ to mean that f is specialized for the statically-given arguments *statArgs* and accepts at runtime the dynamic arguments *dynArgs*. We note that, since we assume a single graph instance, we will usually not explicitly include it in the generated code.

Graph Data Structure. We assume the availability of a graph data structure supporting methods for reading and updating attributes, and scanning the outgoing edges and incoming edges of a given node. The code statements corresponding to these methods are as follows. Let v_n and v_m be variables referencing the graph nodes n and m, respectively. Let a be a node attribute and b be an edge attribute. Let d_a and d_b be variables of the appropriate types for attributes a and b, respectively, having the values d and d', respectively.

- $d_a = \mathsf{get}(a, v_n)$ assigns $Att^G(a, n)$ to d_a and $d_b = \mathsf{get}(a, v_n, v_m)$ assigns $Att^G(b, n, m)$ to d_b.
- $\mathsf{set}(a, v_n, d_a)$ updates the value of the attribute a on the node n to d: $Att' = Att(a, n) \mapsto d$, and $\mathsf{set}(b, v_n, v_m, d_b)$ updates the value of the attribute b on the edge (n, m) to d': $Att' = Att(b, n, m) \mapsto d'$.
- $\mathsf{edge}(v_n, v_m)$ checks whether $(n, m) \in E^G$.
- nodes returns (an iterator to) the set of graph nodes V^G.

In addition, we require that the graph data structure be linearizable[4].

2.4.1 Synthesizing Atomic Operator Application

Let $op = [nt_{1..k}, et_{1..m}, bexp] \rightarrow [nUpd_{1..k}, eUpd_{1..m}]$ be an operator consisting of the following elements, for $i = 1..k$ and $j = 1..m$: (i) node attributes $nt_i = \texttt{nodes}(\texttt{node}\ n_i, a_i\ v_i)$; (ii) edge attributes $et_j = \texttt{edges}(\texttt{src}\ s_j, \texttt{dst}\ d_j, b_j\ w_j)$; (iii) a guard expression $bexp = op^{Gd}$; (iv) node updates $nUpd_i = v_i \mapsto nExp_i$; and (v) edge updates $eUpd_j = w_j \mapsto eExp_j$.

We note that in referring to pattern nodes the naming of variables n_i, s_j, d_j, etc. are insignificant in themselves, but rather stand for different ways of indexing the actual set of variable names. For example n_1 and s_2 may both stand for a variable '`a`'.

Figure 2.9 shows the codes we emit, as procedure definitions, for (a) evaluating an operator, (b) for checking a shape constraint, and (c) for checking a value constraint.

The procedure `apply` must use synchronization to ensure atomicity. There are multiple synchronization protocols one can implement to ensure atomic operator execution. For example, one can use speculative locking to ensure atomicity similar to what is supported by transactional memory systems. For problems like the single-source shortest path problem, which do not

[4]In practice, our graph implementation is optimized for action that don't mutate the graph structure. We rely on the locks acquired by the synthesized code to correctly synchronize concurrent accesses to graph attributes.

mutate the graph structure and which employ operators working over a small bounded number of nodes, one could simply acquire a spinlock on each node. We dub this protocol as *order-and-spin*.

The HIR that Elixir generates simply contains the outlines of atomic sections that must be executed transactionally in order to have correct parallel execution. In Chapter 3 we will discuss how one can encode different synchronization protocols as planning problems and use a planner to generate the right lock acquire and release instrumentation statements in order to implement these atomic sections. To improve clarity of exposition, in this section we present the synchronization instrumentation that is required for correct parallel execution inlined to the HIR code that Elixir generates.

The procedure apply first reads the nodes from the matching variable 'mu' into local variables. It then copies the variables to another set of variables used for locking. We assume a total order over all nodes, implemented by the procedure lock_less, which we use to ensure absence of deadlocks. The statement sort(lock_less, $\mathrm{lk}_{1..k}$) sorts the lock variables, i.e., swaps their values as needed, using the sort procedure. Next, the procedure acquires the locks in ascending order (we use spin locks), thus avoiding deadlocks. Then, the procedure reads the node and edge attributes from the graph and evaluates the guard. If the guard holds the update expressions are evaluated and used to update the attributes in the graph. Finally, the locks are released.

Since operators do not morph the graph checkShape does not require any synchronization. The procedure checkGuard is synchronized using the

same strategy as apply.

Figure 2.10 shows the code we emit for Code(⟦relaxEdge⟧).

2.4.2 Synthesizing Expand

We now define an operation EXPAND, which is used for implementing the `group` static scheduling term, RDX, and DELTA.

Let R be a pattern and $v_{1..m} \subseteq V^R$ and $v_{m+1..k} = V^R \setminus v_{1..m}$ be two complementing subsets of its nodes such that $v_{1..m}$ induces a connected subgraph of R. We define the set of matchings $V^R \to G$ identifying with μ on the node variables $\overline{v}$ by

$$\text{EXPAND}[\![op, \overline{v}]\!](G, \mu) \stackrel{\text{def}}{=} \{\mu' \in V^R \to V^G \mid \mu|_{\overline{v}} = \mu'|_{\overline{v}}\} \ .$$

We now explain how to synthesize a procedure that accepts a matching $\mu \in W \to V^G$, where W is any superset of $v_{1..m}$, and computes all matchings $\mu' \in V^R \to V^G$ such that $\mu(v_i) = \mu'(v_i)$ for $i = 1..m$.

We can bind the variables $v_{m+1..k}$ to graph nodes in different orders and using different methods of the graph API. For example, one can scan the entire set of graph nodes for each unbound pattern node and check whether it is a neighbor of the right set of bound nodes, according to the shape constraint. An alternative, and potentially more efficient way, is to choose an order that enables scanning the edges incident to nodes that are already bound. In this section we will create HIR code that is generic enough to support various implementation and scheduling alternatives. Later, in Chapter 3 we will show

```
def apply[op](mu : record[n_{1..k}]) =
  static for i = 1..k {n_i = mu[n_i]}
  if checkShape[op](mu)
    static for i = 1..k {lk_i = n_i}
    sort(lock_less, lk_1..k)
    static for i = 1..k {lock(lk_i)}
    static for i = 1..k {v_i = get(a_i, n_i)}
    static for j = 1..m {w_j = get(b_j, s_j, d_j)}
    if Code(bexp)
      static for i = 1..k {set(a_i, n_i, Code(nExp_i))}
      static for j = 1..m {set(b_j, s_j, d_j, Code(eExp_j))}
    static for i = 1..k {unlock(lk_i)}
```

(a) Code($[\![op]\!]$).

```
def checkShape[op](mu : record[n_{1..k}]) : bool =
  static for i = 1..k {n_i = mu[n_i]}
  // Now s_j and d_j correspond to μ(s_j) and μ(d_j).
  static for i = 1..k
    static for j = 1..k
      if n_i = n_j // Check if μ is one-to-one.
      return false
  static for j = 1..m
    if ¬edge(s_j, d_j) // Check for missing edges.
      return false
  return true
```

(b) Code($(G, \mu) \models R^{op}$).

```
def checkGuard[op](mu : record[n_{1..k}]) : bool =
  static for i = 1..k {n_i = mu[n_i]}
  static for i = 1..k {v_i = get(a_i, n_i)}
  static for j = 1..m {w_j = get(b_j, s_j, d_j)}
  return Code(bexp)
```

(c) Code($\mu(G, Gd^{op})$).

Figure 2.9: Operator-related procedures.

```
1  def apply[relaxEdge](mu : record[a, b]) =
2    a = mu[a]; b = mu[b];
3    if (a != b ∧ edge(a, b)) // Inline checkShape[relaxEdge](mu).
4      lk_1 = a; lk_2 = b;
5      if lock_less(lk_2, lk_1) // inline sort
6        swap(lk_1, lk_2);
7      lock(lk_1); lock(lk_2);
8      ad = get(dist, a); bd = get(dist, b);
9      w = get(wt, a, b);
10     if ad + w < bd // Inline checkGuard[relaxEdge](mu).
11       set(dist, b, ad + w);
12     unlock(lk_1); unlock(lk_2);
```

Figure 2.10: Code(⟦relaxEdge⟧).

how we can cast the problem of selecting a valid schedule of API calls to bind nodes as a planning problem. We represent each efficient binding order by a permutation of $v_{m+1..k}$, which we denote by $u_{m+1..k}$, and by an auxiliary sequence $T(R, v_{m+1..k}) = (u_{m+1}, w_{m+1}), \ldots, (u_k, w_k)$ where each tuple defines the connection between an unbound node u_j and a previously bound node w_j.

The procedure expand, shown in Figure 2.11, first updates μ' for $1..m$ and then uses $T(R, v_{m+1..k})$ to bind nodes $v_{m+1..k}$. Each node is bound to all possible values by a loop using the procedure expandEdge, which handles one tuple in (u_j, w_j). The loops are nested to enumerate over all combinations of bindings.

We note that a matching computed by the enumeration does not necessarily satisfy the shape constraints of R as some of the pattern nodes may be bound to the same graph node and not all edges in R may be present between the corresponding pairs of bound nodes. It is possible to filter out matchings that do not satisfy the shape constraint or guard via checkShape

```
def expand[op, v_{1..m}, T : record[n_{m+1..k}]]
            (mu : record[n_{1..k}],
             f : record[n_{1..k}] ⇒ Unit) =
  mu' = record[n_{1..k}] // Holds the expanded matching.
  static for i = 1..m {mu'[v_i] = mu[v_i]}
  expandEdge[m + 1,
    expandEdge[m + 2,
    ...
      expandEdge[k, f(mu')]...]

  // Inner function.
  def expandEdge[i, code] =
    [u_i, w_i] = T[i] // u_i is unbound and w_i is bound.
    for s ∈ nodes
      mu'[u_i] = s
      if Edge(w_i, u_i)
      code // inline code
```

Figure 2.11: Code applying function f to matchings in $\textsc{Expand}[\![op, v_{1..m}]\!](G, \mu)$

and checkGuard, respectively.

We use expand to define $\mathsf{Code}(\textsc{Rdx}[\![op]\!](G, \mu))$ in Figure 2.12.

```
def redexes[op](f : record[n_{1..k}] ⇒ Unit) =
  for v ∈ nodes
    mu = record[n_1]
    expand[op, n_1, T(R, n_{2..k})](mu, f')

  def f'(mu : record[n_{1..k}]) =
    if checkShape[op](mu) ∧ checkGuard[op](mu)
      f(mu)
```

Figure 2.12: Code for computing $\textsc{Rdx}[\![op]\!](G, \mu)$ and applying a function f to each redex.

2.4.3 Synthesizing Delta via Automatic Reasoning

We now explain how to automatically obtain an overapproximation of $\textsc{Delta}[\![op, op']\!]\,(G, \mu)$ for any two operators $op = [R, Gd] \to [Upd]$ and $op' = [R', Gd'] \to [Upd']$ and a matching μ, and how to emit the corresponding code.

The definition of Delta given in Section 2.2 is global in the sense that it requires searching for redexes in the entire graph, which is too inefficient. We observe that we can redefine Delta by localizing it to a subgraph affected by the application of the operator op, as we explain next.

For the rest of this subsection, we will associate a matching μ with the corresponding redex pattern R using the notational convention μ_R.

Let μ_R and $\mu_{R'}$ be two matchings corresponding to the operators above. We say that μ_R and $\mu_{R'}$ *overlap*, written $\mu_R \curlywedge \mu_{R'}$, if the matched subgraphs overlap: $\mu_R(V^R) \cap \mu_{R'}(V^{R'}) \neq \emptyset$. Then, the following equality holds:

$$
\begin{array}{l}
\textsc{Delta}[\![op, op']\!]\,(G, \mu_R) = \\
\quad \textbf{let} \quad G' = [\![op]\!](G, \mu_R) \\
\quad \textbf{in} \quad \{\mu_{R'} \mid \mu_{R'} \curlywedge \mu_R, \\
\qquad\qquad\quad (G, \mu_{R'}) \not\models R', Gd', \\
\qquad\qquad\quad (G', \mu_{R'}) \models R', Gd' \quad .
\end{array}
$$

We note that any overapproximation of Delta can be used in correctly computing the operational semantics of an `iterate` statement. However, tighter approximations lead to reduction in useless work. We proceed by developing an overapproximation of the local definition of Delta.

Given a matching μ_R, the set of overlapping matchings $\mu_{R'}$ can be classified into statically-defined equivalence classes, defined as follows. If $\mu_{R'} \curlywedge \mu_R$ then the overlap between $\mu_R(V^R)$ and $\mu_{R'}(V^{R'})$ induces a partial function $\rho : V^R \rightharpoonup V^{R'}$ defined as $\rho(x) = x'$ if $\mu_R(x) = \mu_{R'}(x')$. We call the function ρ the *influence function* of R and R' and denote the domain and range of ρ by ρ_{dom} and ρ_{range}, respectively. Two matchings $\mu^1_{R'}$ and $\mu^2_{R'}$ are equivalent if they induce the same influence function ρ. We can compute the equivalence class $[\rho]$ of an influence function ρ by

$$[\rho] = \textsc{Expand}[\![op', \rho_{range}]\!](G, \mu_R) \ .$$

Let $\mathit{infs}(op, op') = \rho_{1..k}$ denote the influence functions for the redex patterns R and R'. We define the function $\mathit{shift} : \mathit{Match} \times (V^R \to V^{R'}) \to \mathit{Match}$, which accepts a matching μ_R and an influence function ρ and returns the part of a matching $\mu_{R'}$ restricted to ρ_{range}:

$$\mathit{shift}(\mu_R, \rho) \stackrel{\text{def}}{=} \{(\rho(x), \mu_R(x)) \mid x \in \rho_{dom}\} \ .$$

The first overapproximation we obtain is

$$\textsc{Delta1}[\![op, op']\!]\,(G, \mu_R) \stackrel{\text{def}}{=} \bigcup_{\rho \in \mathit{infs}(op, op')} \textsc{Expand}[\![op', \rho_{range}]\!](G, \mathit{shift}(\mu_R, \rho)) \ .$$

A straightforward way to obtain a tighter approximation is to filter out matchings not satisfying the shape and value constraints of op'.

We say that an influence function ρ is *useless* if for all graphs G and all matchings $\mu_{R'}$ the following holds: for $G' = [\![op]\!](G, \mu_R)$ either $(G, \mu_{R'}) \models$

$R^{op'}, Gd^{op'}$, meaning that an active element $elem\langle op', \mu_{R'}\rangle$ has already been scheduled, or $(G', \mu_{R'}) \not\models R^{op'}, Gd^{op'}$, meaning that the application of op to (G, μ_R) does not modify the graph in a way that makes $\mu_{R'}(G')$ a redex of op'. Otherwise we say that ρ is *useful*. We denote the set of useful influence functions by $useInfs(op, op')$. We can obtain a tighter approximation $\textsc{Delta2}[\![op, op']\!]\,(G, \mu_R)$ via useful influence functions:

$$\textsc{Delta2}[\![op, op']\!]\,(G, \mu_R) \stackrel{\text{def}}{=} \bigcup_{\rho \in useInfs(op, op')} \textsc{Expand}[\![op', \rho_{range}]\!](G, shift(\mu_R, \rho)) \ .$$

We use automated reasoning to find the set of useful influence functions.

Influence Patterns. For every influence function ρ, we define an *influence pattern* and construct it as follows.

1. Start with the redex pattern R^{op} and a copy R' of $R^{op'}$ where all variables have been renamed to fresh names.

2. Identify a node variable $x \in V^R$ with a node variable $\rho(x)$ and rename node attribute variables in R' to the variables used in the corresponding nodes of R. Similarly rename edges attributes for identified edges.

Example 2.4.1 (Influence Patterns)**.** *Figure 2.14 shows the six influence patterns for operator relaxEdge (for now, ignore the text below the graphs). Here R^{op} consists of the nodes* `a` *and* `b` *(and the connecting edge) and R' consists of the nodes* `c` *and* `d` *(and the connecting edge). We display identified nodes by listing both variables inside the node ellipse.*

```
1 assume (Gd)
2 assume !(Gd') // comment out if identity pattern
3 update(Upd)
4 assert !(Gd')
```

Figure 2.13: Operator Delta Query.

Intuitively, the patterns determine that candidate redexes are one of the following types: a successor edge of `b`*, a successor edge of* `a`*, a predecessor edge of* `a`*, a predecessor edge of* `b`*, an edge from* `b` *to* `a`*, and the edge from* `a` *to* `b` *itself.*

Query Programs. To detect useless influence functions, we generate a straight-line program over the variables of the corresponding influence pattern, as shown in Figure 2.13.

Intuitively, the program constructs the following verification condition: (1) if the guard of R, Gd, holds; and (2) the guard of R', Gd', does not hold; and (3) the update Upd assign new values; then (4) the guard Gd' does not hold for the updated values. Proving the assertion means that the corresponding influence function is useless.

The case of $op = op'$ and the identity influence function is special. The compiler needs to check whether the guard is strong, and otherwise emit an error message. This is done by constructing a query program where the second `assume` statement is removed.

We pass these programs to a program verification tool (we use Boo-

gie [10] and Z3 [35]) asking it to prove the last assertion. This amounts to checking satisfiability of a propositional formula over the theories corresponding to the attributes types in our language — integer arithmetic and set theory. When the verifier is able to prove the assertion, we remove the corresponding influence function. If the verifier cannot prove the assertion or a timeout is reached, we conservatively consider the function as useful.

Example 2.4.2 (Query Programs)**.** *Figure 2.14 shows the query programs generated by the compiler for each influence pattern. Out of the six influence patterns, the verifier is able to rule out all except (a) and (e), which together represent the edges outgoing from the destination node, with the special case where an outgoing edge links back to the source node. Also, the verifier is able to prove that the guard is strong for (f).* ***This results with the tightest approximation of*** **Delta.**

We note that if the user specifies positive edges weights (`weight : unsigned int`*) then case (e) is discovered to be spurious.*

Figure 2.15 shows the code we emit for DELTA^2. We represent influence functions by appropriate records.

2.4.4 Synthesizing Unordered Statements

We implement the operational semantics defined in Section 2.2.3.3 by utilizing the Galois system runtime, which enables us to: (i) automatically construct a concurrent worklist from a dynamic scheduling expression, and

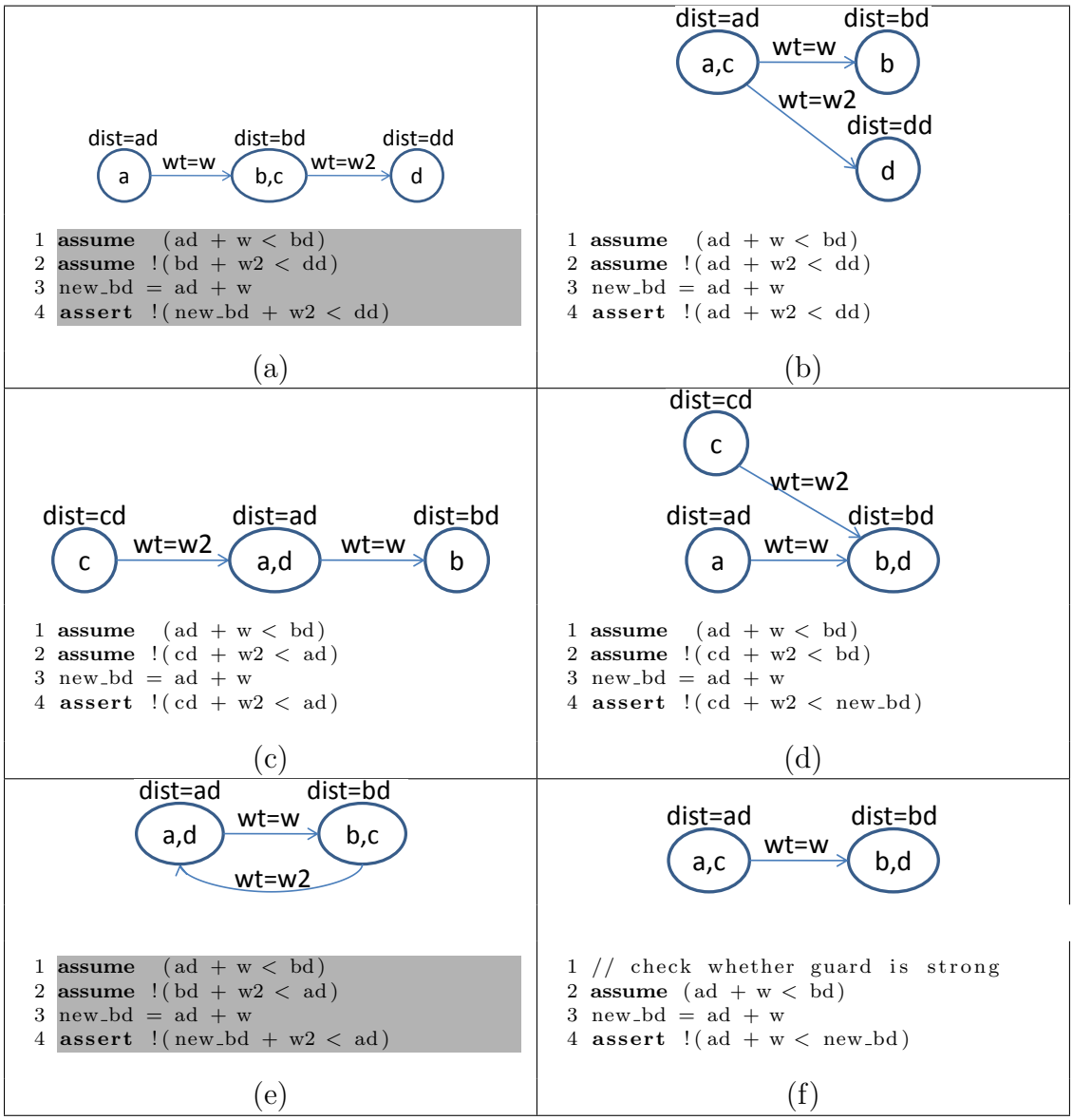


Figure 2.14: Influence patterns and corresponding query programs for relaxEdge; (b), (c), (d), and (f) are spurious patterns.

```
def delta2[op, op′, ρ_{1..q}](mu : record[n_{1..k}],
                              f : record[n_{1..k}] ⇒ Unit) =
  static for i = 1..q // For each useful influence function.
    mu′ = record[n_{1..k}]
    // Assume len(ρ_i) = |ρ_{i dom}|.
    for j = 1..len(ρ_i) // Initialize mu′ for ρ_{i dom}.
      mu′[ρ(n_j)] = mu[n_j]
    expand[op′, ρ_{i range}, T(R′, V^{R′} \ ρ_{i range})](mu′, f′)

  def f′(mu : record[n_{1..k}]) =
    if checkShape[op′](mu) ∧ checkGuard[op′](mu)
      f(mu)
```

Figure 2.15: Code for computing DELTA2$[\![op, op']\!]\,(G, \mu)$ and applying a function f to each matching.

```
def execute[iterate exp] =
  // Initialize worklist.
  wl : Wl[exp]()
  for i = 1..k { redexes[op_i](addToWl) }
  foreach e in wl // Parallel loop via a Galois unordered iterator.
    // e = elem⟨op, μ⟩
    Code([[op]])
    static for i = 1..k
      delta2[op, op_i, useInfs(op, op_i)](μ, addToWl)

  def addToWl(mu) = wl.add(mu)
```

Figure 2.16: Code($[\![$`iterate` $exp]\!]$)

(ii) process the elements in the worklist in parallel by a given function. We use the latter capability by passing the code we synthesize for operator application followed by the code for DELTA2$[\![op, op']\!]\,(G, \mu)$, which inserts the found elements to the worklist for further processing. The code we emit is shown in Figure 2.16.

```
1 assume (Gd)
2 pr1 = pr(op')
3 update(Upd)
4 assume (Gd')
5 assert pr1 = pr(op')
```

Figure 2.17: Ordered Operator Delta Query.

2.4.5 Synthesizing Ordered Statements

In this section we discuss extra reasoning is performed by Elixir for the case of ordered statements.

2.4.5.1 Synthesizing Ordered Delta

For the case of ordered statements we augment the delta computation analysis to also identify cases where an application of an operator $op = [R, Gd] \rightarrow [Upd]$ simply changes the priority of $op' = [R', Gd'] \rightarrow [Upd']$. As mentioned in Section 2.2.4, execution proceeds by exploring the different priority classes in a strict level-by-level order. Thus, whenever op can change the priority of op', we should consider op' as part of the delta of op. The query in Figure 2.17 identifies influence patterns where the priority of op' remains the same and therefore no scheduling of op' is necessary. The set of useless influence patterns consists of all cases for which we can prove assertions of both query programs from Figure 2.13 and Figure 2.17.

2.4.5.2 Customizing Ordered Parallelization

Certain algorithms, such as [87, 8], have additional properties that enable optimizations over the baseline ordered parallelization scheme discussed in Section 2.2.4. For example, in the case of Breadth-First-Search (`BFS`), one can show that when processing work at priority level i, all new work is at priority level $i + 1$. This allows us to optimize the implementation to contain only two buckets: B_c that holds work items at the current priority level, and B_n that holds work items at the next priority level. Hence, we can avoid the overheads associated with the generic scheme, which supports an unbounded number of buckets. Additionally, since B_c is effectively read-only when operating on work at level i, we can exploit this to synthesize efficient load-balancing schemes when distributing the work contained in B_i to the worker threads. Currently Elixir uses these two insights to synthesize specialized dynamic schedulers (using utilities from the OpenMP library) for problems such as breadth-first search.

Automating the Optimizations We now discuss how we use automated reasoning to enable the above optimizations. What we aim to show is that if the priority of active elements at the current level is an arbitrary value k, then all new active elements have the same priority $k + s$, where $s \geq 1$. We heuristically guess values of s by taking all constant numeric values appearing in the program $s = C_1, \ldots, C_n$. We illustrate this process for the `BFS` example. In the case of `BFS` the worklist delta consists only of a case similar to that of

```
1 assume (ad == k)
2 assume (s == C_i)
3 assume (ad + 1 < bd)
4 new_bd = ad + 1
5 assume (cd == new_bd)
6 assume (cd + 1 < dd)
7 assert (ad == k & new_bd == k + s)
```

Figure 2.18: Query program to enable leveled worklist optimization. `C_i` stands for a heuristically guessed value of s.

Figure 2.14(a) with all weights equal to one. The query program we construct is shown in Figure 2.18. The program checks that the difference between the priority of the shape resulting from the operator application and the shape prior to the operator application is an existentially quantified positive constant. Additionally, we must guarantee that when we initialize the worklist all work is at the same priority level. Our compiler emits a simple check on the priority value on each item inserted in the worklist during initialization to guarantee that this condition is satisfied.

Chapter 3

Planning-based Synthesis

How can one effectively transform a program expressed in a high level language to an equivalent and efficient lower level program? This question, which is as old as the first compilers, is the subject of this chapter [1]. As we discussed in the previous chapter, the Elixir front-end starts with a non-deterministic algorithm specification and after computing the operator delta fuses the operators with the schedule to create a high-level, incremental algorithm. This high-level algorithm is a description of the problem that is much closer to conventional imperative code, and it can be expressed as a program in a high-level intermediate representation (HIR), similar to the ones we have in a traditional compiler. The next goal of Elixir is to transform this HIR program to an efficient low-level intermediate representation (LIR), and finally generate explicitly parallel C++ code.

We identify three key problems that need to be solved in order to

[1] Part of the work presented in this chapter has appeared in *"Dimitrios Prountzos, Roman Manevich, Keshav Pingali. 'Synthesizing Parallel Graph Programs via Automated Planning'. In Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI) 2015."* The first author is responsible for the conception and the implementation of the ideas presented in this publication. Additional authors provided assistance with the presentation of the material.

generate efficient parallel code from this HIR:

1. Inserting synchronization to ensure atomic operator execution.
2. Finding a good schedule of HIR statements.
3. Selecting efficient implementations of HIR statements.

One design strategy for the compiler is to implement separate compiler phases for each problem, but this introduces the familiar *phase ordering problem* that prevents generating high-quality code for many problems. *Integrated compilation* tackles phase-ordering by combining compiler phases and *simultaneously* solving them. However, it is unclear how to perform integrated compilation encompassing the above-mentioned tasks. The main contributions of the work discussed in this chapter are:

- We present a framework using a novel approach based on *automated planning* (Section 3.3) for synthesizing efficient parallel code. Our framework (Section 3.4) encodes individual tasks via constraints and can use an off-the-shelf planner to *simultaneously* solve them. This enables the first integrated compilation approach for tasks such as scheduling and synchronization. Moreover, our framework is parametric in the input language and can be potentially applied to other compilation problems.
- We instantiate our framework with Elixir and use it synthesize parallel code for a number of challenging graph problems. To the best of our knowledge, this is the first time that parallel solutions were automatically

synthesized for problems of this complexity. We automatically explore various scheduling, implementation-selection, and synchronization policies that capture algorithmic and implementation insights, such as using efficient iteration patterns based on graph data structure properties.

- We present *Aliasing tracking-based synchronization* (ATS), a novel speculative locking protocol that is synthesized by our system, and provides custom synchronization for each operator (Section 3.5). ATS can handle operators working on an unbounded neighborhood and can outperform generic runtime-based solutions for speculation.

The rest of this chapter is organized as follows: Section 3.1 informally discusses the synthesis challenges and motivates our solutions through a series of examples. Section 3.2 presents informally our planning-based synthesis framework. Section 3.3 gives necessary background on automated planning and defines various planning mechanisms used by our framework. Section 3.4 and Section 3.5 discuss planning-based synthesis in detail, both for the serial and the parallel setting. Finally, Section 3.6 presents we instantiate our framework inside Elixir.

3.1 Generating Parallel Code: Challenges

In this section we present a number of challenges that must be addressed in order to generate efficient parallel code from an HIR program. We use two running examples to illustrate different aspects of the HIR and motivate different challenges. In Section 3.1.1 we disccuss the *triangle counting*

(*Triangles*) problem, while in Section 3.1.2 we present the *maximal independent set* problem (`MIS`). In Section 3.1.3 we discuss three major problems that must be solved to produce efficient parallel code from HIR.

3.1.1 Example: *Triangles*

Figure 3.1(a) illustrates an Elixir program for counting the number of triangles (cycles of exactly three edges) in an undirected graph. The program consists of the following elements:

Data-Structures: A global integer variable (line 1) stores the number of counted triangles. A graph (lines 3,4) consists of two relations, one for nodes and one for edges. Each node is implicitly given a unique number. The relations also specify attributes for nodes and edges; in this problem there are none.

Operators: The `countTriangle` operator (lines 6–14) defines a pattern, where three edges over distinct nodes connect in a cycle and their end-points are ordered (the latter condition is used to avoid counting the same triangle more than once). When such a subgraph is detected, the right-hand side increments the triangle counter.

Schedule: A `foreach` statement (line 16) computes the set of redexes, and applies the count operator once to each redex, in some arbitrary order. It is possible to define, via an optional scheduling tactic (≫ `sched`), the order in which redexes are dispatched in order to achieve greater rates

```
1  "Int counter = 0;"

3  Graph [ nodes(node : Node)
4          edges(src : Node, dst : Node) ]

6  countTriangle(a, b, c) = [
7   distinct({a, b, c}) // Automatically inserted
8    edges(a, b)
9    edges(b, c)
10   edges(c, a)
11   (a < b ∧ b < c ∧ a < c)
12 ] → [
13   "counter++;"
14 ]

16 countTriangles = foreach countTriangle ≫ sched

18 tactic edges: sortedIncreasing
```

(a)

Algorithm	Schedule specification
A-start	1 sched = **group** b,c
B-start	1 sched = **group** a,c
C-start	1 sched = **group** a,b

(b)

Figure 3.1: Elixir program for triangle counting: (a) High-level program parameterized by a scheduling specification; (b) Three scheduling specifications;

of convergence and to tune performance. In this example, it may be beneficial, in order to exploit locality, to co-schedule for each node a, all redexes over its neighbors b and c. To achieve this, we use the scheduling tactic `group b,c`. This tactic specifies that the pattern nodes b, c should be matched in all possible ways, thus forming a composite rewrite rule that groups together individual redex applications. In Figure 3.1(b) we present three different ways of using `group`. These give rise to three different algorithm variants, which start from a different initial node x (a partial redex) and find all possible completions for that redex.

Figure 3.2(a) shows the program `TrH`, which implements this specification. This program, written in pseudo-code corresponding to the Elixir HIR, iterates over all node triplets (a,b,c); for each triplet, it increments the counter if there is an edge between every ordered pair of nodes. The graph ADT needs to support a method **edges**(a,b) that checks whether an edge exists between nodes a and b.

3.1.2 Example: Maximal Independent Set

A *maximal independent set* of nodes in an undirected graph $G = (V, E)$ is a set $S \subseteq V$ such that a node is in S iff its immediate neighbors are not in S. Figure 3.3(a) shows an Elixir program for computing `MIS`. For `MIS`, the graph (lines 2-4) is described by two relations, one for nodes and one for edges. Each node has a *status* attribute, which is initialized to *Unmatched*. The algorithm sets this attribute to *Matched* if the node is added to S and to *NMatched* if

```
1  TrH =
2   for a : nodes do
3    for b : nodes do
4     for c : nodes do
5      if  edges(a, b)
6       if  edges(b, c)
7        if  edges(c, a)
8         if  a < b
9          if  b < c
10          if  a < c
11           "counter++;"
12          fi
13         fi
14        fi
15       fi
16      fi
17     fi
18    od
19   od
20  od
```

(a)

```
1  TrH1 =
2   for a : nodes do
3    for b : nodes do
4     if  a < b
5      if  edges(a, b)
6       for c : nodes do
7        if  b < c
8         if  a < c
9          if  edges(b, c)
10          if  edges(c, a)
11           "counter++;"
12          fi
13         fi
14        fi
15       fi
16      od
17     fi
18    fi
19   od
20  od
```

(b)

```
1  TrH2 =
2   for a : nodes do
3    for b : nodes do
4     if  edges(a, b)
5      if  a < b
6       for c : nodes do
7        if  edges(b, c)
8         if  b < c
9          if  a < c
10          if  edges(c, a)
11           "counter++;"
12          fi
13         fi
14        fi
15       fi
16      od
17     fi
18    fi
19   od
20  od
```

(c)

Figure 3.2: *Triangles* HIR programs: (a) Program corresponding to the specification in Figure 3.1 and `sched = group b, c`; (b) Optimally rescheduled HIR program; (c) Alternative HIR program.

one of its neighbors is added to S. The *match* operator (lines 6-11) defines a graph rewrite rule. The left-hand side, called the *operator guard*, defines a predicated subgraph pattern in which node a is *Unmatched* and none of its neighbors, $\mathcal{N}$(a), is *Matched*. When such a subgraph, dubbed as a *redex*, is detected, the right-hand side of the rewrite rule is executed, which adds a to S and updates the status of a and its neighbors appropriately. Elixir provides special looping instructions to encode operators working on an unbounded number of nodes. The $\forall$ predicate (line 9) holds if $sb \neq Matched$ is true for all $b \in \mathcal{N}$(a). The *map* instruction (line 11) updates to *NMatched* the status of every neighbor of a distinct from a (nodes may have self-loops). Finally, a `foreach` statement (line 12) computes the initial set of redexes, and applies *match* once to each redex, in some non-deterministic order. Optionally, a scheduling tactic could be used to refine the execution order of redexes.

Figure 3.3(b) shows `misH`, the HIR for implementing the specification. This program expresses the execution of multiple operators according to the schedule. For `MIS` this is straightforward: the '**for** a : **nodes do**' statement (line 1) iterates over all nodes and executes *match* transactionally for each a. Similar to the triangle counting problem, the use of the `foreach` schedule directive indicates that no operator delta should be considered.

3.1.3 Producing Efficient Parallel Code from HIR code

Even the generation of efficient *sequential* implementations from Elixir programs is very challenging since it requires solving two difficult tasks: (i)

(a) ***Elixir specification***

```
1  Graph[
2   nodes(n : Node, status:int)
3   edges(src:Node, dst:Node)
4  ]

6  match(a) =
7  [ nodes(a, sa)
8   sa = Unmatched ∧
9   ∀ b. {edges(a, b) nodes(b, sb) :
            sb ≠ Matched }
   ]→
10 [sa := Matched;
11  map { edges(a, c) nodes(c, sc) :
       c ≠ a :
       sc := NMatched };
   ]
12 mis = foreach match
```

(b) ***misH***

```
1 for a : nodes do
2  var sa := status(a);
3  if  sa = Unmatched
4   if forall b : 𝒩(a) {
       status(b) ≠ Matched }
5    status(a) := Matched;
6    map c : 𝒩(a) { c ≠ a :
       status(c) := NMatched };
7   fi
8  fi
9 od
```

(c) ***misV1***

```
1  for a : nodes do
2   lock a ctx ∅;
3   var sa := status(a);
4   if  sa = Unmatched
5    lock 𝒩(a) ctx a;
6    if forall b : 𝒩(a) {
        status(b) ≠ Matched }
7     status(a) := Matched;
8     map c : 𝒩(a) { c ≠ a :
        status(c) := NMatched };
9     unlock a,𝒩(a);
10   else unlock a,𝒩(a) fi //forall b
11  else unlock a fi //sa=Unmatched
12 od
```

(d) ***misV2***

```
1  for a : nodes do
2   lock a ctx ∅;
3   var sa := status(a);
4   if  sa = Unmatched
5    if forall b : 𝒩(a) {
       status(b) ≠ Matched } with
           𝒩(a) ctx a
6     status(a) := Matched;
7     mapAndUnlock c : 𝒩(a) {
        c ≠ a : status(c) := NMatched
       } with a,𝒩(a) ctx a,𝒩(a);
8    fiUnlock 𝒩(a), a ctx 𝒩(a), a
9   else unlock a fi
10 od
```

Figure 3.3: (a) MIS Elixir specification; (b) HIR; (c) ATS-instrumented HIR; (d) Alternative ATS-instrumented variant.

finding a good schedule of HIR statements, and (ii) selecting efficient implementations of HIR statements (in the context of conventional compilers, analogs of these tasks are instruction scheduling and instruction selection). To generate efficient parallel code, we also have to insert synchronization to ensure that operator execution is transactional. We argue that unless all three problems are solved simultaneously, there is a phase-ordering problem that prevents the generation of efficient parallel code.

Scheduling of HIR Statements Intuitively conjunctions, disjunctions, nested node iterators, and invariant predicates within node iterators in Elixir programs give rise to opportunities for scheduling HIR statements in different orders, and some orders may be far more efficient than others.

A simple example is an Elixir guard $p_1(a) \wedge p_2(b)$ (for example, lines 8-9 of Figure 3.3(a)), which can be implemented by HIR of the form $if\ p_1(a)\ if\ p_2(b)$.... or of the form $if\ p_2(b)\ if\ p_1(a)$.... Depending on the selectivity of the predicates, one order may be more efficient than the other.

A more important scheduling opportunity arises from invariant predicates within node iterators. `TrH` and `TrH1` in Figure 3.2 show an example. Since the predicates $a<b$ and **edges**(a, b) are invariant within the '**for** c : **nodes do**' loop, they can be lifted out and the execution of the loop can be made conditional on these predicates as shown in `TrH1`. In a sparse graph, the predicate **edges**(a,b) is false for most pairs of nodes (a,b), so the optimized code is far more efficient than the original code. Even for a very dense graph, executing

(a)

```
1  TrL1 =
2   for a : nodes do
3    for b : Succ(a) do
4     if a < b
5      for c : Succ(b) do
6       if b < c
7        if a < c
8         if edges(c, a)
9          "counter++;"
10        fi
11       fi
12      fi
13     odSucc
14    fi
15   odSucc
16  od
```

(b)

```
1  TrL2 =
2   for a : nodes do
3    for b : sortedLTSucc(a) do
4     for c : sortedLTSucc(b) do
5      if a < c
6       if sortedEdges(c, a)
7        "counter++;"
8       fi
9      fi
10    odLTSucc
11   odLTSucc
12  od
```

Tiles
tile(**for** b : Succ(a) **do**) = **for** b : **nodes do**, **if** edges(a, b)
tile(**for** b : sortedLTSucc(a) **do**) = **for** b : **nodes do**, **if** edges(a, b), **if** $a < c$

Figure 3.4: *Triangles* LIR programs: (a) LIR program using the successors tile; (b) LIR program under different tiling.

the c loop conditionally depending on the predicate (a<b) will halve the total execution time. Note that these kinds of transformations are well beyond the capabilities of conventional loop invariant removal algorithms [4] since these algorithms only move invariant computations out of loops, and cannot make the execution of a loop dependent on the value of an invariant predicate within it.

Implementation Selection It may be possible to improve performance by exploiting how the graph is stored in memory. A common representation for sparse graphs is the Compressed Sparse Row (CSR) format which permits indexed access to the neighbors of a node. For this format, the HIR code pattern '**for** b : **nodes do if edges**(a,b)...' can be implemented more efficiently by the code '**for** b : **Succ**(a) **do**...', where **Succ**(a) are the successors of node `a`, leading to the code `TrL1` in Figure 3.4(a). Note that to obtain `TrL1` from `TrH1`, it is necessary to reschedule `TrH1` to obtain the code `TrH2` shown in Figure 3.2(c), and then detect the efficient iteration pattern supported by CSR.

We will call this kind of pattern matching and replacement *tiling* since it is similar to the tiling approach to instruction selection in retargetable compilers. The synthesis methodology that we describe in this chapter is parameterized by a set of tiles, which represent, among other things, efficient iteration patterns of this sort that are supported by the graph representations used with the generated code. For example, assume that node successors are sorted in increasing order. Then, instead of linearly scanning all of a's neighbors b in the range $[first, last)$ and checking whether $a < b$ for each b, we can use a custom iterator '**for** b : **sortedLTSucc**(a) **do**' that initially performs binary search to find the first element $b_{first} : a < b_{first}$, and then linearly scans all nodes in $[b_{first}, last)$, which definitely satisfy this constraint. Figure 3.4(b) shows the implementation `TrL2` exploiting this property, as well as the tiles that lead to implementations such as `TrL1` and `TrL2`.

Synchronization Producing parallel code adds extra complexity to code generation, since it is necessary to insert locking code to ensure transactional execution of the operators. Transactional execution can be achieved using synchronization protocols such as *order-and-spin* locking or *speculative locking*.

Order-and-spin locking associates an exclusive spin-lock with each node. A total order $\prec$ is imposed on all nodes, and locks on nodes must be acquired in this order, so as to avoid deadlock. This can be achieved by sorting all nodes accessed by the threads according to $\prec$ before acquiring any lock. This approach is attractive for problems with operators working over a bounded number of nodes, since the sorting can be customized and inlined inside the operator. For example, it can be an effective solution for problems such as the single-source shortest-path problem that we presented in Chapter 2. For operators such as `match` that work on an unbounded number of nodes, it may not be the most effective approach because the upfront sorting of a large number of nodes may be quite expensive. Moreover, this scheme is difficult to implement for problems where operators destructively update the graph structure.

In this chapter, we focus on *speculative locking* since it is used in existing graph frameworks [2, 92]. At a high level, correct parallel execution of *match* in `MIS` requires the following actions: (i) lock `a` and its neighbors, (ii) perform the checks on the status fields of these nodes, (iii) set these fields appropriately, and (iv) release all the locks. If a lock cannot be acquired in step (i), all currently held locks are released, and *match* is retried later. Each of these four steps can be implemented by code that touches node `a` and iterates over

its neighbors; we call this the baseline version.

Rescheduling this code to interleave some of these steps produces variants that may perform better. For example, steps (i) and (ii) can be interleaved so that the status of a neighbor b is examined as soon as it is locked; if b's status is $Matched$, the operator execution can be terminated without examining more neighbors. Although this seems desirable, note that if the probability of conflicts is high, the baseline version that acquires all locks before performing any checks might perform better since it reduces wasted computation because of aborts. Which version performs better therefore may depend on the graph structure, the thread count, etc. Similar choices arise in steps (iii) and (iv). Fusing the status updates with lock releases results in tighter atomic sections and fewer conflicts potentially, whereas the baseline version may permit the use of vector store instructions. Synchronization therefore introduces new scheduling opportunities.

Moreover, implementing any of these variants requires book-keeping code to keep track of locks acquired by an operator execution. An operator-agnostic generic implementation is the *stamp-and-log* strategy: each thread maintains a runtime log of locked nodes and releases the log contents when the operator execution terminates. A stamp associating each node with its current owner is used during lock acquires to detect conflicts. This strategy is used by systems that delegate concurrency management to a runtime system [2]. *However, compile-time reasoning of locks acquired along different paths in the HIR code permit the generation of synchronization code that is customized to*

the operator and does not need such runtime structures.

ATS relies on static information about node may-aliases, and per-program-point information about the set of node references through which lock acquires have already been performed. This allows ATS to: (i) statically insert the right lock releases for program-points where operator execution may terminate; (ii) synthesize custom conflict-detection checks using alias-checking with already acquired nodes. `misV1` is an ATS synchronized version of `misH`. In line 5 $\mathcal{N}$(a) are locked in a context where only a is locked (**ctx** a). We need to perform a `lock`(b) only for $b \in \mathcal{N}$(a) such that $b \neq a$. This is because a, which is already locked, may be aliased to b — elements of $\mathcal{N}$(a) are not aliased to one-another, so no further checks are needed. If `lock`(b) fails, then the thread definitely does not own b and a conflict occurs. Such thread-local alias checks obviate the need for a stamp and are amenable to further compile-time optimization. Similarly, line 4 evaluates a predicate in **ctx** a. If it's false, we simply release a (line 11) and terminate operator execution. Statically computing this information allows simply emitting an '`unlock` a', obviating the need for a runtime log.

3.1.4 The Need for an Integrated Solution

How should scheduling, synchronization and implementation selection be implemented in a compiler that generates parallel code from HIR programs? A staged approach with separate compiler passes per task is easy to implement but introduces the phase-ordering problem. We illustrate this using the

Triangles example from Figure 3.2 and Figure 3.4. Starting with `TrH`, we can apply scheduling (T_S) and implementation selection (T_{IS}) in either order, using phase-local optimization heuristics. For T_S, the obvious heuristic is to nest node iterators within conditionals whenever possible; moreover, complex conditionals involving graph data (*e.g.*, **if edges**(a, b)) should be nested within scalar ones (**if** $a < b$) if possible. T_{IS} favors maximal tile usage since tiles encode efficient implementations of HIR statement sequences.

If T_S followed by T_{IS} are applied to `TrH`, T_S produces `TrH1`, an optimal schedule, which is left unchanged by T_{IS} since there is no opportunity to apply tiling. If T_{IS} is followed by T_S, no tiling is possible in `TrH`, so T_{IS} produces `TrH`, and T_S then produces `TrH1`, which does not use tiles at all. In contrast, our planning approach starts from `TrH` and produces `TrL1`, which has an optimal schedule and makes maximal tile usage. Conceptually, during the search for an optimal plan, it considers `TrH2`, which permits the use of two tile instances, thus leading to `TrL1`. When synchronization is involved, finding the optimal solution becomes even harder with the staged approach whereas planning remains equally effective. Moreover, planning is superior to exhaustive search of the implementation space, which would consider many more sub-optimal variants that do not use tiling.

3.2 A Planning-based Synthesis Framework

In this chapter, we show that these parallel code generation problems can be formulated using constraints, and that these constraints can be solved

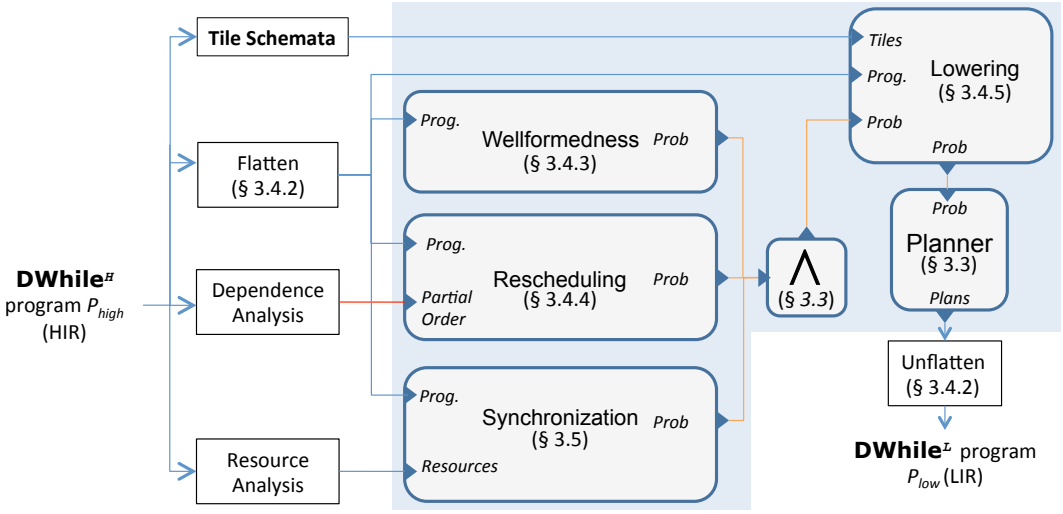


Figure 3.5: Planning framework architecture. *Prob* : planning problem.

efficiently using *planning*. A STRIPS-style planning problem [43] is specified by an initial state, a goal state, and a set of actions that can be used to transition from one state to another (Section 3.3 provides a detailed definition). The planning problem is to synthesize a sequence of actions that lead from the initial state to the goal state. Properties that a solution must satisfy are encoded by *temporal constraints*. There are two main advantages to this approach.

Integration: Searching for solutions that *simultaneously* solve all constraints avoids the phase-ordering problem and produces better code.

Engineering: Each code generation problem is defined declaratively and succinctly; different correctness and profitability concerns are seamlessly composed together, allowing easy construction and experimentation.

Figure 3.5 shows our system, which is parameterized by: (i) the HIR description; (ii) a dependence analysis; (iii) a resource analysis; and (iv) a tile schema for each statement type in the low-level language. At a high level, the system works as follows. The HIR program P_{high} is fed to several planning problem construction units, and each unit emits a planning problem related to a different code generation problem. Individual problems are then combined to define a single composite planning problem that is fed to a planner, which emits the LIR program. One detail is that since planners deal with sequences rather than nested structures, the HIR program is *flattened* by producing an in-order representation of its abstract syntax tree, and this sequence is actually the input to the planning problem construction units. Although our planning-based transformations are rewrites on sequences of actions, they can alternatively be reinterpreted in a more traditional form as tree rewrites on a structured IR. At the other end, the planner produces an in-order representation of the LIR program, which is *unflattened* to produce the actual LIR program.

To permit the system to be used for other code generation problems, the descriptions of the HIR, LIR and the tiles are inputs to the system, as shown in Figure 3.5. Our current system has the following planning problem construction units for tasks related to parallel code generation for graph programs: (i) ensuring that the output program is syntactically and semantically correct (*Wellformedness*), (ii) is equivalent to the input program (*Rescheduling*), (iii) is properly synchronized (*Synchronization*), and (iv) is implemented

by statements in the low-level language (*Lowering*). Since rescheduling must respect dependences, it takes the results of a *dependence analysis* as input. The *Resource Analysis* module extracts all accesses to shared resources in the program, so that these can be synchronized properly; for our code generation problem, these are accesses to shared-memory variables.

`MIS` Synthesis as a Planning Problem We now explain the flow though some of the modules in Figure 3.5 by showing how to derive `misV1` and `misV2` from `misH`.

First we decompose the structured program `misH` to its basic syntactic units. Their set $\mathcal{U}$ is fed to the problem construction units. Different `misH` schedules correspond to permutations of $\mathcal{U}$'s contents. However, not all permutations encode programs that are both syntactically correct and semantically equivalent to `misH`. To automatically get desired solutions, we encode a planning problem requiring each unit-action appear exactly once, and augment it with temporal constraints expressing syntactic wellformedness and the results of a dependence analysis. Such constraints restrict solutions to plans encoding syntactically correct programs that are equivalent to (satisfy the same dependences) `misH`. For example, consider $i, j \in \mathcal{U}$ corresponding to '**if** $sa = Unmatched$' and '$status(a) := Matched$', respectively. We encode the control dependence between i, j using the temporal constraint $\neg j \,\mathcal{W}\, i$. This requires that the first plan state s_i where i has executed precedes the first plan state s_j where j has executed.

The ATS planning problem augments $\mathcal{U}$ with a set $\mathcal{L}$ of lock/unlock statements. $\mathcal{L}$ contains multiple instruction variants for each node to be locked, one per history of previous lock acquisitions. For example, in `misH` node a can be locked before $\mathcal{N}$(a) or after it. Hence, $\mathcal{L}$ includes '**lock** a **ctx** $\emptyset$' and '**lock** a **ctx** $\mathcal{N}$(a)'. The encoding of ATS planning actions enables only valid combinations in plans, and allows locking customization to specific schedules. For example, in `misV1` the combination is **lock** a **ctx** $\emptyset$, **lock** $\mathcal{N}$(a) **ctx** a. Such statements encode all the information necessary to perform conflict detection and which locks to release in case of aborts. Temporal constraints enforce global correctness properties of ATS. For example, to encode two-phase locking, which guarantees serializable execution, we require that all locks happen before unlocks. Additionally, to guarantee operator *cautiousness*[2], which enables transactional execution without storing rollback information, we require all locks to occur before shared state updates. ATS is a very good example of the value that planning adds to the field of program transformations. Here, the planning system does not merely find a permutation that reorders statements subject to the partial order dictated by program dependences but *it synthesizes the right sequence of actions that constitute a custom version of a locking protocol for a specific program.*

The input HIR instantiates a set of tile-schemas. Lowering adapts the planning problem to also use tile-releated actions. Solutions to the new

[2]Cautious operators are formally defined in Chapter 6.

planning problem encode low-level programs. For example, the tile

$$tile[\forall(b) \textbf{ with } rs_1 \textbf{ ctx } rs_2] \triangleq \textbf{lock } rs_1 \textbf{ ctx } rs_2, \forall(b)$$

combines locking and the $\forall$ evaluation in `misV2` (Figure 3.3(d), line 5) .

3.3 Planning with Temporally Extended Goals

This section provides background on automated planning with temporal goals. We also describe two operations employed by our framework: (i) conjunction of planning problems, and (ii) translation of problems over individual actions to problems over constant-length "macro" actions.

Classical STRIPS-style Planning Problems. A planning problem is a quadruple $P = \langle Flnts^P, Init^P, Act^P, Goal^P \rangle$ where $Flnts^P$ is a set of propositional facts, called *fluents*; $Init^P \subseteq Flnts^P$ represents the *initial state*; Act^P represents the set of *actions* (defined next); and $Goal^P \subseteq Flnts^P$ represents the *goal*. In the sequel, we shall drop the superscript P when it is obvious from the context. An action $o \in Act$ is represented by an identifier $Id(o)$ and four sets of propositions called the *Add*, *Del*, Pre^t, and Pre^f. *Add* describes the fluents that o makes true, *Del*, the fluents that o makes false, Pre^t (Pre^f), the fluents that must be true (false) in order for o to be applicable. We will often conflate an action with its identifier, when no confusion arises.

Action Notation. We lift negation to sets of fluents by writing $\neg S$ for $\{\neg f \mid f \in S\}$. We denote actions by $\langle I \rangle\, o\, \langle O \rangle$ where o is an identifier; $I =$

$I^t \cup \neg I^f$ and $O = O^t \cup \neg O^f$; and I^t, O^t, I^f, and O^f are sets of fluents. Thus, $Pre^t(o) = I^t$, $Pre^f(o) = \neg I^f$, $Add(o) = O^t$, and $Del(o) = \neg O^f$.

States and State Transformers. A state $\sigma \subseteq Flnts$ represents a truth-assignment $\sigma^b : Flnts \to \{0, 1\}$ such that $\sigma^b(p) = 1$ if and only if $p \in \sigma$. An action $o \in Act$ denotes a partial state transformer $[\![o]\!] : \Sigma \rightharpoonup \Sigma$ such that $[\![o]\!]\sigma = \sigma'$ if $Pre^t(o) \subseteq \sigma$, $\sigma \cap Pre^f(o) = \emptyset$, and $\sigma' = (\sigma \setminus Del(o)) \cup Add(o)$ hold.

Plans. A *plan* π is a sequence of actions $o_1, \ldots, o_k$[3] such that $[\![o_k]\!] \circ \ldots \circ [\![o_1]\!]$ $Init \supseteq Goal$. We write $Plans(P)$ for the set of plans of a planning problem P. For a given plan length, it is possible to efficiently encode the planning problem as a propositional formula, which can be handed to a SAT solver. Shortest plans can be found by searching for plans of increased length [75].

Planning with Temporally Extended Goals. Temporal goals specify the conditions that plans must satisfy. We express such conditions in (a subset of) linear temporal logic (LTL), whose models are the sequences of states generated by the corresponding plans, starting from the initial state. Planning problems may be extended by a set of temporal goals, specified by LTL formulas.

[3] In practice, we will be interested in the sequence of action identifiers.

In Section 3.4 and Section 3.5, we define temporal goals via the following operators:

Formula	Description
$\mathsf{F}\,\varphi \stackrel{\text{def}}{=}$	φ occurs at least once.
$\mathsf{F!}\,\varphi \stackrel{\text{def}}{=}$	φ occurs exactly once.
$\varphi\,\mathcal{W}\,\varphi' \stackrel{\text{def}}{=}$	φ' occurs and φ occurs (at least) until φ', or φ always occurs.
$\varphi \sqsubset \varphi' \stackrel{\text{def}}{=}$	φ first occurs (if at all) before φ'.
$\varphi_1 \sqsubset \ldots \sqsubset \varphi_n \stackrel{\text{def}}{=}$	$\bigwedge_{i=1}^{n-1} \varphi_i \sqsubset \varphi_{i+1}$.
$(a, a') \otimes (b, b') \stackrel{\text{def}}{=}$	(Balanced parentheses) $(a \sqsubset a' \sqsubset b \sqsubset b') \vee (b \sqsubset b' \sqsubset a \sqsubset a')$ $\vee (a \sqsubset b \sqsubset b' \sqsubset a') \vee (b \sqsubset a \sqsubset a' \sqsubset b')$.

Conjoining Planning Problems. To allow solving tasks in a modular way, we define an appropriate conjunction operation. This enables us to encode subtasks by individual planning problems and then conjoin them into a planning problem that solves the entire task. We have that for two planning problems P and Q, the following holds: $\mathit{Plans}(P \wedge Q) \supseteq \mathit{Plans}(P) \cap \mathit{Plans}(Q)$.

Inverse Homomorphism. A *macro action* is a sequence of actions, written as $m = o_1; \ldots; o_k$. A sequence of actions can be composed into a single action: $[\![m]\!] = [\![o_k]\!] \circ \ldots \circ [\![o_1]\!]$. We write $\overline{m} = o_1, \ldots, o_k$ for the corresponding sequence of actions identifiers. Given a planning problem P and set of macro actions M over Act^P, we wish to obtain another planning problem P_M such that $\mathit{Plans}(P_M) = \{m_1, \ldots, m_k \mid \text{for } i \in [1, k] : m_i \in$

M and $\overline{m_1}, \ldots, \overline{m_k} \in \mathit{Plans}(P)\}$. That is, the language $\mathit{Plans}(P_M)$ is induced by the inverse homomorphism h^{-1} where $h(m) \stackrel{\text{def}}{=} \overline{m}$. This language can be obtained by transforming the planning problem appropriately, which we denote by $\mathit{InvHom}(P, M) = P_M$.

3.4 Formal Synthesis Framework

This section shows how compilation tasks other than the insertion of synchronization can be formulated as planning problems. Synchronization is treated in Section 3.5.

3.4.1 The Parametric Language DWhile

We now describe the class of data-intensive programming languages targeted by our synthesis framework. It is parameterized by: (i) type of atomic state-changing statements, (ii) Boolean expressions, and (iii) data range expressions. We call the languages obtained by instantiating these parameters with the grammar shown in Figure 3.6(a) DWhile languages. Programs may refer to global variables, defined externally, via update statements. We use attribute grammars [77] (AG for short) to impart semantic conditions to our parametric language and define functions. A statement **for** x : r **do** S **od** has a dual role: (i) iterating over a range of data values (defined by r), and (ii) introducing a scope in which the local immutable variable x is bound and initialized to the single-value range r. We use the predicate $\mathit{val}(r)$ to test whether a range expression denotes a single value.

Meta Variable	Description
x	A program variable $x \in Var$
b	Boolean expression
r	Data range expression
rs	A sequence of range expressions
Upd	State update

Attribute	Value Type (inherited/synthesized)
$boundVars(\cdot)$	2^{Var} (inherited)
$vars(\cdot)$	2^{Var} (synthesized)

Production	Semantic Rules
$S ::= \textbf{for}\ x : r\ \textbf{do}^L\ B_1\ \textbf{od}^L$	if $x \in boundVars(S)$ then error, if $vars(r) \not\subseteq boundVars(S)$ then error, $boundVars(B_1) = boundVars(S) \cup \{x\}$.
$\mid \textbf{while}\ b\ \textbf{do}^L\ B_2\ \textbf{od}^L$	if $vars(b) \not\subseteq boundVars(S)$ then error, $boundVars(B_2) = boundVars(S)$.
$\mid \textbf{if}\ b^L\ B_t\ \textbf{else}^L\ \textbf{skip}^L\ \textbf{fi}^L$	if $vars(b) \not\subseteq boundVars(S)$ then error, $boundVars(B_t) = boundVars(S)$.
$B ::= S$	$boundVars(S) = boundVars(B)$.
$\mid A$	$boundVars(A) = boundVars(B)$.
$A ::= AtomUpd$	$boundVars(AtomUpd) = boundVars(A)$.
$\mid \textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2^L; A_1$	if $vars(rs_1, rs_2) \not\subseteq boundVars(A)$ then error. $boundVars(A_1) = boundVars(A)$.
$\mid \textbf{if}\ b^L\ A_t\ \textbf{else}^L\ R\ \textbf{exit}^L\ \textbf{fi}^L$	if $vars(b) \not\subseteq boundVars(A)$ then error, $boundVars(A_t) = boundVars(R) = boundVars(A)$.
$\mid \textbf{for}\ x : r\ \textbf{do}^L\ A_b\ \textbf{od}^L$	if $\neg val(r)$ then error, if $vars(r) \not\subseteq boundVars(A)$ then error, if $x \in boundVars(A)$ then error, $boundVars(A_b) = boundVars(A) \cup \{x\}$.
$R ::= \epsilon$	
$\mid \textbf{unlock}\ rs^L$	if $vars(rs) \not\subseteq boundVars(R)$ then error.
$AtomUpd ::= Upd; R;\ \textbf{commit}$	$boundVars(Upd) = boundVars(R) = boundVars(AtomUpd)$.
$Upd ::= Upd^L$	if $vars(Upd) \not\subseteq boundVars(Upd)$ then error.
$\mid \textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2^L$	if $vars(rs_1, rs_2) \not\subseteq boundVars(Upd)$ then error.
$\mid Upd_1; Upd_2$	$boundVars(Upd_1) = boundVars(Upd_2) = boundVars(Upd)$.

(a)

Meta variable	Description
L	A label $L \in Label$

U	$::=$	$Upd^L \mid \textbf{commit}^L \mid \textbf{if}\ b^L \mid \textbf{else}^L \mid \textbf{exit}^L \mid \textbf{fi}^L$
	$\mid$	$\textbf{while}\ b\ \textbf{do}^L \mid \textbf{for}\ x : r\ \textbf{do}^L \mid \textbf{od}^L$
	$\mid$	$\textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2^L \mid \textbf{unlock}\ rs^L \mid \textbf{skip}^L$
F	$::=$	$U \mid U\ F$

(b)

Figure 3.6: (a) AG for DWhile, and (b) A regular grammar for flat (labelled) DWhile.

A statement **var** $x := r; S$ introduces a scoped local variable and is syntactic sugar for **if** $\neg$**empty**(r) **for** $x : r$ **do** S **od fi**. The AG ensures that: (i) local variables are only accessed within their scope, never hiding other variables; (ii) there exists at most one atomic section[4]; and (iii) updates appear only in the inner-most nesting level[5]. In the sequel, we fix a high-level language, DWhile^H, and a low-level language, DWhile^L. The synchronization-related statements **lock** rs_1 **ctx** rs_2 and **unlock** rs, which are explained in Section 3.5, do not appear in the input program. We slightly abuse notation by conflating meta variables and their terminals/non-terminals. Finally, we abbreviate "**if** $b\ S$ **else** N **fi**" (N is either **exit** or **skip**) by "**if** $b\ S$ **fi**".

3.4.2 Flattening and Unflattening DWhile Programs

Our synthesis technique operates over a deconstructed form of DWhile programs, which we call *flat programs*, defined by the regular grammar in Figure 3.6(b). Flat programs consist of a sequence of *units*. The function $\mathit{flat} : \mathsf{DWhile} \to F$ labels each unit in the input program and returns them in order, taking care to associate the same label with units matching a given HIR statement: $\{\mathbf{if}\ b^L, \mathbf{else}^L, \mathbf{exit}^L, \mathbf{fi}^L\}$, $\{\mathbf{if}\ b^L, \mathbf{else}^L, \mathbf{skip}^L, \mathbf{fi}^L\}$, $\{\mathbf{while}\ b\ \mathbf{do}^L, \mathbf{od}^L\}$, and $\{\mathbf{for}\ x : r\ \mathbf{do}^L, \mathbf{od}^L\}$.

For the remainder of this section, we fix a labelled high-level program

[4] To simplify the exposition we allow at most one atomic section and let sequential composition appear only inside an atomic section.

[5] This condition is only necessary to handle our specific synchronization protocol and can be removed for sequential code.

$S \in \mathsf{DWhile}^H$ and $F^\flat = \mathit{flat}(S)$.

We invert flattening by defining the (pseudo-inverse) function $\mathit{unflat} : F \to \mathsf{DWhile}$ such that $\mathit{flat}(\mathit{unflat}(F^\flat)) = F^\flat$. *unflat* can be implemented by an LALR(1) parser operating on the sequence of tokens that are the units of $F^\flat$.

Permuting Flat Programs. We are now interested in defining planning problems that encode the constraints between $F^\flat$ and the output of the planner $F^{\flat\prime}$ such that $F^{\flat\prime}$ is a *permutation* of $F^\flat$ that represents an equivalent-meaning program. Let $F^\flat$ be the sequence of units $u_1 \cdot \ldots \cdot u_n$. We write $F^\flat(i) = u_i$, and $|F^\flat| = n$ for the length of $F^\flat$. We abbreviate $\{1, \ldots, n\}$ by $1..n$. A permutation $\Pi : 1..n \to 1..n$ induces a transformation $\Pi : F \to F$ over flat programs, defined as $\Pi(F^\flat) = F^\flat(\Pi(1)) \cdot \ldots \cdot F^\flat(\Pi(n))$. We define the equivalence relation $F_1 \approx^\Pi F_2$ if and only if F_1 is a permutation of F_2.

3.4.3 Encoding Wellformedness by a Planning Problem

We say that a flat program $F^{\flat\prime}$ is *wellformed* if there exists a program $S' \in \mathsf{DWhile}$ such that $\mathit{flat}(S') = F^{\flat\prime}$.

Lemma 1. *Figure 3.7 defines the planning problem $WF(S)$ whose plans are all wellformed permutations of $\mathit{flat}(S)$:*

$$\mathit{Plans}(\,WF(S)) = \{F^{\flat\prime} \mid F^{\flat\prime} \approx^\Pi \mathit{flat}(S), F^{\flat\prime} \text{ is wellformed}\} \ . \tag{3.1}$$

Attribute	**Value Type**
$prn(\cdot)$	$\wp(\,U^{Label} \times U^{Label})$ (synthesized)
$ite(\cdot)$	$\wp(\,U^{Label} \times U^{Label} \times U^{Label})$ (synthesized)
$boundFnts(\cdot)$	$\wp(\mathcal{B}[Var])$ (synthesized)
$boundActs(\cdot)$	$\wp(Act)$ (synthesized)

Production	**Semantic Rules**
$N ::= Upd^L$	$prn(N) = ite(N) = boundFnts(N) = \emptyset$, $boundActs(N) = \{\ \langle \mathcal{B}[vars(Upd^L)]\rangle\ Upd^L\ \langle\rangle\ \}$.
$\mid N_1; N_2$	$prn(N) = prn(N_1) \cup prn(N_2)$, $ite(N) = ite(N_1) \cup ite(N_2)$, $boundFnts(N) = boundFnts(N_1) \cup boundFnts(N_2)$, $boundActs(N) = boundActs(N_1) \cup boundActs(N_2)$.
$\mid$ **if** b^L N_1 **else**^{L} N_2 **fi**^{L}	$prn(N) = \{(\textbf{if}\ b^L, \textbf{fi}^L)\} \cup prn(N_1) \cup prn(N_2)$, $ite(N) = \{(\textbf{if}\ b^L, \textbf{else}^L, \textbf{fi}^L)\} \cup ite(N_1) \cup ite(N_2)$, $boundFnts(N) = boundFnts(N_1) \cup boundFnts(N_2)$, $boundActs(N) = boundActs(N_1) \cup boundActs(N_2) \cup$ $\{\ \langle \mathcal{B}[vars(b)]\rangle\ \textbf{if}\ b^L\ \langle\rangle\ \}$.
$\mid$ **while** b **do**^{L} N_b **od**^{L}	$prn(N) = \{(\textbf{while}\ b\ \textbf{do}^L, \textbf{od}^L)\} \cup prn(N_b)$, $ite(N) = ite(N_b)$, $boundFnts(N) = boundFnts(N_b)$, $boundActs(N) = boundActs(N_b) \cup$ $\{\ \langle \mathcal{B}[vars(b)]\rangle\ \textbf{while}\ b\ \textbf{do}^L\ \langle\rangle\ \}$.
$\mid$ **for** $x : r$ **do**^{L} N_b **od**^{L}	$prn(N) = \{(\textbf{for}\ x : r\ \textbf{do}^L, \textbf{od}^L)\} \cup prn(N_b)$, $ite(N) = ite(N_b)$, $boundFnts(N) = \{\mathcal{B}[x]\} \cup boundFnts(N_b)$, $boundActs(N) = boundActs(N_b) \cup$ $\{\ \langle \neg\mathcal{B}[x], \mathcal{B}[vars(r)]\rangle\ \textbf{for}\ x : r\ \textbf{do}^L\ \langle \mathcal{B}[x]\rangle,$ $\langle\rangle\ \textbf{od}^L\ \langle \neg\mathcal{B}[x]\rangle\ \}$

(a)

$Flnts^{WF(S)}$	=	$\{flat(S)\} \cup boundFnts(S)$
$Act^{WF(S)}$	=	$\{\langle\rangle\ u\ \langle only(u)\rangle\ \mid u \in flat(S)\} \wedge boundActs(S)$
$Init^{WF(S)}$	=	$\emptyset$
$Goal^{WF(S)}$	=	$\bigcup_{i=1}^{5} Goal_i^{WF(S)}$
$Goal_1^{WF(S)}$	=	$\{\mathsf{F!}\,u \mid u \in \{flat(S)\}\}$
$Goal_2^{WF(S)}$	=	$\{po \sqsubset pc \mid (po, pc) \in prn(S)\}$
$Goal_3^{WF(S)}$	=	$\{i \sqsubset e \sqsubset f \mid (i, e, f) \in ite(S)\}$
$Goal_4^{WF(S)}$	=	$\{p \otimes p' \mid p, p' \in prn(S), p \neq p'\}$
$Goal_5^{WF(S)}$	=	$\{(i \sqsubset po \sqsubset e \Rightarrow pc \sqsubset e) \mid$ $(po, pc) \in prn(S), (i, e, f) \in ite(S)\}$.

(b)

Figure 3.7: (a) AG for computing delimiters, fluents for tracking bound variables, and actions for tracking sets of bound variables over units. To avoid clutter, we handle productions of similar form together by letting the meta non-terminals N, N_1, N_2 stand for potions of the right-hand sides of productions in Figure 3.6(a). (b) planning problem for wellformedness. We write $\{flat(S)\}$ for the set of units in the sequence $flat(S)$ and $\mathcal{B}[vars(e)]$ for the set $\{\mathcal{B}[z] \mid$ variable z appears in $e\}$.

To ensure syntactic wellformedness, the planning problem WF contains a fluent u and the action $\langle\rangle\, u\, \langle \mathit{only}(u)\rangle$ per unit u in the original program where $\mathit{only}(u) \stackrel{\text{def}}{=} \{u\} \cup \{\neg v \mid v \in \mathit{flat}(S) \setminus \{u\}\}$. This ensures each action emits the corresponding fluent at the instant it appears in a plan by setting it at the post state and turning off unit-fluents from the previous action. This allows us to express temporal goals over the units of the output program. The first goal establishes that the output program is a permutation of the input program. We refer to units of the form **if** b^L, **while** b **do**^{L}, and **for** $x : r$ **do**^{L} as *opening delimiters* and units of the form **fi**^{L} and **od**^{L} as *closing delimiters*, as they immediately precede and succeed compound statements. Units of the form **else**^{L} are considered as a closing delimiter (of the then-branch statement) immediately followed by an opening delimiter (of the else-branch statement). The other goals establish that delimiters appear in correct order and form nested scopes. To ensure semantic wellformedness, we use fluents of the form $\mathcal{B}[x]$, per variable appearing in a **for** statement. A **for** $x : r$ **do**^{L} unit adds a $\mathcal{B}[x]$ fluent and the corresponding **od**^{L} removes it to signify that only units in the sub-plan between these units, which correspond to the body of the loop, may access the variable.

3.4.4 Encoding Rescheduling by a Planning Problem

Let $[\![S]\!]$ denote the semantics of a program S. The plans in $\mathit{Plans}(\mathit{WF}(S))$ are wellformed permutations of $\mathit{flat}(S)$, however, they do not necessarily preserve the semantics of S. We add goals to ensure that the semantics is pre-

served.

Let $\preceq \subseteq 1..n \times 1..n$ be a partial order over $1..n$. We say that a permutation $\Pi : 1..n \to 1..n$ is *monotone* w.r.t $\preceq$, written $\Pi_{\preceq}$, if $i \preceq j$ implies $\Pi(i) \preceq \Pi(j)$. We say that a partial order $\preceq \subseteq 1..n \times 1..n$ is *dependence preserving* if every monotone permutation $\Pi_{\preceq}$ induces an equivalent program: $[\![S]\!] = [\![\mathit{unflat}(\Pi_{\preceq}(\mathit{flat}(S)))]\!]$. A dependence preserving partial order $\preceq$ induces an equivalence relation among programs: S and S' are *dependence-equivalent*, written $S \approx^{\preceq} S'$, iff there exists a monotone permutation $\Pi_{\preceq} : 1..n \to 1..n$ such that $S' = \mathit{unflat}(\Pi_{\preceq}(\mathit{flat}(S)))$.

We say that $\mathit{Dependences} : \mathsf{DWhile}^H \to \mathbb{N} \times \mathbb{N}$ is a *dependence analysis* if $\mathit{Dependences}(S)$ is a dependence preserving partial order for every $S \in \mathsf{DWhile}^H$. Notice that in our definition a dependence analysis returns a result over flat programs. This allows us to uniformly express transformations such as loop and condition reordering, hoisting statements out of loops, and reordering updates. We encode a dependence analysis by temporal goals as follows:

$$\mathit{Goal}^{\mathit{Dependences}(S)} \stackrel{\text{def}}{=} \{u_i \sqsubset u_j \mid i \preceq j \in \mathit{Dependences}(S), i \neq j\} \ . \tag{3.2}$$

Lemma 2. *Define* $\mathit{Equiv}(S) = \mathit{WF}(S) \wedge \mathit{Goal}^{\mathit{Dependences}(S)}$ *to be* $\mathit{WF}(S)$ *extended with the dependence analysis goals. The plans of* $\mathit{Equiv}(S)$ *are all flat*

programs that can be unflattened to a dependence-equivalent program:

$$Plans(Equiv(S)) = \{F^{\flat\prime} \mid S \approx^{\preceq} unflat(F^{\flat\prime})\} \ . \tag{3.3}$$

3.4.5 Encoding Lowering by a Planning Problem

Tiles and Tilings. Let $\mathsf{FDWhile}^H$ and $\mathsf{FDWhile}^L$ denote the flat languages corresponding to DWhile^H and DWhile^L, respectively. A *tile* associates a sequence of (high-level) units from $\mathsf{FDWhile}^H$ with a (low-level) unit from $\mathsf{FDWhile}^L$, written as $tile(lu) = hu_1, \ldots, hu_k$. We say that a flat low-level program $F^L = lu_1, \ldots, lu_m$ is a *tiling* of the flat high-level program: $F^H = tile(F^L) = tile(lu_1), \ldots, tile(lu_m)$. We also say that lu_i *covers* $tile(lu_i)$ in F^H. Intuitively, tiles provide customized implementations that take advantage of, e.g., specific data structure implementations and properties of the runtime platform, to achieve better efficiency.

The Tile-Based Lowering Problem. For $S^H \in \mathsf{DWhile}^H$, a set of tiles M, and a dependence analysis *Dependences*, find a program $S^L \in \mathsf{DWhile}^L$ such that there exists a dependence-equivalent program $S^{H\prime} \approx^{\preceq} S^H$ and $flat(S^L)$ is a tiling of $flat(S^{H\prime})$ where $\preceq = \ Dependences(S^H)$.

Multi Tiles. Tiles implementing loops and conditions usually cover only one delimiter of the loop or condition statement. To cover the remaining delimiters, we couple them with additional tiles. These tiles appear in tandem, covering

all of the delimiters of a high-level statement (2 for loops and up to 3 for conditionals). We call these *multi tiles*.

Example 1. *The following tile takes advantage of a graph data structure where the successors of a node can be efficiently accessed:*

$$tile[\textbf{\textit{for}}\ b : Succ(a)\ \textbf{\textit{do}}] = \textbf{\textit{for}}\ b : nodes\ \textbf{\textit{do}},\ \textbf{\textit{if}}\ \ \textbf{\textit{edges}}(a, b)\ \ .$$

The tile above is coupled with the following tile, which is used as a closing delimiter: $tile[\textbf{\textit{odSucc}}] = \textbf{\textit{fi}},\ \textbf{\textit{od}}$.

Theorem 1. *For* $S^H \in \mathsf{DWhile}^H$ *and a set of (multi) tiles* M*, define the planning problem*

$$Lower_M(S^H) \stackrel{\text{def}}{=} InvHom(Equiv(S^H), M)\ \ . \tag{3.4}$$

Then, $\pi \in Plans(Lower_M(S^H))$ *if and only if* $unflat(\pi)$ *is a solution to the tile-based lowering problem for* S^H *and* M*.*

3.5 Planning-Based Synchronization

We now turn to parallel DWhile programs and the problem of synchronizing them both correctly and efficiently. Let $[\![r]\!]\sigma$ mean the set of objects denoted by range expression r in a program state σ. To support parallelism, we assume that the outermost loop takes parallel semantics.[6] We have that

$$[\![\textbf{for}\ x : r\ \textbf{do}\ S\ \textbf{od}]\!]\sigma = [\![S(x := v_1) \| \ldots \| S(x := v_k)]\!]\sigma$$

[6] We also assume that the parallel loop does not contain **while** loops.

where $[\![r]\!]\sigma = \{v_1, \ldots, v_k\}$ and $[\![S(x := v_i)]\!]$ denotes the **atomic** execution of the loop body in the context where x is bound to v_i. The tasks $S(x := v_i)$ execute in parallel while ensuring serializability. When a task aborts due to a conflict, it is re-executed. General techniques for synchronizing arbitrary programs usually rely on variants of transactional memory [115, 61]. We consider these approaches as a reference for comparison. We define an efficient speculative lock-based synchronization technique (Section 3.5.1), dubbed ATS, that can achieve better performance than the reference synchronization techniques. Finally, we define a planning problem (Section 3.5.2) to automatically insert synchronization statements that realize ATS.

Resources and Resource Expressions. We assume an analysis that computes for each unit u the sets of expressions $rd(u)$ and $wt(u)$, which denote the (overapproximation of) shared runtime objects that it may directly access for reading and writing, respectively. We define $res(u) = rd(u) \cup wt(u)$. For example, $res(mis) = \{\mathrm{a}, \mathcal{N}(\mathrm{a})\}$ for `misH` in Figure 3.3(b).

Stable Ranges. We further assume that the range expressions appearing in an DWhile program are invariant for each update statement *Upd* appearing in it: $[\![r]\!]\sigma = [\![r]\!] \circ [\![Upd]\!]\sigma$. This condition must be checked for each instantiation of DWhile in order for our synchronization technique to work correctly. For the class of Elixir programs considered in this thesis, this amounts to checking that the sets of nodes and edges remain constant.

3.5.1 Alias Tracking-Based Synchronization (**ATS**)

Our synchronization technique is a variant of two-phase locking where each shared object *obj* is associated with an atomic bit field *isLk*. When $obj.isLk = 1$ it means that the object is owned for exclusive access by some task. However, *isLk* does not convey which task owns the object. This means that when an attempt to acquire *obj* by a test-and-set instruction through the resource expression r fails there are two possible reasons: (i) the *obj* is currently owned by another task; or (ii) *obj* has been previously acquired by the current task, perhaps through another resource expression r' aliased with r. To differentiate between the two cases, it is possible to track the set of resource expressions *rs* that were used earlier in the execution to acquire objects and dynamically check whether the objects in r are a subset of *rs*: $[\![r]\!]\sigma \subseteq [\![rs]\!]\sigma$. If so, the object is owned by the current task. To achieve this form of synchronization we introduce the following instrumentation statements.

"**unlock** *rs*" releases the objects in $[\![rs]\!]$, by setting $isLk = 0$, taking care to reset the bit of each object at most once. This is done by checking each object for aliasing against the sub-resource expressions of *rs* already used for releasing objects.

"**lock** rs_1 **ctx** rs_2" attempts to acquire the objects in $[\![rs_1]\!]$ where rs_2 denotes the set of objects already owned by the current task, which we call *context*. If $[\![rs_1]\!]\sigma \subseteq [\![rs_2]\!]\sigma$ the operation succeeds; otherwise, if locking rs_1 fails then the objects denoted by rs_2 and the portion of rs_1 successfully acquired are unlocked and the task is re-executed.

We say that a DWhile program is correctly synchronized by ATS when all shared data structures are linearizable [63] and the following conditions hold: ($C1$) **Isolation:** each resource is acquired before accessed; ($C2$) **Two-phase locking:** all unlocks happen after all locks; ($C3$) **Release:** all locks are released when execution aborts or commits; ($C4$) **Cautious:** all locks are acquired before any updates occur, which ensures that on abort, no rollback operations are required to restore state to the one at the beginning of the task; and ($C5$) **Locks tracking:** "**lock** rs_1 **ctx** rs_2" statements execute with the correct context. $C1$–3 ensure serializability. In Figure 3.3(c), `misV1` is correctly synchronized by ATS[7]. It is obtained from `misH` by applying instrumentation, as explained next.

3.5.2 Instrumenting DWhile for ATS via Planning

Figure 3.8 defines the planning problem $Synch(S)$ used for instrumenting DWhile programs, which ensures that $C1$–5 hold on every execution path.

Encoding a Lockset Analysis. Let $res(S)$ denote the set of all resources in S. We encode a flow-sensitive dataflow may-analysis, which tracks the set of acquired resources via the powerset lattice $\langle \wp(res(S)), \subseteq, \cup, \cap, \emptyset, res(S) \rangle$. We encode lattice elements by the set of fluents $\{locked[r] \mid r \in res(S)\}$ and define the shorthand notation $\mathcal{D}[rs] \stackrel{\text{def}}{=} locked[rs] \cup \neg locked[res(S) \setminus rs]$ to refer to a specific lattice element in action pre-/postconditions. The definition of

[7] For simplicity, we removed the **exit** and **commit** statements.

Act_2 requires that expressions rs_1 have not been acquired and in such a case adds the correct dataflow facts to the postcondition. Act_3 ensures that a **rel** statement releases all locks. To capture the control flow of conditions, Act_4 records the lattice element upon entry to the condition and Act_5 re-establishes that element in the else branch.

Notice that since update statements may only appear before the commit statement (by a control dependence), then $Goal_3$ restricts lock statements to appear before **commit**. Together with $Goal_4$, they restrict lock statements to appear inside the atomic section.

We ensure $C1$ via the fluents $\{read[r], write[r] \mid r \in res(S)\}$ and $Goal_1$. We ensure $C2$ by $Goal_2$. We ensure $C3$ by Act_6 — all locks should be released as a precondition to committing or exiting a condition via the else branch (which is where a **exit** unit would appear). We ensure $C4$ by $Goal_3$. Finally, we ensure $C5$ by having plans that choose **lock** rs_1 **ctx** rs_2 statements that satisfy the lockset analysis defined above.

3.6 Elixir

Figure 3.9 summarizes the overall structure of Elixir when instantiated with our planning-based synthesis framework. Our synthesizer accepts an Elixir program and generates an implementation in three phases described next.

Enhancement. The first phase achieves two main tasks: (1) in case of a fixed-point iteration loop, it uses automatic reasoning to infer the code needed

Meta variable		Description
r		Resource expression $r \in \mathit{res}(S)$
r'		Data range expression
rs		Resource set expression $rs \subseteq \mathit{res}(S)$
u		Any unit type referenced below $u \in \mathit{flat}(S)$
$\mathit{Flnts}^{\mathit{Synch}(S)}$	$=$	$\{\mathit{locked}[r], \mathit{read}[r], \mathit{write}[r]\} \cup \{\mathit{afterUpd}\} \cup$ $\{\textbf{if}\ \ b^L\ \textbf{ctx}\ rs\} \cup \{\textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2 \setminus rs_1\} \cup$ $\{\mathit{flat}(S)\} \cup \{\mathcal{B}[\mathit{vars}(\textbf{unlock}\ rs)]\} \cup$ $\{\mathcal{B}[\mathit{vars}(\textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2 \setminus rs_1)]\}$
$\mathit{Act}^{\mathit{Synch}(S)}$	$=$	$\bigwedge_{i=1}^{7} \mathit{Act}_i^{\mathit{Synch}(S)}$
$\mathit{Act}_1^{\mathit{Synch}(S)}$	$=$	$\{\langle\rangle\ u\ \langle \mathit{read}[\mathit{rd}(u)] \cup \mathit{write}[\mathit{wt}(u)]\rangle\}$
$\mathit{Act}_2^{\mathit{Synch}(S)}$	$=$	$\{\langle \mathcal{D}[rs_2 \setminus rs_1], \mathcal{B}[\mathit{vars}(\textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2 \setminus rs_1)]\rangle$ $\textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2 \setminus rs_1$ $\langle \mathit{locked}[rs_1], \mathit{only}(\textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2 \setminus rs_1)\rangle\}$
$\mathit{Act}_3^{\mathit{Synch}(S)}$	$=$	$\{\langle \mathcal{D}[rs_1], \mathcal{B}[\mathit{vars}(\textbf{unlock}\ rs_1)]\rangle \textbf{unlock}\ rs_1$
	$=$	$\langle \mathcal{D}[\emptyset], \mathit{only}(\textbf{unlock}\ rs_1)\rangle\}$
$\mathit{Act}_4^{\mathit{Synch}(S)}$	$=$	$\{\langle \mathcal{D}[rs]\rangle\ \textbf{if}\ \ b^L\ \langle \textbf{if}\ \ b^L\ \textbf{ctx}\ rs\rangle\}$
$\mathit{Act}_5^{\mathit{Synch}(S)}$	$=$	$\{\langle \textbf{if}\ \ b^L\ \textbf{ctx}\ rs\rangle\ \textbf{else}^L\ \langle \mathcal{D}[rs]\rangle\}$
$\mathit{Act}_6^{\mathit{Synch}(S)}$	$=$	$\{\langle \mathcal{D}[\emptyset]\rangle\ \textbf{commit}\ \langle\rangle, \langle \mathcal{D}[\emptyset]\rangle\ \textbf{exit}\ \langle\rangle\}$
$\mathit{Act}_7^{\mathit{Synch}(S)}$	$=$	$\{\langle\rangle\ \mathit{Upd}\ \langle \mathit{afterUpd}\rangle\}$
$\mathit{Init}^{\mathit{Synch}(S)}$	$=$	$\emptyset$
$\mathit{Goal}^{\mathit{Synch}(S)}$	$=$	$\bigcup_{i=1}^{4} \mathit{Goal}_i^{\mathit{Synch}(S)}$
$\mathit{Goal}_1^{\mathit{Synch}(S)}$	$=$	$\{\mathsf{F}\ \mathit{locked}[r]\} \cup \{\mathit{locked}[r] \sqsubset (\mathit{read}[r] \vee \mathit{write}[r])\}$
$\mathit{Goal}_2^{\mathit{Synch}(S)}$	$=$	$\{\textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2 \setminus rs_1 \sqsubset \textbf{unlock}\ rs\}$
$\mathit{Goal}_3^{\mathit{Synch}(S)}$	$=$	$\{\mathit{locked}[r] \sqsubset \mathit{afterUpd}\}$
$\mathit{Goal}_4^{\mathit{Synch}(S)}$	$=$	$\{\textbf{for}\ x : r'\ \textbf{do} \sqsubset \textbf{lock}\ rs_1\ \textbf{ctx}\ rs_2 \setminus rs_1 \mid \neg \mathit{val}(r')\}$

Figure 3.8: ATS problem. To avoid clutter, set formers don't specify that: $L \in \mathit{Label}, u \in \mathit{flat}(S), rs, rs_2 \subseteq \mathit{res}(S), rs_1 \subseteq \mathit{res}(S) \setminus \{\}$.

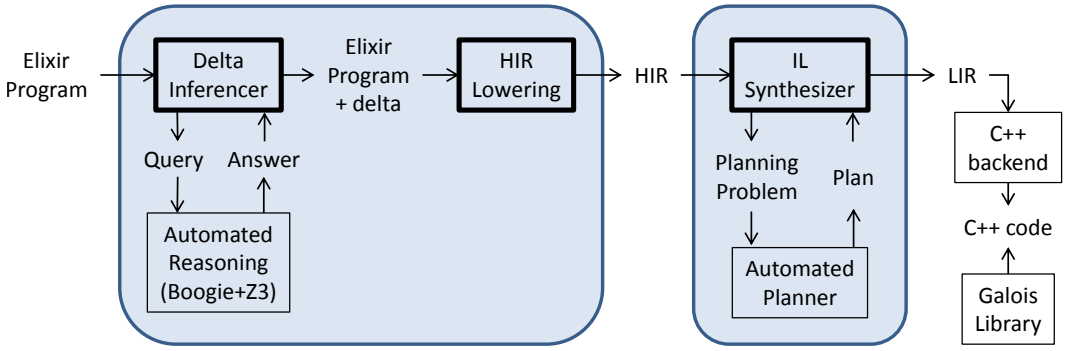


Figure 3.9: Architecture of the Elixir synthesizer.

to compute the fixpoint, which amounts to adding future redexes to the work-list; and (ii) lowering the program to a high-level intermediate language (HIR) program, which in particular explicitly represents the chosen (static) scheduling tactic.

Lowering. The second phase first simplifies the input by heuristically applying rewrite rules to remove redundant conditions from the guard of operators and unify delta clauses. It then produces a low-level intermediate language (LIR) program, as described in Section 3.4 and Section 3.5.

Code Generation. A C++ back-end generates a C++ implementation using spin locks for synchronization and the Galois system [2].

Chapter 4

Experimental Evaluation of Elixir

One of the main difficulties in writing high-performance graph analytics programs is that there is usually no single implementation that performs well for all graph types. For example, some implementations may perform well for high-diameter graphs like road-networks but perform poorly for low-diameter graphs like social-networks, and vice versa. Consequently, it may be necessary to have several implementations of the same basic algorithm, and choose the appropriate one for a given input graph, using some insight about the graph.

Elixir is the first system that allows to automatically synthesize efficient parallel implementations for the complex domain of sparse graph problems. To demonstrate its effectiveness in generating efficient parallel implementations and enabling input adaptivity, we now present an in-depth performance analysis for several challenging sparse graph problems [1].

[1] Part of the work presented in this chapter has appeared in *"Dimitrios Prountzos, Roman Manevich, Keshav Pingali. 'Synthesizing Parallel Graph Programs via Automated Planning'. In Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI) 2015,"* and *"Dimitrios Prountzos, Roman Manevich, Keshav Pingali. 'Elixir: A System for Synthesizing Concurrent Graph Programs'. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) 2012."* The first author is responsible for the conception and the implementation of the ideas presented in these publications. Additional authors provided assistance with the presentation of the material.

Our evaluation methodology is as follows: For each problem we consider an implementation space consisting of different Elixir schedules capturing algorithmic insights, and different plans capturing different implementation-level insights. Using Elixir we automatically generate a large number of variants and then use exhaustive search over the implementation space to find the best-performing variants for a number of interesting inputs. Subsequently, we compare these best-performing solutions against hand-optimized implementations by expert programmers.

The hand-optimized solutions we compare against are implemented on top of the state-of-the-art Galois, Cilk and OPENMP frameworks. While we cannot know what would be the fastest hand-tuned implementation for the problems we study, we believe that all the solutions we compare against are very competitive. The interested reader is referred to [132] for a recent study by Intel, which compares existing graph frameworks and shows that Galois performs competitively against other frameworks as well as hand-written expert implementations. Similarly, [109] compares favorably Galois with other popular frameworks, such as Ligra [139] and GraphLab [92].

Additionally, we note that some of the solutions we compare against use customized, elaborate synchronization requiring expert parallel-programming skills. For example, the hand-written single-source shortest path, breadth-first-search, and connected-components solutions on top of Galois, as well as all the Cilk-based solutions, use specialized lock-free synchronization (not automatically supported by Galois), which can be more efficient than the default

Galois synchronization.

This chapter is organized into two main sections: In Section 4.1 we focus on the use of the Elixir scheduling language as a means to explore different implementations. In Section 4.2 we focus on the use of the planning infrastructure to explore complementary dimensions of the implementation space. Each section contains detailed performance analysis for a number of interesting case-studies.

4.1 Exploring Elixir Schedules

In this section we demonstrate a subset of the capabilities of Elixir by exploring an implementation space consisting purely of different Elixir schedules and considering one arbitrary plan for each such schedule. To evaluate the effectiveness of Elixir, we perform studies on three problems: single-source shortest path (`SSSP`), breadth-first-search (`BFS`), and betweenness centrality (`BC`). In the following paragraphs we present detailed experimental results for each of these problems and show how varying the schedule leads to programs with different performance.

Section 4.1.1 discusses in details the dimensions of the implementation space that we considered in our case-studies. Section 4.1.2 presents details of the experimental setup that we considered. Section 4.1.3 presents in detail results for the `SSSP` problem, Section 4.1.4 discusses results for the `BFS` problem, and Section 4.1.5 presents results for the `BC` problem.

Dimension	Value Range
Worklist (WL)	{`CF`, `CL`, `OBM`, `BS`, `LGEN`, `LOMP`}
Group (GR)	{a, b, `NONE`}
Unroll Factor (UF)	$\{0, 1, 2, 10, 20, 30\}$
VC Check (VC)	{`ALL`,`NONE`,`LOCAL`}
SC Check (SC)	{`ALL`,`NONE`}

Table 4.1: Dimensions explored by our synthesized algorithms.

4.1.1 Design Space

Table 4.1 shows the dimensions of the design space supported in Elixir, and for each dimension, the range of values explored in our evaluations.

Worklist Policy (WL): The dynamic scheduler is implemented by a worklist data structure. To implement the `LIFO`, `FIFO`, and `approx metric` policies, Elixir uses worklists from the Galois system [88, 108]. These worklists can be composed to provide more complex policies. To reduce overhead, they manipulate chunks of work-items. We refer to the chunked versions of FIFO/LIFO as `CF/LF` and to the (approximate) metric-ordered worklist composed with a `CF` as `OBM`. We implemented a worklist (`LGEN`) to support general, level-by-level execution (`metric` policy). For some programs, Elixir can prove that only two levels are active at any time. In these cases, it can synthesize an optimized, application-specific scheduler using OpenMP primitives (`LOMP`). Alternatively, it can use a bulk-synchronous worklist (`BS`) provided by the Galois library.

Grouping: In the case of the `SSSP` `relaxEdge` operator, we can group either on a, or b, creating a "push-based" or a "pull-based" version of the algorithm. Additionally, Elixir uses grouping to determine the type of worklist items. For example, worklist items for `SSSP` can be edges `(a,b)`, but if the `group b` directive is used, it is more economical to use node `a` as the worklist item. In our benchmarks, we consider using either edges or nodes as worklist items, since this is the choice made in all practical implementations.

Unroll Factor: Unrolling produces a composite operator. This operator explores the subgraph in a depth-fist order.

Shape/Value constraint checks (VC/SC): We consider the following class of heuristics to optimize the worklist manipulation. After the execution of an operator *op*, the algorithm may need to insert into the worklist a number of matchings μ, which constitute the delta of *op*. Before inserting each such μ, we can check whether the shape constraint (SC) and/or the value constraint (VC) is satisfied by μ, and if it is not, avoid inserting it, thus reducing overhead. Eliding such checks at this point is always safe, with the potential cost of populating the worklist with useless work.

In practice, there are many more choices such as the order of checking constraints and whether these constraints are checked completely or partially. In certain cases, eliding check c_i may be more efficient since performing c_i may require holding locks longer. Elixir allows the user to specify which SC/VC

checks should be performed and provides three default, useful polices: `ALL` for doing all checks, `NONE` for doing no checks, and `LOCAL` for doing only checks that can be performed by using graph elements already accessed by the currently executing operator. The last one is especially useful in the context of parallel execution. In cases where both VC and SC are applied, we always check them in the order SC, VC.

4.1.2 Implementation and Experimental Details

We use Elixir to automatically enumerate and synthesize a number of program variants for each problem, and compare the performance of these programs to the performance of existing hand-tuned implementations. In the `SSSP` comparison, we use a hand-parallelized code from the Lonestar benchmark suite [79]. In the `BFS` comparison, we use a hand-parallelized code from Leiserson and Schardl [87], and for `BC`, we use a hand-parallelized code from Bader and Madduri [8]. In all cases, our synthesized solutions perform competitively, and in some cases, they outperform the hand-optimized implementations. More importantly, these solutions were produced through a simple enumeration-based exploration strategy of the design space, and do not rely on expert knowledge from the user's part to guide the search.

Elixir produces both serial and parallel C++ implementations. Intuitively, Elixir generates a (parallel) loop iterating over the contents of a worklist W containing the redexes that remain to be executed. The dynamic component of the schedule denotes the order in which the we iterate over the contents

of W. The static component corresponds to a hard-coded schedule of multiple operator instances that are executed on each loop iteration, starting from an initial (potentially partial) redex. Each iteration may conditionally schedule new redexes. Different plans correspond to different implementations of this loop iteration. Elixir relies on a host runtime to provide a parallel loop construct. The synthesized code also assumes there is a graph data structure that supports a generic API with methods such as '**for** b : `Succ(a)` **do**'. In our experiments we used parallel loops, graphs, and work-lists from the Galois runtime. The graph implementations are variants of the Compressed Sparse Row (CSR) format. Implementations of standard collections such as sets and vectors are taken from the C++ standard library. Galois provides its own synchronization but we disabled this feature, and use Elixir-synthesized synchronization following the *order-and-spin* policy for the three case-studies presented in this section.

In our experiments, we use the following input graph classes:

Road networks: These are real-world, road network graphs of the USA from the DIMACS shortest paths challenge [1]. We use the full USA network (*USA-net*) with 24M nodes and 58M edges, the Western USA network (*USA-W*) with 6M nodes and 15M edges,and the Florida network (FLA) with 1M nodes and 2.7M edges.

Scale-free graphs: These are scale-free graphs that were generated using the tools provided by the SSCA v2.2 benchmark [6]. The generator is based

on the Recursive MATrix (R-MAT) scale-free graph generation algorithm [27]. The size of the graphs is controlled by a $SCALE$ parameter; a graph contains $N = 2^{SCALE}$ nodes, $M = 8 \times N$ edges, with each edge having strictly positive integer weight with maximum value $C = 2^{SCALE}$. For our experiments we removed multi-edges from the generated graphs. We denote a graph of $SCALE = X$ as $rmatX$.

Random graphs: These graphs contain $N = 2^k$ nodes and $M = 4 \times N$ edges. There are $N - 1$ edges connecting nodes in a circle to guarantee the existence of a connected component and all the other edges are chosen randomly, following a uniform distribution, to connect pairs of nodes. We denote a graph with $k = X$ as $randX$.

We ran our experiments on an Intel Xeon machine running Ubuntu Linux 10.04.1 LTS 64-bit. It contains four 6-core 2.00 GHz Intel Xeon E7540 (Nehalem) processors. The CPUs share 128 GB of main memory. Each core has a 32 KB L1 cache and a unified 256 KB L2 cache. Each processor has an 18 MB L3 cache that is shared among the cores. For `SSSP` and `BC` the compiler used was GCC 4.4.3. For `BFS`, the compiler used was Intel C++ 12.1.0. All reported running times are the minimum of five runs. The chunk sizes in all our experiments are fixed to 1024 for `CF` and 16 for `CL`.

One aspect of our implementation that we have not optimized yet is the initialization of the worklist, before the execution of a parallel loop. Our current implementation simply iterates over the graph, checks the operator

Dimension	Value Ranges
Group	$\{a, b,$ `NONE`$\}$
Worklist	{`CF`, `OBM`, `LGEN`}
Unroll Factor	$\{0, 1, 2, 10, 20, 30\}$
VC check	{`ALL`,`NONE`}
SC check	{`ALL`,`NONE`}

Table 4.2: Dimensions explored by our synthetic `SSSP` variants.

guards and populates the worklist appropriately when a guard is satisfied. In most algorithms, the optimal worklist initialization is much simpler. For example, in `SSSP` we just have to initialize the worklist with the source node (when we have nodes as worklist items). A straightforward way to synthesize this code is to ask the user for a predicate that characterizes the state before each parallel loop. For `SSSP`, this predicate would assert that the distance of the source is zero and the distance of all other nodes is infinity. With this assertion, we can use our delta inference infrastructure to synthesize the optimal worklist initialization code. This feature is not currently implemented, so the running times that we report (both for our programs and programs that we compare against) exclude this part and include only the parallel loop execution time.

4.1.3 Single-Source Shortest Path

We synthesize both ordered and unordered versions of single-source shortest-path (`SSSP`). In Table 4.2, we present the range of explored values in each dimension for the synthetic `SSSP` variants. In Table 4.3, we present the

Variant	GR	WL	UF	VC	SC	f_{Pr}
`v50`	b	`OBM`	2	✓	✓	ad/Δ
`v62`	b	`OBM`	2	×	✓	ad/Δ
`v63`	b	`OBM`	10	×	✓	ad/Δ
`dsv7`	b	`LGEN`	0	✓	✓	ad/Δ

Table 4.3: Chosen values and priority functions (f_{Pr}) for best performing `SSSP` variants (✓denotes `ALL`, × denotes `NONE`).

combinations that lead to the three best performing asynchronous `SSSP` variants (`v50`, `v62`, `v63`) and the best performing delta-stepping variant (`dsv7`). In Figure 4.1 we compare their running times with that of an asynchronous, hand-optimized Lonestar implementation on the FLA and USA-W road networks. We observe that in both cases the synthesized versions outperform the hand-tuned implementation, with the leveled version also having competitive performance.

All algorithms are parallelized using the Galois infrastructure, they use the same worklist configuration, with $\Delta = 16384$, and the same graph data-structure implementation. The value of Δ was chosen through enumeration and gives the best performance for all variants. The Lonestar version is a hand-tuned lock-free implementation, loosely based on the classic delta-stepping formulation [104]. It maintains a worklist of pairs $[v, dv^*]$, where v is a node and dv^* is an approximation to the shortest path distance of v (following the original delta-stepping implementation). The Lonestar version does not implement any of our static scheduling transformations. All synthetic variants perform fine grained locking to guarantee atomic execution of

operators, checking of the VC and evaluation of the priority function. For the synthetic delta-stepping variant dsv7 Elixir uses LGEN since new work after the application of an operator can be distributed in various (lower) priority levels. An operator in dsv7 works over a source node a and its incident edges (a, b), which belong to the same priority level.

In Figure 4.2 we present the runtime distribution of all synthetic SSSP variants. Here we summarize a couple of interesting observations from studying the runtime distributions in more detail. By examining the ten variants with the worst running times, we observed that they all use a CF (chunked FIFO) worklist policy and are either operating on a single edge or the immediate neighbors of a node (through grouping), whereas the ten best performing variants all use OBM. This is not surprising, since by using OBM there are fewer updates to node distances and the algorithm converges faster. To get the best performance though, we must combine OBM with the static scheduling transformations. Interestingly, combining the use of CF with grouping and aggressive unrolling (by a factor of 20) produces a variant that performs only two to three times worse than the best performing variant on both input graphs.

4.1.4 Breadth-First Search

We experiment with both ordered and unordered versions of the BFS. In Table 4.4 and Table 4.5, we present the range of explored values for the synthetic BFS variants and the combinations that give the best performance, respectively. In Figure 4.3, we present a runtime comparison between the

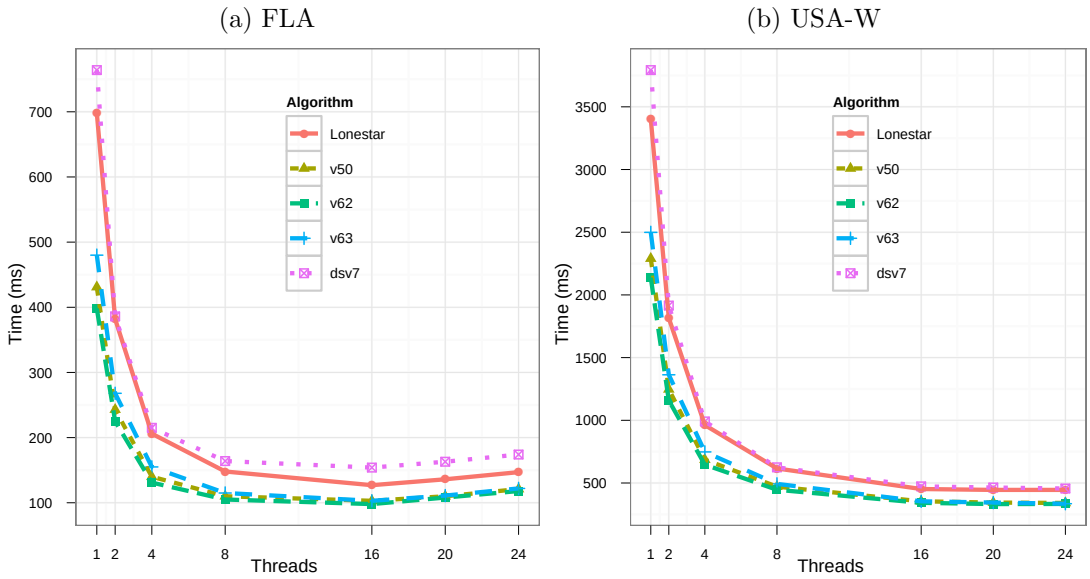


Figure 4.1: Runtime comparison of SSSP algorithms.

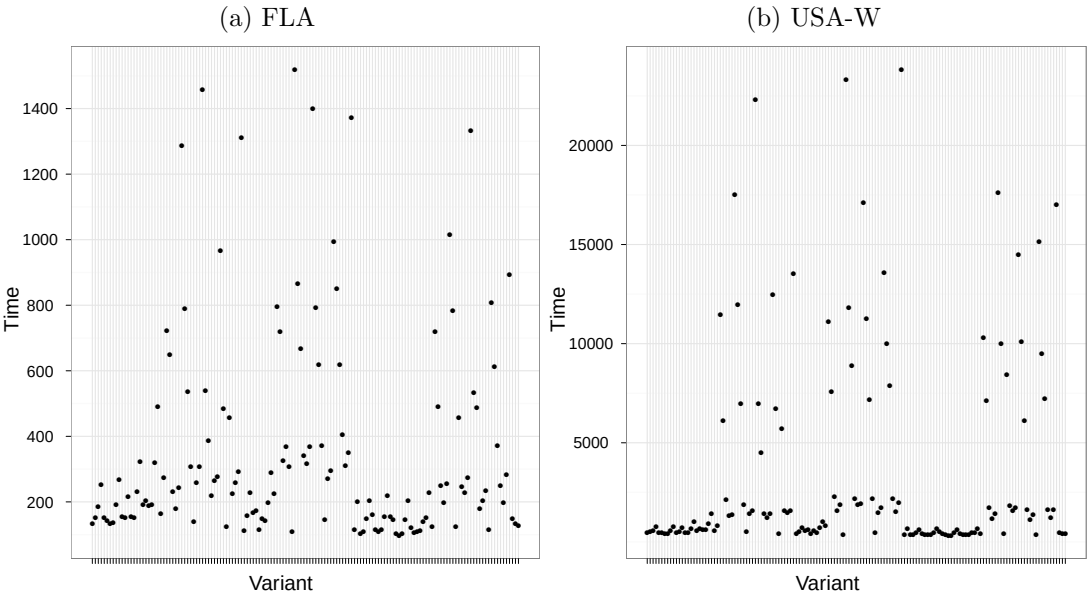


Figure 4.2: Runtime distribution of all synthetic SSSP variants.

three best-performing `BFS` variants (both asynchronous and leveled), and two highly optimized, handwritten, lock-free parallel `BFS` implementations. The first handwritten implementation is from the Lonestar benchmark suite and is parallelized using the Galois system. The second is an implementation from Leiserson and Schardl [87], and is parallelized using Cilk++. We experiment with three different graph types. For the *rmat20* and *rand23* graphs, the synthetic variants perform competitively with the other algorithms For the *USA-net* graph, they outperform the hand-written implementations at high thread counts (for 20 and 24 threads).

To understand these results, we should consider the structure of the input graphs and the nature of the algorithms. Leveled `BFS` algorithms try to balance exposing parallelism and being work-efficient by working on one level at a time. If the amount of available work per level is small, then they do not exploit the available parallel resources effectively. Asynchronous `BFS` algorithms try to overcome this problem by being more optimistic. To expose more parallelism, they speculatively work across levels. By appropriately picking the priority function, and efficiently engineering the algorithm, the goal is reduce the amount of mis-speculation introduced by eagerly working on multiple levels. Focusing on the graph structure, we observe that scale-free graphs exhibit the small-world phenomenon; most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of "hops". This means that the diameter of the graph and the number of levels is small (12 for *rmat20*). The random graphs that we consider also have a small

diameter (17 for *rand23*). On the other hand, the road networks, naturally, have a much larger diameter (6261 for *USA-net*). The smaller the diameter of the graph the larger the number of nodes per level, and therefore the larger the amount of available work to be processed in parallel. Our experimental results support the above intuitions. For low diameter graphs we see that the best performing synthetic variants are, mostly, leveled algorithms (`v17`,`v18`, `v19`). For *USA-net* which has a large diameter, the per-level parallelism is small, which makes the synthetic asynchronous algorithms (`v11`, `v12`, `v14`) more efficient than others. In fact, at higher thread counts (above 20) they manage to, marginally, outperform even the highly tuned hand-written implementations. For all three variants we use $\Delta = 8$. This effectively, merges a small number of levels together and allows for a small amount of speculation, which allows the algorithms to mine more parallelism. Notice that, similarly to `SSSP`, all three asynchronous variants combine some static scheduling (small unroll factor plus grouping) with a good dynamic scheduling policy to achieve the best performance.

The main take-away message from these experiments is that no one algorithm is best suited for all inputs, especially in the domain of irregular graph algorithms. This validates our original assertion that a single solution for an irregular problem may not be adequate, so it is desirable to have a system that can synthesize competitive solutions tailored to the characteristics of the particular input.

For level-by-level algorithms, there is also a spectrum of interesting

Dimension	Value Ranges
Group	{b, NONE}
Worklist	{OBM, LOMP, BS}
Unroll Factor	$\{0, 1, 2\}$
VC check	{ALL,NONE}
SC check	{ALL,NONE}

Table 4.4: Dimensions explored by our synthetic BFS variants.

choices for the worklist implementation. Elixir can deduce that BFS under the metric ad scheduling policy can have only two simultaneously active priority levels. Therefore, it can use a customized worklist in which a bucket B_k holds work for the current level and a bucket B_{k+1} holds work for the next. Hence, we can avoid the overheads associated with LGEN, which supports an unbounded number of buckets. BS is a worklist that can be used to exploit this insight. Additionally, since no new work is added to B_k while working on level k, threads can scan the bucket in read-only mode, further reducing overheads. Elixir exploits both insights by synthesizing a custom worklist LOMP using OpenMP primitives. LOMP is parameterized by an OpenMP scheduling directive to explore load-balancing policies for the threads querying B_k (in our experiments we used the STATIC policy).

4.1.5 Betweenness Centrality

The betweenness centrality (BC) of a node is a metric that captures the importance of individual nodes in the overall network structure. Informally, it is defined as follows. Let $G = (V, E)$ be a graph and let s, t be a fixed pair

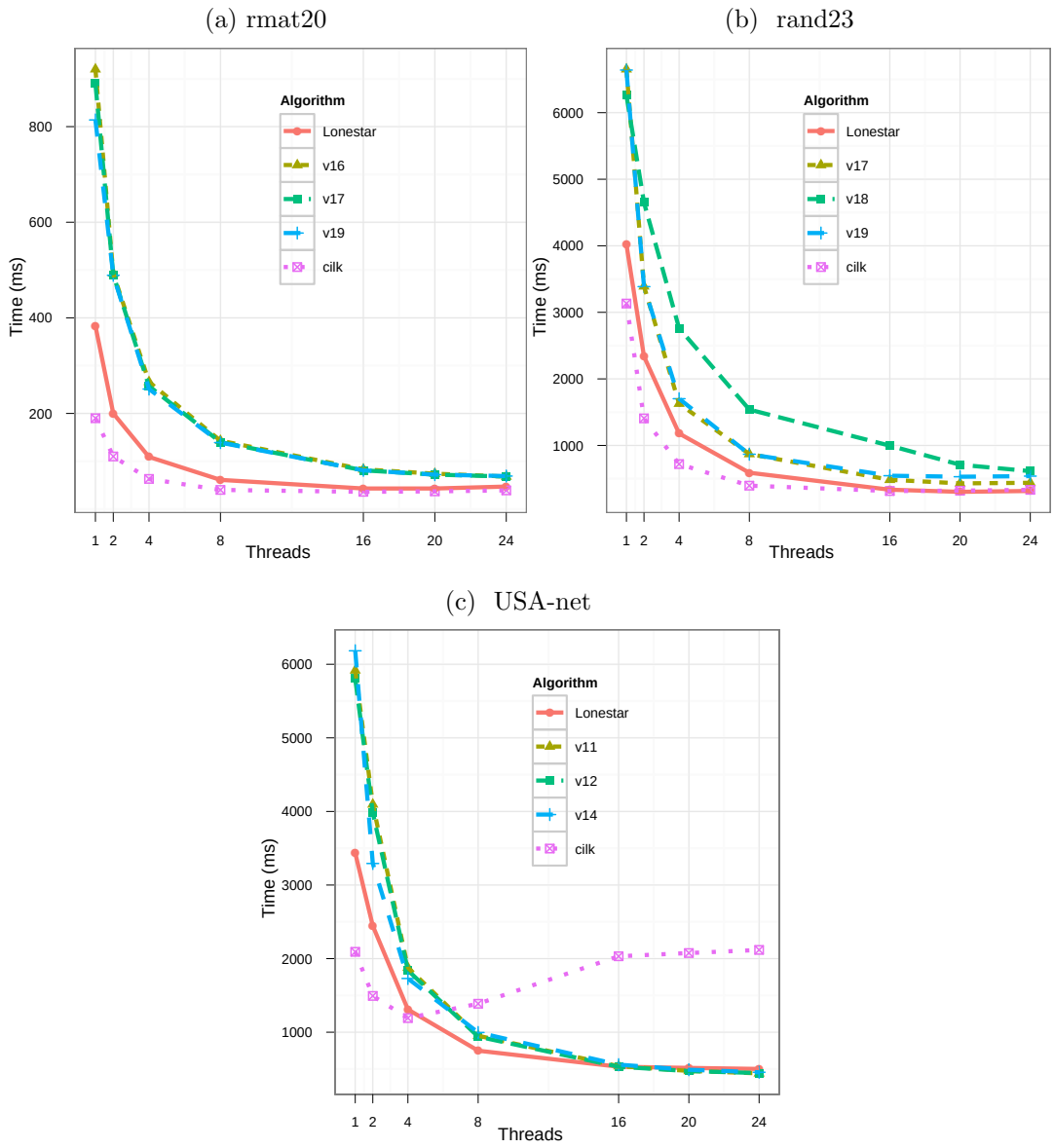


Figure 4.3: Runtime comparison of BFS algorithms.

Variant	GR	WL	UF	VC	SC	f_{Pr}
v11	b	OBM	1	✓	✓	ad/Δ
v12	b	OBM	2	✓	✓	ad/Δ
v14	b	OBM	1	✓	×	ad/Δ
v16	b	OBM	0	✓	×	ad/Δ
v17	b	BS	0	✓	✓	ad
v18	b	LOMP	0	✓	✓	ad
v19	b	BS	0	✓	×	ad

Table 4.5: Chosen values and priority functions for BFS variants. We chose $\Delta = 8$. (✓denotes ALL, × denotes NONE.)

of graph nodes. The betweenness score of a node u is the fraction of shortest paths between s and t that include u. The betweenness centrality of u is the sum of its betweenness scores for all possible pairs of s and t in the graph. The most well known algorithm for computing BC is Brandes' algorithm [20]. In short, Brandes' algorithm considers each node s in a graph as a source node and computes the contribution due to s to the betweenness value of every other node u in the graph as follows: In a first phase, it starts from s and explores the graph forward building a DAG with all the shortest path predecessors of each node. In a second phase it traverses the graph backwards and computes the contribution to the betweenness of each node. These two steps are performed for all possible sources s in the graph. For space efficiency, practical approaches to parallelize BC (e.g. [8]) focus on processing a single source node s at a time, and parallelize the above two phases for each such s. Additionally, since it is computationally expensive to consider all graph nodes as possible source nodes, they consider only a subset of source nodes (in practice this provides a

Dimension	Forward Phase Ranges	Backward Phase Ranges
Group	{b, NONE}	{a}
Worklist	{LOMP, BS}	{CF}
Unroll Factor	{0}	{0}
VC check	{ALL,NONE}	{LOCAL}
SC check	{ALL,NONE}	{ALL,NONE}

Table 4.6: Dimensions explored by the forward and backward phase in our synthetic BC variants.

Variant	GR	WL	UF	VC	SC	f_{Pr}
v1	NONE	BS	0	(✓,L)	(✓,✓)	ad
v14	b	LOMP	0	(✓,L)	(✓,✓)	ad
v15	b	BS	0	(✓,L)	(×,×)	ad
v16	b	LOMP	0	(✓,L)	(×,×)	ad
v24	b	LOMP	0	(×,L)	(×,×)	ad

Table 4.7: Chosen values and priority functions for BC variants (✓denotes ALL, × denotes NONE, L denotes LOCAL). For the backward phase there is a fixed range of values for most parameters (see Table 4.6). In the SC column the pair (F, B) denotes that F is used in the forward phase and B in the backward phase. f_{Pr} is the priority function of the forward phase.

good approximation of betweenness values for real-world graphs [7]).

In Table 4.6 and Table 4.7, we present the range of explored values for the synthetic BC variants and the combinations that give the best performance, respectively. We synthesized solutions that perform a leveled parallelization of the forward phase and an asynchronous parallelization of the backward phase. In Figure 4.4 we present a runtime comparison between the three best performing BC variants, and a hand-written, OpenMP parallel BC implementation

by Bader and Madduri [8], which is publicly available in the SSCA benchmark suite [6]. All algorithms perform the computation outlined above for the same five source nodes in the graph, *i.e.* they execute the forward and backward phases five times. The reported running times are the sum of the individual running times of all parallel loops.

We observe that in the case of the *USA-W* road network our synthesized versions manage to outperform the hand-written code, while in the case of *rmat20* graph the hand-written implementation outperforms our synthesized versions. We believe this is mainly due to the following reason. During the forward phase, both the hand-written and synthesized versions build a shortest path DAG by recording for each node u, a set $p(u)$ of shortest path predecessors of u. The set $p(u)$ therefore contains a subset of the immediate neighbors of u. In the second phase of the algorithm, the hand-written version walks the DAG backward to update the values of each node appropriately. For each node u, it iterates over the contents of $p(u)$ and updates each $w \in p(u)$ appropriately. Our synthetic codes instead examine all incoming edges to u and use $p(u)$ to dynamically identify the appropriate subset of neighbors and prune out all other in-neighbors. In the case of *rmat* graphs, we expect that the in-degree of authority nodes to be large, while in the road network case the maximum in-degree is much smaller. We expect therefore our iteration pattern to be a bottleneck in the first class of graphs. A straightforward way to handle this problem is to add support in our language for multiple edge types in the graph. By having explicit predecessor edges in the graph instead of considering $p(u)$

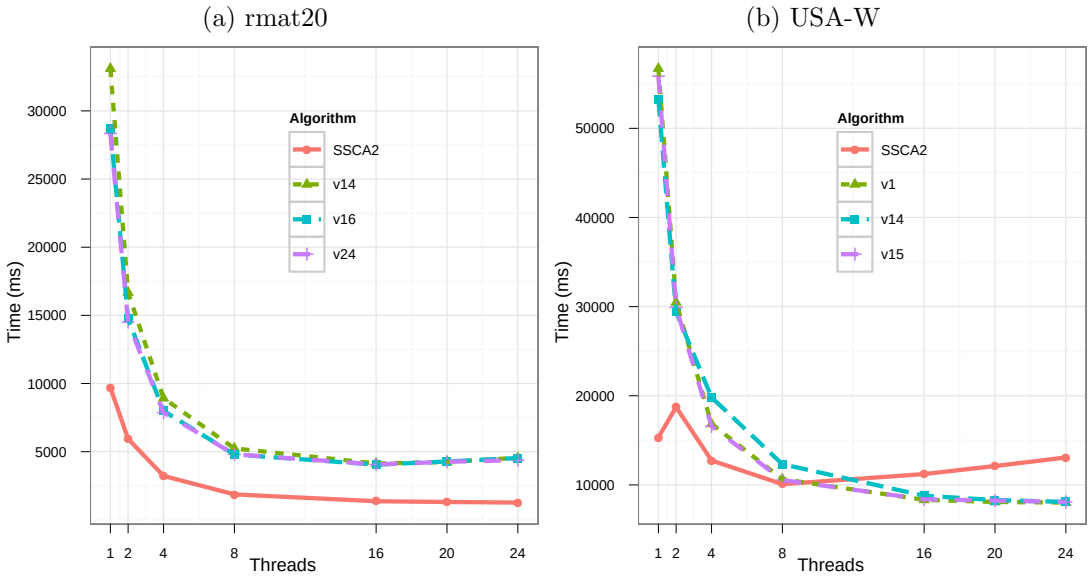


Figure 4.4: Runtime comparison of BC algorithms.

as yet another attribute of u, our delta inference algorithm will be able to infer the more optimized iteration pattern. We plan to add this support in future extensions of our work.

4.2 Exploring Elixir Plans

In this section we demonstrate a different subset of the capabilities of Elixir by fixing the number of Elixir schedules to a small number and exploring an implementation space consisting of different plans for each of the Elixir schedules. Such plans capture insights about different ways of implementing synchronization policies or encoding different iteration patterns over the elements of a graph by using the graph ADT API. In the following paragraphs

we present results for the following four complex graph problems: maximal independent set, triangle counting, preflow-push, and connected components.

Section 4.2.1 gives details of the experimental evaluation and experimental setup of the case-studies. Section 4.2.2 presents results for the maximal independent set problem, Section 4.2.3 presents experiments for the triangle counting problem, Section 4.2.4 discusses results for the connectected components problem, and Section 4.2.5 discusses the preflow-push problem.

4.2.1 Implementation and Experimental Details

In the case-studies presented in this section we used parallel loops, graphs, and work-lists from the Galois runtime, similarly to the experiments in Section 4.1. Our case-studies require speculation-based synchronization for atomic operator execution. Again, we disable the default Galois speculation-based synchronization scheme and guide Elixir to synthesize customized speculation implementations based on ATS.

We performed our experiments on a 40-core machine with Intel Xeon E7-4860 hyper-threaded processors, with 24MB of L3 cache and 128GB of main memory. The operating system is Scientific Linux 6.3 and the compiler is GCC 4.8.1 (-O2). We used four kinds of graphs: (i) the DIMACS USA road network (24M nodes and 58M edges), (ii) the *wikipedia-2007* graph (3.5M nodes and 8M edges), (iii) the *rmat24* graph (a=0.5,b=c=0.1,d=0.3), which is a synthetic scale-free graph (16M nodes,268M edges), and (iv) a number of random graphs dubbed *randX* (2^X nodes,4×2^X edges). Since all our input

problems are defined over undirected graphs we preprocess the inputs to add symmetric edges, in case they are not already present in the graph. We also removed multi-edges (otherwise we would not even have a graph). We present graphs showing the runtime distribution of Elixir solutions, as well as graphs comparing their performance against the manual implementations. In order to improve clarity of exposition in the comparison graphs we normalize runtimes against the fastest single-thread runtime across all implementations (manual and synthesized) and we report speedups over that baseline. All reported times are the medians of five runs and baseline times are presented in figure captions.

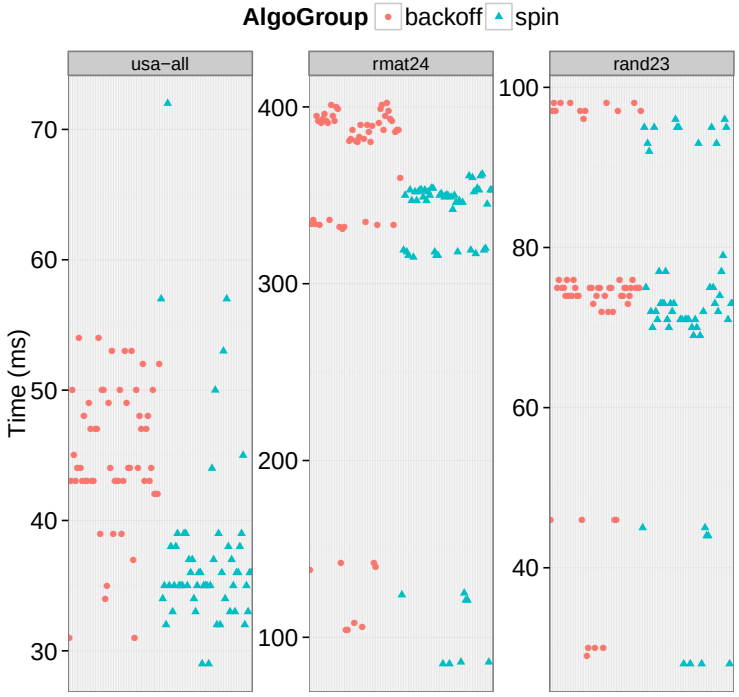
4.2.2 Maximal Independent Set

We explore a space of `MIS` implementations by considering different variations of ATS speculative locking and different HIR schedules. We generated 128 variants in total. Their main differences are:

Lock acquire and release for $\mathcal{N}(\mathbf{a})$: The *incremental* strategy fuses the evaluation of the $\forall$ predicate with locking $\mathcal{N}(\mathrm{a})$, whereas *one-shot* locks $\mathcal{N}(\mathrm{a})$ before evaluating it. Different release policies can be implemented by releasing subsets of locks during the execution of *map* and releasing the rest at the operator end.

Conflict resolution: The *spin* strategy keeps trying to dispatch the same work-item *wi* till it succeeds; the *back-off* strategy chooses a different

(a) Elixir Variants Runtime Distribution



(b) Runtime Comparison

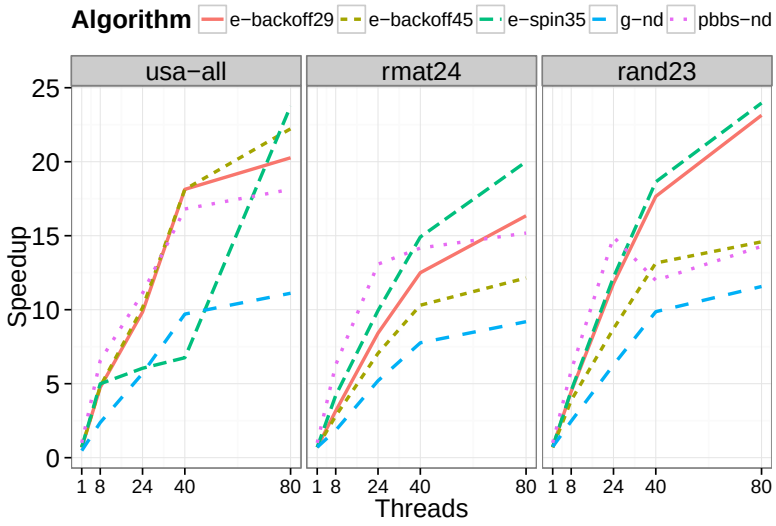


Figure 4.5: `MIS` variants runtime-distribution and comparison with hand-written codes. Base-time (ms): 689 (usa-all), 1700 (rmat24), 671 (rand23).

work-item and inserts *wi* in a special abort queue where it is processed by a special thread to guarantee forward progress (the default Galois policy).

The product of all these choices gives us a set of variants that use different ways to grow and shrink the atomic section corresponding to the *match* operator, and to resolve conflicts. Figure 4.5a presents the distribution of best runtimes for all Elixir variants on three input graphs. In Figure 4.5a we cluster variants in two families, based on the conflict resolution policy. Figure 4.5b presents comparisons of the best Elixir variants for each input and family against hand-written implementations. For performance comparisons, we used the Galois program `g-nd`, which employs the `match` operator but uses the default transactional execution scheme of the Galois system implemented with *stamp-and-log* within the Galois data structure library; conflicts are resolved using back-off. Additionally, we used *pbbs-nd*, the non-deterministic parallelization from the Problem Based Benchmark Suite (PBBS) [16]. *pbbs-nd* is a lock-free parallelization using the Cilk runtime (compiled with ICC 13.1). In Figure 4.5b the name *e-spin*$\langle i \rangle$ (*e-backoff*$\langle i \rangle$) denotes the i-th spin-based (backoff-based) Elixir variant. The key observations are:

First, for all inputs, spinning is a better conflict resolution policy than back-off, the default Galois policy. The best synthesized spin version, *spin-35* outperforms the Galois version by more than a factor of 2. This shows the advantages of customizing synchronization policies to applications. Additionally,

as shown in Figure 4.5a even among Elixir variants spin-based variants tend to be better performing than backoff-based variants. One could expect that backoff is always better since it prevents live-locks from happening. However, depending on the sparsity of the input graphs and the operator structure, the probability of live-locks may be small. In that case, it may be more beneficial to simply retry dispatching the same redex, instead of paying the overhead of an always-on runtime mechanism that prevents livelocks.

Second, one-shot locking performs better than incremental locking, as is seen from the performance of *e-spin35* and *e-backoff29*, which use one-shot locking, and *e-backoff45* which uses incremental locking. It is likely that this is because conflicts are detected earlier and there is less wasted work from aborts, even though locks are held longer.

Third, early release helps marginally. For example, for the road network graph and the backoff family the runtime of the best performing early-release variant is 84% of the runtime of the best variant without this optimization. For other graphs the best early-release variant runtime is 96% of the best non-early-release runtime. Reducing the size of atomic sections is definitely a useful optimization, since it decreases the probability of conflict. In our setting, the benefit would be mostly observable when `match` is applied to a high-degree node. Since each `match` application on a node a renders all its neighbors *NMatched*, the probability of successfully applying `match` on an *Unmatched* high-degree node is quite low. Finally, we note that the best Elixir variants perform competitively with *pbbs-nd*, and for some inputs can even outperform

it. The results can be partially attributed to parallelizing solutions on top of different runtime systems.

4.2.3 Triangle Counting

An Elixir solution to the *Triangles* problem uses a single, *count* operator that checks whether a triple of nodes a, b, c form a triangle. We define a space of implementations by experimenting with different schedules for the *count* operator, and by conditionally customizing the synthesized implementations to exploit structural invariants of the input graph. First, with respect to Elixir schedules, we use the `group` tactic to create a "blocked" composite operator that co-schedules multiple instances of *count*. For example, the schedule '`count` $\gg$ `group a,b`' starts from a specific node a and considers all possible bindings for b and c. Alternatively, we can start from b or c and perform similar blocked explorations. Second, we consider input graphs where the neighbors of each node are sorted in increasing order. Communicating this information to our system allows the planner to use tiles encoding specialized strategies of iterating over the neighbors of each node.

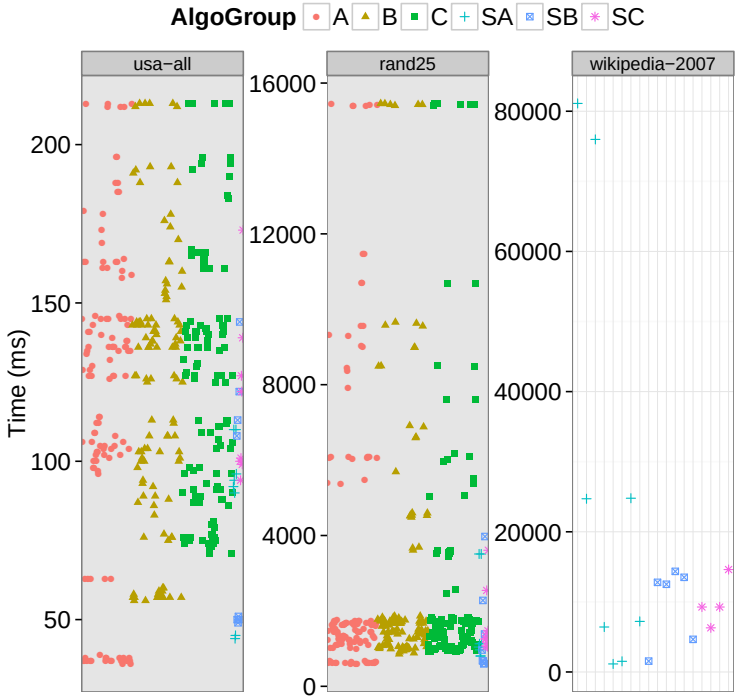
These two design parameters give us six different algorithm families: *A, B, C, SA, SB, SC*, with the first letter denoting the starting node of the blocked operator, and the conditional prefix S denoting whether sorted property is taken into consideration or not. For each family, our planner enumerates a number of solutions that correspond to the different schedules of evaluating the operator. In total, we have 384 variants. Figure 4.6a presents the distribution

of best runtimes for the Elixir variants on three input graphs. Figure 4.6b compares the best performing variants against a version of the *node-iterator* algorithm [133] (`galois-ni`) implemented using the Galois system. The key observations are:

First, no single variant performs best for all input graphs. In order to get the best performing solution for each input we need to customize the implementation to properties of the input graph. The relative performance of Elixir variants in Figure 4.6a shows that the best performing variants for the road network and random graphs use simple implementations of iterating over the node neighbors. Since the average node degree is small and uniform, a simple strategy of iterating through all neighbors can incur less overhead than a more elaborate strategy that tries to find the right set of neighbors to start iterating over. This effect is more obvious in the road network graph, while in the random graph the differences between the different families are smaller. When we move to the scale-free Wikipedia graph however, which has few nodes with very high connectivity, the more elaborate iteration pattern is essential in getting performance. For this input, we do not report times for variants in the A, B, C families because their running time (80 threads) exceeded a timeout of 300 seconds.

Second, we note that the schedule of evaluating the operator constraints greatly affects performance. For example, as we can see in Figure 4.6a variants of the A, SA families outperform Elixir variants in SB across all examined inputs. Moreover, there is variation even within the same family. Studying

(a) Elixir Variants Runtime Distribution



(b) Runtime Comparison

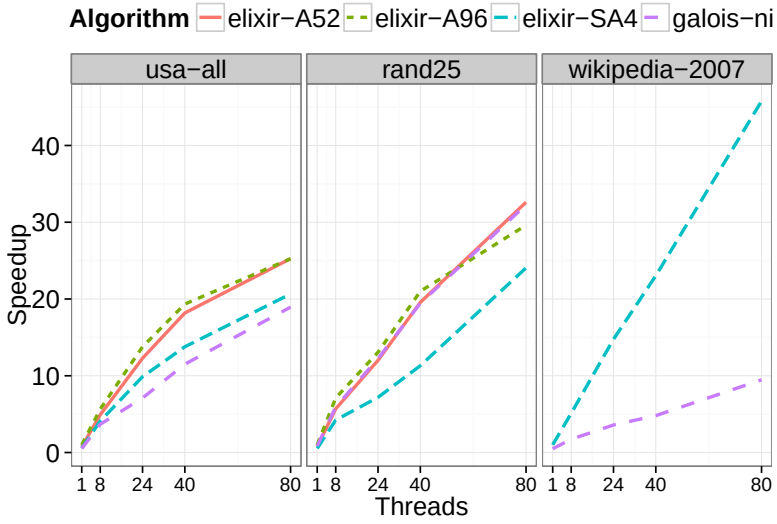


Figure 4.6: *Triangles* variants runtime-distribution and comparison with hand-written code. Base-time (ms): 909 (usa-all), 19367 (rand25), 52590 (wikipedia-2007).

the Wikipedia graph results reveals that the best Elixir variant in the SB family, which takes 1.5 seconds, iterates over all neighbors a of b, and for each a examines all potential c's. A sub-family that first iterates over all neighbors c, and for each c iterate again over the neighbors of b to find an appropriate a gives a best runtime of 12.8 seconds. Similar performance variations exist in the SA family, with the best variant being more than 4 times faster than `galois-ni` on Wikipedia (5.5 seconds), while being slower on the random graph.

The key take away is that finding the best implementation requires picking the right combination of schedule for the operator constraints, and appropriate specialization to the input graph properties. Programmer intuition can be an unreliable guide, and the ability to quickly experiment with different schedules and tiles to find the best variant for a specific input and architecture is important.

4.2.4 Connected Components

The classic formulation [69] applies the *hook* and *compress* operators non-deterministically up to a fixpoint. A popular scheduling heuristic, which is adapted both in the PRAM literature [138] and in multicore implementations [33], is to alternate *hook* and *compress* rounds, selecting the number of applied operators in each round heuristically. We generate 3200 variants by considering different operator schedules yielding different mixes of operators per round, and different plans for each schedule.

Figure 4.7 presents a comparison of the best variants for each input against two manually parallelized solutions in Galois. The first, *g-uf*, is based on the union-find-based algorithm [34] (union and find are merely *hook*/*compress* schedules). The second solution, *g-l*, is a parallelization of the label-propagation algorithm, as implemented in the Ligra framework [139]. The *g-l* solution relies on an implementation of the Ligra API on top of Galois, which performs competitively with the original Ligra implementation [109]. Both solutions use lock-free synchronization and represent a level of performance that cannot be obtained (automatically) by systems such as Galois, but instead requires expert parallel programming knowledge.

In all cases, the synthetic variants perform competitively with the hand-written implementations. For the road-network graph, *g-uf* takes 59% of the runtime of the best Elixir variant, and both are roughly two orders of magnitude faster than *g-l*. In the other cases, *g-l* outperforms *g-uf* and the best Elixir variant is faster than *g-l* (takes 95% of the *g-l* time on *rmat24* and 67% on *rand23*). The best runtime difference between the best and worst variants is at most $\times 3.3$ for *rmat24*, $\times 2$ for *usa-all*, and $\times 3.5$ for *rand23*.

To understand these results a bit better we focus on a key characteristic of the above solutions. All algorithms work greedily to identify component representatives. *g-l* is more conservative, since it performs a local search to guess a node representative. The *g-uf* and Elixir variants employ different *hook*/*compress* mixes to perform more expanded searches, with *g-uf* being the most aggressive. An expanded search improves the convergence rate, and

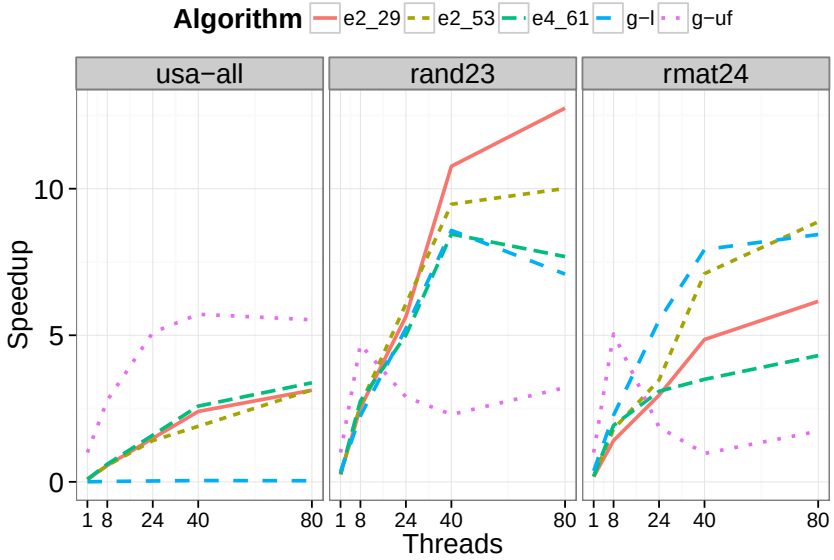


Figure 4.7: Elixir CC variants comparison with hand-written codes. Base-time (ms): 2007 (usa-all), 2282 (rand23), 7813 (rmat24).

it can be useful for long diameter graphs, such as the road network (note the very poor performance of *g-l* on this graph). However it can lead to more contention, so for graphs with smaller diameter (*rmat24*, *rand23*), *g-uf* performance decreases, whereas *g-l* improves. Unsurprisingly, how greedy the algorithm should be depends on the input. The problem with the hand-implemented solutions is that their strategy is fixed. This is a key benefit of Elixir, since it enables a more adaptive approach by automatically generating solutions with varying mixes of operators and different evaluation orders of each individual operator.

4.2.5 Preflow-Push

The preflow-push algorithm [50] is an efficient solution to the maximum-flow problem. The main algorithm kernel non-deterministically applies the *push* and *relabel* operators until reaching a fixpoint. We guide Elixir to generate solutions closely matching the static schedule of the *discharge* kernel in the *relabel-to-front* algorithm [34]. This schedule considers a node a and alternates between pushing flow to all its neighbors b and relabeling a. In addition, we use a worklist with *fifo* policy (chunk-size: 32) to dynamically schedule new redexes. For this schedule, we select the first 50 variants returned by the planner. We compare their performance against a hand-written solution on top of Galois, which uses the same worklist policy and a roughly similar static operator schedule.

Figure 4.8 shows the performance of the best-performing Elixir solutions for each input and the Galois code. We observe that for both inputs Elixir variants are competitive with the hand-written code. On the *usa-all* graph *elixir8* is the fastest and its runtime is roughly 93% of the time taken by the hand-written code. On the *rand23* graph the Galois code takes roughly 91% of the time taken by *elixir3* and 63% of the time taken by *elixir8*. For lack of space we do not present analytical plots of the best runtimes distribution of the Elixir variants. We note however, that the time of the best Elixir variant is 60% of the time of the worst Elixir variant for the *rand23* graph and 72% for the *usa-all* graph.

The three solutions differ primarily in the synchronization implemen-

tation. The Elixir variants use ATS to synchronize individual operators participating in the schedule, while the Galois variant considers the entire static schedule as a single composite transaction synchronized by *stamp-and-log*. The static schedule constitutes a cautious composite operator [115] (each individual operator is also cautious). Consequently, no state rollback is necessary to in the case a transaction aborts.

A notable difference between the two Elixir variants is the synchronization of *relabel*. This operator checks a predicate over a node and its neighbors, similar to *match* in `MIS`. In *elixir8* the locking of neighbors is performed incrementally as each neighbor is visited, while in *elixir3* locking and predicate evaluation are not fused. Elixir currently does not support synthesizing solutions that emulate the synchronization of the Galois variant, which shares locks across operators. This requires encoding a more complete dataflow analysis in our planning framework, and is a subject of future work.

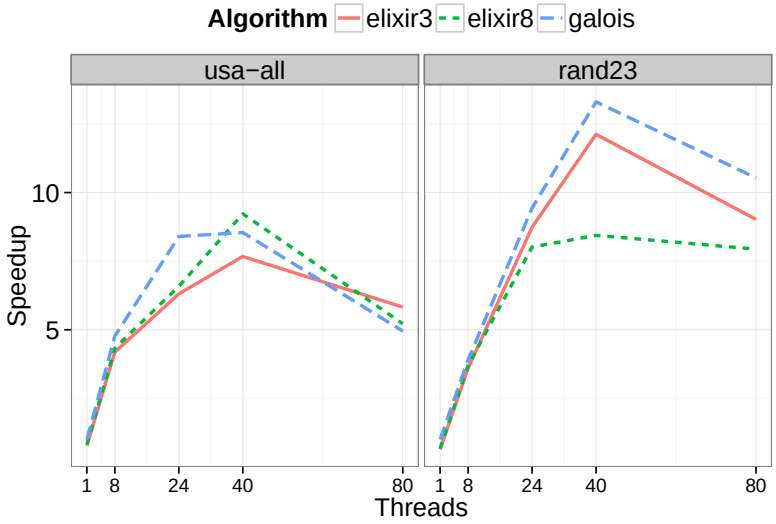


Figure 4.8: Elixir preflow-push variants comparison with Galois implementation. Base-time (ms): 8921 (usa-all), 15836 (rand23).

Chapter 5

Betweenness Centrality: An Exercise in Parallel Algorithm Design

In the previous three chapters we introduced the Elixir methodology for expressing algorithms at the high level and for automatically generating parallel implementations from such high-level specifications. Focusing on the Elixir language itself, we saw that its distinctive characteristic comes from separating the main logic of an algorithm from the specification of how to orchestrate that logic to efficiently compute a solution. Additionally, with respect to the specification of the schedule, the Elixir language is informed by a refined view of the different ingredients of the schedule — the static and dynamic components and the operator delta.

Given that Elixir provides a language that requires the programmer to think in a way that deviates from the traditional notations for expressing algorithms, a natural question to ask is how worthwhile is it for the programmer to undertake this task. Of course, in our opinion, the existence of a tool like Elixir that takes such a sequential, non-deterministic specification and automatically produces efficient parallel code is a worthwhile enough reason to consider this

approach. In this chapter [1] we would like to demonstrate an additional reason:

Utilizing the concepts of the Elixir language and methodology can help the programmer gain insights that will enable the design of new parallel algorithms for the problems they consider.

To support this point, we present a case-study on the derivation of new parallel algorithms for a well known graph analytics problem. In particular, we study the problem of computing the betweenness centrality of nodes in a graph. Our exercise starts by studying existing solutions for this problem and understanding them through the Elixir lens. Armed with the concepts of the Elixir language we dissect these solutions to operators and the different ingredients of the schedule. Having lifted existing solutions to this elemental form, we then discuss how to extend the algorithm logic to derive new families of algorithms for this problem that expose more concurrency. Subsequently, we discuss how to optimize the different aspects of the schedule (dynamic, static, operator delta) to arrive to increasingly efficient versions of the new algorithms.

The rest of the chapter is organized as follows. Section 5.1 presents background on the problem of betweenness centrality. Section 5.2 presents an operator formulation of BC and describes how existing algorithms for BC can

[1] Part of the work presented in this chapter has appeared in *"Dimitrios Prountzos, Keshav Pingali. 'Betweenness Centrality: Algorithms and Implementations'. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) 2013."* The first author is responsible for the conception and the implementation of the ideas presented in this publication. Additional authors provided assistance with the presentation of the material.

be viewed as implementations of different schedules for applying the operators to the graph. Section 5.3 describes how to derive and optimize a new asynchronous algorithm for BC by appropriately controlling operator scheduling. Finally, Section 5.4 presents our experimental evaluation on two multicore architectures using inputs from multiple graph classes.

5.1 The Problem of Betweenness Centrality

Centrality metrics are essential in understanding network structure, since they capture the relative importance of individual nodes in the overall network. Here, we examine Betweenness Centrality (BC) [44], a commonly used metric that is based on shortest path computation. If $G = (V, E)$ is a graph and s, t are a specific pair of graph nodes, the betweenness score of a node v for this node pair is the fraction of shortest paths between s and t that include v. The betweenness centrality of v is the sum of its betweenness scores for all possible pairs of s and t in the graph. More formally, let σ_{st} be the number of shortest paths between s and t, and let $\sigma_{st}(v)$ be the number of those shortest paths that pass through v. The betweenness centrality of node v is defined as: $BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$.

BC is useful in the study of diseases in sexual networks [90], finding important actors in terrorist networks [78, 31], lethality in biological networks [70, 36], and contingency analysis for power grid component failures [71]. BC is also used as a heuristic in other algorithms; for example [48] proposes an algorithm for community detection and clustering in large networks, based

on the BC of the network edges.

5.1.1 Brandes' Algorithm: a Basis for Parallel BC Algorithms

An efficient sequential algorithm for computing BC was proposed by Brandes [20], and it has been the basis for many parallelization approaches [8, 93, 40, 137, 32]. Below, we outline the main ideas behind Brandes' algorithm. We define the *dependency* of a source vertex s on a vertex v as:

$$\delta_s(v) = \sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

The betweenness centrality of a vertex v is then expressed using Eq. 5.1. The key insight is that $\delta_s(v)$ satisfies the recurrence 5.2, where $pred(s, w)$ is a list of immediate predecessors of w in the shortest paths from s to w.

$$BC(v) = \sum_{s \neq v \in V} \delta_s(v) \tag{5.1}$$

$$\delta_s(v) = \sum_{w:\ v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w)) \tag{5.2}$$

Brandes' algorithm uses this insight and it works as follows. Each $s \in V$ is considered as a source of shortest-paths and the contribution of s to $BC(v)$, for all $v \neq s$ is computed in two phases. In the first phase, a shortest-path computation is performed from s, that computes $pred(s, v)$ and σ_{sv} for all nodes. The predecessor lists induce a DAG D over the graph G. In a second phase, D is traversed backward (in non-increasing distance order) and for each $v \in V$, $\delta_s(v)$ is computed based on 5.2, and the contribution to $BC(v)$

```
1  Graph G = /* read input graph */
2  Worklist wl = {v: v ∈ G.nodes}
3  foreach s: Node ∈ wl {
4    forall v: Node ∈ G: // Compute shortestpath DAG D
5      compute σ_sv
6      compute pred(s, v)
7    forall v: Node ∈ D: // Traverse DAG D backward
8      compute δ_s(v)
9      BC(v) += δ_s(v)
10   forall (u, v): Edge ∈ G: // Reset graph attributes
11     reset (u,v)
12 }
```

Figure 5.1: Pseudocode for Brandes' algorithm.

is computed based on 5.1. The process is described in Figure 5.1. Between processing successive sources, node and edge attributes are reset.

5.1.2 Understanding Parallelism in BC

This algorithm has parallelism at multiple levels. First, we can process multiple source nodes in parallel (loop in line 3 in Figure 5.1). In this coarse-grained parallelization strategy, each thread picks an arbitrary graph node s and computes its contribution to the betweenness values of other nodes. Each of these computations is independent, and the updates on each $BC(v)$ form a simple reduction. This parallelization strategy is simple and effective, but each outer loop iteration that is performed in parallel requires its own storage, so the space overhead of this scheme can be substantial. Therefore, it is used only for relatively small graphs. We will refer to approaches using this strategy

as *outer-level* schemes. Alternatively, we can expose parallelism by focusing on processing a single source node and performing each of the computation steps in parallel (loops in lines $4, 7, 10$ in Figure 5.1). This fine-grained, *inner level* approach is more space-efficient since we only need to maintain a single graph instance, but poses a more challenging goal for parallelization due to non-trivial data dependencies. Finally one can combine the two techniques by processing several source nodes in parallel and performing the per-node computations in parallel.

5.1.3 Previous Work

Examples of the outer-level approach are [154, 71, 23]. Parallel performance is excellent, as expected, but the size of the input graph is very small or a big distributed cluster is used [23]. Bader et al. in [8] were the first to present an approach that targets both outer-level and inner-level parallelism. This work focuses on unweighted graphs, where the shortest path exploration can be performed by a breadth-first-search (BFS) exploration. Both of the main phases of the algorithm are performed in a level-parallel manner. Within level i all nodes are processed in parallel but only edges between nodes in levels i and $i+1$ are allowed to be processed. Similarly, in the backward DAG traversal, only nodes between levels i and $i-1$ are processed in parallel. The strong ordering between levels is achieved by using barriers. Subsequent work by Madduri et al. [93], improves the algorithm to use successors instead of predecessors in the computation of the DAG D, which produces a more

efficient, locality-friendly algorithm. [32] targets outer-level parallelism and also performs prefetching and appropriate re-layout of the graph nodes to improve locality. [148] presents a variation of [8] where the graph is logically partitioned among processors, locking is coarsened to a lock per partition, and the predecessor lists are distributed across partitions; [147] extends this work with architecture specific optimizations for the IBM Cyclops64 processor. Similarly, in [137] a GPU level-synchronous parallelization is presented, where graph edges are partitioned among the threads.

Edmonds et al. [40] present an approach that targets fine-grained parallelism and focuses on a distributed memory environment. This work deals with both weighted and unweighted graphs. In the case of weighted graphs, the level-parallel BFS approach is not applicable. Their solution breaks up the DAG construction phase into a number of sub-phases, separated again by barriers. Initially a label-correcting single-source shortest-path (SSSP) algorithm is employed [103, 114] to compute the shortest path distances and predecessor lists. Then, using the predecessor lists the node successors are computed. Finally, a third sub-phase computes σ_{sv} in a level-parallel BFS style, using the node successors. The backward traversal of the DAG is performed without using barriers by using the predecessor lists. [153] presents a serial adaption of Brandes that deals with weighted graphs by adding virtual nodes to turn them into unweighted graphs, where a BFS exploration can be performed. These algorithms compute the exact value of BC. To reduce the computational cost, a number of approximation algorithms have been proposed [21, 7, 45]. For

example, instead of computing the contribution of all source nodes $s \neq v$ to $BC(v)$ in Eq. 5.1, we can compute the contributions of a subset of source nodes.

5.1.4 Goals and Contributions

Our work makes three contributions.

- We show that the problem of computing BC can be formulated abstractly in terms of the *operator formulation* of algorithms [115].
- We show that existing parallel BC algorithms can be viewed as implementations of different Elixir schedules for applying the operators to the graph, permitting all these algorithms to be formulated in a single framework.
- The full set of our operators can correctly compute BC *under any arbitrary schedule* and can therefore be the basis for a new class of algorithms that can potentially mine more parallelism. This is especially true for the case of weighted graphs, where a purely level-synchronous approach cannot be used. Using these operators, we derive a new parallel algorithm by carefully controlling and optimizing the scheduling of operators. It is space-efficient because it targets fine-grained parallelism. It is able to expose a lot of parallelism because it breaks away from the level parallel mode of execution. It deals with both weighted and unweighted graphs, and, as we show experimentally, has good scalability. Our current im-

plementation targets multicore systems; it is straightforward to adopt it to a distributed setting.

5.2 A Framework for Expressing BC Algorithms

In this section, we formulate the computation of betweenness centrality in terms of the *operator formulation* of algorithms [115]. Operators act on graph nodes and edges and update their attributes.We describe a generic algorithm that computes BC by repeatedly applying operators in an unspecified order to the graph until a fixpoint is reached (that is, when no new operator applications can happen). We also show that existing algorithms for BC can be viewed as particular schedules for applying these operators.

To introduce the notion of operators, we consider the simpler problem of computing the breadth-first-search level of nodes in a directed graph. Each node u has a field $l(u)$ (for level), initialized to 0 for *Root* , and to ∞ for all other nodes. When the algorithm terminates, the level of a node will be equal to the length of the shortest path from *Root* to that node. The algorithm discovers paths from *Root* incrementally, so during the execution of the algorithm, the level of a node v is equal to the length of the shortest path to v that has been discovered so far.

Figure 5.2 shows the operator formulation for BFS. This operator is applied to a single edge of the graph and to the two nodes that are its end-points. An operator has a left hand side that specifies the precondition (predicate) under which it can be applied; an edge that satisfies this precondition is called

an *active edge*. An active edge can be updated as shown in the right-hand side of the operator.

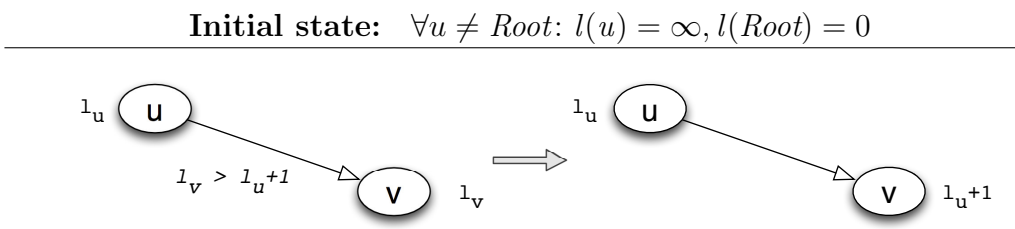


Figure 5.2: BFS expressed using a single operator for computing shortest paths.

The operator of Figure 5.2 can be described in words as follows: an edge (u, v) is active if $l(v) > l(u) + 1$; such an edge can be updated by setting $l(v)$ to $l(u)+1$. If there are several active edges in a graph, an implementation is allowed to update them concurrently, provided that active edge attributes are updated atomically. It can be shown that as long as the scheduling of active edges is fair (that is, the selection of an active edge is not postponed indefinitely), (i) the computation will terminate within some finite number of steps, and (ii) upon termination, for each node u, $l(u)$ will be its BFS level.

5.2.1 Operators for BC

We now show how to express BC using a small set of operators. To keep the presentation simple, we focus first on unweighted graphs, and then extend our approach to the case of weighted graphs. Our solution operates in two phases. In the first phase, it builds the BFS DAG. As in the BFS computation described above, the first phase discovers and records paths from

Root to other nodes incrementally. The second phase performs a bottom-up walk of the DAG to compute betweenness centrality.

For each node u we maintain a number of attributes:

- the shortest path distance ($l(u)$) of u,
- the number of shortest paths ($\sigma(u)$) of length $l(u)$ from *Root* to u,
- a list of nodes ($preds(u)$), each of which is the predecessor of u on a shortest path from *Root* to u, and
- a list of nodes ($succs(u)$), each of which is a successor of u on a shortest path from *Root*. Our implementation actually maintains only the number of the successors of a node and not the full list; for expository purposes we describe the operators using the successor list attribute.

Additionally, we associate with each edge (u, v) a level ($l(uv)$) and a path-count ($\sigma(uv)$) attribute; these are used for book-keeping during the algorithm execution, as explained below.

5.2.1.1 Operators for the DAG Construction Phase

The goal of the first phase is to construct the shortest-path DAG D and also compute the path-count $\sigma(u)$ for each node u. There are four operators, shown in Figure 5.3. Below, we discuss each in detail.

Shortest Path (SP): This is the same as the BFS operator except that it also resets $\sigma(v)$, $preds(v)$ and $succs(v)$ to their default values. This is the only operator that modifies levels of nodes. It is enabled when the

following *guard* predicate $g_{SP}(uv)$ is true:

$$g_{SP}(uv) := l(v) > l(u) + 1$$

First Update (*FU*): This operator is applied to an edge connecting nodes at successive levels. It updates $\sigma(v)$ with the current value of $\sigma(u)$ and it also updates the predecessor and successor lists of v and u. The operator is enabled when g_{FU} holds:

$$g_{FU}(uv) := l(v) = l(u) + 1 \wedge l(uv) \neq l(u)$$

The second constraint ensures that the operator is applied only once, since after the operator application, $l(uv) = l(u)$ as long as $l(u)$ is stable. There may be several incoming edges to node v at level $l + 1$ from nodes at level l, and this operator will be applied once for each such edge. These applications will be preceded by an application of the SP operator that brings node v to level $l + 1$.

Update Sigma (*US*): This operator is applied on an edge connecting nodes at successive levels, and it propagates changes to the path-count ($\sigma(u)$) of u to v. The previous update of $\sigma(v)$ from $\sigma(u)$ is stored in $\sigma(uv)$, which is used to compute the correct incremental update. The operator is enabled when g_{US} holds:

$$g_{US}(uv) := l(u) = l(uv) \wedge l(v) = l(u) + 1 \wedge \sigma(uv) \neq \sigma(u)$$

Correct Node (*CN*): This operator corrects the successor list of u in case its neighbor v moves to a lower level after having received some updates from u. Note that it is unnecessary to remove u from $preds(v)$ since $preds(v)$ would have been set to $\emptyset$ when v moved to a lower level as a result of applying SP. The operator is enabled when g_{CN} holds:

$$g_{CN}(uv) := l(u) \geq l(v) \wedge l(uv) = l(u) \wedge l(u) \neq \infty$$

Example 2 (Sample execution of the first phase)**.** *In Figure 5.4 we show a sample execution of the operators for the DAG construction phase of the algorithm. The Root node is s. Then, nodes a, b, c are at distance 1 and node d is at distance 2. Initially, all nodes other than s have distance ∞. First, we explore all nodes across the path (s, a, c, d). This results in a sequence of SP and FU applications that set $l(a) = 1, l(c) = 2, l(d) = 3$, update all path-counts to 1, and set successors and predecessors accordingly. Subsequently, we explore the path (s, b, c, d) and perform a similar sequence of operator applications. Note that, when we process (c, d), we apply US(cd) to correct c's contribution to d's path-count. When we explore path (s, c, d), the levels of c and d are lowered. Finally, we must also update the information of a and b to correctly reflect that c is no longer their successor. This is done by applying CN(ac), CN(bc). At this point, no more operators can be applied and thus we have reached the fixpoint.*

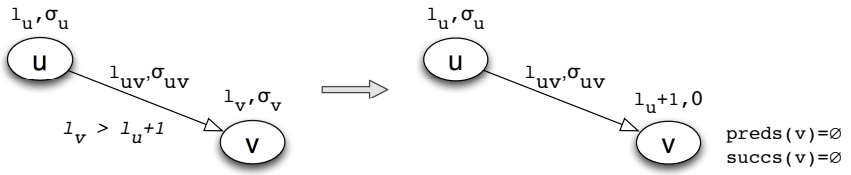
Initial state:

$\forall u \in \textbf{nodes} \setminus \mathit{Root}\colon [\sigma, l, \mathit{preds}, \mathit{succs}](u) = (0, \infty, \emptyset, 0)$

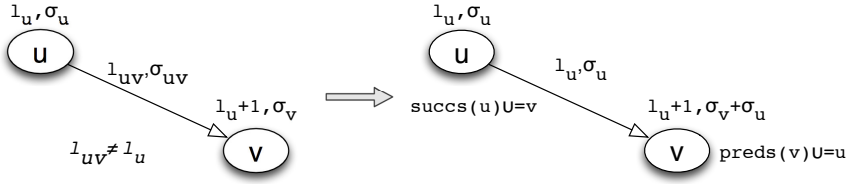
$[\sigma, l, \mathit{preds}, \mathit{succs}](\mathit{Root}) = (1, 0, \emptyset, 0)$

$\forall (u, v) \in \textbf{edges}\colon [l, \sigma](u, v) = (\infty, 0)$

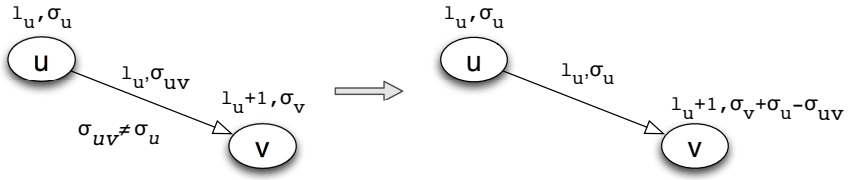
(a) Shortest Path



(b) First Update



(c) Update Sigma



(d) Correct Node

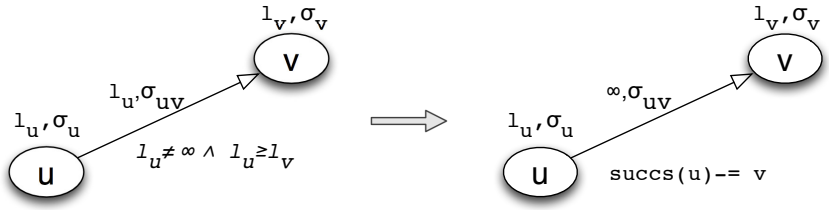


Figure 5.3: Operators for shortest-path DAG construction phase for unweighted graphs.

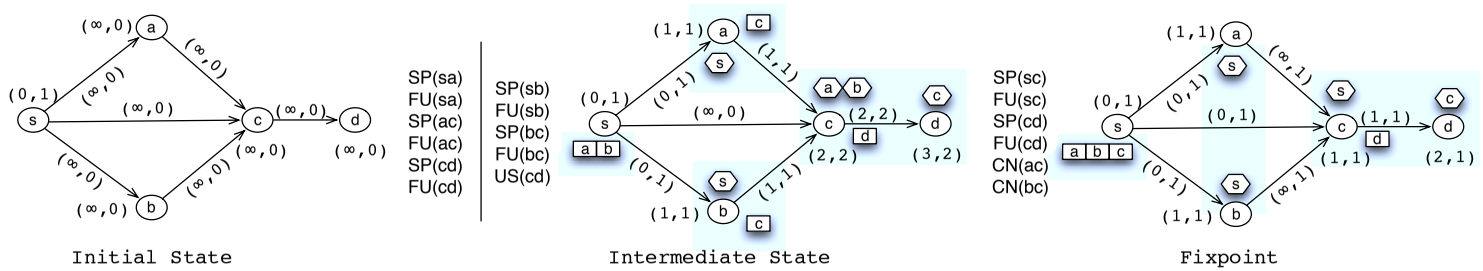


Figure 5.4: Sample sequential execution of the algorithm. The graph state is shown before, during and after the algorithm execution. Each node u is decorated with $(l(u), \sigma(u))$. Square boxes represent $succs(u)$ and hexagon boxes represent $preds(u)$. In the first fragment, operators in the left column execute before the ones in the right column.

5.2.1.2 Operators for Backward DAG Traversal

The goal of the second phase is to update the dependency $\delta(u)$ and the contribution to the centrality $BC(u)$ for each node u, based on equations 5.2 and 5.1, respectively. This is achieved by applying the single operator in Figure 5.5a until we reach a fixpoint. The operator is applied on a single edge (u, v) of the graph, such that $u \in preds(v)$. Hence, (u, v) is also an edge of the shortest-path DAG D. The operator guard is:

$$g := succs(v) = \emptyset \wedge u \in preds(v)$$

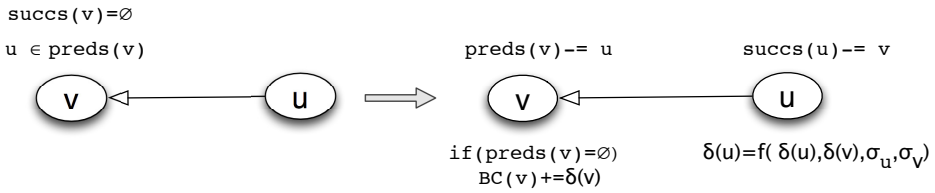
When the operator is applied, the value of $\delta(u)$ is updated based on the value of $\delta(v)$ as specified by Eq.5.2, that is:

$$\delta(u) \stackrel{+}{=} \frac{\sigma_{su}}{\sigma_{sv}}(1 + \delta(v))$$

Additionally, v is removed from the successors of u, and u from the predecessors of v. Finally, $BC(v)$ is updated conditionally, based on $\delta(v)$. This happens during the update of the last predecessor u of v. The operator applies when $succs(v) = \emptyset$, that is, when v has no successors, or has received updates from all its successors. This way, the backward traversal of the DAG D is performed in a data driven manner, breaking away from a level-parallel implementation. In Figure 5.5b we present a composite operator that is produced by merging together a number instances of the above backward traversal operator. This is an instance of an optimization we call *operator merging*, discussed in detail in Section 5.3.3.

Initial state: $\forall u \in \mathbf{nodes}(:,\delta)_s(u) = 0$

(a) Initial formulation



Initial state: $\forall u \in \mathbf{nodes}(:,\delta)_s(u) = 0$

(b) Formulation after operator merging

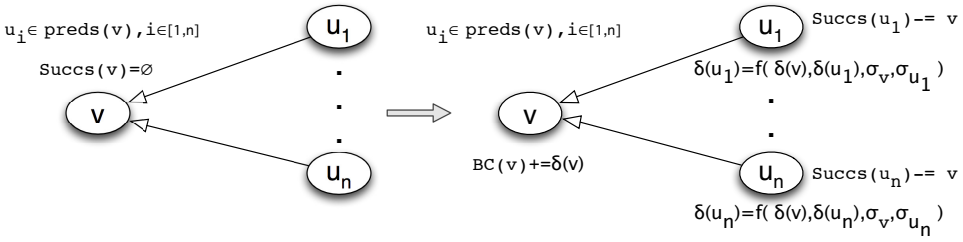


Figure 5.5: Operator for backward DAG traversal phase

5.2.2 Operators for Weighted Graphs

In Figure 5.6 we show the operators for the weighted case. We assume that each edge (u, v) has a strictly positive weight $w(uv)$. SP discovers a new shortest path from *Root* to v through u when $l(u) + w(uv) < l(v)$ and sets $l(v) = l(u) + w(uv)$. Similarly, g_{FU}, g_{US} are changed to properly identify successive nodes on shortest paths from *Root*. Finally, g_{CN} is modified to check for $l(u) + w(uv) > l(v)$ in order to capture the case when a shorter path v has been discovered after v has received an update from u. It is easy to see that the operators in Figure 5.3 are derived from the ones in Figure 5.6 by setting $w_e = 1$ for each edge e in the graph. The second phase is the same as in the unweighted case.

5.2.3 Characterizing BC Algorithms

Algorithms for BC in the literature can be viewed as implementations of particular schedules for the operators of Figures 5.3, 5.5 and 5.6. Some of the operators are not required for certain schedules.

Unweighted graphs: We first describe algorithms for unweighted graphs. The algorithms by Bader [8] and Madduri [93] build the BFS DAG level by level; each level is built in parallel, with barrier synchronization between levels. The construction of each level essentially involves executions of the SP and FU operators of Figure 5.3; US and CN are unnecessary in such level-by-level algorithms since nodes reach their final levels in a single step rather

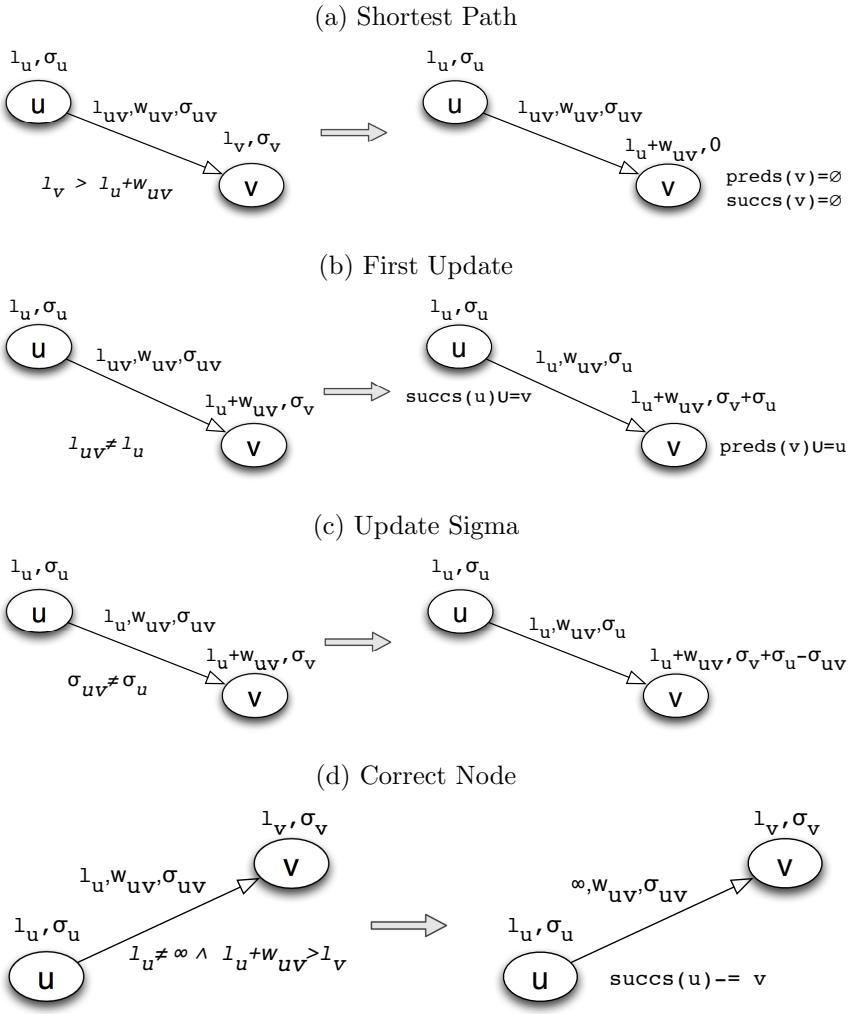


Figure 5.6: Operators for shortest-path DAG construction phase for weighted graphs. Initialization same as in Figure 5.3

than being lowered to that level by relaxation, and the path-counts of nodes at the current level are finalized before moving to the next level. The second phase of [8], which traverses the DAG backward to compute betweenness centrality, can be described by applications of the operator in Figure 5.5b; once again, these applications are not performed in a data-driven manner but in a level-parallel manner using an auxiliary stack data structure populated in the first phase. The second phase of [93] operates in a similar manner except that each node "pulls" information from all its successors instead of "pushing" information to its predecessors, and can be described by a similar operator. Approaches that exploit just coarse grained [32, 71] or both coarse and fine-grained parallelism [8] are also straightforward to describe with the same operators. An open question is what is the best policy for mapping operators from multiple iterations to the available computational resources. Approaches such as [148, 137] perform the level synchornous approach using the operators of Figure 5.3. Logically partitioning the graph among threads and having each thread be responsible for its partition, is a different way of scheduling the operators and of achieving their atomic execution. Similarly, [23] can be seen as a SIMDization of the operators of Figure 5.3. Data structure optimizations in the implementation of the graph abstract data type, such as the re-layout of graph nodes [32] or the distributed storage of the predecessor list [148], are orthogonal to our description of the algorithm and their effect on improving performance is complementary.

Weighted graphs: The algorithm for weighted graphs by Edmonds et al. [40] breaks down the forward pass in a number of sub-phases. It performs a first phase where applications of SP and FU are applied asynchronously in order to update the distance of nodes and the predecessor lists. Then in a subsequent phase, the path-counts are updated by using applications of the US operator in a level-synchronous manner. The US operator in that phase is simplified in that no corrections in the estimation of σ are necessary, due to the constraint on the schedule. Essentially, [40] acts as a label-correcting algorithm for only a subset of the node attributes $(l(u), preds(u))$ and as a label-setting algorithm for the rest $(\sigma(u), succs(u))$. Therefore, it is restricted to level-synchronous schedules for the computation of the latter. Our operators, on the other hand, provide a label-correcting capability for all node attributes and are able to merge the above two phases in a single, fully asynchronous phase, potentially exposing more parallelism. Finally, the backward DAG traversal is performed asynchronously using the operator in Figure 5.5b.

5.2.4 Correctness of BC Operators

In this section, we state the main correctness results of our operator formulation of BC.

Forward Pass We start with the operators for the forward pass, presented in Figure 5.3. We prove that the first phase terminates, and that upon termination all node attributes have correct values. We consider the most general

setting where operators are allowed to execute in any order. The computation is modeled by a *history*, which is a sequence of operator applications, each of which is considered to be an instantaneous event that changes the state of the graph. We denote an operator application on an edge (u, v) by $op(uv), op \in \{SP, FU, US, CN\}$. To capture meaningful computations, we restrict attention to *well-formed histories*, which are histories where each time an operator is enabled on an edge, its execution cannot be postponed indefinitely. Correctness follows from the following two theorems.

Theorem 2. *(Termination) Any well-formed history H of events $op(uv), op \in \{SP, FU, US, CN\}$ of the operators in Figure 5.3 to a graph $G = (V, E)$ has finite length.*

Theorem 3. *At the fixpoint, the following facts hold for an arbitrary node v:*

(a) *$l(v)$ is equal to the length of the shortest path to v from Root.* ***(b)*** *u is the predecessor of v in a shortest path to v from Root $\iff$ $u \in preds(v)$ and $v \in succs(u)$.* ***(c)*** *$\sigma(v)$ is the number of shortest paths from Root to v.*

The full proofs are presented in Appendix B. Here we briefly state the main ideas. Th. 2 is proven by showing that each operator can appear only a finite number of times in an arbitrary history H. We prove that the SP operator appears only a finite number of times, and that successive SP applications partition H into "windows", within which we can only have a finite number of FU, US, CN applications. Th. 3(a): We consider the partitioning of nodes $P = \{P_0 \dots P_n\}$, where P_0 contains the *Root* and P_i contains

nodes at shortest path distance i, that is, nodes directly connected to nodes in P_{i-1} but not to nodes in $P_0, \ldots, P_{i-2}$. We show that at the fixpoint the partitioning induced by our algorithm is equivalent to P. Th. 3(b): ($\Rightarrow$) We examine an arbitrary history, and show that for an arbitrary edge (u, v) there always exists an *FU* application that updates appropriately $succs(v), preds(v)$ after u, v finally settle as successive nodes on a shortest path from *Root*. ($\Leftarrow$) Considering $u \in preds(v)$ and $v \in succs(u)$ at the fixpoint we show by induction on $l(u)$ that there exists a shortest path to v through u. Th. 3(c): By induction on the shortest path length, using the fact that at the fixpoint $\sigma(v) = \sum_{u \in preds(v)} \sigma(u)$.

Backward Pass We now discuss the correctness of the operator in Figure 5.5a for the backward pass. Proving termination is straightforward. Initially there is a fixed number of predecessor edges between the nodes comprising the shortest-path DAG. Each operator application depends on finding one such predecessor edge (u, v) and removes it from the graph. Therefore, the number of predecessor edges decreases monotonically and eventually becomes zero. At that point no more applications are enabled and the algorithm terminates. Correctness at the fixpoint follows from the theorem below:

Theorem 4. *Let $preds_{init}(v), BC_{init}(v)$ denote the values of the respective node attributes at the beginning of the backward pass. At the fixpoint, the following facts hold for an arbitrary node v:* ***(a)*** $\delta(v) = \sum_{w:\, v \in preds_{init}(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$ ***(b)*** *if $v \neq Root$ then $BC(v) = BC_{init}(v) + \delta(v)$, else $BC(v) = BC_{init}(v)$.*

5.3 Derivation of New Asynchronous Algorithms

The operators in Figures 5.3, 5.6, 5.5a can be applied in any order, and as long as no enabled operator application is postponed indefinitely, the implementation will terminate and produce the right BC values. However, the number of operator applications, which is one measure of work-efficiency, may be very different for different orders. In addition, some orders may exploit locality better than others. Getting a scalable solution greatly depends on performing the right operator scheduling. Exploiting scheduling to improve parallelism is a well-studied area [17, 18, 30, 15]. Here, we derive asynchronous algorithms by choosing particular scheduling policies for operator execution.

A simple approach is to repeatedly scan all the edges of the graph in some order, applying all applicable operators to an edge when it is visited. This is inefficient since most edges will not have any operators that can be applied to them. Instead, we use a worklist-based approach. The unit of work in our setting is processing a single edge, which may trigger applications at neighboring edges. Therefore, we maintain a multiset of edges, implemented as a dynamic worklist *Wl*. At each step, we pick an edge e from *Wl*, apply an operator to it, and add to *Wl* new edges that may need processing. In a concurrent setting, *Wl* must support concurrent `add` and `poll` operations. Atomic operator execution is achieved by acquiring locks on the edge end-points during the operator execution.

In Section 5.3.1 we discuss the most performance-critical aspect of our design: the policy for processing *Wl* elements. In Section 5.3.2 we discuss how

to incrementally update the *Wl*. In Section 5.3.3, Section 5.3.4 we describe key scheduling optimizations for improving the basic algorithm. Finally, in Section 5.3.5 we present two algorithm variants that we can derive by exploiting all the scheduling insights.

5.3.1 Choosing a Dynamic Scheduling Policy

During the algorithm execution, *Wl* will usually contain many edges where operators may be applied. An important design decision is picking a work-efficient order to process edges, since this will affect the convergence rate of the algorithm [103, 58, 108].

During the DAG construction phase, the shortest-path exploration (SP operator) is the backbone of our algorithm. The number of all other operator applications is ultimately a function of the times we mispredict the length of the shortest path to a node. Hence, a good ordering policy in our case is one that efficiently identifies shortest-paths, but which is also flexible enough to allow threads to optimistically try to expose parallelism. In the unweighted case, for example, a level-parallel approach allows only nodes within successive levels to be processed concurrently and can stifle parallelism if there are not enough nodes in a level. However, an approach that allows exploration of arbitrary paths on the graph may end up performing too much wasted work since it may mispredict the distance of nodes multiple times.

Our approach is loosely based on the Δ-stepping approach of Meyer *et al.* [103] and effectively tackles such problems. We partition the edges (u, v)

in *Wl* into equivalence classes based on an approximation $l^*(v)$ of $l(v)$ defined as $l^*(v) := \frac{l(u)+w(uv)}{\Delta}$. The term $l(u) + w(uv)$ is the length of the path from the *Root* to v through u. Δ is a user specified parameter that defines a range of distance values that fall in the same equivalence class. Each equivalence class is associated with a bucket B_i. Each time an edge (u, v) is inserted in *Wl*, we compute $l^*(v)$ and place it in the appropriate B_i. Intuitively, we want to explore shorter paths before longer ones in order to minimize the number of mispredictions on the level of a node. Hence, our policy places these edges in lower-numbered buckets. Threads query the buckets for work in increasing order. The deviation from the classic Δ-stepping is that *there is no barrier between the processing of successive buckets*. Each thread queries buckets in increasing order repeatedly until no more work is left, but different threads can be simultaneously operating on different buckets and, consequently, can extract more parallelism in case there is limited amount of work in a particular bucket. The role of Δ is to control the interplay between exposing parallelism, decreasing the probability of mispredicting levels, and controlling the overhead of iterating over and querying buckets in a concurrent setting. This strategy is enabled only because our operators are general enough to restore the correct values of attributes, in case of mispeculation. Note that the equivalence class of an edge can change after the point it is inserted in B_i. This however can only affect the performance and not the correctness of our algorithm. To mitigate the overhead even further, each thread gets a chunk of edges out of B_i each time it queries *Wl*.

The backward DAG traversal phase operates in a data-flow driven manner. When the δ value of a node is fully updated, we propagate changes backward by scheduling updates to its DAG predecessors. This scheme avoids a level-parallel traversal of the shortest-path DAG and can expose more concurrency in the case where the DAG contains independent components. Edges to be processed are inserted into a worklist. A FIFO or LIFO based worklist policy gave us the best performance.

The best value of Δ and of the other scheduling parameters varies across inputs. Section 5.4 provides details about the chosen values.

5.3.2 Incremental Solutions via the Operator Deltas

We first discuss the policy for the DAG construction phase. Finding the minimal set of edges that need to be processed, and therefore need to be added to *Wl*, due to an operator application on edge (u, v) is challenging in a concurrent setting. This is because the immediate neighborhoods of u and v are, in general, unbounded. Therefore, we over-approximate the minimal set of edges by using the following policy: Whenever an operator changes the attributes of a node v, we add to *Wl* all edges adjacent to v that *may* need to be updated due to the change to v. We determine these edges for each operator as follows.

SP: An $SP(uv)$ moves v to a new level and identifies a new shortest path P to v. All outgoing edges of v are inserted to *Wl* so that the exploration along P continues. Additionally, incoming edges of v must be examined

to correctly update neighboring nodes $w \neq u$ that lie on paths P' (longer than P). Such nodes may have recorded v as their successor. Hence, we add all adjacent edges to v to Wl.

***FU*, *US*:** Both operators update $\sigma(v)$, so all outgoing edges of v are added to Wl to further propagate this update.

***CN*:** This operator simply corrects the successor list of the source node u of the edge it is applied on and does not enable any other operators. Hence no updates to Wl are required.

To initialize Wl, we insert all outgoing edges of the *Root* to initiate the shortest-path explorations.

For the backward DAG traversal we use a data-flow driven policy that considers the minimal set of edges. Whenever $succs(v) = \emptyset$ an operator is enabled on each edge (u, v) between v and $u \in preds(v)$. In the baseline scheme, we add each such (u, v) to Wl. When operator merging is applied (see Section 5.3.3), Wl can simply contain nodes; in this case we insert v to Wl. We initialize Wl with all (u, v) (or v), such that v is at the fringe of the DAG generated in the first phase. Such nodes are easy to identify after the first phase, since they have no successors. We perform a scan over all nodes and populate Wl appropriately, when this property is satisfied.

5.3.3 Static Scheduling Optimization I: Operator Merging

In certain cases it is beneficial for performance reasons to merge multiple operators together and create a new composite operator. Operator merging is essentially a transformation that binds scheduling decisions at compile-time and provides benefits similar to classical loop optimizations. Co-scheduling the application of multiple operators can improve locality. Also, in a similar spirit to loop fusion, combining multiple operators together improves locality and reduces loop overheads. We identify two cases where merging is applicable and describe additional performance benefits it provides.

In the first phase, every $SP(uv)$ will lead to a subsequent $FU(uv)$. Merging the two firstly improves locality, since the data of v is already local to the thread due to $SP(uv)$. Second, it reduces the locking overhead, by eliminating locking for $FU(uv)$. Finally, it reduces the pressure on Wl, since we need insert the outgoing edges of v to Wl only once in the combined operator.

In the second phase, we can merge all applications of the operator between v and its predecessors u_i. The composite operator is shown in Figure 5.5b. Merging firstly permits all updates starting from v to be performed in a single operator application. Thus, we can avoid the removal of u from $preds(v)$, because now we do not need to keep track of whether processing v is over. In order to update $BC(v)$ correctly the composite operator needs to be applied once per node v and its predecessors. This can be achieved by simply inserting a single instance of v into Wl, upon receiving an update from the

last successor. A special check to avoid updating *Root* is also necessary. Note that we can perform fine-grained locking on each u_i and do not need to access atomically an unbounded number of nodes.

5.3.4 Optimization II: Heuristics for Tighter Operator Deltas

In Section 5.3.2 we discussed a policy that over-approximates the set of edges that need to be processed after an operator application, and therefore need to be added to *Wl*. We identify various ways of improving that policy. Our optimizations reduce the number of redundant checks on parts of the graph for enabled operators and the number of `add` calls to *Wl*, thus leading to fewer accesses to shared resources. In practice they are important in speeding up the first phase of the algorithm.

First, when an $SP(uv)$ is executed, if $preds(v) = \emptyset$, then no incoming neighbor u of v needs to be processed, since $v \in succs(u)$ iff $u \in preds(v)$. Hence, we can avoid inserting incoming edges of v to *Wl*.

Second, when a $US(uv)$ is executed, if $succs(v) = \emptyset$, then we can avoid adding outgoing edges (v, w) of v to *Wl*. The reason is that in order for $US(vw)$ to be executed, w must already be a successor of v (an $FU(uv)$ must have been executed). Since v has no recorded successors, all (v, w) are already in *Wl* waiting to be processed, and we can avoid re-inserting them. This way, we expose opportunities to batch up all path-count updates and avoid redundant *Wl* population.

Third, when using the combined $SP(uv), FU(uv)$ operator, we can

avoid adding outgoing edges (v, w) after an $FU(uv)$ if $succs(v) = \emptyset$, since they are already there due to the application of the combined operator. Note that all these optimizations are inexpensive since they require simple checks on operator-local state, and do not require acquiring extra locks.

5.3.5 Putting it All Together: Derivation of Two Asynchronous Variants

We now describe how to combine all the ideas presented in the previous sections in order to produce two asynchronous algorithms.

In the first algorithm, *async1*, during the forward phase (line 4 in Figure 5.1), each thread extracts an edge out of the worklist and tries to find an operator to apply to it by checking the operator guards in some arbitrary order, while also merging applications of SP with FU. The order we chose is $[CN, SP \circ FU, FU, US]$. Note that the guards are mutually exclusive, therefore any order of checking the guards is guaranteed not to postpone an operator application indefinitely. For the backward phase (line 7 in Figure 5.1), each thread extracts a node out of the worklist and tries to perform the composite operator in Figure 5.5b to all predecessor nodes. In Figure 5.7 we show in pseudocode for the forward pass of *async1*.

In the second algorithm, *async2*, we consider a node and all its immediate neighbors, and statically co-schedule operator applications on the edges connecting them. Additionally, we perform potential `CN` applications in place whenever we discover a new shorter path to a node. The motivation behind

```
1  Worklist Wl = {(srcNode, w): (srcNode, w) ∈ G.edges}

3  foreach (u, v) ∈ Wl {
4    lock(u,v)
5    if (g_CN(u, v)) {
6      apply CN ; unlock(u,v)
7    } else if (g_SP∘FU(u, v)) {
8      apply SP ∘ FU; unlock(u,v)
9      Wl = Wl ∪ {(v, w): w ∈ outNbrs(v)}
10     if (vHasPreds)
11       Wl = Wl ∪ {(w, v): w ∈ inNbrs(v)}
12   } else if (g_FU(u, v)) {
13     apply FU; unlock(u,v)
14     if (vHasSuccs)
15       Wl = Wl ∪ {(v, w): w ∈ outNbrs(v)}
16   } else if (g_US(u, v)) {
17     apply US; unlock(u,v)
18     if (vHasSuccs)
19       Wl = Wl ∪ {(v, w): w ∈ outNbrs(v)}
20   } else { unlock(u,v) }
21 }
```

Figure 5.7: Pseudocode for forward pass of *async1*.

these design choices is to exploit spatial and temporal locality by having a a single thread work on an entire neighborhood. A potential issue with this design though is that it may be harder to load balance work, in case the node degrees are not evenly distributed. This can be more problematic in an environment with high degree of parallelism. The pseudocode for *async2* is presented in Figure 5.8. Note that in this algorithm we insert nodes instead of edges into the worklist. The second phase is the same as in *async1*.

To guarantee atomic execution of operators we acquire fine-grained spinlocks on the end-points of each edge that an operator works on. The graph nodes are totally ordered based on their runtime (allocation) addresses. We acquire locks respecting this order to avoid deadlock between threads working on the same edge. To amortize locking overhead we acquire the locks in the beginning of the loop iteration and release them after the first successful operator application, or at the end if no operator is applicable. In case of a successful operator application we release a lock on a node immediately after the last access to a node attribute. Inserting edges/nodes in *Wl* after an operator application is done without holding any locks. This is because this action does not access any data attributes, and also because the graph structure is not mutated. This way we significantly reduce the length of atomic sections and allow more fine-grained thread interleavings.

```
1  Worklist Wl = {srcNode}

3  foreach u ∈ Wl {
4    forall v ∈ outNbrs(u) {
5      lock(u,v)
6      if (g_{SP∘FU}(u, v)) {
7        apply SP ∘ FU; unlock(u,v)
8        Wl = Wl ∪ {v}
9        if (vHasPreds) {
10         // Inline CN applications
11         forall w ∈ inNbrs(v) {
12           lock(v,w)
13           if (g_CN(w, v)) { apply CN }
14           unlock(v,w)
15         }
16       }
17     } else if (g_FU(u, v)) {
18      apply FU; unlock(u,v)
19      if (vHasSuccs) Wl = Wl ∪ {v}
20     } else if (g_US(u, v)) {
21      apply US; unlock(u,v)
22      if (vHasSuccs) Wl = Wl ∪ {v}
23     } else { unlock(u,v) }
24   }
25 }
```

Figure 5.8: Pseudocode for forward pass of *async2*.

5.4 Experimental Evaluation

We implemented two versions of the algorithm, based on the operators in Figure 5.3 and Figure 5.6 and compared their performance against a number of publicly available versions of BC algorithms for weighted and unweighted graphs. We ran our experiments on two architectures. First, an Intel Xeon machine (**Nehalem**) running Scientific Linux 6.3 with four 6-core 2.00 GHz Intel Xeon E7540 processors that share 128 GB of memory. Second, a Sun T5440 machine (**Niagara**) running SunOS 5.10. It contains two 8-core 1.4 GHz Sun UltraSPARC T2 Plus (Niagara 2) processors, and provides 128 concurrent hardware threads, sharing 32 GB of memory. On the Nehalem, the compiler used was GCC 4.7.1. On the Niagara, the compiler used was GCC 4.5.1. We report the average time (t) of 5 runs and the standard deviation (sd). We consider the following classes of input graphs:

- Real-world road network graphs of the USA from the DIMACS shortest paths challenge [1]. We use the full USA network (*USA-net*) with 24M nodes and 58M edges and the central USA network (*USA-ctr*) with 14M nodes and 34M edges.

- A network of scientific co-authorships [112] (*coauth*) with 391K nodes and 873K edges, where edge weights are converted to integers (by multiplying all weights by 1000).

- Scale-free graphs generated using the Recursive MATrix (R-MAT) scale-free graph generation algorithm [27]. The size of the graphs is controlled

by a $SCALE$ parameter; a graph contains $N = 2^{SCALE}$ nodes, $M = 8 \times N$ edges, with each edge having strictly positive integer weight with maximum value $C = 2^{SCALE}$. The RMAT graphs we used were generated using the tools provided by the SSCA v2.2 benchmark [6]. The parameters used for the graph construction were the default ones, as specified by the generator ($a = 0.55, b = 0.1, c = 0.1, d = 0.25$). For our experiments we removed multi-edges from the graphs. We denote a graph of $SCALE = X$ as $rmatX$.

- Random graphs containing $N = 2^k$ nodes and $M = 4 \times N$ edges. There are $N - 1$ edges connecting nodes in a circle to guarantee the existence of a connected component and all the other edges are created randomly, following a uniform distribution. A graph with $k = X$ is denoted as $randX$.

For large-scale graphs, it is computationally infeasible to compute BC by doing shortest path computations from every node of the graph. Therefore, like previous studies [7, 93, 40, 32], we perform shortest path computations for only a subset of nodes.

5.4.1 Experiments on Weighted Graphs

We consider two algorithms for weighted graphs: (i) our algorithm *async1*, which is based on the operators of Figure 5.6, with all scheduling optimizations enabled, and (ii) a serial reference implementation of Brandes'

algorithm (**boost-s**) for weighted graphs, available in the Boost Graph Library [140] V. 1.47.0. The only parallel algorithm for weighted graphs with a publicly available implementation that we are aware of is [40]. However, that solution targets a distributed-memory environment while our implementation targets shared-memory multicores, so a direct comparison of performance is not meaningful.

5.4.1.1 Implementation Details

We parallelized *async1* in C++ using the Galois system [2]. Each of the two major phases of the algorithm in Figure 5.1 is implemented as a parallel `foreach` loop over a worklist of edges and nodes, respectively. Below we discuss in more detail several aspects of our implementation.

Graph data-structure Our graph implementation is based on the compressed sparse row (CSR) format, with node and edge data stored in two different arrays. Our graphs were not initialized in a NUMA aware manner. In our experience this does not have an observable impact on our experimental machines, since they do not have deep NUMA hierarchies.

Worklist policies We use worklist implementations provided by the Galois system [2, 108]. In Table 5.1, we present the scheduling parameters that gave us the best performance. These values were obtained by performing a small manual search on the parameter space. A more exhaustive search can

Nehalem	Forward		Backward
coauth	$\Delta = 512$	$CF128$	$CL256$
USA-net	$\Delta = 32768$	$CF64$	$CL256$
rmat25	$\Delta = 32768$	$CF256$	$CL256$
Niagara	Forward		Backward
coauth	$\Delta = 2048$	$CF128$	$CL256$
USA-net	$\Delta = 32768$	$CF128$	$CL256$
USA-ctr	$\Delta = 32768$	$CF128$	$CL256$

Table 5.1: Scheduling parameters for weighted graph experiments.CFx (CLx): Chunked FIFO (LIFO) with chunk size x.

potentially lead to improved performance; we plan to examine this in future work. Recall that in the forward pass, we use a delta-stepping-like policy that prioritizes the work-items into buckets using the parameter Δ. Within each bucket, our scheme follows a `FIFO` or `LIFO` policy with additional chunking of work. For the backward pass, we use either a chunked `FIFO` or a chunked `LIFO` policy. For example, for the *coauth* graph forward pass on the Nehalem we use $\Delta = 512$ and each bucket is implemented using a `FIFO` policy with chunk size of 128 edges. For the backward pass we use a `LIFO` with chunk size 256.

5.4.1.2 Analysis of Results

Table 5.2 presents results for various real-world and one synthetic graph. Our algorithm scales very well on all graphs across both architectures. On Nehalem, *async1* achieves (self-relative) scalability of 9.5× on both the USA road-network, and the *rmat25*. The high thread count on the Niagara, allows *async1* to mine the available parallelism through the fine-grained operator execution. For the two road-networks it achieves its best scalability, 32× and 37×,

Nehalem	coauth (500 steps)		USA-net (10 steps)		rmat25 (10 steps)	
Threads	*t*	*sd*	*t*	*sd*	*t*	*sd*
boost-serial	68	0	510	4	3020	53
1	32	3	285	1	1493	102
4	12	0	78	0	383	27
8	10	0	43	1	223	6
12	10	0	34	0	170	8
16	8	0	31	0	163	6
20	8	0	30	0	159	10
24	8	0	30	0	157	15

Niagara	coauth (100 steps)		USA-net (10 steps)		USA-ctr (10 steps)	
Threads	*t*	*sd*	*t*	*sd*	*t*	*sd*
boost-serial	83	0	1872	107	1086	1
1	110	0	1583	6	981	5
16	9	0	115	0	69	0
32	6	0	65	0	37	0
48	6	0	53	0	29	0
64	7	0	49	0	26	0
96	10	0	53	1	26	0
128	15	0	61	2	31	2

Table 5.2: Average execution time (sec.) and stdv. of *async1* for weighted graphs

at 64 threads. By exploiting cross-level parallelism *async1* is able to achieve scalability even on the really small co-author network (4× on Nehalem, 18× on Niagara). We report the *boost-s* runtime to provide a reference against a publicly available serial implementation.

5.4.2 Experiments on Unweighted Graphs

We evaluated a number of BC algorithms for unweighted graphs. Below, we describe each of them and give details about the parallelization strat-

egy and scheduling policy. We note that the graph format for all unweighted algorithms is based on the CSR format.

outer This is a parallelization only of the outer loop. The graph state is replicated P times, where P is the number of threads. Each thread performs an iteration of the outer loop which is mostly independent from other iterations. The updates on the betweenness value of each node form a reduction which is straightforward to handle. The serial algorithm executed in the inner loop by each thread uses the successor lists to represent the DAG, as discussed in [93]. The algorithm was implemented in the Galois system.

async1*, *async2 These are our algorithms based on the operators of Figure 5.3 with all scheduling optimizations enabled. *async1* was used on the Niagara experiments, while *async2* was used on the Nehalem experiments. The algorithms were implemented in the Galois system following the same design choices as the weighted version in Section 5.4.1.1. We now discuss the scheduling parameters that we used. On both architectures we select $\Delta = 1$. The intuition behind this is that in the case of unweighted graphs the diameter is low, which potentially increases the amount of work per level. Setting $\Delta = 1$ focuses the search for work in individual levels, while still allowing for cross-level speculation. We note that setting $\Delta = 1$ *does not make them level-synchronous algorithms*. As discussed in Section 5.3.1 these algorithms allow threads to simultaneously work on an arbitrary number of buckets. Setting $\Delta = 1$ simply reduces the speculation window for the threads but does

not restrict them to a single bucket. If bucket B_i becomes temporarily empty, threads move to buckets $T_j, j > i$, and can return later to T_i if new work exists there. This strategy is applicable only because our operators are general enough to restore the correct values of attributes in case of mispeculation. On the Nehalem, where *async2* is used, each bucket is processed using a `LIFO` policy. For `rmat25` we use a chunk size of 8 nodes and for `rand26` we use chunks of 32 nodes. On the Niagara, where *async1* is used, each bucket is processed using a `FIFO` policy with chunks of edges of size 512. The second phase uses a `LIFO` policy with chunk sizes of 16 (Nehalem) and 2048 (Niagara).

preds This algorithm is an inner-level, level-synchronous parallelization presented in [8]. The implementation is part of the SSCA v2.2 benchmark [6]. The implementation uses OpenMP directives to parallelize the loops and define their schedules, and OpenMP barriers for the synchronization of levels. The OpenMP scheduling policy for the forward phase is `dynamic` and for the backward phase is `static`.

succs/succs-serial This algorithm is an inner-level, level synchronous parallelization presented in [93]. The implementation is our adaption of the implementation provided in GraphCT v.0.5 [39]. GraphCT is a parallel toolkit for analyzing massive graphs on the massively multithreaded Cray XMT; the implementation is optimized for the Cray and uses compiler directives specific to that system. The algorithm, as presented in [93] admits a lock-free

implementation. Our adaption is a parallelization using OpenMP. We replace the original atomic intrinsics with the equivalent ones provided by GCC to implement it. We experimented with various scheduling policies and present results for the `guided` policy, which performed the best. This algorithm also serves as the serial baseline that we compare all algorithms against.

5.4.2.1 Analysis of Results

In Table 5.3 we present the running time of all four algorithms for 100 iterations of the outer loop. Due to large running times, we present results only for high thread counts on the `rmat25` graph, on the Nehalem. In Table 5.4, Table 5.5 we focus on the inner-loop parallelization strategies and present results for 10 iterations of the outer loop.

First, we discuss performance on the Nehalem (Table 5.3, Table 5.4). We can make a number of interesting observations. We note that for `rmat25` *outer* is the best performing. We expect this to be the case when the graph fits in the available memory. However, this comes at the cost of high memory usage; for 24 threads *outer* requires about 63% of the 128 GB of main memory and it will exceed the machine's memory capacity for larger graphs. For this reason, we do not present results for *outer* on `rand26` or on the Niagara, which has less memory and a larger number of threads. Second, *preds* starts by being more than 2× slower than *succs* (due to the locality benefits of *succs*), something that is in accordance with previous studies [93]. At higher thread counts, though, the performance gap between the two is reduced. Interestingly,

on our system *preds* is sensitive to thread pinning. Avoiding pinning makes *preds* more than 2× slower than *succs* even on high thread counts (e.g. on 24 threads *preds* takes 174 sec. for 10 steps on `rmat25` on Nehalem). Focusing on *async2*, on the Nehalem, we observe that it is slower than the other algorithms on the `rmat25` graph, while it is the fastest on the `rand26`. We believe this is due to the structure of the input graphs. The average longest BFS level for the outer loop iterations we performed provides an approximation of the graph diameter. It is 15.9 (stdv. 0.7) on `rmat25` and 20.1 (stdv. 0.57) on `rand26`. By increasing the "diameter" of the explored graph, hence decreasing the amount of work per level, an algorithm that tries to mine work from multiple levels becomes comparatively better.

On the Niagara, we experiment with a different variant of our algorithm, *async1*. In *async1* the unit of work is the processing of a single edge instead of a node and its immediate neighbors. Intuitively, this can lead to more fine-grained distribution of work, which can be more suitable for an environment with a high degree of parallelism. *async1* is able to scale to high thread counts on both inputs, as we see in Table 5.5. We do not report performance numbers on *preds*, due to execution problems on this platform. Regarding *succs*, its best running time is 325 seconds (stdv. 35) for `rmat25` and 1044 seconds (stdv. 138) for `rand26`, both on 16 threads. *succs* did not scale beyond 16 threads. We believe this behavior is partly due to the poor exploitation of parallel resources by the level-parallel approach (more threads means less work per thread on each level) and partly to scaling issues in the OpenMP runtime on

the Niagara.

5.4.3 Assessing the Effectiveness of Scheduling

To illustrate the effect of scheduling on the performance we consider one more experiment using *async2* on the *rmat25* unweighted graph on the Nehalem architecture. We execute two outer loop iterations and record the number of executed operators. We consider the following variants of *async2*: The first, `Ord`, is the version we already presented with all scheduling optimizations enabled. The second variant, `Ord-d`, is a de-optimized version of `Ord` where, the optimizations presented in Section 5.3.4 are disabled. Both `Ord` variants use the same worklist policy. Finally, `CF` is a variant of *async2* with a `FIFO` worklist policy (with chunk size of 16 nodes) and all other scheduling optimizations enabled. Using a `FIFO` schedule for the unweighted operators gives us the optimal order of processing the edges in the sequential setting. The `CF` worklist policy maintains this FIFO order for each thread in the parallel setting, while relaxing the constraint of maintaining a total FIFO ordering among threads, to achieve better scalability. In Table 5.6, we report the average number of operator applications and runtime of the forward pass per outer loop iteration.

Firstly, note that for `Ord` and `CF` the operator counts for one thread are identical. This is natural since, with $\Delta = 1$, on one thread `Ord` emulates the best schedule (`FIFO`). Additionally, we see that the number of *US* applications and *CN* evaluations is zero. This is normal, since in the best schedule no

corrections should be done to the path-count and we should never mispredict the level of a node. This is not the case for `Ord-d`, though, which disables some of the optimizations and performs unnecessary work. Also, we note that not all nodes and edges will be reachable from all possible roots, in a directed graph, therefore the number of operator applications will not, in general, match the node and edge count of the graph.

Focusing on runtime performance, we observe a correlation between the number of operator applications and the running time. `CF` deviates quickly from the optimal schedule in terms of operator applications. For 4 threads and above we observe an increase on the number of SP operators, which translates to increases in the number of other operators. Although we increase the number of threads, the extra work eliminates the gain from the increased degree of parallelism, something that hinders both scalability and absolute performance. `Ord-d` does not deviate from the best schedule in terms of SP applications, due to the use of the right scheduling policy, but lacking the worklist maintenance optimizations ends up performing much more work (observe the increased values of *e(CN)*, *None*). `Ord`, on the other hand, combines both a good policy for processing the worklist elements along with a careful policy for populating the worklist with new work, and manages to outperform both other variants. As the thread count increases it experiences a much smaller increase in the number processed elements (mainly manifesting as increased *e(CN)* and *None* values), but scalability is mainly hindered by a slower per-operator processing time.

Nehalem	async2		succs		preds		outer	
Threads	***t***	***sd***	***t***	***sd***	***t***	***sd***	***t***	***sd***
16	985	8	785	33	1051	4	275	2
20	1010	6	771	6	997	36	242	1
24	1063	6	802	73	960	8	209	4

Table 5.3: Execution time (sec) and stdv. for executing 100 outer-loop iterations on rmat25.

Nehalem	rmat25 (10 steps)						rand26 (10 steps)					
	preds		succs		*async2*		preds		succs		*async2*	
Threads	***t***	***sd***	***t***	***sd***	***t***	***sd***	***t***	***sd***	***t***	***sd***	***t***	***sd***
serial			261	16					601	44		
1	1014	7	385	1	806	11	1636	4	850	4	1449	24
4	309	2	155	1	205	1	515	4	308	7	364	0
8	161	1	92	1	115	1	266	1	199	8	191	1
12	120	1	80	3	101	1	206	4	202	27	144	0
16	101	0	78	4	99	2	195	5	191	19	128	0
20	91	0	76	2	101	1	187	1	191	18	125	0
24	86	1	74	1	106	1	183	0	190	9	126	0

Table 5.4: Average execution time (sec) and stdv on Nehalem.

Niagara	rmat25 (10 steps)		rand26 (10 steps)	
	async1		*async1*	
Threads	***t***	***sd***	***t***	***sd***
succs-serial	1794	39	3265	89
1	5019	81	6444	99
16	350	13	409	6
32	209	12	210	6
48	148	5	151	6
64	123	8	123	10
96	121	14	106	14
128	127	24	98	15

Table 5.5: Average execution time (sec) and stdv on Niagara.

P	SP	FU	US	e(CN)	CN	None	Time/Iter.(sec)
				`Ord`			
1	25.3	43.5	0*	0*	0*	190.9	56.5
4	25.3	43.5	0.0	0.0	0.0	190.9	14.13
8	25.4	44.1	4.1	5.6	0.6	198.8	8.1
12	25.4	44.0	3.3	6.4	0.5	198.9	6.6
16	25.5	44.0	3.2	9.3	0.6	202.0	6.3
20	25.5	43.7	0.8	16.7	0.3	209.2	6.6
24	25.7	43.8	1.2	27.2	0.6	222.3	7.2
				`Ord-d`			
1	25.3	43.5	0*	265.2	0*	1934.0	332.2
4	25.3	43.5	0.0	265.5	0.0	1934.8	83.6
8	25.3	43.5	0.0	265.5	0.0	1934.9	50.5
12	25.4	43.5	0.0	265.7	0.0	1935.1	44.2
16	25.4	43.5	0.0	265.7	0.0	1935.2	42.5
20	25.4	43.5	0.0	265.9	0.0	1936.0	42.9
24	25.4	43.5	0.0	266.0	0.0	1935.8	44.5
				`CF`			
1	25.3	43.5	0*	0*	0*	190.9	55.2
4	26.4	44.2	9.8	23.1	1.8	497.6	24.6
8	28.3	47.1	18.6	68.9	6.1	979.5	25.6
12	28.6	47.0	17.9	77.2	6.3	995.6	23.2
16	29.4	47.6	19.4	104.3	7.2	1036.7	24.5
20	27.6	45.8	18.5	55.7	4.4	841.8	21.1
24	30.1	48.7	21.3	125.1	8.5	1188.7	29.7

Table 5.6: Average number of operator applications (millions) and runtime of the forward pass per outer-loop iteration, in an execution of 2 iterations on unweighted *rmat25*(Nehalem). *P*: thread count, *e(CN)*: is the number of *CN* evaluations, *CN*: is the number of actual *CN* applications. 0* denotes an absolute zero value. *None* denotes the number of checked edges for which no operator is enabled.

Chapter 6

Optimization of Irregular Programs via Shape Analysis

6.1 Challenge Overview

In this chapter [1] we focus on the problem of optimizing the overheads of speculative parallelization of irregular algorithms. The problem setting we consider is the following: A programmer attempts to parallelize an irregular algorithm in a conventional programming language (Java in our case). One general-purpose solution to this problem is to use optimistic parallelization: computations are performed speculatively in parallel, but the runtime system monitors conflicts between concurrent computations, and rolls back offending computations as needed. There are many implementations of this high-level idea such as thread-level speculation [126], transactional memory [62, 55], and the Galois system [81]. For concreteness, our results are presented in the context of the Galois system but they are applicable to other systems as well.

[1] Part of the work presented in this chapter has appeared in *"Dimitrios Prountzos, Roman Manevich, Keshav Pingali, Kathryn S. McKinley. 'A Shape Analysis for Optimizing Parallel Graph Programs'. In Proceedings of the 38th ACM SIGPLAN Symposium on Principles of Programming Languages, (POPL) 2011. "* The first author is responsible for the conception and the implementation of the ideas presented in this publication. Additional authors provided assistance with the presentation of the material.

In the Java version of the Galois system, applications programmers write algorithms in sequential Java augmented with a construct called the *Galois unordered-set iterator*[2]. This iterator iterates in some unspecified order over a set of *active nodes*, which are nodes in the graph where computations need to be performed. The body of the iterator is considered to be an *operator* that is applied to the active node to perform the relevant computation, known as an *activity*. An activity may touch other nodes and edges in the graph, and these are collectively known as the *neighborhood* for that activity. These nodes and edges must be accessed by invoking methods from graph classes provided in the Galois library.

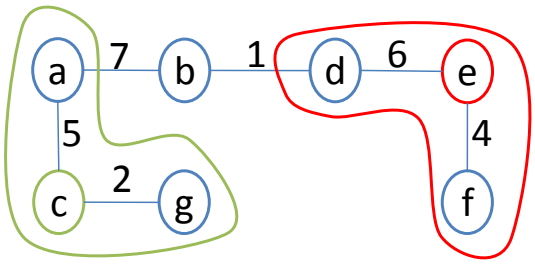


Figure 6.1: Neighborhoods in Boruvka's MST algorithm

We illustrate these concepts using Boruvka's minimal spanning tree (MST) algorithm [41], the running example in this chapter. The MST starts as a forest with each node in its own component. The algorithm iteratively contracts the graph by *non-deterministically* choosing a graph node, examining all edges incident on that node to find the lightest weight edge, and contracting

[2]The Galois system also supports ordered-set iterators, but we do not consider these in this chapter.

that edge, which is added to the MST. The algorithm terminates when the graph has been contracted to a single node. Figure 6.1 shows an undirected graph. For active node **e**, the neighborhood of the corresponding activity consists of nodes **d**, **e** and **f**, and the edges between these nodes, since these are the edges that must be examined to find and contract the lightest weight edge connected to **e**.

In most algorithms, each neighborhood is a small portion of the overall graph, so it is possible to work on many active nodes concurrently provided the corresponding neighborhoods do not overlap. For example, in Figure 6.1, the neighborhood for the activity at node **c** is disjoint from the neighborhood for the activity at node **e**, so these activities can be performed in parallel. However, the activity at node **b** cannot be performed concurrently with the activity at **e** since the neighborhoods overlap.

In the Galois system, this concurrency is exploited by adding all graph nodes to the work-set and executing iterations of the Galois set-iterator speculatively in parallel. All concurrency control is performed within the library graph classes. Conceptually, an exclusive lock called an *abstract lock* is associated with each graph element, and this lock is acquired by an activity when it touches that element by invoking a graph API method. If the lock has already been acquired by another activity, a conflict is reported to the runtime system, which rolls back offending activities. To permit rollback, methods that modify the state of the graph also record undo actions that are executed on rollback. The idea of handling conflicts at the abstract data type level rather than at

the memory level is also used in boosted TM systems [61].

Compared to static parallelization, optimistic parallelization has several overheads.

1. *Wasted work from aborted activities*: Because conflicts between activities are detected online, an activity may be rolled back after it has performed a lot of computation.

2. *Conflict checking*: Abstract locks must be acquired and released by activities, and this is an overhead even if no activities are ever aborted.

3. *Undo actions*: These must be registered for every graph API call that might mutate the graph.

In this chapter, we present a novel shape analysis that can be utilized to reduce the overheads of conflict checking and registering undo actions (reducing the number of aborted activities is mainly a scheduling problem, and is dealt with elsewhere [80]). Our main contributions are the following.

- *Shape Analysis*: We develop a novel shape analysis for programs with set and graph data structures, which infers properties for optimizing speculative parallel graph programs. We utilize the structure of stores arising in our programs to design a *hierarchy summarization abstraction*, which uses a finite set of reachability relations relative to a given property (the "object-is-locked" property), to abstract stores into shape graphs .

Our abstraction assigns unary predicates *only to root objects*, capturing reachability facts from root objects to objects deeper in the heap. Thus, the size of an abstracted store is *linear in the number of variables*, and the number of abstracted stores at a program point depends on the number of explored variable-alias sets, which tends to be constant in our programs (≈6). Therefore the number of abstract states explored by our analysis in practice is linear in the size of the program, circumventing the state-space explosion that is the bane of existing shape analyses.

- *Predicate Discovery*: We develop a simple yet effective technique for discovering predicates relevant for inferring the set objects that are always locked, at each program location, from data structure specifications and "footprints" of data structure method specifications.

- *Evaluation of effectiveness*: We implement our shape analysis in the TVLA framework and use a Java-to-TVLA front-end to analyze several benchmarks from the Lonestar Benchmark Suite [79], a collection of real-world graph-based applications that exhibit irregular behavior. Our analysis takes at most 16 seconds on each benchmark and infers all available optimizations. These optimizations result in substantial improvements in running time, ranging from 2× to 12×.

- *Bounded-model checking*: We also describe a bounded-model checking phase, which helps programmers to rewrite their program to enable more optimizations.

Several existing heap abstractions, including Canonical Abstraction [130], Boolean heaps [117], indexed predicate abstraction [85], and generalized typestates [86], abstract the heap by recording a set of unary predicates for **every object** and summarizing the heap by collapsing equivalence classes of objects with the same set of predicate values. Such abstractions achieve high precision, as they express every Boolean combination of intersection and union of objects satisfying those predicates. However, the size of a summarized heap can be exponential in the number of predicates, and the summarization of a set of stores can be doubly-exponential. We call these *bottom-up abstractions*, since they typically express reachability facts for objects in the depth of the heap relative to heap roots. Our experience with bottom-up abstraction shows that heaps are partitioned very finely, leading to state space explosion. As we discuss in Section 6.4, our *top-down abstraction* runs several orders of magnitude faster than an implementation of the bottom-up abstraction approach when analyzing our benchmarks.

The rest of the chapter is organized as follows. Section 6.2 provides an overview of our optimizations and shape analysis on Boruvka's MST example. Section 6.3 presents our shape analysis via hierarchy summarization and predicate discovery. Section 6.4 describes the static analysis implementation and gives experimental results that demonstrate the effectiveness of our approach.

6.2 Solution Overview

This section introduces the programming model, the performance optimizations, and our shape analysis informally, using Boruvka's MST algorithm as the running example.

6.2.1 Boruvka's MST algorithm

Pseudocode for the algorithm is shown in Figure 6.2. The Galois iterator on line 25 iterates over the graph nodes in some non-deterministic order, performing edge contractions. In lines, 31-38 we examine the neighbors of the active node `a`, and identify the neighbor `lt`, which is connected to `a` by a lightest weight edge. In lines 44-59 we contract the two components by removing `lt` from the graph, and updating all of `lt`'s neighbors to become neighbors of `a`. This is done by the loop in lines 45-58. If a neighbor `n` of `lt` is already connected to `a`, we update the data value of the edge connecting them (lines 49-54). Otherwise, we add an edge connecting the two nodes (lines 55-57).

In Boruvka's algorithm, the neighborhood of an active node `a` consists of the immediate neighbors of `a` and `lt` and their related edges and data. In more complex examples like Delaunay mesh refinement, the neighborhood of an activity can be an unbounded subgraph.

```
1  class GaloisRuntime {
2    @rep static set<Object> locks; // abstract
         locks
3    // Flag options
4    static int LOCK_UNDO=0; // acquire locks +
         log undo
5    static int UNDO   =1; // log undo
6    static int LOCK   =2; // acquire locks
7    static int NONE   =3; // no locks and no
         undo
8  }

10 class Weight {
11   static Weight MAX_WEIGHT;
12   int v;
13   // We record the endpoints of the edge that
14   // holds the weight in the input graph.
15   final Node<Void> initSrc, initDst;
16   int compareTo(Weight other);
17 }
18 class Boruvka {
19   void main() {
20     Graph<Void,Weight> g = ...// read from file
21     GSet<Node> wl = new GSet<Node>();
22     wl.addAll(g.getNodes(NONE), NONE);
23     GBag<Weight> mst = new GBag<Weight
           >();

25     // Galois iterator
26     foreach (Node a : wl) { // in any order
27 L1:   Set<Node> aNghbrs = g.
       getNeighbors(a, LOCK);
28       // Find neighbor incident to lightest edge
29       Weight minW = Weight.MAX_WEIGHT;
30       Node lt = null;
31 L2:   for (Node n : aNghbrs) { // Iterator nIter
32         Edge e = g.getEdge(a, n, NONE);
33         Weight w = g.getEdgeData(e, NONE);
34         if (w.compareTo(minW) < 0) {
35           minW = w;
36           lt = n;
37         }
38       }
39       if (lt == null) // no neighbors
40         continue;
41       // Contract edge (a, lt)
42 L3:   g.getNeighbors(lt, LOCK); // avoids
       undo in L4
43 L4:   g.removeEdge(a, lt, NONE);
44 L5:   Set<Node> ltNghbrs = g.getNeighbors(lt,
       NONE);
45 L6:   for (Node n : ltNghbrs) { // Iterator nIter
46         Edge e = g.getEdge(lt, n, NONE);
47         Weight w = g.getEdgeData(e, NONE);
48         Edge an = g.getEdge(a, n, NONE);
49         if (an != null) { // merge edges
50           Weight wan = g.getEdgeData(an,
                 NONE);
51           if (wan.compareTo(w) < 0)
52             w = wan; // use minimal weight
53 L7:       g.setEdgeData(an, w, NONE);
54         }
55         else { // new neighbor for a
56 L8:       g.addEdge(a, n, w, NONE);
57         }
58       }
59 L9:   g.removeNode(lt, NONE);
60 L10:  mst.add(minW, NONE);
61 L11:  wl.add(a, NONE); // put node back on
       worklist
62     } } }
```

Figure 6.2: Simplified implementation of Boruvka's algorithm.

6.2.2 Speculative Execution in Galois

The graph data structure `Graph<ND,ED>` is parameterized by data objects referenced by nodes and edges, respectively. The work list of active nodes is stored in a set `GSet<Node>`. The `Weight` objects, which record the weights of the MST edges and their end-points (nodes) in the original graph, are stored in another collection `GBag<Weight>`, which only allows addition operations in a concurrent context. The last argument to a data structure method is a flag that tells the runtime system whether the method should attempt to acquire abstract locks and whether it should log an inverse method call. The default value `LOCK_UNDO` is always a safe choice, ensuring correctness of speculative execution.

The Galois system protects user-defined data types, such as `Weight`, using a read-write lock (allowing concurrent read operations but at most one write operation) and maintaining backup copies of such objects.

Iterations of the Galois set iterator are executed speculatively in parallel, and this execution has transactional semantics: an iteration either completes and commits, or is rolled back and retried.

6.2.3 Data Structure Specifications

Figure 6.3 shows the syntax for a lightweight specification of abstract data types (for all clients), defining their abstract state, abstract locks acquired by each method, and operational semantics of each method in terms of abstract fields. The set of variables, includes method parameters, the special

Syntactic Categories

TName	Types
OFld	Pointer fields
SFld	Set fields
Field	All fields
PVar	Pointer variables
BVar	Boolean variables
SVar	Set-valued variables
Var	All variables

Data Types (EBNF)

TypeDecl ::=	`class` *TName*{*FieldDecl** *MethodDef**}
FieldDecl ::=	[`@rep`] [`static`] *TName OFld*; |
	`@rep` [`static`] *set*⟨*TName*⟩ *SFld*;
MethodDef ::=	`@locks`(*Path**) `@op`(*Stmt**) `Java-code`
Stmt ::=	*Var* = *Expr* | *Var.Field* = *Expr*
Expr ::=	*Path* | *Path* + *Path* | *Path* − *Path* | `choose`(*Path*) |
	Path `in` *Path* | *Path* `notIn` *Path* |
	`isEmpty`(*Path*) | `new` *TName*((*Field* = *Var*)*)
Path ::=	*Var*.(*Var* + *Field*[:*SVar*] + `rev`(*Field*)[:*SVar*])$^+$

Figure 6.3: EBNF grammar for specified data structures. The notation $[x]$ means that `x` is optional.

`ret` parameter for returning values, static variables, and temporary variables used to define the semantics of methods. The formal semantics of this language can be found in Appendix A. Our analysis operates in terms of these specifications, ignoring the internal details of library ADT's. We assume the correctness of the specifications; approaches such as [156] can be used for their verification.

Figure 6.4 shows a graph type built from the `Node` and `Edge` types and the parametric types `ND` and `ED`, used to store user-defined data on the graph nodes and edges. In our example, nodes do not store any data objects and thus their `nd` fields are null. Figure 6.5 shows a bag, a set, and an iterator type.

Specifying Abstract Data Types. The `@rep` annotations in Figure 6.4 define the abstract state of a data structure in terms of *set-fields* [82], i.e., fields whose values are sets of objects.

For example, the abstract state of `Graph` is given by a pair of sets — `ns` and `es` — representing the set of graph nodes and edges, respectively. [3] We represent an undirected edge by a single edge, directed arbitrarily.

`GaloisRuntime` contains a static (i.e., global) `locks` set, representing the set of abstract locks acquired by an iteration. [4]

[3] The full specification includes additional first-order constraints.

[4] Galois implements a lock coarsening scheme by maintaining a single set of abstract locks, shared among all data structure instances.

```
1  @rep set<Node> ns; // graph nodes
2  @rep set<Edge> es; // graph edges

4  @locks(n.rev(src).dst, n.rev(dst).src)
5  @op(nghbrs = n.rev(src).dst + n.rev(dst).src,
6      ret = new Set<Node<ND>>(cont=nghbrs))
7  Set<Node<ND>> getNeighbors(Node<ND> n, int opt);

9  @locks(f.rev(src).dst.t, t.rev(src).dst.f)
10 @op(f.rev(src):eft.dst.t, t.rev(src):etf.dst.f,
11     ret = choose(eft + etf))
12 Edge<ED> getEdge(Node<ND> f, Node<ND> t, int opt);

14 @locks(f, t, f.rev(src).dst.t, t.rev(src).dst.f)
15 @op(f.rev(src):eft.dst.t, t.rev(src):etf.dst.f,
16     ret = (eft+etf) in es,
17     ne = new Edge<ED>(src=f, dst=t, ed=d),
18     es += ne)
19 boolean addEdge(Node<ND> f, Node<ND> t, ED d,
20                 int opt);

22 @locks(n.rev(src).dst, n.rev(dst).src)
23 @op(ret = n in ns, ns = n,
24     es = n.rev(src) + n.rev(dst))
25 boolean removeNode(Node<ND> n, int opt);

27 @locks(f.rev(src).dst.t, t.rev(src).dst.f)
28 @op(f.rev(src):eft.dst.t, t.rev(src):etf.dst.f,
29     ret = (etf + eft) in es, es = (etf + eft))
30 boolean removeEdge(Node<ND> f, Node<ND> t, int opt);

32 @locks(n)
33 @op(ret = n.nd)
34 ND getNodeData(Node<ND> n, int opt);

36 @locks(e, e.src, e.dst)
37 @op(ret = e.ed)
38 ED getEdgeData(Edge<ED> e, int opt);

40 @locks(e, e.src, e.dst)
41 @op(e.ed = d)
42 void setEdgeData(Edge<ED> e, ED d, int opt);

44 }

46 class Node<ND> {
47  ND nd; // data object
48 }

50 class Edge<ED> {
51   Node src; // edge origin
52   Node dst; // edge destination
53   ED ed;    // data object
54 }
```

Figure 6.4: Graph specification samples.

```
1  class GSet<E> { // boosted set
2   @rep set<E> gcont;// set contents
3   @locks(e)
4   @op(ret = e in gcont)
5   boolean contains(E e);
6   @locks(e)
7   @op(ret = e notIn gcont, gcont += e)
8   boolean add(E e);
9  }

11 class GBag<E> {//boosted bag for reduction operations
12  @rep set<E> bcont; // bag contents
13  @locks() // No locks required!
14  @op(bcont += e)
15  void add(E e);
16  @op(ret = new Set<E>(bcont))
17  Set<E> toSet(); // used only in sequential code
18 }

20 interface Set<E> { // sequential set from java.util
21  @rep set<E> cont; // set contents
22 }

24 interface Iterator<E> { // iterator from java.util
25  @rep Set<E> all; // underlying set
26  @rep set<E> past; // past iteration elements
27  @rep E at;          // element at current  iteration
28  @rep set<E> future; // future iteration elements
29 }
```

Figure 6.5: Set, bag and iterator specification samples.

Example 3. *Figure 6.6 shows an abstract store representing the input graph of Figure 6.1 where* `a` *references the active node* **a** *and* `lt` *references* **c** *— the node connected to* **a** *by the lightest edge, discovered on the second iteration after iterating over* **b**. *The figure does not show objects used by the internal (concrete) representation of specified data types. Instead, it uses the (*`@rep`*) set fields to indicate that an object is contained in a set field of a data structure.*

Filled locks denote objects in `GaloisRuntime.locks`*; hollow locks denote objects for which our analysis infers that lock protection is not required.*

Path Language. We use a language of access path expressions (access paths for short) to denote the set of objects that can be obtained by following variables and fields in a store: a variable (`x`) denotes the object it references; a pointer field `e.f` denotes an object obtained by traversing the field `f` forward from an object denoted by the prefix expression `e`; a set field denotes any object stored in the set stored in a given object; a reverse field, written `rev(f)` or $\overleftarrow{f}$, denotes objects obtained by traversing field `f` backwards. We formalize path expressions in Section 6.3.

We use the notation `:x` inside a path expression to denote a set of intermediate objects during an access path traversal. We write `+` and `-` for set union and difference respectively.

Example 4 (Legend)**.** *The following paths are derived from the abstract store in Figure 6.6:*

- *`a.rev(src)` represents the outgoing edges of `a`:* $\{2, 3\}$.
- *`a.rev(dst)` represents the incoming edges of `a`:* $\emptyset$.
- *`a.rev(src).dst + a.rev(dst).src` represents all of the graph nodes adjacent to `a`:* $\{b, c\}$.
- *`a.rev(src):x.dst.lt` sets the temporary variable `x` to the edge from `a` to `lt`:* $x = \{3\}$ *(2 is not on a path from `a` to `lt`).*

Specifying Abstract Locks. A `@locks` annotation defines the set of abstract locks a method should acquire by a set of path expressions. To keep specifications succinct, access paths expressions in `@locks` stand for all of their prefixes (e.g., `n.rev(src).dst` stands for `n`, `n.rev(src)`, and `n.rev(src).dst`).

We call a node referenced by `n` along with its incident edges and adjacent nodes the immediate neighborhood of `n`. We specify the set of locks for such a neighborhood by `@locks(n.rev(src).dst, n.rev(dst).src)`.

Example 5 (Commutativity via abstract locks)**.** *The `removeNode` method specifies locks for the immediate neighborhood of the node being removed. A call to `removeNode(c,LOCK)` attempts to lock the `Node` object `c`, the `Edge` objects referencing `c` via the `src` field or the `dst` field (the edges incident to `c`), and the `Node` objects that are neighbors of `c`. This ensures the concurrent method calls `removeNode(c,LOCK)` and `removeNode(e,LOCK)` will not cause their respective iterations to abort, since the immediate neighborhoods of `c` and `e` do not overlap.*

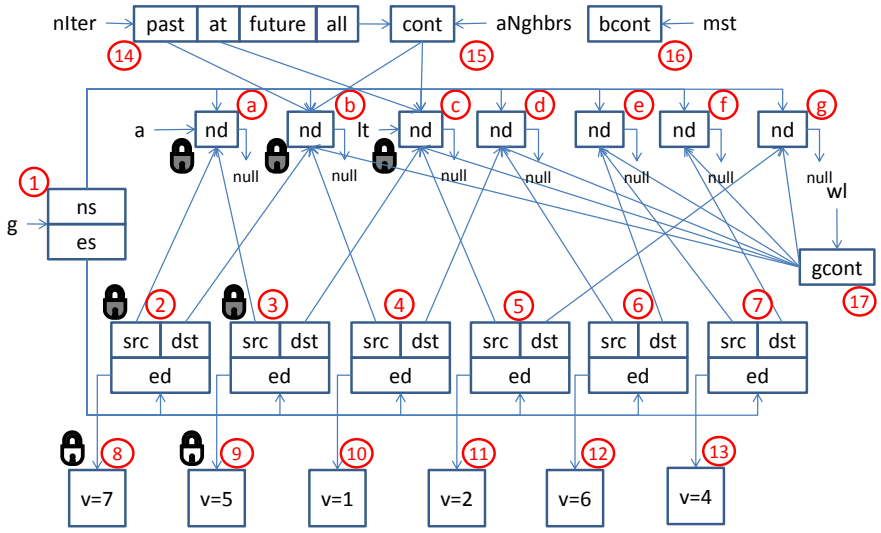


Figure 6.6: An abstract store arising at `L2`, using as input the graph from Figure 6.1. Object are shown by rectangles sub-divided by their fields; circles are used to name objects; locks show objects contained in the global `GaloisRuntime.locks` set. We label `Node` objects by the same labels used in Figure 6.1 and other objects by a running index.

Specifying Method Semantics. An `@op` annotation defines the semantics of a method by a simple imperative language. The language allows a sequence of statements, using set expressions over method parameters, static fields, and temporary variables. Expressions of the form `a in b`, `a notIn b`, and `isEmpty(a)`, test whether `a` is contained in `b`, `a` is not contained in `b`, and whether `a` is an empty set, respectively. (We overload these expressions to treat reference variables as singleton sets.) Statements of the form `a += exp` and `a -= exp` are shorthand for `a = a + exp` and `a = a - exp`, respectively. `choose(exp)` non-deterministically chooses an object from a set denoted by `exp`.

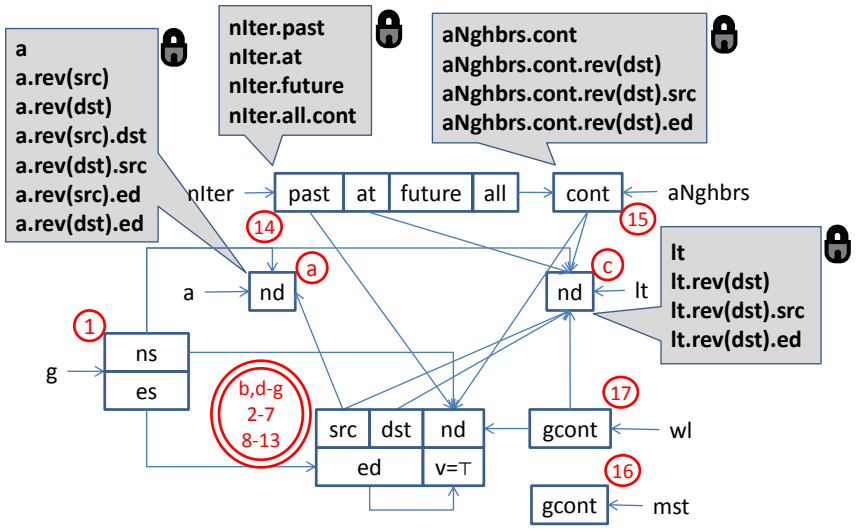


Figure 6.7: A shape graph obtained by applying hierarchy summarization abstraction to the store in Figure 6.6. Grey boxes represent sets of locked objects. v=⊤ indicates that the numeric value of v has been abstracted away.

6.2.4 Optimization Opportunities

Our static analysis enables the following optimizations.

Eliminating Usage of Concurrent Data Structures. The following conditions allow replacing a concurrent implementation of a data structure by a sequential implementation: the data structure is iteration private, or the data structure is never modified. We use a purity analysis [131] to discover objects that are never modified inside an iteration (such as `Weight` in the running example).

Reducing Rollback Logging. Logging inverse method calls for iterations that commit represents wasted work, as the log is cleared when the iteration commits and the logged method calls are never used.

Our static analysis finds a minimal set of *failsafe points* — program locations in the client program such that an iteration reaching them cannot abort. The analysis computes an under-approximation of the set of objects that are always locked at a program location. If the set of locks computed for a location `L` subsumes the set of locks computed for all locations reachable from `L`, then `L` is a failsafe point. An iteration reaching a failsafe point will never fail to acquire a lock and therefore cannot abort. We eliminate logging inverse actions for method calls appearing after a failsafe point.

If no method call before a failsafe point modifies shared data structures, rollback logging is not needed anywhere in the iteration. Algorithms with this property are called *cautious* algorithms [102].

Eliminating Redundant Locking. Our analysis can also be used to find method calls for which all locks have already been acquired by preceding method calls. Lock acquisitions can be eliminated for these calls. Furthermore, our analysis finds user-defined objects "dominated" by other locked objects, *i.e.*, objects that can only be accessed after a unique abstract lock is acquired. We eliminate lock operations for such objects as well.

6.2.5 Optimizing the Running Example by Static Analysis

We develop a sound static analysis to automatically infer available optimizations of the kind discussed above. The input to our analysis is a Java program with a single parallel loop, given by the `foreach` construct, operating over a library of specified boosted data structures. The output of our analysis is an assignment of option flags to each ADT method call and a list of (user-defined) types that do not need "transactional" protection.

The core component of our analysis is a shape analysis that under-approximates the set of objects that are always locked at a program point. Intuitively, our analysis abstracts stores into bounded-size shape graphs by collapsing all objects not referenced by variables together and recording for each root object a set of path expressions denoting the set of locked objects.

Figure 6.7 shows a shape graph obtained by applying our abstraction to the store in Figure 6.6. The object labeled by a double-circle shows the set of collapsed objects. The grey box pointing to **a** expresses the fact that the immediate neighborhood of **a** is locked, along with the `Weight` objects referenced by its incident edges. This shape graph represents an intermediate invariant inferred by our analysis at `L2`. The full invariant is given by a set of the shape graphs at that point, at the fixpoint.

Below, we provide sample invariants that our analysis infers for Figure 6.2 and the corresponding path expressions denoting sets of objects all of which are locked:

1. At `L2`, the immediate neighborhood of `a` is locked:
   `a + a.rev(dst).src + a.rev(src).dst`.

2. At `L4`-`L9`, the immediate neighborhoods of `a` and `lt` are locked:
   `a + a.rev(dst).src + a.rev(src).dst +`
   `lt + lt.rev(dst).src + lt.rev(src).dst`.

3. At `L2` and `L6`, all graph nodes accessible by the iterator `nIter` (past, present, and future iterations) are locked: `nIter.past + nIter.at + nIter.future`.

4. At `33`, `47`, `50`, and `53`, the edges referenced by `e` and `an` and the nodes they reference are locked:
   `e + an + e.src + e.dst + an.src + an.dst`.

*Inv*1 is part of the invariant needed to prove that `L4` is a failsafe point (before executing the statement). *Inv*1 needs to be maintained from `L2` and on. It is also used to eliminate locking in lines `32`-`33`. *Inv*2 helps establish `L4` as a failsafe point, since all accesses to nodes and edges in the second loop are to objects known to be locked. Also, it helps eliminate redundant locking at `L9`. *Inv*3 helps establish the failsafe point at `L4` by the fact that the node referenced by `n` is locked at `46` and `48`, and eliminate locking at `32`. Finally, *Inv*4 establishes that the calls to `getEdgeData` and `setEdgeData` in lines `33`, `47`, `50`, and `53` do not lock new objects.

Additionally, our analysis infers that `Weight` objects are read-only, which enables eliminating all lock operations and backup copy maintenance for

them. Points `L7`, `L8`, `L10`, `L11` are after the failsafe point and do not require storing inverse actions. At `L10`, the calls to the `add` method of `Bag` do not require acquiring locks, and trivially commute.

We apply these optimizations to the code of Figure 6.2 by setting the `LOCK` option at `L1` and `L3`, which eliminates rollback logging, and setting the `NONE` option in all other calls, eliminating both abstract locking and rollback logging.

This implementation of Boruvka's algorithm is cautious: our analysis infers that the failsafe point is `L4` and that no modifications are made to the graph between `L1` and `L4`. If we remove the statement at `L3`, the failsafe point is at `L5`, which requires logging an inverse method call for `g.removeEdge(a, lt)`.[5]

6.3 A Shape Analysis for Graph Programs

This section presents our static analysis for enabling the optimizations described in previous sections. The core component of the analysis is a shape analysis that under-approximates the set of objects that are always locked, at each program location. This section is organized as follows: (1) we discuss the class of programs and stores that our shape analysis addresses; (2) we define Canonical Abstraction [130] and partial join [94] in our setting; (3) we define

[5]Swapping `L4` and `L5` makes the code cautious once again, but breaks sequential correctness, since in the Galois library it is illegal to remove an edge while iterating over the neighbors of a node incident to it.

***H**ierarchy **S**ummarization **A**bstraction* (*HSA*); (4) we present a technique for discovering predicates relevant to our analysis; (5) we explain how the results of the shape analysis are used; (6) we contrast our abstraction with ***B**ackward **R**eachability **A**bstraction* (*BRA*), a commonly used form of shape abstraction; (7) we discuss how our analysis can aid the programmer by providing non-cautiousness counterexamples; and finally (8) we discuss limitations of our analysis.

6.3.1 A Class of Programs and Stores

We analyze Java programs (excluding recursive procedures) where the implementation of specified data structures is replaced by the abstract fields in the `@rep` annotations and the semantics of methods is given by the `@op` annotations.

Figure 6.8 defines stores in terms of pointer fields and set-valued fields defined by the `@rep` annotations. We define the meaning of path expressions (recursively), which denote sets of objects reachable from a variable by following fields in specified directions and going through specified variables. The last definition in Figure 6.8 provides the meaning of variables assigned to intermediate objects along path expressions, such as `f.rev(src):eft.dst.t`.

Bounded-depth Hierarchical Stores. We define the set of types reachable from an object o (by forward paths) to be the set of types of all objects in $[\![o.p]\!]$ for all path expressions p.

Stores

T_O	Objects
$\mathit{Stack} : \mathit{PVar} \to T_O \cup$	Stacks
$\quad \mathit{SVar} \to 2^{T_O} \cup$	
$\quad \mathit{BVar} \to \{\mathtt{T}, \mathtt{F}\}$	
$\mathit{Heap} : (T_O \times \mathit{OFld}) \to T_O \cup$	Heaps
$\quad (T_O \times \mathit{SFld}) \to 2^{T_O}$	
$\mathit{Store} : \mathit{Stack} \times \mathit{Heap}$	Stores
$\sigma = (S^\sigma, H^\sigma)$	Store notation

Semantics of Path Expressions

$[\![\mathit{Path}]\!] : \mathit{Store} \to 2^{T_O}$

Base case. Variables:

$$[\![x]\!](\sigma) = \begin{cases} \{S^\sigma(x)\}, & x \in \mathit{PVar}; \\ S^\sigma(x), & x \in \mathit{SVar}. \end{cases}$$

Inductive case. $p \in \mathit{Path}$, $[\![p]\!](\sigma)$ is known, and e is a field or variable:

$$[\![p.e]\!](\sigma) = \begin{cases} [\![p]\!](\sigma) \cap [\![e]\!](\sigma), & e \in \mathit{PVar} \cup \mathit{SVar}; \\ \{H^\sigma(o, e) \mid o \in [\![p]\!](\sigma)\}, & e \in \mathit{OFld}; \\ \bigcup\limits_{o \in [\![p]\!](\sigma)} H^\sigma(o, e), & e \in \mathit{SFld}; \\ \{o \in T_O \mid H^\sigma(o, f) \in [\![p]\!](\sigma)\}, & e = \mathtt{rev}(f), f \in \mathit{OFld}; \\ \{o \in T_O \mid H^\sigma(o, f) \cap [\![p]\!](\sigma) \neq \emptyset\}, & e = \mathtt{rev}(f), f \in \mathit{SFld}. \end{cases}$$

For an object $o \in T_O$ and path $p \in \mathit{Path}$, we define
$[\![o.p]\!](\sigma) = \textbf{let } y \text{ be fresh}, \sigma' = (S^\sigma | y \mapsto o, H^\sigma) \textbf{ in } [\![y.p]\!](\sigma')$

Meaning of intermediate variables:
The expression $x.p{:}v.q$ assigns to v the set of objects that are both on a path from x to q and in $x.p$. For $x \in \mathit{Var}, v \in \mathit{SVar}, p, q \in \mathit{Path}$
$[\![v]\!](\sigma) = [\![x.p]\!](\sigma) \cap \{o \in T_O \mid [\![o.q]\!](\sigma) \neq \emptyset\}$.

Figure 6.8: Stores and semantics of path expressions.

Our work focuses on the class of *bounded-depth hierarchical stores* — stores where the set of types reachable from $[\![\texttt{o.f}]\!]$ is a proper subset of the set of types reachable from o, for every object o and field f. Such stores are acyclic — the length of any unidirectional path, i.e., a path where all fields are either forward or reverse, is linearly bounded by the number of program types.

6.3.2 Canonical Abstraction and Partial Join

We implement our shape analysis using the TVLA system [89], which allows defining stores by first-order predicates, program statements by first-order transition formulae (formulae relating the values of predicates after a statement to those before), and abstract states by first-order *abstraction predicates*. The system automatically generates sound abstract operations and transformers, yielding a sound abstract interpretation for a given program.

TVLA uses Canonical Abstraction [130], which abstracts stores into 3-valued logical structures. To focus our presentation on the important details of our analysis, we simplify our description of TVLA's abstraction and use *shape graphs* for abstract states instead of 3-valued structures.

Definition 6.3.1 (Shape Graph)**.** *Let $P = AP \cup NAP$ be a set of predicates consisting of two disjoint sets of unary predicates called* abstraction predicates *(AP) and* non-abstraction predicates *(NAP). A shape graph G is a tuple (N^G, P^G, E^G) where N^G is a set of abstract objects, $P^G : N^G \to 2^P$ assigns predicates to objects, and $E^G : OFld \cup SFld \to N^G \times N^G$ is a set of* may-edges

for each field. We denote the set of shape graphs over P by ShapeGraph[P].

We call the set of abstraction predicates assigned to an abstract node $v \in N^G$ its *canonical name*: $\mathit{CName}(v) \stackrel{\text{def}}{=} P^G(v) \cap AP$. A shape graph G is *bounded* if no two abstract nodes have the same canonical name. This means that the number of abstract nodes in a bounded shape graph is *exponentially bounded* by the number of abstraction predicates.

We define the abstraction function $\beta[P] : \mathit{Store} \to \mathit{BGraph}[P]$, which maps a store $\sigma = (S^\sigma, H^\sigma)$ into a bounded shape graph G as follows. We use the helper function $P^\sigma : T_O \to 2^P$, which evaluates the predicates in P for each object, and $\mu^{\sigma,G} : T_O \to N^G$, which maps store objects having an equal canonical name to an abstract node representing their equivalence class in G. The predicate assignment function assigns to abstract nodes the predicates common to all objects they represent, and a field edge exists between two abstract nodes if there exist two objects represented by the abstract nodes that are related by that field.

$$\mu^{\sigma,G}(o_1) = \mu^{\sigma,G}(o_2) \iff P^\sigma(o_1) \cap AP = P^\sigma(o_2) \cap AP$$

$$\begin{aligned} P^G(n) = {} & \bigcap_{o \text{ s.t. } n=\mu^{\sigma,G}(o)} P^\sigma(o) \\ E^G(f) = {} & \{(n_1, n_2) \mid \exists o_1, o_2 \colon n_1 = \mu^{\sigma,G}(o_1), n_2 = \mu^{\sigma,G}(o_2). \\ & \quad \begin{cases} o_2 = H^\sigma(f)(o_1), & f \in \mathit{OFld}; \\ o_2 \in H^\sigma(f)(o_1), & f \in \mathit{SFld}. \end{cases} \\ & \} \ . \end{aligned}$$

We say that a shape graph G' *subsumes* a shape graph G, written $G \sqsubseteq G'$, if there exists an onto function $\mu^{G,G'} : N^G \to N^{G'}$, such that

$P^G(n) \supseteq P^{G'}(\mu^{G,G'}(n))$ for all $n \in N^G$, and $(n_1, n_2) \in E^G(f)$ implies that $(\mu^{G,G'}(n_1), \mu^{G,G'}(n_2)) \in E^{G'}(f)$ for all $n_1, n_2 \in N^G$, $f \in \mathit{OFld} \cup \mathit{SFld}$.

The meaning of a shape graph G is given by the function $\gamma[P] : \mathit{ShapeGraph}[P] \to 2^{\mathit{Store}}$ defined as $\gamma[P](G) = \{\sigma \mid \beta[P](\sigma) \sqsubseteq G\}$.

We say that two shape graphs G and G' are *congruent* if there exists a bijection between their sets of abstract nodes $\mu^{G,G'} : N^G \to N^{G'}$, which preserves the abstraction predicates: $P^G(n) \cap AP = P^{G'}(\mu^{G,G'}(n)) \cap AP$ for all $n \in N^G$. Two congruent shape graphs G and G' can be subsumed by a congruent shape graph $G'' = G \sqcup G'$, by intersecting corresponding predicate values and taking the union of corresponding edges using the bijections $\mu^{G'',G} : N^{G''} \to N^G$ and $\mu^{G'',G'} : N^{G''} \to N^{G'}$:

$$\begin{array}{rcl} N^{G''} & = & N^G \\ P^{G''}(n) & = & P^G(\mu^{G'',G}(n)) \cap P^{G'}(\mu^{G'',G'}(n)) \\ (n_1, n_2) \in E^{G''}(f) & \Leftrightarrow & (\mu^{G'',G}(n_1), \mu^{G'',G}(n_2)) \in E^G(f) \text{ or} \\ & & (\mu^{G'',G'}(n_1), \mu^{G'',G'}(n_2)) \in E^{G'}(f) \ . \end{array}$$

We use TVLA's partial join operator [94], which merges congruent shape graphs into a single shape graph, and keeps non-congruent shape graphs in a set:

$$\{G\} \sqcup \{G'\} \stackrel{\text{def}}{=} \begin{cases} \{G \sqcup G'\}, & G \text{ and } G' \text{ are congruent;} \\ \{G, G'\}, & \text{else.} \end{cases}$$

The abstraction of a set of stores $\alpha[P] : 2^{\mathit{Store}} \to 2^{\mathit{BGraph}}[P]$ is defined as $\alpha[P](\Sigma) = \bigsqcup_{\sigma \in \Sigma} \beta[P](\sigma)$.

6.3.3 Hierarchy Summarization Abstraction

Our abstraction is defined relative to a set of *abstraction paths*, denoted by *AbsPaths*, which represent possible paths from variables to locked objects. The next subsection discusses a technique to discover a set of useful abstraction paths for a set of data structures.

Let $\sigma = (S^\sigma, H^\sigma)$ be a store. For a pointer variable `x` and an abstraction path p, we define a unary predicate expressing the fact that v is a root object referenced by `x` and **all** objects reachable from it by the path p are locked:

$$\mathit{ForwardReach}[x, p](v) \stackrel{\text{def}}{=} [\![x]\!]^\sigma = \{v\} \wedge [\![x.p]\!]^\sigma \subseteq [\![\texttt{locks}]\!]^\sigma \ .$$

(Our implementation optimizes the use of predicates by removing the program variables and having just one abstraction predicate for all variables of a given type.)

We encode hierarchy summarization abstraction via shape graphs and the set of predicates P^{HSA}, shown in Table 6.1, and the abstraction paths in Table 6.2. Since a pointer variable points to at most one node, the number of abstract nodes in a bounded shape graph $G \in \mathit{BGraph}[P^{HSA}]$ is equal to at most the number of heap roots + 1 (in the case where there exist non-root objects). The canonical names in such a shape graph are the sets of aliased pointer variables. We call such sets *aliasing configurations*. In practice, the average number of different aliasing configurations discovered by our analysis is a small constant ($\approx$6), which means that the set of bounded shape graphs our analysis explores is linear in the number of program locations.

Predicates	Meaning
Abstraction Predicates	
$\{x(v) \mid x \in Var\}$	`x` references v
Non-abstraction Predicates	
$\{ForwardReach[x,p](v) \mid x \in Var, p \in AbsPaths\}$	Hierarchy summarization predicates

Table 6.1: P^{HSA} predicates for hierarchy summarization abstraction.

Type	Abstraction Paths
`Graph`	`es`, `es.src`, `es.dst`, `ns`, `ns.`$\overleftarrow{\texttt{src}}$, `ns.`$\overleftarrow{\texttt{dst}}$, `ns.`$\overleftarrow{\texttt{dst}}$`.ed`, `ns.`$\overleftarrow{\texttt{dst}}$`.ed`, `ns.`$\overleftarrow{\texttt{src}}$`.dst`, `ns.`$\overleftarrow{\texttt{dst}}$`.src`, `ns.nd`, `ns.`$\overleftarrow{\texttt{src}}$`.dst.nd`, `ns.`$\overleftarrow{\texttt{dst}}$`.src.nd`, `es.ed`, `es.src.nd`, `es.dst.nd`
`Node`	`a`, `lt`, `n`, $\overleftarrow{\texttt{src}}$, $\overleftarrow{\texttt{dst}}$, $\overleftarrow{\texttt{src}}$`.dst`, $\overleftarrow{\texttt{dst}}$`.src`, `nd` $\overleftarrow{\texttt{src}}$`.ed`, $\overleftarrow{\texttt{dst}}$`.ed`, $\overleftarrow{\texttt{src}}$`.dst.nd`, $\overleftarrow{\texttt{dst}}$`.src.nd`
`Edge`	`e`, `an`, `src`, `dst`, `ed`, `src.nd`, `dst.nd`
`Weight`	$\overleftarrow{\texttt{ed}}$, $\overleftarrow{\texttt{ed}}$`.src`, $\overleftarrow{\texttt{ed}}$`.dst`, $\overleftarrow{\texttt{ed}}$`.src.nd`, $\overleftarrow{\texttt{ed}}$`.dst.nd`
`Set`	`cont`, `cont.`$\overleftarrow{\texttt{src}}$, `cont.`$\overleftarrow{\texttt{dst}}$, `cont.`$\overleftarrow{\texttt{src}}$`.dst`, `cont.`$\overleftarrow{\texttt{dst}}$`.src`, `cont.`$\overleftarrow{\texttt{src}}$`.ed`, `cont.`$\overleftarrow{\texttt{dst}}$`.ed`, `cont.`$\overleftarrow{\texttt{src}}$`.dst.nd`, `cont.`$\overleftarrow{\texttt{dst}}$`.src.nd`, `cont.nd`
`GSet`	`gcont`, `gcont.nd`
`GBag`	`bcont`
`Iterator`	`all`, `all.cont`, `all.cont.nd`, `past`, `at`, `future`, `past.nd`, `at.nd`, `future.nd`

Table 6.2: Abstraction paths for the running example. We omit Java Generics parameters when no confusion is likely.

Figure 6.7 shows the result of applying $\beta[P^{HSA}]$ to the store in Figure 6.6 and the predicates in Table 6.1. Heap roots are labeled with path expressions that denote the sets of objects that are reachable from them and are definitely locked. At `L2`, we would expect that node **a**, its neighbors, and the edges connecting **a** are locked. This is specified by the path expressions labeling node **a**. For example, `a.rev(src).dst`, `a.rev(dst).src` refer to all the neighbors of **a**. Additionally, the current element that we are iterating over is node **c**, which is the lightest neighbor of **a**; this node has its single incoming edge and edge data locked. All other (non-root) nodes, edges, and `Weight` objects are collapsed by our abstraction.

6.3.4 Predicate Discovery

We now describe heuristics for generating the set of abstraction paths from the data structures in a program, and show how it finds paths expressing the invariants described in Section 6.2. Our technique constructs paths in three phases: (a) building the *type dependence graph*, (b) discovering *variable-to-lock* paths in method specifications, and (c) combining variable-to-lock paths and all forward paths in the type dependence graph.

Definition 6.3.2 (Type Dependence Graph)**.** *A* type dependence graph *for a program, contains a* type node N_T *for each program type* T*, labeled by the set of program variables of that type; and a* field edge *from type node* N_T *to type node* $N_{T'}$*, labeled by a field of type* T' *or set*$\langle T' \rangle$ *declared in type* T*.*

Figure 6.9 shows the type dependence graph for Figure 6.2. For the

rest of this section, we fix the set of variables and fields, and define the set of well-formed path expressions, *WFPath*.

Definition 6.3.3 (Well-formed Path Expressions)**.** *Define the* type-node pair *of a path expression element as follows: $TNPair(x) = (N_T, N_T)$ for a variable x of type T; $TNPair(f) = (N_T, N_{T'})$ for a field f of type T' or $set\langle T' \rangle$ declared in a type T; and $TNPair(\overleftarrow{f}) = (N_{T'}, N_T)$ for a reversed field expression $\overleftarrow{f}$, if $TNPair(f) = (N_T, N_{T'})$.*

Let p be a path expression $e_1.e_2.\ldots.e_k$ and let the corresponding sequence of type-node pairs be $(N_1, N'_1), \ldots, (N_k, N'_k)$. We say that p is well-formed *if the sequence of type-nodes*
$N_1, N'_1, \ldots, N_k, N'_k$ is an undirected path in the type dependence graph. We define the type-node pair of p to be $TNPair(p) = (N_1, N'_k)$.

Example 6. *For example, $\texttt{nIter.at.}\overleftarrow{\texttt{gcont}}\texttt{.wl}$ is well-formed, whereas $\texttt{g.nd}$ and $\overleftarrow{\texttt{ed}}\texttt{.es}$ are not.*

In the sequel, we consider only well-formed path expressions. We say that a path expression p contains a cycle if the corresponding path in the type dependence graph contains a cycle. A forward path is a (well formed) path expression that contains no reversed field sub-expressions.

Definition 6.3.4 (Forward Closure)**.** *The* forward closure *of a path expression p, written Forward(p), is the set of all path expressions of the form $p.p'$ where p' is a forward path not containing program variables ($p.p'$ is well-formed) and p'*

does not introduce cycles other than ones already contained in p. The forward closure of a type T is the set of all forward paths starting from type T, not containing program variables.

Path closures of sets of path expressions and types are obtained by taking the union of the closures of all set members.

Example 7. *Forward*($\texttt{Edge}$) = {$\texttt{ed}$, $\texttt{src}$, $\texttt{dst}$, $\texttt{src.nd}$, $\texttt{dst.nd}$} *and*

$$\textit{Forward}(\texttt{an.src.}\overleftarrow{\texttt{dst}}) = \{\texttt{an.src.}\overleftarrow{\texttt{dst}}, \texttt{an.src.}\overleftarrow{\texttt{dst}}\texttt{.ed}\}$$

.

The forward closure of the types in the type dependence graph represent data access patterns where a sequence of method calls is used to obtain an object of type T from an object of a type T' higher in the hierarchy. For example, in lines `32-33` of Figure 6.2, a sequence of method calls is used to obtain an edge from the graph and a `Weight` from an edge. In particular, the forward closure gives us the paths needed to express *Inv*3 and *Inv*4.

However, these paths ignore the effect of methods, which create more intricate paths, such as the ones needed for *Inv*1 and *Inv*2. Those are discovered by "summarizing" method specifications, as explained next.

6.3.4.1 Discovering Paths in Method Footprints

We now explain how to find variable-to-lock paths, which represent possible paths between objects referenced by the method parameters (and

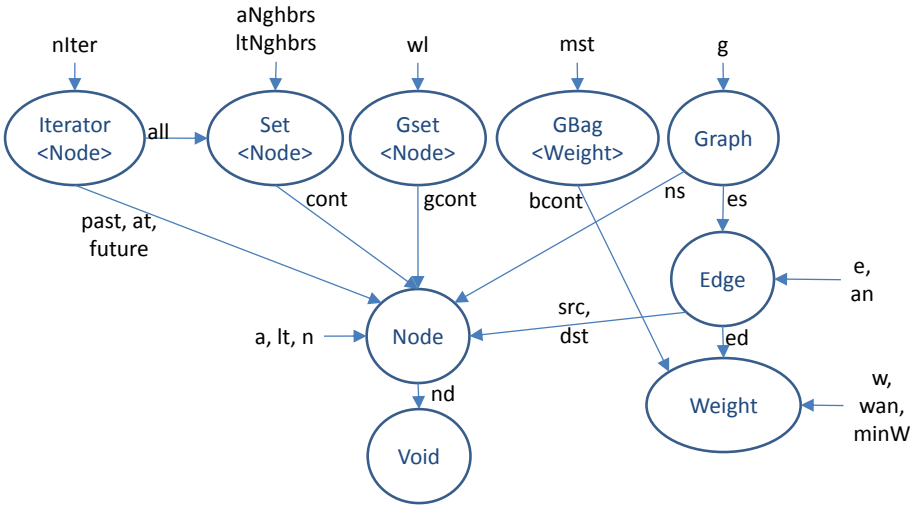


Figure 6.9: Type dependence graph for Figure 6.2.

returned value) and objects accessed by the `@locks` specification, after the `@op` specification "executes".

To find these paths, we construct a *footprint graph* for each method. Intuitively, this graph represents the set of objects accessed by the method, sometimes referred to as the "footprint" of the method. The idea of "footprint analysis" was defined by Calcagno et al. [26] to infer method preconditions and postconditions. We put this idea to use for a different purpose.

We create a footprint graph by the following steps:

Handling statements We interpret the statements in `@op` in the order they appear. For each statement, we create a graph representing every path expression on the right-hand side of an assignment. This is done by creating a new node for each position in the expression, connecting them

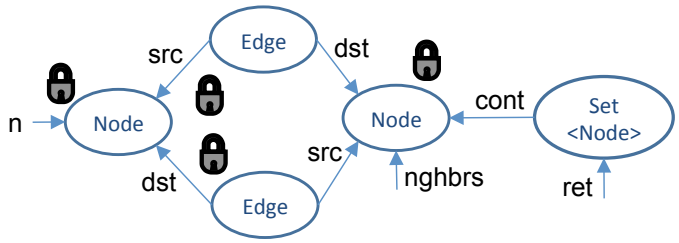


Figure 6.10: Footprint graph for `Graph.getNeighbors`. A lock is shown next to each node labeled by `locks`.

by the respective fields, and labeling nodes by the variables along the expression. If the left-hand side of the assignment is a pointer or set variable (`locks`), we use it to label the last node of each path graph. If it is a field of the type containing the method, we create a node of that type labeled by `this` and connect an edge field from that node to the last node of every path graph created for the right-hand side expression.

Creating `@locks` paths We create path graphs for all path expressions in `@locks` that do not already appear in `@op`.

Merging We merge nodes labeled by a common (pointer or set) variable.

Setting `locks` We label every node matching a path expression in `@locks` by `locks`.

Example 6.3.5. *Figure 6.10 shows the footprint graph for the* ***`getNeighbors`*** *method of* ***`Graph`****. The top node represents the outgoing edges of* ***`n`****, the lower node represents the incoming edges of* ***`n`****. Both are connected to some neighbor of* ***`n`****. The node on the right represents the returned set*

Type	Variable-to-Lock Paths
`Graph`	`es`, `es.src`, `es.dst`, `ns`, $\texttt{ns.}\overleftarrow{\texttt{src}}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}$, $\texttt{ns.}\overleftarrow{\texttt{src}}\texttt{.dst}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}\texttt{.src}$
`Node`	`Var(Node)`, $\overleftarrow{\texttt{src}}$, $\overleftarrow{\texttt{dst}}$, $\overleftarrow{\texttt{src}}\texttt{.dst}$, $\overleftarrow{\texttt{dst}}\texttt{.src}$
`Edge`	`Var(Edge)`, `src`, `dst`
`Weight`	$\overleftarrow{\texttt{ed}}$, $\overleftarrow{\texttt{ed}}\texttt{.src}$, $\overleftarrow{\texttt{ed}}\texttt{.dst}$
`Set<Node>`	`cont`, $\texttt{cont.}\overleftarrow{\texttt{src}}$, $\texttt{cont.}\overleftarrow{\texttt{dst}}$, $\texttt{cont.}\overleftarrow{\texttt{src}}\texttt{.dst}$, $\texttt{cont.}\overleftarrow{\texttt{dst}}\texttt{.src}$
`GSet`	`gcont`
`GBag<E>`	$\emptyset$

Table 6.3: Variable-to-Lock paths for the running example. `Var(T)` denotes an arbitrary variable to an object of type `T`.

containing the neighbors of ***n****. We use this graph to obtain paths expressing that* ***getNeighbors*** *has the effect of locking the immediate neighborhood of* ***n****.*

We define the function $\mathit{VarToLock} : \mathit{TName} \rightarrow 2^{\mathit{WFPath}}$ associating a set of variable-to-lock paths with each program type.

We create a set of variable-to-lock paths for every type node from all footprint graphs as follows. For each footprint graph, we take all the acyclic non-empty paths from a node labeled by a method parameter (including `this` and the return parameter `ret`) to any node labeled by `locks`. We associate these paths with the type node corresponding to the type of the parameter. We denote the set of variable-to-lock paths of type T by $\mathit{VarToLock}(T)$.

Table 6.3 shows the variable-to-lock paths that we get for the running example. These paths enable us to express *Inv*1 and *Inv*2.

We combine the sets of paths defined earlier to obtain the set of ab-

straction paths:

$$AbsPaths \stackrel{\text{def}}{=} \bigcup_{t \in TName} Forward(t) \cup Forward(\, VarToLock(t)) \ .$$

Here, expressions of the form `Var(T)` appearing in $VarToLock(t)$ are substituted by the set of paths $\{x \in Var \mid x \text{ is of type } T\}$.

6.3.5 Putting it All Together

Our overall static analysis consists of the following stages:

Preprocessing We use a lightweight purity analysis [131] to detect objects that do not require concurrency control and fields that are never used inside the parallel loop, e.g., the `initSrc` and `initDst` fields of `Weight`. The remainder of the analysis does not consider path expressions in `@locks` containing unused fields and sets the `opt` flags of read-only objects to `NONE`.

Shape Analysis We execute a forward shape analysis using hierarchy summarization abstraction and TVLA-generated abstract transformers. The fixpoint is a set of bounded shape graphs at every program location.

Finding Redundant Locks We use abstract operations in TVLA to conservatively check whether every shape graph at a program location represents stores that lock all objects defined by a `@locks` specification of a method executing at that location. If so, we set the `opt` argument of that method call to `UNDO` (if it was not already set to `NONE`).

Finding Failsafe Points We perform a backward BFS traversal over the CFG (control flow graph) to find earliest program locations where all following method calls are labeled by `NONE` or `UNDO` (meaning they do not acquire locks). These program locations are the program failsafe points, We set the optimization argument of all method calls dominated by failsafe points to `NONE`.

6.3.6 Backward Reachability Abstraction

A common abstraction idiom for shape abstraction uses *coloring*, which records a set of unary (object-)predicates with **every object** in the store. These predicates are used to partition the set of objects into equivalence classes. Examples are Canonical Abstraction [130], Boolean heaps [117], Indexed predicate abstraction [85], and generalized typestates [86].

These abstractions typically employ *backward reachability predicates* that use paths in the heap to relate objects to variables. For example, most TVLA-based analyses and analyses using Boolean heaps distinguish between disjoint data structure regions (e.g., list segments and sub-trees) by using transitive reachability from pointer variables. Indexed predicate abstraction [85] uses predicates that assert that cache clients are contained in one of two lists (`sharer_list` and `invalidate_list`). Lam et al. [86] use set containment predicates as the generalized typestate of an object.

We call such abstractions *bottom-up*, since they record properties of objects deep in the heap with respect to (shallow) root objects. These ab-

stractions achieve high precision as they express every Boolean combination of intersection and union of objects satisfying the unary predicates. However, the size of an abstracted store can be exponential in the number of predicates, which might lead to state space explosion in cases where objects satisfy many different subsets of predicates.

We define backward reachability abstraction by using the set of abstraction paths presented earlier to define backward reachability predicates. For a pointer variable `x` and an abstraction path p, we define a unary predicate expressing the fact that v is a locked object reachable from `x` by the path p:

$$\mathit{BackwardReach}[x, p](v) \stackrel{\text{def}}{=} v \in [\![\texttt{locks}]\!]^{\sigma} \cap [\![x.p]\!]^{\sigma} \quad .$$

We obtain a backward reachability abstraction $\beta[P^{BRA}]$ from the predicates shown in Table 6.4. *BRA* is strictly more precise than *HSA*. However, it can be very expensive — the number of abstract nodes in a shape graph obtained by $\beta[P^{BRA}]$ can be exponential in the number of backward-reachability predicates. State space explosion manifests when stores create overlaps between different interacting sets (set fields), which is often the case in our programs. Applying $\beta[P^{BRA}]$ to the store in Figure 6.6, will conflate all objects not locked and not referenced by a program variable. Compared to Figure 6.7, `Edge` objects `2` and `3`, for example, will remain needlessly distinguished. Situations such as iterating over the neighbors of a node, exploring multiple neighborhoods simultaneously or sharing objects between multiple collections, cause the number of useless distinctions to increase.

Predicates	Meaning
Abstraction Predicates	
$\{x(v) \mid x \in Var\}$	x references v
$\{BackwardReach[x,p](v) \mid x \in Var, p \in AbsPaths\}$	Backward-reachability predicates

Table 6.4: Predicates for backward-reachability abstraction.

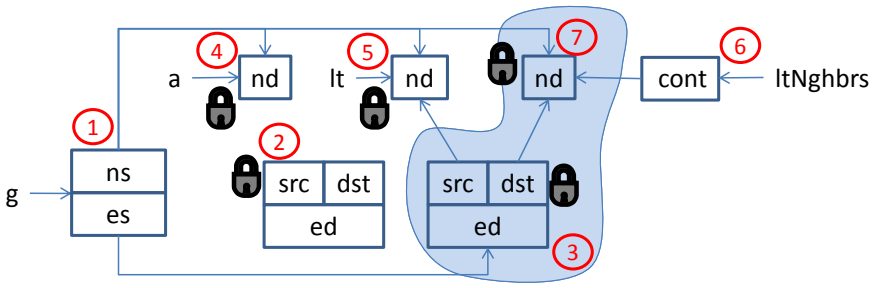


Figure 6.11: A counterexample at location L5 for the non-cautious implementation of BVK.

6.3.7 Producing Non-Cautiousness Counterexamples

When the code of a parallel loop body is not cautious, our analysis can sometimes provide a counterexample to demonstrate the violation of the cautious property at appropriate program points. To find such counterexamples, we assume the small scope conjecture [68], which says that counterexamples usually manifest in small graphs.

A graph with three nodes and two edges is sufficient to provide us with a counterexample for the case of BVK, as shown in Figure 6.11. The region of the graph where the violation happens is highlighted. This is the smallest counterexample found by our analysis, taking about 300 seconds to produce.

6.3.8 Limitations

We recognize the following limitations of our analysis.

Bounded-depth hierarchy. As discussed at the beginning of this section, we assume a class of stores where a finite-depth hierarchy property exists. This allows us to ensure a bound on the number of hierarchy summarization paths used to define our abstraction. This precludes us from handling benchmarks where data structures such as lists and trees are explicitly manipulated (and cannot be abstracted away by a `@rep` specification). Generalizing our analysis to handle recursive data structures may be done by considering abstraction paths with regular expressions over the pointer fields of the data structure.

Temporary violation of invariants. Our abstraction is geared to infer invariants of the form $\forall o.R(o) \implies p(o)$ where $R(o)$ expresses a heap region (by abstraction paths) and $p(o)$ is a property we wish to summarize for the objects in the region $R(o)$ (the is-locked property in our analysis). When the property p is temporarily violated for the objects in $R(o)$ and then restored, our analysis is not able to restore the invariant. For example, assume an invariant $\forall o.R(o) \implies p(o)$ holds at program point 1. Then a point 2 a single object in $R(o)$, referenced by a pointer variable `x`, is made to have $\neg p(o)$ and at point 3 it is removed from $R(o)$. In order to regain the invariant $\forall o.R(o) \implies p(o)$ at point 3, we may need to refine our abstraction in order to express an invariant such as $\forall o.(R(o) \wedge \neg \mathtt{x}(o)) \implies p(o)$.

6.4 Experimental Evaluation

The shape analysis described in Section 6.3 was implemented in TVLA, and used to optimize four benchmarks from the Java Lonestar suite [79]. These benchmarks were chosen because they exhibit very diverse behavior. We describe them below.

- BVK: Boruvka's MST algorithm. This benchmark adds and removes nodes and edges from a graph.
- DMR: Delaunay mesh refinement. This benchmark uses iterative refinement to produce a quality mesh. In each iteration, a neighborhood of a bad triangle, called the *cavity* of that triangle, is removed from the mesh and replaced with new triangles. DMR uses a large number of collections with intricate patterns of data sharing, so it is a "stress test" for the analysis.
- SP: Survey propagation, a heuristic SAT solver. Most iterations only update node labels, but once in a while, an iteration removes a node (corresponding to a "frozen variable" [22]) and its incident edges.
- PFP: Preflow-push maxflow algorithm [34]. This algorithm only updates labels of nodes and edges, and does not modify the graph structure.

Prog.	**IR Size**	**Graph Calls**	**Set Calls**	**Field Acc.**	**Optimal**
BVK	340	17/20	4/4	23/23	✓
DMR	1,168	26/30	30/30	164/164	✓
SP	925	32/34	16/16	123/123	✓
PFP	479	6/8	3/3	28/28	✓

Table 6.5: Program characteristics and static analysis results. x/y measures Optimized/Total.

Program	Graph Calls	Set Calls	Field Acc.	Optimal
BVK	20/3	4/0	23/0	3 ✓
DMR	30/4	30/0	164/0	4 ✓
SP	34/2	16/0	123/0	2 ✓
PFP	8/2	3/0	28/0	2 ✓

Table 6.6: Static analysis results.

6.4.1 Static Analysis Evaluation

Table 6.5 reports the results of static analysis of our benchmarks. We measure the size of benchmarks by the number of intermediate language (Jimple) instructions in the client program, excluding the code implementing the data structures accompanied by a specification. Columns 3 to 5 show the number of static optimization opportunities that our analysis enables. Galois protects application-specific objects (e.g., the cavity in DMR) using a variant of object-based STM, which can also benefit from our optimizations. Column 5 refers to those objects. In all cases, our analysis was precise enough to identify the maximum number of sites that were eligible for optimization, and it discovered the minimal set of latest failsafe points. The optimal result that we compare against was determined manually. Since our analysis is sound,

Analysis	Total SGs	Avg. # Abs. Nodes	Avg. # SGs CFG Location	Time (sec)
		BVK		
HSA	13,594	9	6.25	6
BRA	412,862	15	250	3,406
		DMR		
HSA	35,763	13	6.46	16
BRA	1,043,116	20	268	14,909
		SP		
HSA	25,421	13	6.26	12
BRA	394,765	21	158	12,446
		PFP		
HSA	17,692	10	6.96	7
BRA	71,800	17	45	972

Table 6.7: *HSA*, *BRA* performance statistics.(*SG: Shape Graph*)

we need to consider only the relatively few calls where the analysis does not suggest conflict detection or rollback logging optimizations.

6.4.1.1 Comparing Analyses: HSA vs. BRA

In Table 6.7, we compare our analysis using hierarchy summarization abstraction (*HSA*), with an analysis using backward reachability abstraction (*BRA*). The first column reports the total number of shape graphs (SG) explored by the analysis, which is a measure for the amount of work performed. We also report the average size of a shape graph, the average number of SG's per CFG location (our analysis uses roughly 1.43 CFG locations for a Jimple instruction) at the fixed point, and the running time of the analysis.

As expected, *HSA* generates a constant number of SG's at each program

location, whereas in *BRA* the number of SG's increases as the benchmarks become more complex (from 45 SG's for PFP to 268 for DMR). The benefits of *HSA* are more striking as the complexity of the benchmark increases. For PFP, *BRA* generates roughly 6 times more SG's than *HSA*, per CFG location. For DMR, in which the number of collections increases, *BRA* produces 41 times more structures. Additionally, we observe that in *BRA* we have more refined and, consequently, larger SG's. For all benchmarks the average SG size in *BRA* is roughly 1.6 times larger than in *HSA*. These facts lead to a significant state space explosion, which translates to increased work performed by *BRA* (for DMR we see a 29-fold increase in the number of generated SG's), and to increased running times. Thus, *HSA* is as precise as *BRA* but more efficient.

6.4.2 Experimental Evaluation of Optimizations

This section provides detailed performance results for each benchmark. To evaluate the performance gains obtained by different levels of sophistication of the analysis, we considered the following variants for each benchmark.

- O1: Baseline version: accesses within parallel loops to *all* objects are protected.
- O2: Iteration-private objects are not protected.
- O3: O2+ dominated shared objects are not protected.

- O4: O3+ duplicate lock acquisitions and unnecessary undo operations are eliminated.

Even in the baseline version, we do not protect object accesses made outside of parallel loops since the analysis required to enable this is trivial. At level O2, iteration-private objects are identified and accesses to them are not protected; this optimization by itself can be accomplished by a combination of flow-insensitive points-to and escape analysis. Optimization levels O3 and O4 target shared data; for these levels, a shape analysis similar to ours is necessary.

We performed our experiments using the Galois runtime system and a Sun Fire X2270 Nehalem server running Ubuntu Linux version 8.04. The system contains two quad-core 2.93 GHz Intel Xeon processors, which share 24 GB of main memory. Each core has two 32 KB L1 caches and a unified 256 KB L2 cache. Each processor has an 8 MB L3 cache that is shared among the cores. We used the Sun HotSpot 64-bit server JVM, version 1.6.0. Each variant was executed nine times in the same instance of the JVM. We drop the first two iterations to account for the overheads of JIT compilation, and report results for the run with the median running time. To reduce the non-determinism caused by garbage collection, we executed each application with the maximum possible heap size available (22 GB).

Because of the don't-care non-determinism of unordered-set iterators, different executions of the same benchmark/input combination may perform

different numbers of iterations. Since our optimizations focus on reducing the overhead of each iteration and not on controlling the total number of iterations, we focus on a performance metric called *throughput*, which is the number of committed iterations per millisecond. For completeness, we also present other measurements such as the total running time, the number of committed iterations, the abort ratio, etc. Table 6.8 shows detailed results for all benchmarks. Figure 6.12 shows how the throughput changes across different thread counts.

6.4.2.1 Boruvka's Algorithm

We do not provide results for level O2, since the number of iteration private objects is insignificant. The number of committed iterations is exactly the same across all thread counts (this is a natural property of the algorithm since each committed iteration adds one edge to the MST). The analysis is successful in reducing the number of locks per iteration, and it correctly infers that the operator implementation is cautious.

The Boruvka algorithm takes roughly 141 seconds to run if we use 1 thread and optimization level O1, and 75 seconds if we use 8 threads and optimization level O4. At optimization level O4, no undo's are logged and the number of acquired locks in each iteration is substantially reduced. However, overall speedup is limited by the high abort ratio (for example, for 8 threads, the abort ratio is between 68% and 75% for all levels of optimization). The abort ratio decreases as the optimization level increases because if the time

Th.	BVK			DMR				SP				PFP			
	O1	O3	O4	O1	O2	O3	O4	O1	O2	O3	O4	O1	O2	O3	O4
Lock Acquisitions/Iteration															
1	885	637	64	1,429	1,269	97	28	99	98	49	6	109	111	44	8
2	1,114	799	93	1,429	1,269	97	28	99	99	50	6	109	111	44	8
4	1,270	896	118	1,429	1,269	97	28	99	100	50	6	109	111	44	8
8	1,512	1,020	153	1,430	1,268	97	28	100	100	50	6	110	111	45	9
Undos/Iteration															
1	40.69	40.77	0.00	27.77	8.45	8.45	0.00	4.82	4.82	0.00	0.00	5.85	2.93	0.00	0.00
2	47.42	47.30	0.00	27.77	8.45	8.44	0.00	4.85	4.85	≈ 0.00	0.00	5.85	2.93	0.00	0.00
4	47.46	47.72	0.00	27.77	8.45	8.44	0.00	4.86	4.94	≈ 0.00	0.00	5.85	2.94	0.00	0.00
8	46.61	47.25	0.00	27.79	8.44	8.45	0.00	4.90	4.89	≈ 0.00	0.00	5.92	2.95	0.00	0.00
Committed Iterations (Millions)															
1	1.60	1.60	1.60	1.62	1.62	1.62	1.62	21.61	21.52	21.49	21.71	8.62	8.34	8.41	8.41
2	1.60	1.60	1.60	1.61	1.62	1.62	1.62	21.52	22.15	22.25	21.70	8.48	8.41	8.39	8.38
4	1.60	1.60	1.60	1.62	1.62	1.62	1.62	21.75	21.68	23.84	23.15	9.33	8.52	8.50	9.25
8	1.60	1.60	1.60	1.61	1.62	1.62	1.62	21.85	23.79	24.24	25.12	9.80	11.23	13.54	16.39
Abort Ratio %															
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	42.50	40.28	36.99	0.00	0.01	0.00	0.01	0.97	0.95	1.04	1.23	0.13	0.14	0.11	0.11
4	61.52	58.57	55.18	0.03	0.02	0.01	0.02	3.51	3.60	3.50	3.45	0.61	0.35	0.38	0.87
8	75.70	71.25	68.57	0.05	0.04	0.02	0.03	8.39	8.62	11.54	7.19	1.02	3.08	5.04	3.83
Running Time (sec)															
1	141	107	81	171	132	24	22	180	176	24	14	104	94	22	18
2	155	109	81	93	77	14	12	109	110	24	17	75	69	13	10
4	190	102	77	49	39	8	7	65	58	14	11	114	103	7.3	6.6
8	219	98	75	28	22	5	5	42	42	10	9	108	108	7.7	9.4
Throughput (Iterations/ms)															
1	11.38	15.01	19.65	9	12	68	75	120	122	893	1,533	83	89	389	463
2	10.34	14.71	19.72	17	21	118	131	198	201	934	1,274	113	122	659	798
4	8.41	15.62	20.87	33	41	212	234	333	371	1,729	2,074	82	83	1,167	1,402
8	7.31	16.38	21.40	57	73	323	347	515	573	2,404	2,868	91	104	1,762	1,739

Table 6.8: Performance metrics. BVK input is a random graph of 800,000 nodes and 5-10 neighbors per node. DMR input is a random mesh with 549,998 total triangles, 261,100 bad. SP input is a 3-SAT formula with $1,000$ variables and $4,200$ clauses. PFP input is a random graph of $262,144$ nodes and capacities in the range [0, 10000].

to execute an iteration is reduced, the iteration holds its locks for a smaller amount of time, reducing the likelihood of conflicts. This high abort ratio is intrinsic to the algorithm. The MST is built bottom-up, so towards the end of the execution, only the top few levels of the tree remain to be built and there is not much parallel work.

A Non-Cautious Boruvka Implementation. As we discussed in Section 6.2, removing the call to *getNeighbors* at `L3` results in non-cautious iterations. Our analysis successfully infers that the failsafe point along this program path moves from `L3` to `L5`. The only difference in the inferred method flags is in `L4`, where the call to `removeEdge` requires the `UNDO` flag instead of `NONE`. This example shows the utility of our analysis for optimizing programs in which the operator implementation is not cautious.

6.4.2.2 Delaunay Mesh Refinement

The number of committed iterations for this application is fairly stable across thread counts and optimization levels. Lock acquisitions drop dramatically in going from O2 to O3. The analysis deduces correctly that the operator implementation is cautious, which is why the number of undo's per iteration drops to zero at optimization level O4 (the number of undo's per iteration is stable in going from O2 to O3 because the re-triangulated cavity is constructed in private storage and then stored into the shared graph). The abort ratio is very small even for 8 threads.

The reductions in the average number of acquired locks and logged undo's per iteration are reflected directly in the running time. DMR takes 171 sec. to run if we use 1 thread and optimization level O1, and only 5 sec. if we use 8 threads and optimization level O4. This is roughly a factor of 34 improvement in the running time, of which a factor of roughly 8 comes from optimizations and a scaling factor of roughly 4 comes from increasing the number of threads. Since the number of committed iterations is fairly stable across all optimization levels and thread counts, the same improvement factors can also be seen in throughput.

6.4.2.3 Survey Propagation

The number of committed iterations is fairly stable for this benchmark. The analysis is successful in reducing the number of locks per iteration. The number of undo's per iteration is fairly small even at optimization level O1 because the graph is mutated only when a variable is frozen, which happens in very few iterations. The analysis correctly infers that the operator implementation is cautious.

The SP algorithm takes roughly 180 seconds to run if we use 1 thread and optimization level O1, and 9 seconds if we use 8 threads and optimization level O4. Most of this benefit comes from the optimizations; at optimization level O4, we observe a speedup of roughly 1.6 on 8 threads. We see a 5.5× improvement in throughput for 8 threads when the optimization level goes from O1 to O4, and by 19% from O3 to O4.

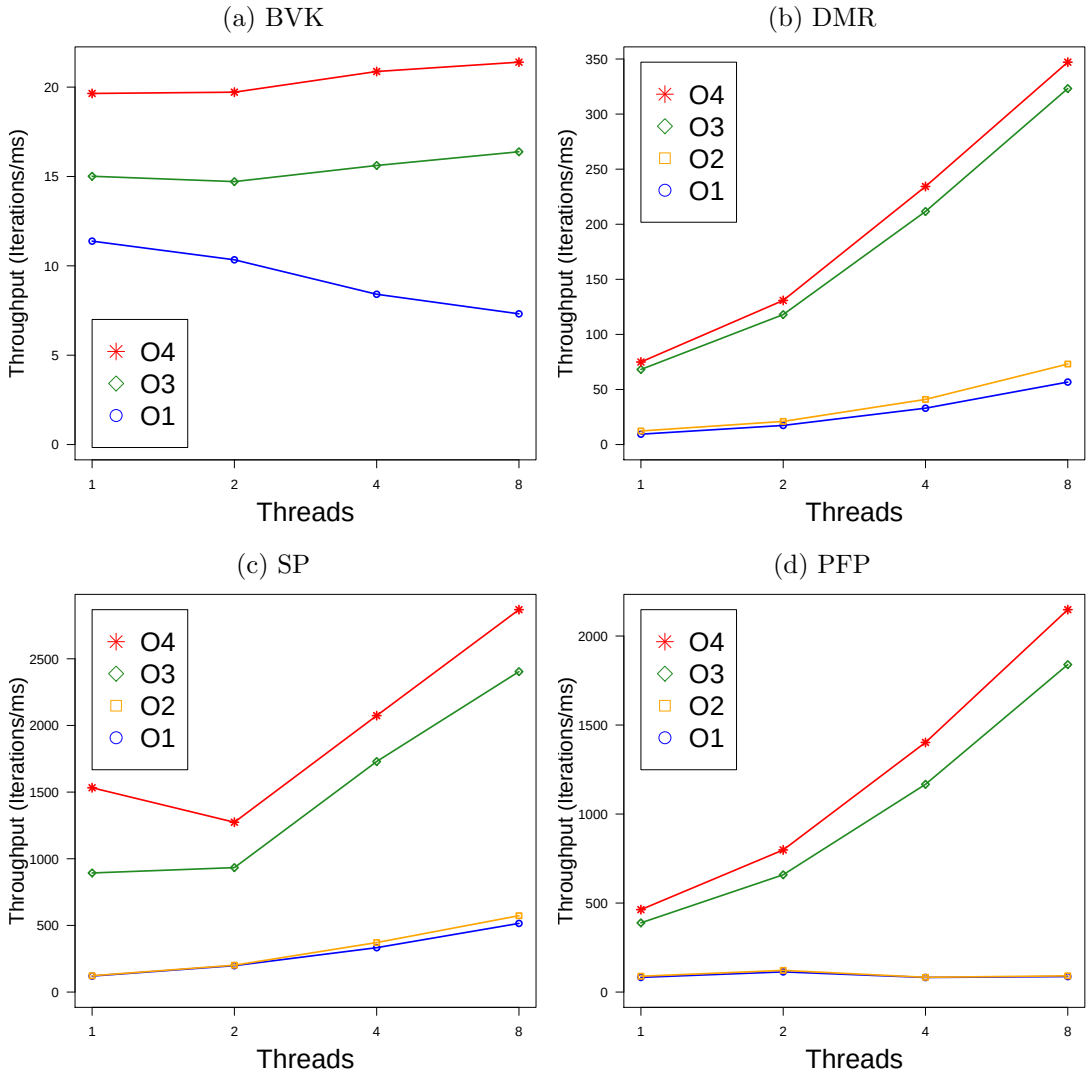


Figure 6.12: Benchmark throughputs.

6.4.2.4 Preflow-push Maximal Flow

A distinctive characteristic of PFP is its schedule sensitivity - because of don't-care non-determinism, different schedules can perform very different amounts of work. This can be seen in the 8-thread numbers: at optimization level O4, the program executes twice as many iterations on 8 threads as it does on a single thread. The number of undos per iteration is 0 for O3, since the graph structure is not mutated by the algorithm.

The preflow-push algorithm takes roughly 104 seconds to run if we use 1 thread and optimization level O1, and the best parallel time is 6.6 seconds if we use 4 threads and optimization level O4. This is a 16-fold improvement, of which roughly 6-fold improvement comes from the optimizations, and an improvement of roughly 3-fold comes from exploiting parallelism.

6.4.2.5 Summary of Results

Our analysis eliminates all costs related to rollback logging for our benchmarks, and reduces the number of lock acquisitions by a factor ranging from 10× to 50×, depending on the application and the number of threads. These improvements translate to a noticeable improvement (ranging from 2× up to 12×) in the running time, which is consistent across different thread counts, and robust against pathologies of speculation (e.g. high abort ratio).

Chapter 7

Related Work

In this chapter we discuss work that is related to the synthesis and static analysis techniques that we presented in previous chapters.

7.1 Elixir and Program Synthesis

In this section we discuss work that is related to the techniques we present in Chapter 2 and Chapter 3.

7.1.1 Program Synthesis Systems

Program synthesis is a well studied problem [98, 5, 51]. We give a brief description of some notable contributions. In deductive synthesis systems a program is generated through an iterative refinement of a high-level specification. On each step a set of well-defined proof rules are used, each of which corresponds to some programming construct. This approach was pioneered by Manna, Waldinger and others in the 1970's [98, 97, 96]. Some notable systems following this approach are KIDS [143] and Specware [142]. These systems start from a high-level, possibly non-algorithmic specification, design strategies and subsequent optimization techniques, which are guided by

an experienced user. Examples of applications synthesized by this approach include constraint solvers, garbage collectors [113], and more recently graph algorithms [107, 106]. More recently, approaches such as template-based synthesis [146] have been used to develop a variety of interesting algorithms, such as Bresenhams line drawing algorithm. In this approach, a user supplies a template or outline of the intended program structure, and the tool fills in the details.

A different way of specifying user intent is followed by inductive synthesis systems. In such systems one starts from instances/examples and generalizes to produce a program that explains all instances that meet a specification. This generic paradigm of "programming by examples" has been successfully applied to many different problem domains [52]. One other notable example of this approach is the Sketch system [145]. Sketch starts from a partial program with unspecified integer holes, whose values are difficult for the programmer to identify. The synthesizer infers the content of holes using a SAT-based combinatorial search over the space of possible sketch completions. Sketching is applicable to the class of finite programs i.e., functions that take inputs of bounded size and perform a finite computation. Alternatively, a model checker can be used to eliminate invalid candidate programs, by attempting to verify a candidate programs. Another line of work focuses on synthesis from logic specifications [67]. The user writes a logical formula and a system synthesizes a program from that. Both of the above works are also instances of the counterexample guided iterative synthesis (CEGIS) paradigm [49, 136], which

provides a very effective way of solving a synthesis constraint.

7.1.2 Synthesis for Concurrency and Parallelism

[105] generates parallel tree traversals for attribute grammar evaluation. Our work could be used synergistically to parallelize individual traversals. [11] explores SIMD loop synthesis by extracting the equivalence relation from the loop body and using it as specification to synthesize the parallel loop. Our technique can synthesize code with loops but is less ambitious in the sense that it lowers from a high to a low-level program rather than synthesizing from a pre-post specification. Another line of work that focuses on concurrency is Sketching [144] and Paraglide [151]. There, the goal is to start from a (possibly partial) sequential implementation of an algorithm and infer synchronization to create a correct concurrent implementation. Automation is used to prune out a large part of the state space of possible solutions or to verify the correctness of each solution [152]. Our plans encode correct programs by construction. Not all plans encode the tightest synchronization to optimize different aspects of the computation. [60] synthesize concurrent data-structures from relational specifications by generating a set of plans and choosing the most profitable ones.

7.1.3 Data-Structure Synthesis

Work on data-structure synthesis [141, 12, 59, 95, 60] is complementary to Elixir. Currently, Elixir compiles against a generic graph API and

uses hand-implemented graphs from the Galois library. Combining the two approaches would provide even more complete solution to the synthesis of irregular algorithms.

7.1.4 DSLs and Synthesis for High-Performance

The SPIRAL system uses recursive mathematical formulas to generate divide-and-conquer implementations of linear transforms [124]. Divide-and-conquer is used in the Pochoir compiler [149], which generates code for finite-difference computations, given a finite-difference stencil, and in the synthesis of dynamic programming algorithms [123]. This approach cannot be used for synthesizing high-performance implementations of graph algorithms since most graph algorithms cannot be expressed using mathematical identities; furthermore, the divide-and-conquer pattern is not useful because the divide step requires graph partitioning, which usually takes longer than solving the problem itself. The Tensor Contraction Engine [13] takes integrals used in quantum chemistry applications as input, and uses loop transformations like loop fusion and fission to generate parallel code.

Green-Marl [66] is an orchestration language for graph analysis. Basic routines like BFS and DFS are assumed to be primitives written by expert programmers, and the language permits the composition of such traversals. Elixir gives programmers a finer level of control and provides a richer set of scheduling policies; in fact, BFS is one of the applications presented in Chapter 4 for which Elixir can automatically generate multiple parallel variants,

competitive with handwritten third-party code. There is also a greater degree of automation in Elixir since the system can explore large numbers of scheduling policies and synchronization methods automatically. Green-Marl provides support for nested parallelism, which Elixir currently does not support.

In [108] Nguyen *et al.* describes a synthesis procedure for building high performance worklists. Elixir uses their worklists for dynamic scheduling, and adds static scheduling and synthesis from a high-level specification of operators.

7.1.5 Compiler Techniques

Several papers propose approaches to tackle the phase-ordering problem by using Lightweight Modular Staging and rewrite rules to optimize programs [128, 150]. Equality preserving rewrite rules used by Tate et al. cannot support synchronization synthesis with global constraints (e.g. cautiousness). ILP-based techniques have been used to generate embedded processor code for basic blocks and for software pipelining [42], and for scheduling for spatial architectures [110].

7.1.6 Superoptimization

Superoptimizers find optimal straight-line machine code sequences. [100] exhaustively enumerates programs of increasing length or cost, which limits applicability to small sequences. [134] use MCMC sampling as a search technique and achieves better scaling. Denali [73, 74] uses SAT-based constraints

and equality-preserving rewritings to find an optimal loop-free sequence for a guarded multi-assignment input program. Compared to our solution, none of these approaches handle more general programs involving conditionals and loops. Denali does not handle dependences and does not consider instrumentation transformations. [54] synthesize loop-free programs of component compositions using SMT-based reasoning. [72] use planning to generate straight-line code of library API calls, and uses programmer-compiler interaction to prune undesirable compositions. Our work handles more general programs and instrumentation transformations, but this work also focuses on information flow between planner generated and host application code.

7.1.7 Term and Graph Rewriting.

Term and graph rewriting [129] are well-established research areas. Systems such as GrGen [46], PROGRES [135] and Graph Programming (GP) [116] are using graph rewriting techniques for problem solving. The goals however are different than ours, since in that setting the goal is to find a schedule of actions that leads to a correct solution. If a schedule does not lead to a solution, it fails and techniques such as backtracking are employed to continue the search. In our case, every schedule is a solution and we are interested in schedules that generate efficient solutions. Additionally, none of these systems is focused on concurrency and the optimization of concurrency overheads.

Graph rewriting systems try to perform efficient incremental graph pattern matching using techniques such as Rete networks [24, 47]. In a simi-

lar spirit, systems that are based on dataflow constraints are trying to efficiently perform incremental computations using runtime techniques [37]. Unlike Elixir, none of these approaches focuses on parallel execution. In addition, Elixir tries to synthesize efficient incremental computations using compile-time techniques to infer high quality deltas.

7.1.8 Finite-differencing.

Finite differencing [111] has been used to automatically derive efficient data structures and algorithms from high level specifications [25, 91]. This work is not focused on parallelism. Differencing can be used to come up with incremental versions of fixpoint computations [25]. Techniques based on differencing rely on a set of rules, which are most often supplied manually, to incrementally compute complicated expressions. Elixir automatically infers a sound set of rules for our problem domain, tailored for a given program, using an SMT solver.

7.2 Static Analysis and Speculation Optimization

In this section we discuss related work to the static analysis material presented in Chapter 6.

7.2.1 Shape Analysis of Complex Heaps.

Shape analysis is a well-established field with many important contributions [130, 89, 127, 117, 14]. Prior work on shape analysis has focused

mostly on analyzing data structure implementations to infer heap structure. In contrast, we use data structure specifications to abstract away data structure representations, and we focus on unstructured graphs.

The Jahob system [83] verifies that a data structure implementation meets its specification, and it uses the abstract state to simplify the verification of data structure clients. Our analysis assumes that a given specification is correct. Checking that the implementation and specification of the method semantics match and that the `@locks` specification ensures that only commutating methods can execute concurrently is an interesting challenge.

Maron et al. [99] use specialized predicates to model sharing patterns between objects stored in data structures, and use this information to statically parallelize benchmarks from the JOlden suite and SPECjvm98 benchmarks. Our benchmarks operate on unstructured graphs and are not amenable to static parallelization. We exploit the fact that our execution model is speculative to avoid tracking correlations between different data structures, which increases the cost of the analysis considerably.

7.2.2 Optimizing Speculative Parallelism.

In the Galois system, the optimizations described here are performed manually [102]. Our shape analysis automates these optimizations, reducing the burden on the programmer and ensuring correctness of optimized code. Failsafe points extend the notion of cautious operators. Our running example shows that non-cautious code too can be optimized by turning conflict detec-

tion and rollback logging off for a subset of the calls, obtaining performance improvement similar to the cautious version. Additionally, in [102] the system optimizes locking only after the failsafe point in contrast to our analysis, which optimizes locking regardless of whether an operator is cautious.

Prabhu et al. [118] use value speculation to probabilistically reduce the critical path length in ordered algorithms. Their static analysis focuses mainly on array programs. Value speculation is orthogonal to our approach, and the benchmarks examined in our case-studies do not benefit from value speculation. Furthermore, our heap abstractions are very different because we need to handle complex ADTs such as unstructured graphs.

7.2.3 Compiler Optimizations for Transactional Memories.

Harris et al. [56], Adl-Tabatabai et al. [3], and Dragojevic et al. [38] use compiler optimizations to reduce the overheads of transactional memory. They also handle immutable, and transaction local objects. Additionally, they describe extending traditional compiler optimizations such as common subexpression elimination (CSE) to reduce the overheads of logging. Although CSE helps to reduce repeated logging for a single object, its effectiveness for our benchmarks is limited by the extensive use of collections. Their approaches cannot capture global properties such as failsafe points. Other optimizations they propose are complementary to ours.

7.2.4 Lock Inference For Atomic Sections

McCloskey et al. [101], Hicks et al. [64], and Cherem et al. [29] describe analyses that infer locks for atomic sections. These techniques are overly conservative for our benchmarks since they would always infer that an iteration might touch the whole graph.

Chapter 8

Conclusions and Future Work

8.1 Concluding Remarks

Parallelism is ubiquitous today. Limitations in improving single-core processor performance have led to the emergence of a host of multicore and heterogeneous parallel architectures. At the same time, the emergence of new problem domains where algorithms work on irregular sparse graphs forces a much wider audience of programmers to deal with parallelism. Since the best solution for such irregular algorithms is usually input dependent, programmers must consider multiple candidates in order to find the one that works best for their setting. It is therefore imperative to provide programming abstractions and tools that (a) enable non-experts to deal effectively with the intricacies of parallelism, such as data-races and deadlocks, and (b) allow programmers to easily experiment with many algorithm variants.

This dissertation has presented the design of the Elixir system, the first solution that addresses both of the above-mentioned issues for the very complex domain of irregular problems. Elixir enables high productivity by allowing programmers to express parallelism implicitly and to generate automatically many parallel program variants to find the best performing one. Elixir also

achieves competitive performance by using sophisticated compiler analyses to transform the input program to an efficient parallel implementation. The key novelties of Elixir include:

- A novel specification language that separates the algorithm logic (operators) from the specification of how to schedule this logic to achieve efficient implementations. Elixir is based on a very refined view of the schedule that for the first time allows to express very interesting algorithm variants as different operator schedules. This specification language can also act as a conceptual tool to understand parallelism in existing algorithm implementations and design new parallel solutions by adapting either the operators or the schedule.

- Automated reasoning for synthesizing efficient parallel incremental computations. Elixir utilizes theorem proving techniques to infer what the effects of each operator are, and uses this information to knit together the operators and schedule in order to produce work-efficient parallel algorithms.

- Integrated compilation via automated planning. Elixir casts the problem of synthesizing parallel code as an automated planning task. It encodes different transformations required to generate efficient code as different planning problems and then composes them into a single planning problem. Solving this resulting problem corresponds, effectively, to simultaneously performing the individual transformations. This methodology

constitutes the first integrated compilation approach for the domain of irregular graph problems. Additionally, since this approach is parametric on the description of the problem domains and transformations, it can be applied to different problem domains.

In the spirit of increasing programmer productivity, this dissertation has also introduced static analysis techniques that improve the performance of irregular codes expressed in contemporary programming abstraction such as transactional memories and the Galois system. We have presented interesting optimizations for reducing the overheads of speculative parallelization and show how to employ shape analysis to reason about the behavior of irregular programs at compile time and optimize their runtime execution in a manner completely transparent to the user.

8.2 Future Directions

The Elixir system, as presented in this dissertation, is a firm, first step towards simplifying parallel programming for multicore processors, particularly for domains like big data machine learning. However, parallel platforms continue to grow in complexity; in particular, we now have cloud computing and heterogeneous platforms that include multicores, GPU's and FPGA's. The current work on Elixir opens up several promising avenues for future work.

Support for machine-learning algorithms on big data. Machine-learning is an important emerging problem domain. Given the enormous size of data

processed by machine-learning algorithms, we need parallel processing to achieve efficiency. One major obstacle for the adoption of parallelism for such problems comes from the fact that data-scientists usually do not have any experience with parallel programming. In our opinion, the most promising way to help this group of programmers is to develop domain-specific languages (DSLs) that allow programmers to implicitly express parallelism. Many machine-learning problems admit a dual interpretation as graph problems and as sparse matrix problems. Therefore, one interesting avenue for future work is to design a DSL that includes the graph abstractions of Elixir and new matrix-related abstractions in order to express such problems naturally.

Another issue is the variety of runtime frameworks these algorithms can be parallelized in. The choices include shared-memory frameworks such as Galois, traditional MPI-based distributed systems, NoSQL systems and graph-databases, among others. It is therefore important to develop compilation techniques from the above DSLs to multiple runtimes so that users can write a single program that runs over multiple platforms transparently.

Synthesis for heterogeneous architectures. A complementary approach to traditional multi-core parallelism for achieving performance is through the use of heterogeneous architectures. Currently, Elixir targets multicore processors. A future direction is to investigate synthesis techniques for more exotic architectures, such as GPUs and the Intel MIC. Such architectures favor a SIMD type of parallelism and may require different scheduling policies

than the more asynchronous multicores. An interesting challenge would be to identify the kinds of algorithmic schedules that perform well on such architectures and appropriately enrich the Elixir language and system to support them. Another opportunity is to exploit the synthesis techniques of Elixir to build programs that take advantage of the more exotic ISA of these architectures. One could also use the Elixir methodology as a starting point for exploring synthesis techniques for hardware software co-design. Architectures like FPGAs are becoming more prevalent in their use to accelerate irregular workloads in the cloud [125]. We believe that the right combination of DSLs and compilation/synthesis techniques will increase the applicability of these technologies.

Auto-tuning. Elixir enables the automatic exploration of an enormous and interesting program space for irregular problems. However, given the size of this space, the input sensitivity of irregular problems and the complexity of modern architectures, neither analytical modeling nor exhaustive search are effective solutions for enabling users to quickly identify the best performing programs. To address this issue, one could explore the use of smart search strategies to reduce the auto-tuning time and make the system usable at a larger scale. The research challenge is to identify what kind of algorithmic insights can be used to create an effective auto-tuning process for irregular problems. We believe that a plausible approach that combine high-level features such as characteristics of the of input graph (diameter, node degrees etc.)

and Elixir algorithm schedules, with low-level features such as efficient implementation patterns and synchronization policies used by the planning-based synthesizer.

Synthesis of incremental algorithms. Elixir can also serve as the basis for a framework that synthesizes incremental algorithms for irregular graph problems. Incremental algorithms compute an initial result for a specific input and then update the result smartly based on input changes, without recomputing the answer from scratch. Such a system would be particularly useful in today's big-data world, since one can think of many scenarios where the input to an irregular graph problem dynamically keeps changing – think of the constant stream of updates in the structure of social network graphs, for example. The current automated reasoning in Elixir for computing the operator deltas can serve as a starting point for such a system.

Synthesis of lock-free algorithms. The baseline parallel execution mode for irregular algorithms requires that operators execute atomically. Elixir and other frameworks achieve this by acquiring locks and implementing some variation of speculative execution. However, we can often improve performance by relaxing the atomic execution requirement and using lock-free synchronization. For example, the hand-written breadth-first-search algorithms that we compared Elixir solutions against in Chapter 4 can deliver improved performance by employing such intricate synchronization schemes. These implementations

require expert knowledge and are currently outside the reach of parallel processing frameworks. Automating their construction is an interesting research problem. A solution would require the interaction of programmers with a synthesis system. The programmer states properties that ensure algorithm correctness and the system tries to prove that these properties are preserved under the relaxed synchronization scheme.

Synthesis for education scenarios. Elixir provides a high-level language for expressing different algorithm variants for an irregular problem. It would be interesting to explore the design of synthesis techniques for education scenarios [53] based on the Elixir language. For example, an interesting experiment would be to design a system where students express solutions in Elixir and automatically receive feedback both about correctness and performance of their algorithms.

Appendices

Appendix A

Shape Analysis Data-Structure Specifications Semantics

We now define the sequential semantics of method calls.

A.1 Semantics of Expressions and Statements

Let $\sigma = (S^\sigma, H^\sigma)$. We define the meaning of expressions relative to σ. Let *rangef* denote the range of a function f. The notation $f[x \mapsto y]$ stands for the function $f' = \lambda v \; . \; \begin{cases} y, & v = x; \\ f(v), & \text{else.} \end{cases}$

The meaning of path expressions is already defined in Section 6.3.

$$
\begin{aligned}
[\![\text{exp}_1 + \text{exp}_2]\!] &\stackrel{\text{def}}{=} [\![\text{exp}_1]\!] \cup [\![\text{exp}_2]\!] \\
[\![\text{exp}_1 - \text{exp}_2]\!] &\stackrel{\text{def}}{=} [\![\text{exp}_1]\!] \setminus [\![\text{exp}_2]\!] \\
[\![\texttt{choose}(\text{exp})]\!] &\stackrel{\text{def}}{=} o \text{ s.t. } o \in [\![\text{exp}]\!] \\
[\![\text{exp}_1 \,\texttt{in}\, \text{exp}_2]\!] &\stackrel{\text{def}}{=} \begin{cases} \mathsf{True}, & [\![\text{exp}_1]\!] \subseteq [\![\text{exp}_2]\!]; \\ \mathsf{False}, & \text{else.} \end{cases} \\
[\![\text{exp}_1 \,\texttt{notIn}\, \text{exp}_2]\!] &\stackrel{\text{def}}{=} \begin{cases} \mathsf{True}, & [\![\text{exp}_1]\!] \not\subseteq [\![\text{exp}_2]\!]; \\ \mathsf{False}, & \text{else.} \end{cases} \\
[\![\texttt{isEmpty}(\text{exp})]\!] &\stackrel{\text{def}}{=} \begin{cases} \mathsf{True}, & [\![\text{exp}]\!] = \emptyset; \\ \mathsf{False}, & \text{else.} \end{cases} \\
[\![\texttt{new T}(f_1 = v_1, \ldots, f_k = v_k)]\!] &\stackrel{\text{def}}{=} \\
& o \text{ s.t. } o \notin \mathit{rangeS} \cup \mathit{rangeH}(f)(v) \\
& \text{for every field } f \text{ and } v \in T_o, v \neq o \\
& \text{and } H(f_i)(o) = S(v_i) \text{ for every } i = 1, \ldots, k \qquad .
\end{aligned}
$$

The semantics of statements of the form `x = exp` and `x.f = exp` is

given, respectively, by

$$[\![\texttt{x = exp}]\!] = (S^\sigma[x \mapsto [\![\texttt{exp}]\!]], H^\sigma)$$

and

$$[\![\texttt{x.f = exp}]\!] = (S^\sigma, H^\sigma(f)[S^\sigma(x) \mapsto [\![\texttt{exp}]\!]]) \ .$$

The semantics of a sequence of statements $st_1; \ldots; st_k$ is given by composition:

$$[\![st_1; \ldots; st_k]\!] \stackrel{\text{def}}{=} [\![st_k]\!] \circ \ \ldots \ \circ [\![st_1]\!] \ .$$

A.2 Semantics of `@op` and `@locks` Specifications

Let a method specification m be `@locks`$L_1, \ldots, L_k$ `@op` $O_1, \ldots, O_n$. The meaning of m is the meaning of the sequence of statements

```
GaloisRuntime.locks += L_1;
...
GaloisRuntime.locks += L_K;
O_1; ...; O_n .
```

Appendix B

Betweenness Centrality Proofs of Correctness

In this section we give proofs of correctness of the operators for the first phase, presented in Figure 5.3, and of the operator for the second phase, presented in Figure 5.5a.

B.1 Correctness of Forward Phase

We prove that the first phase terminates, and that upon termination all node attributes have correct values. Correctness follows from the following two theorems.

Theorem 2. *(Termination) Any well-formed history H of events $op(uv), op \in \{SP, FU, US, CN\}$ of the operators in Figure 5.3 to a graph $G = (V, E)$ has finite length.*

Theorem 3. *At the fixpoint, the following facts hold for an arbitrary node v:*

(a) *$l(v)$ is equal to the length of the shortest path to v from Root.* ***(b)*** *u is the predecessor of v in a shortest path to v from Root $\iff$ $u \in preds(v)$ and $v \in succs(u)$.* ***(c)*** *$\sigma(v)$ is the number of shortest paths from Root to v.*

We consider the most general setting where operators are allowed to

execute in any order. The only requirement is each time an operator is enabled on an edge, its execution cannot be postponed indefinitely. The computation is modeled by a *history*, which is a sequence of operator applications, each of which is considered to be an instantaneous event that changes the state of the graph. We denote an operator application on an edge (u, v) by $op(uv), op \in \{SP, FU, US, CN\}$. To capture meaningful computations, we restrict attention to *well-formed histories* which are histories that satisfy the following condition.

Definition 1. *A sequence of operator applications is said to be a* well-formed history *if every* $op(uv), op \in \{FU, US, CN\}$ *where* $l(u) = l$*, is preceded by an* $SP(wu)$ *that sets* $l(u) = l$ *and there is no other* $SP'(nu)$ *between* $SP(wu)$ *and* $op(uv)$ *that sets* $l(u) = l'$*, where* $l' \neq l$*.*

B.1.1 Termination

The proof of termination relies on the following lemma, which asserts that the level of *Root* is always 0, and the levels of all other nodes are positive integers. It can proven by induction on the length of the sequence of operator applications.

Lemma 3. $l(Root) = 0 \quad \wedge \forall u \in (\boldsymbol{nodes}(\setminus, \{)Root\}) \colon l(u) > 0$

We are now ready to prove Theorem 2.

Proof. To prove that the computation terminates, we argue that each individual operator can appear only a finite number of times in a history. We first

argue that there can only be a finite number of SP applications in a history. This is because (i) node levels are modified only by the SP operator, (ii) each application of this operator strictly lowers the level of some node, and (iii) node levels must be non-negative (from Lemma 3). Therefore, there can only be a finite number of applications of the SP operator in a history, as in the example of Figure 5.4.

$$\cdots, SP(uv), \overbrace{\cdots, FU(vm), \underbrace{\cdots, US(uv), \cdots}_{\text{h''}}, CN(vq), \cdots}^{\text{h}}, SP(vw), \cdots$$

Figure B.1: SP applications partition the execution history in windows.

We now argue that in the subsequence of the history between two successive applications of the SP operator, such as the subsequence h in Figure B.1, there can only be a finite number of FU, US, and CN applications (for completeness, we should also consider the subsequence of the history before the first and after the last application of SP, but the analysis is similar). Let us call such a subsequence a *window*; the operator applications in a window do not change the levels of nodes. In such a window, there can be at most one application of the CN operator on an edge. This is because (i) the CN operator can be applied only to edges (u, v) for which $l(u) \geq l(v)$, and (ii) once this operator has been applied to an edge (u, v) and $l(uv)$ is set to ∞, neither this operator nor any other operator is applicable to this edge. A similar argument shows that there can be at most one application of the FU operator on a given edge, within a window. Therefore, there can only be a finite number of FU

and *CN* applications in a history.

To complete the proof, we must show that there can be only a finite number of applications of the *US* operator between two successive applications of $op(uv), op \in \{SP, FU, CN\}$, such as the subsequence h'' in Figure B.1. We observe that a *US* operator can be applied only to an edge (u, v) for which $l(v) = l(u)+1$. If we consider the sub-graph consisting only of these edges, we see that this sub-graph must be a DAG; each application of the *US* operator propagates information up one edge of this DAG. Each path starting at the *Root* of the DAG has finite length; assume that L is the length of the longest path. Then, a safe upper bound on the number of *US* applications between any two applications of the other operators is $O(E_D^{|L|+1})$, where E_D is the number of edges of the DAG.

□

B.1.2 Correctness at the Fixpoint

We now prove Theorem 3, which states that at the fixpoint we compute the correct values of node attributes.

B.1.2.1 Correctness of Node Levels

We now prove Th. 3(a), which asserts that at the fixpoint, the level of a node is equal to the shortest distance from *Root* to that node.

Proof. Consider the following partitioning P of the graph nodes. Partition

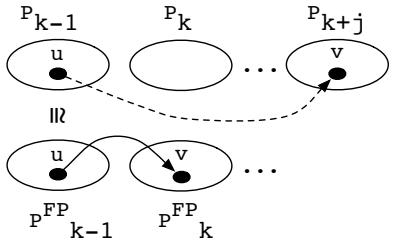


Figure B.2: v cannot belong to P_{k+j}, this implies existence of the dotted edge, which is impossible.

P_0 contains just the *Root* node. Partition P_1 contains all the nodes directly connected to the *Root*. Similarly, partition P_i contains all the nodes directly connected to nodes in P_{i-1}; nodes in P_i are directly connected to nodes in P_{i-1} but not to nodes in $P_0, \ldots, P_{i-2}$. Finally, partition P_∞ contains all nodes that are not reachable from the *Root*. Hence, partitioning P places all nodes u with minimum distance from the *Root* , $l_{min}(u) = i$ in partition P_i:

$$P = \{P_i \colon \forall u \in P_i \,.\, l_{min}(u) = i\}$$

At the fixpoint, let the distances of nodes be $\{l^{FP}(v_1), \ldots, l^{FP}(v_{|V|})\}$. This final solution induces another partitioning P^{FP} on the nodes, such that:

$$P^{FP} = \{P_i^{FP} \colon \forall u, v \in P_i^{FP} \,.\, l^{FP}(u) = l^{FP}(v)\}$$

Assume that P and P^{FP} are different. Then, there must be at least one node that belongs to different partitions in P and P^{FP}. By Lem. 3, $l^{FP}(\mathit{Root}) = 0$, hence P_0 and P_0^{FP} are similar. Therefore, assume that $P_0 = P_0^{FP}, \ldots, P_{k-1} = P_{k-1}^{FP}$ and P_k and P_k^{FP} are the first partitions that differ by, at least, node v. We consider two cases:

$v \in P_k \wedge v \notin P_k^{FP}$**:** Then, $v \in P_{k+j}^{FP}, j \geq 1$. Since $v \in P_k$, there is at least one node $u \in P_{k-1}$ (also in P_{k-1}^{FP}) that is directly connected to v, such that $l^{FP}(v) - l^{FP}(u) \geq 2$. Hence, the SP operator can fire once more, a contradiction since we are at the fixpoint.

$v_k \notin P_k \wedge v_k \in P_k^{FP}$**:** Then, $v \in P_{k+j}, j \geq 1$. $v \in P_k^{FP}$ implies that during the algorithm execution, an $SP(uv)$ was applied to an edge (u, v) between some node $u \in P_{k-1}^{FP}$ and v, which added v to P_k^{FP}. Therefore, (u, v) directly connects node $u \in P_{k-1}$ (since $P_{k-1} = P_{k-1}^{FP}$) and node $v \in P_{k+j}, j \geq 1$, a contradiction since nodes in P_{k+j} are only directly connected to nodes in P_{k+j-1}. This case is depicted in Figure B.2.

We have shown that $P^{FP} = P$, hence at the fixpoint the algorithm induced partitioning P^{FP} places each graph node at its minimum distance from the *Root*. □

B.1.2.2 Correctness of *succs* and *preds* Lists

We now prove Th. 3(b), which asserts that at the fixpoint, u is the predecessor of v in a shortest path to v from *Root* iff $u \in preds(v)$ and $v \in succs(u)$. We initially state the following lemma.

Lemma 4. *Let (u, v) be an edge of the graph. Once $l(u), l(v)$ settle on the shortest path distances, $FU(uv)$ is guaranteed to execute before the fixpoint.*

Proof. Consider a history h that describes the execution of the algorithm up to the fixpoint. We examine h and identify the actions α: $SP(w_1u)$, β: $SP(w_2, v)$

that set u, v to their shortest path distances $l(u) = k$, $l(v) = k + 1$, where $k < \infty$. Without loss of generality, assume that α precedes β. Consider an action γ: $FU(uv)$, in which $l(u) = k$ and $l(v) = k + 1$. We want to argue that γ executes after β, before we reach the fixpoint. Assume it does not. After β the relation $l(v) = l(u) + 1$ holds until the fixpoint. Then, $g_{FU}(uv)$ is not enabled because $l(uv) = k = l(u)$, after the execution of β. Only the CN, FU operators change $l(uv)$. A $CN(uv)$ would set $l(uv) = \infty > k$. Hence, $l(uv) = k$ because of some δ: $FU(uv)$ that was executed before β. But this is impossible, because for δ to be enabled we should have $l(v) = k + 1$ which does not hold before β. Therefore, γ is enabled after β and will be executed in order to reach the fixpoint. □

Proof of Th. 3(b). The proof is based on an examination of the history h.

(⇒) Assume that there is a shortest path P from *Root* to v and that $l(v) = l$. Only SP changes the level of a node, hence in h there will be an α: $SP(wv)$ that sets $l(v) = l$ (depicted in Figure B.3). Only the SP, FU, CN operators change the predecessor and successor attributes, so we consider only applications of these operators after α. We consider an arbitrary u_i that is a predecessor of v on a shortest path from *Root*. After α, by Lem. 4, $g_{FU}(u_iv)$ is enabled and a β: $FU(u_iv)$ will be executed. β establishes that $u_i \in preds(v)$ and $v \in succs(u_i)$. In the application of β, $l(v) = l$ and $l(u_i) = l - 1$, that is, u_i and v are at their minimum distance from the *Root*. Therefore, after β no $SP(wu_i), SP(wv), CN(u_iv)$ is applied since their guards are disabled. Hence, both $u_i \in preds(v)$ and $v \in succs(u_i)$ are preserved until we reach the fixpoint.

($\Leftarrow$) Assume that at the fixpoint $u \in \mathit{preds}(v)$ and $v \in \mathit{succs}(u)$. Then, there must be an action $FU(uv)$ that establishes this relationship between u and v. We examine h backwards and identify the *last* application $\alpha\colon FU(uv)$ that does this; when this operator executes, let $l(u) = k$ and $l(v) = k + 1$. After α, there can be no other $\beta\colon SP(uv)$, since that would enable $g_{FU}(uv)$ again and lead to one more application $\alpha'\colon FU(uv)$, which contradicts that assumption α is the last such action. Therefore, at the fixpoint the nodes u, v connected through the edge (u, v) have $l(u) = k$, $l(v) = k + 1$.

An induction on k establishes that there must be a shortest path from *Root* to v on which (u, v) is the last edge.

- If $k = 0$, then, by Lem. 3, $u \equiv \mathit{Root}$. Hence, the edge (u, v) is a path of length one from the *Root* to v.
- Otherwise, by the inductive hypothesis, there is a shortest path from the *Root* to every node up to distance k. Hence, there is a path P_1 from *Root* to u. Then, there is a path $P_2 = P_1 \cdot (u, v)$ from the *Root* to v through u of length $k + 1$, with minimum length.

□

$$\cdots, \underbrace{SP(wv)}_{\alpha}, \cdots, \underbrace{FU(u_i v)}_{\beta}, \cdots$$

Figure B.3: A possible action sequence during the execution of history h.

B.1.2.3 Correctness of the Path-Count σ

Finally, we prove Th. 3(c), which asserts that at the fixpoint $\sigma(v)$ is the number of shortest paths from *Root* to v. We first prove the following lemma:

Lemma 5. *At the fixpoint,* $\sigma(Root) = 1$ *and for all* $v \neq Root$, $\sigma(v)$ *is the sum of all* $\sigma(u)$, *where* $u \in preds(v)$.

Proof. Consider a history h that describes the execution of the algorithm up to the fixpoint. We first consider the *Root* node. By Lem. 3, $l(Root) = 0$ and $\forall v \neq Root$: $l(v) > 0$. Hence, no operator $op(uv), v \equiv Root, op \in \{SP, FU, US, CN\}$ is enabled and $\sigma(Root) = 1$, its initial value.

We now consider an arbitrary node $v \neq Root$ that has $l(v) = k > 0$ at the fixpoint. We examine h and identify the action α: $SP(uv)$ that sets $l(v) = k$. In general, a node v at level k will have a number of incoming neighbors at level $k-1$. Assume, without loss of generality, that v has n such neighbors $u_1, u_2, \ldots, u_n$ and that α: $SP(u_1v)$ set $l(v) = k$.

Action α sets $\sigma(v) = 0$. The only operators that update the value of $\sigma(v)$ after this are $FU(u_iv)$, $US(u_iv)$. After α no other α': $SP(u_iv)$ exists. Hence, by Th. 3(a), at the fixpoint we will have a number of shortest paths P_v from *Root* to v, each through some u_i. By Th. 3(b), $u_i \in preds(v), i \in [1, n]$ at the fixpoint.

After α, by Lem. 4, $g_{FU}(u_iv)$ will be enabled for all $i \in [i, n]$ and $\sigma(v)$ will be updated from each $\sigma(u_i)$ *once* through a β_i: $FU(u_iv)$. The path-count

of each u_i may be updated incrementally though, so $\sigma(v)$ will require more updates. If a $US(wu_i)$ is executed before β_i then β_i propagates this update to $\sigma(v)$. Now, we argue that each $US(wu_i)$ that happens after β_i is followed by an $US(u_iv)$. Assume it does not. Then, after $US(wu_i)$ we have $\sigma(u_i) = \mu$ and $\sigma(u_iv) = \lambda$, where $\lambda < \mu$. This means that we can extend history h to $h' = h \cdot US(u_iv)$, where the state at the end of h' is different than the state at the end of h. This is because $US(u_iv)$ will set $\sigma'(v) = \sigma(v) + \mu - \lambda > \sigma(v)$. Hence, at the end of h we have not reached the fixpoint, a contradiction. Hence, we have shown that for every $u_i \in preds(v)$, all updates to $\sigma(u_i)$ are propagated to $\sigma(v)$, which guarantees that at the fixpoint $\sigma(v)$ is the sum of all $\sigma(u_i)$. □

Proof of Th. 3(c). We now argue by induction on the length of the shortest path from the *Root* to v that for all v, $\sigma(v)$ is the number of shortest paths from *Root* to v.

Induction Basis: $v \equiv Root$. Then, by Lem. 5, $\sigma(v) = 1$, which is the correct number of paths from *Root* to itself.

Inductive Step: At the fixpoint, consider a node v with shortest path distance from the *Root* equal to k. By Lem. 5, $\sigma(v)$ is equal to the sum of $\sigma(u)$ where $u \in preds(v)$. By Th. 3(b) each such u is a predecessor of v along a shortest path, hence its shortest path distance is $k - 1$. Then, by the inductive hypothesis, $\sigma(u)$ equals to the number of shortest paths

from *Root* to u. Hence, $\sigma(v)$ is indeed equal to the number of shortest paths from the *Root* to v.

□

B.2 Correctness of the Backward Pass

We now present proofs for the correctness of the second phase. We consider the simple version of the operator presented in Figure 5.5a.

This operator, as part of its execution on an edge (u, v), modifies $preds(v)$ and $succs(u)$ by removing u and v from the respective collection. For the purpose of the proof we consider $preds_{ex}(v)$ and $succs_{ex}(v)$ for each node v, which contain the elements removed from $preds(v)$ and $succs(v)$, respectively, by operator applications involving v. We will also use the suffix *init* to denote the values of node attributes at the beginning of the backward pass. For example, $preds_{init}(v)$ is the initial value of $preds(v)$.

B.2.1 Termination

Proving termination is straightforward. Initially there is a fixed number of predecessor edges between the nodes comprising the shortest-path DAG. Each operator application depends on finding one such predecessor edge (u, v) and removes it from the graph. Therefore, the number of predecessor edges decreases monotonically and eventually becomes zero. At that point no more applications are enabled and the algorithm terminates.

B.2.2 Correctness at the Fixpoint

We consider the following facts that hold at the fixpoint of the forward pass and are invariants of the second phase. They are easy to prove by induction on the length of the operator application sequence.

Lemma 6. *Let* (u, v) *be an edge of the graph.* $u \in \mathit{preds}(v) \iff v \in \mathit{succs}(u)$.

Lemma 7. *For an arbitrary node* u*, if* $\mathit{preds}(u) \neq \emptyset$*, then the contents of* $\mathit{preds}(u), \mathit{preds}_{ex}(u)$ *are mutually exclusive and* $\mathit{preds}_{init}(u) = \mathit{preds}(u) \cup \mathit{preds}_{ex}(u)$.

We now prove the following two lemmas for the fixpoint of the second phase.

Lemma 8. *At the fixpoint of the second phase, for an arbitrary node* v *we have that* $\mathit{succs}(v) = \emptyset$.

Proof. Assume this is not the case. Then, there exist nodes x, y such that (x, y) is an edge and $y \in \mathit{succs}(x)$. By Lem. 6, $x \in \mathit{preds}(y)$. We consider cases for the contents of $\mathit{succs}(y)$.

$\mathit{succs}(y) = \emptyset$**:** Then we can apply the operator on (x, y), a contradiction since we are at the fixpoint.

$\mathit{succs}(y) \neq \emptyset$**:** Then, there exists $z \neq x \colon z \in \mathit{succs}(y)$. By this reasoning we can form a chain C of successors. C has to be finite because the shortest

path DAG is finite. Assume $C = \{y, \ldots, pw, w\}$, where $succs(w) = \emptyset$ and $w \in succs(pw)$ (note that it may be that $pw = y$). Then, by Lem. 6, $pw \in preds(w)$. Hence, there exists an edge (pw, w) such that an operator is applicable, a contradiction since we are at the fixpoint.

□

Lemma 9. *At the fixpoint of the second phase, for an arbitrary node v we have $preds_{ex}(v) = preds_{init}(v)$.*

Proof. Assume that there exists v such that at the fixpoint $preds_{ex}(v) \neq preds_{init}(v)$. Then, we consider two cases for an edge (w, v):

$w \in preds_{ex}(v) \wedge w \notin preds_{init}(v)$**:** Then by observing the execution history, there exists an operator application a during which w was added to $preds_{ex}(v)$ and was removed from $preds(v)$. But $preds(v) \subseteq preds_{init}(v)$, a contradiction.

$w \notin preds_{ex}(v) \wedge w \in preds_{init}(v)$**:** Then, by Lem. 8 and Lem. 7, at the fixpoint $w \in preds(v)$ and $succs(v) = \emptyset$. Hence, an operator is applicable on (w, v), a contradiction since we are at the fixpoint.

□

We now prove the following invariant about the value of $\delta(v)$.

Lemma 10. *For an arbitrary node v, $\delta(v)$ is given by:*

$$\delta(v) = \sum_{w\colon v \in preds_{ex}(w) \wedge succs(w)=\emptyset} t(v,w) \tag{B.1}$$

, where $t(v,w) = \frac{\sigma_{sv}}{\sigma_{sw}}(1+\delta_s(w))$

Proof. By induction on the length of the operator application sequence.

Induction Basis: Consider an arbitrary node v. Initially, we have $\delta(v) = 0$ and $preds_{ex}(v) = \emptyset$, hence B.1 holds.

Inductive Step: Consider an arbitrary edge (u,v) where an operator is applied. We have $u \in preds(v) \wedge succs(v) = \emptyset$. By the inductive hypothesis and Lem. 7, $\delta(u) = k$, which does not include a contribution $t(u,v)$ from v. After the operator application we have that $u \in preds_{ex}(v) \wedge succs(v) = \emptyset$ and $\delta'(u) = k + t(u,v)$, hence B.1 holds.

□

We are now ready to prove the main correctness theorem for the backward phase.

Theorem 4. *Let $preds_{init}(v), BC_{init}(v)$ denote the values of the respective node attributes at the beginning of the backward pass. At the fixpoint, the following facts hold for an arbitrary node v:* ***(a)*** $\delta(v) = \sum_{w\colon v \in preds_{init}(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1+\delta_s(w))$ ***(b)*** *if $v \neq Root$ then $BC(v) = BC_{init}(v) + \delta(v)$, else $BC(v) = BC_{init}(v)$.*

Proof. At the fixpoint for all nodes w we have, by Lem. 8, that $succs(w) = \emptyset$. Hence, for an arbitrary node v we have, by Lem. 9 and Lem. 10, that $\delta(v) = \sum_{w:\ v \in preds_{init}(w)} t(v, w)$. Regarding $BC(v)$, the operator updates it by adding to it $\delta(v)$ while processing the edge (w, v) between v and the last $w \in preds(v)$. At that point $\delta(v)$ has stabilized at its final, correct value. Since *Root* has no predecessors, no update to its BC occurs. □

Bibliography

[1] 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/~challenge9/.

[2] The Galois system. http://iss.ices.utexas.edu/?p=projects/galois, 2014.

[3] A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, 2006.

[4] A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.

[5] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FM-CAD*, 2013.

[6] D. Bader., J. Gilbert, J. Kepner, and K. Madduri. Hpcs scalable synthetic compact applications graph analysis (SSCA2) benchmark v2.2, 2007.

[7] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, Berlin, Heidelberg, 2007.

[8] D.A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11), 2006.

[9] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *ICPP*, 2006.

[10] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.

[11] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. From relational verification to simd loop synthesis. In *PPoPP*, 2013.

[12] Don Batory and Jeff Thomas. P2: A lightweight dbms generator. *J. Intell. Inf. Syst.*, 9(2), 1997.

[13] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, A. Sibiryakov, D. E. Bernholdt, A. Bibireata, D. Cociorva, X. Gao, S. Krishnamoorthy, and S. Krishnan. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. In *Proceedings of the IEEE*, 2005.

[14] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[15] G.E. Blelloch, J.T. Fineman, P.B. Gibbons, and H.V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, 2011.

[16] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, 2012.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPOPP*, 1995.

[18] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5), 1999.

[19] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 2011.

[20] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25, 2001.

[21] U. Brandes and C. Pich. Centrality Estimation in Large Networks. *International Journal of Birfucation and Chaos*, 17(7), 2007.

[22] A. Braunstein, M. Mèzard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2), 2005.

[23] A. Buluc and J.R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *Int. Journal of High Perf. Computing Applications*, 2011.

[24] H. Bunke, T. Glauser, and T. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In *Graph Grammars and Their Application to Computer Science*, 1991.

[25] J. Cai and R. Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11(3), 1989.

[26] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, 2007.

[27] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM Data Mining*, 2004.

[28] D. Chazan and W. L. Miranker. Chaotic relaxation. *Linear Algebra and Applications*, 2, 1969.

[29] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, 2008.

[30] R.A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *IPDPS*, 2010.

[31] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47, 2004.

[32] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on a multi-threaded, multi-core platform. In *IPDPS*, 2011.

[33] Guojing Cong, Gheorghe Almasi, and Vijay Saraswat. Fast pgas connected components algorithms. In *PGAS*, 2009.

[34] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[35] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[36] A. Del Sol, H. Fujihashi, and P. O'Meara. Topology of small-world networks of protein–protein complex structures. *Bioinformatics*, 2005.

[37] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *OOPSLA*, 2011.

[38] A. Dragojevic, Y. Ni, and A. Adl-Tabatabai. Optimizing transactions for captured memory. In *SPAA*, 2009.

[39] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, and C. Corley. Massive social network analysis: Mining twitter for social good. In *ICPP*, 2010.

[40] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *HiPC*, 2010.

[41] D. Eppstein. *Spanning trees and spanners*, pages 425–461. Elsevier, 1999.

[42] Mattias Eriksson and Christoph Kessler. Integrated code generation for loops. *ACM Trans. Embed. Comput. Syst.*, 11S(1), June 2012.

[43] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, 1971.

[44] L. C. Freeman. A set of measures of centrality based on betweenness. 1977.

[45] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[46] R. Geiß, G. Batz, D. Grund, S. Hack, and A. Szalkowski. Grgen: A fast spo-based graph rewriting tool. In *Graph Transformations*, 2006.

[47] A. H. Ghamarian, A. Jalali, and A. Rensink. Incremental pattern matching in graph-based state space exploration. In *GraBaTs*, 2010.

[48] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12), 2002.

[49] E Mark Gold. Language identification in the limit. *Information and Control*, 10(5), 1967.

[50] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[51] Sumit Gulwani. Dimensions in program synthesis. In *FMCAD*, 2010.

[52] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC*, 2012.

[53] Sumit Gulwani. Example-based learning in computer-aided STEM education. *Commun. ACM*, 57(8), 2014.

[54] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[55] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[56] T. L. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*, 2006.

[57] Tim Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3), 2005.

[58] M.A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *PPOPP*, 2011.

[59] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, 2011.

[60] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[61] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[62] M. Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[63] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[64] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *TRANSACT*, 2006.

[65] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12), December 1986.

[66] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, 2012.

[67] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, 2010.

[68] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[69] Joseph F. JaJa. *An introduction to parallel algorithms*. Addison-Wesley, 1992.

[70] H. Jeong, S. P. Mason, A. L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411, May 2001.

[71] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS*, 2010.

[72] Troy A. Johnson and Rudolf Eigenmann. Context-sensitive domain-independent algorithm composition and selection. In *PLDI*, 2006.

[73] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[74] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *TOPLAS*, 28(6), 2006.

[75] Henry A Kautz, Bart Selman, et al. Planning as satisfiability. *ECAI*, 92.

[76] Ken Kennedy and John Allen, editors. *Optimizing compilers for modren architectures:a dependence-based approach*. Morgan Kaufmann, 2001.

[77] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2), 1968.

[78] V. Krebs. Mapping networks of terrorist cells. *Connections*, 2002.

[79] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[80] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA*, 2008.

[81] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[82] V. Kuncak and M. C. Rinard. Decision procedures for set-valued fields. *Electr. Notes Theor. Comput. Sci.*, 131, 2005.

[83] V. Kuncak and M. C. Rinard. An overview of the jahob analysis system: project goals and current status. In *IPDPS*, 2006.

[84] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? WWW '10, 2010.

[85] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[86] P. Lam, V. Kuncak, and M. C. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39(3), 2004.

[87] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, 2010.

[88] A. Lenharth, D. Nguyen, and K. Pingali. Priority queues are not good concurrent priority schedulers. Technical Report TR-11-39, UT Austin, Nov 2011.

[89] T. Lev-Ami and M. Sagiv. TVLA: A framework for implementing static analyses. In *SAS*, 2000.

[90] F. Liljeros, C.R. Edling, L. Amaral, H.E. Stanley, and Y. Åberg. The web of human sexual contacts. *Nature*, 411, 2001.

[91] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM TOPLAS*, 31(6), 2009.

[92] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[93] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D/ G. Chavarría-miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDS*, 2009.

[94] R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.

[95] Roman Manevich, Rashid Kaleem, and Keshav Pingali. In *CPC*, 2012.

[96] Z. Manna and R. Waldinger. Synthesis: Dreams ? programs. *IEEE Transactions on Software Engineering*, 5(4), 1979.

[97] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1), 1980.

[98] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3), 1971.

[99] M. Marron, D. Stefanovic, D. Kapur, and M. V. Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *LCPC*, 2008.

[100] Henry Massalin. Superoptimizer: A look at the smallest program. In *ASPLOS*, 1987.

[101] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.

[102] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPOPP*, 2010.

[103] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA*, 1998.

[104] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 2003.

[105] Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, and Rastislav Bodik. Parallel schedule synthesis for attribute grammars. In *PPoPP*, 2013.

[106] S. Nedunuri, W. R. Cook, and D. R. Smith. Theory and Techniques for Synthesizing a Family of Graph Algorithms. *ArXiv e-prints*, July 2012.

[107] Srinivas Nedunuri, DouglasR. Smith, and WilliamR. Cook. Theory and techniques for synthesizing efficient breadth-first search algorithms. In *FM*, 2012.

[108] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS*, 2011.

[109] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[110] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *PLDI*, 2013.

[111] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM TOPLAS*, 4(3), 1982.

[112] G. Palla, I. J. Farkas, P. Pollner, I. Derenyi, and T. Vicsek. Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics*, 10, 2008.

[113] Dusko Pavlovic, Peter Pepper, and Douglas Smith. Formal derivation of concurrent garbage collectors. In *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*. 2010.

[114] R. Pearce, M. Gokhale, and N.M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC*, 2010.

[115] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo,

D. Prountzos, and X. Sui. The TAO of parallelism in algorithms. In *PLDI*, 2011.

[116] D. Plump. The graph programming language GP. In *CAI*, 2009.

[117] A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.

[118] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *PLDI*, 2010.

[119] D. Prountzos, R. Manevich, K. Pingali, and K. S. McKinley. A shape analysis for optimizing parallel graph programs. In *POPL*, 2011.

[120] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A system for synthesizing concurrent graph programs. In *OOPSLA*, 2012.

[121] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Synthesizing parallel graph programs via automated planning. In *PLDI*, 2015.

[122] Dimitrios Prountzos and Keshav Pingali. Betweenness centrality: Algorithms and implementations. In *PPoPP*, 2013.

[123] Y. Pu, R. Bodik, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *OOPSLA*, 2011.

[124] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 2005.

[125] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi, and Xiao Doug Burger. *A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services*. In *ISCA*, 2014.

[126] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10(2), 1999.

[127] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[128] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL*, 2013.

[129] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., 1997.

[130] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3), 2002.

[131] A. Salcianu and M. C. Rinard. Purity and side effect analysis for java programs. In *VMCAI*, 2005.

[132] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. SIGMOD, 2014.

[133] Thomas Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Universität Karlsruhe, 2007.

[134] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[135] A. Schürr, A. J. Winter, and A. Zündorf. Handbook of graph grammars and computing by graph transformation. chapter The PROGRES approach: language and environment. World Scientific Publishing Co., Inc., 1999.

[136] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, 1983.

[137] Z. Shi and B. Zhang. Fast network centrality analysis using gpus. *BMC Bioinformatics*, 2011.

[138] Yossi Shiloach and Uzi Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1), 1982.

[139] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[140] J. G. Siek. and and A. Lumsdaine L.Q. Lee. *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*. Addison-Wesley Professional, 2001.

[141] Yannis Smaragdakis and Don Batory. Distil: a transformation library for data structures. In *DSL*, 1997.

[142] Douglas R. Smith. Mechanizing the development of software. In *Client Resources on the Internet, IEEE Multimedia Systems*, 1999.

[143] D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9), 1990.

[144] A. Solar-Lezama, C.G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[145] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI*, 2007.

[146] S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *POPL*, 2010.

[147] G. Tan, V. C. Sreedhar, and G. R. Gao. Analysis and performance results of computing betweenness centrality on IBM Cyclops64. *J. Supercomput.*, 56, 2011.

[148] Guangming Tan, Dengbiao Tu, and Ninghui Sun. A parallel algorithm for computing betweenness centrality. In *ICPP*, 2009.

[149] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *SPAA*, 2011.

[150] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *POPL*, 2009.

[151] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, 2008.

[152] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[153] J. Yang and Y. Chen. Fast computing betweenness centrality with virtual nodes on large sparse networks. *PLoS ONE*, 6(7), 2011.

[154] Q. Yang and S. Lonardi. A parallel algorithm for clustering protein-protein interaction networks. *Comp. Systems Bioinformatics*, 2005.

[155] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, S. V. N. Vishwanathan, and Inderjit S. Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *CoRR*, 2013.

[156] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.