Elixir: A System for Synthesizing Concurrent Graph Programs [*]

Dimitrios Prountzos Roman Manevich Keshav Pingali
The University of Texas at Austin, Texas, USA
dprountz@cs.utexas.edu,roman@ices.utexas.edu,pingali@cs.utexas.edu

Abstract

Algorithms in new application areas like machine learning and network analysis use "irregular" data structures such as graphs, trees and sets. Writing efficient parallel code in these problem domains is very challenging because it requires the programmer to make many choices: a given problem can usually be solved by several algorithms, each algorithm may have many implementations, and the best choice of algorithm and implementation can depend not only on the characteristics of the parallel platform but also on properties of the input data such as the structure of the graph.

One solution is to permit the application programmer to experiment with different algorithms and implementations without writing every variant from scratch. Auto-tuning to find the best variant is a more ambitious solution. These solutions require a system for automatically producing efficient parallel implementations from high-level specifications. Elixir, the system described in this paper, is the first step towards this ambitious goal. Application programmers write specifications that consist of an *operator*, which describes the computations to be performed, and a *schedule* for performing these computations. Elixir uses sophisticated inference techniques to produce efficient parallel code from such specifications.

We used Elixir to automatically generate many parallel implementations for three irregular problems: breadth-first search, single source shortest path, and betweenness-centrality computation. Our experiments show that the best generated variants can be competitive with handwritten code for these problems from other research groups; for some inputs, they even outperform the handwritten versions.

* This work was supported by NSF grants 0833162, 1062335, and 1111766 as well as grants from the IBM, Intel and Qualcomm Corporations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program synthesis

General Terms Algorithms, Languages, Performance, Verification

Keywords Synthesis, Compiler Optimization, Concurrency, Parallelism, Amorphous Data-parallelism, Irregular Programs, Optimistic Parallelization.

1. Introduction

New problem domains such as machine learning and social network analysis are giving rise to applications with "irregular" data structures like graphs, trees, and sets. Writing portable parallel programs for such applications is challenging for many reasons.

The first reason is that programmers usually have a choice of many algorithms for solving a given problem: even a relatively simple problem like the single-source shortest path (SSSP) problem in a directed graph can be solved using Dijkstra's algorithm [11], the Bellman-Ford algorithm [11], the label-correcting algorithm [22], and delta-stepping [22], among others. These algorithms are described in more detail in Sec. 2, but what is important here is to note that which algorithm is best depends on many complex factors.

- There are complicated trade-offs between parallelism and work-efficiency in these algorithms; for example, Dijkstra's algorithm is very work-efficient but it has relatively little parallelism, whereas the Bellman-Ford and label-correcting algorithms 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 amount of parallelism in irregular graph algorithms is usually dependent also 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.
- The best algorithmic choice may depend on the core architecture. If SIMD-style execution is supported efficiently by the cores as is the case with GPUs, the Bellman-Ford algorithm may be preferable to the label-correcting or delta-stepping algorithms. Conversely, for MIMD-style execution, label-correcting or delta-stepping may be preferable.

Another reason for the difficulty of parallel programming of irregular applications is that 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, as we show experimentally in Sec. 6. 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 CAS operations. These choices can affect performance substantially but even expert programmers cannot always make the right choices.

1.1 Synthesis of Irregular Programs

One promising approach for addressing these problems is *program synthesis*. 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 mathematics can be used as a specification language [29]. However, irregular graph problems do not have mathematical structure that can be exploited to generate implementation variants.

In this paper, 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* [25]. 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.

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-efficency 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 paper. 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 paper 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?

Currently, there are two main restrictions in the specification language. First, Elixir supports only operators for which neighborhoods contain a fixed number of nodes and edges. Second, Elixir does not support mutations on the graph structure, so algorithms such as Delaunay mesh refinement cannot be expressed currently in Elixir. We believe Elixir can be extended to handle such algorithms, but we leave this for future work.

The rest of this paper is organized as follows. Sec. 2 presents the key ideas and challenges, using a number of algorithms for the `SSSP` problem. Sec. 3 formally presents the Elixir graph programming language and its semantics. Sec. 4 describes our synthesis techniques. Sec. 5 describes our auto-tuning procedure for automatically exploring implementations. Sec. 6 describes our empirical evaluation of

Elixir. Sec. 7 discusses related work and Sec. 8 concludes the paper.

2. Overview

In this section, we present the main ideas behind Elixir, using the `SSSP` problem as a running example. In Sec. 2.1 we discuss the issues that arise when `SSSP` algorithms are written in a conventional programming language. In Sec. 2.2, 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 Sec. 2.3 we sketch how Elixir addresses the two most important synthesis challenges: how to synthesize efficient implementations, and how to insert synchronization.

2.1 `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 Sec. 1, there are many sequential and parallel algorithms for solving the `SSSP` problem such as Dijkstra's algorithm [11], the Bellman-Ford algorithm [11], the label-correcting algorithm [22], and delta-stepping [22]. In all algorithms, each node a has an integer attribute ad 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 ad is lowered and (i) there is an edge (a, b) of length w_{ab}, and (ii) $bd > ad + w_{ab}$, the value of bd is lowered to $ad + w_{ab}$. However, the order in which edge relaxations are performed can be different for different algorithms, as we discuss next.

In Fig. 1 we present Java-like 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 bd 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,

```
Label Correcting

INITIALIZATION:
for each node a in V {
  if (a==Src) ad = 0;
  else ad = INFINITY;
}
RELAXATION:
Wl = new worklist();
// init worklist
for each e=(Src,_,_) {
  Wl.add(e);
}
while (!Wl.empty()) {
  (a,b,w) = Wl.get();
  if ($ad + w < bd$) {
    bd=ad+w;
    for each e:outEdg(b)
      Wl.add(e);
  }
}
```

```
Bellman Ford

INITIALIZATION:
for each node a in V {
  if (a==Src) ad = 0;
  else ad = INFINITY;
}
RELAXATION:
for i = 1..$|V| - 1$ {
  for each e=(a,b,w) {
    if ($ad + w < bd$) {
      bd=ad+w;
}}
```

Figure 1: Pseudocode for label-correcting and Bellman-Ford `SSSP` algorithms.

where $|V|$ is the number of graph nodes. Although both algorithms are built using the same basic ingredient, as Fig. 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.

2.2 `SSSP` in Elixir

Fig. 2 shows several `SSSP` algorithms written in Elixir. The major components of the specification are the following.

2.2.1 Operator Specification

In Fig. 2, lines 1–2 define the graph. Nodes and edges are represented abstractly by relations that have certain attributes. Each node has a unique ID 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 *predicated subgraph 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 between them, and whose `dist` attributes satisfy the constraint $ad + w_{ab} < bd$. 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; it is a special case of the general notion of neighborhoods in the operator formulation [25].

The right-hand side of a rewrite rule specifies updates to some of the value attributes of nodes or edges in a subgraph matching the predicated subgraph pattern on the left-hand side of that rule. In this paper, we restrict attention to *local computation* algorithms [25] that are not allowed to morph the graph structure by adding or removing nodes and edges.

Elixir allows disjunctive operators of the form

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

where all operators op_i share a common same shape constraint. The Betweenness Centrality programs discussed in Sec. 6 use disjunctive operators with 2 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 subgraph that matches the left-hand side of the operator and applying the operator there. The statement terminates when no sub-graphs match the left-hand side of the operator. 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 problem-dependent, so it is necessary to give the application programmer control over scheduling. Sec. 2.2.2 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. Sec. 2.3.1 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. Sec. 2.3.2 describes some of the main issues in doing this.

```
1  Graph [ nodes(node : Node, dist : int )
2          edges(src : Node, dst : Node, wt : int ) ]
3
4  source : Node
5
6  initDist = [ nodes(node a, dist d) ] →
7             [ d = if (a == source) 0 else ∞ ]
8
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 ]
14
15 init = foreach initDist
16 sssp = iterate relaxEdge ≫ sched
17 main = init ; sssp
```

Algorithm	**Schedule specification**
Dijkstra	sched = **metric** ad ≫ **group** b
Label-correcting	sched = **group** b ≫ **approx metric** ad ≫ **unroll** 2
Δ-stepping-style	DELTA : **unsigned int** sched = **metric** (ad + w) / DELTA

Bellman-Ford	NUM_NODES : **unsigned int**
	// override sssp
	sssp = **for** i =1..(NUM_NODES −1)
	step
	step = **foreach** relaxEdge

Figure 2: Elixir programs for `SSSP` algorithms.

2.2.2 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 $ad_1 < ad_2$, give e_1 a higher

priority for execution. In Elixir, this is expressed by the specification `metric ad`.

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 neighbors b of node a. 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 this paper 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 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. Thus, it applies the grouping refinement for every operator referring to `v`.

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 support `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. 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.* [23] to define a series dynamic scheduling policies that combine `metric` with `LIFO,FIFO` policies and use implementations of these worklists from the Galois framework [2].

Fig. 2 shows the use of Elixir scheduling constructs to define a number of `SSSP` implementations. The label-correcting variant [22] 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 [22] 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 [11] works in a SIMD style by performing a series of rounds in which it processes all edges in the graph.

2.3 Synthesis Challenges

This section gives a brief description of the main challenges that Elixir addresses. First, we discuss how Elixir optimizes worklist manipulation and second how it synchronizes code to ensure atomic operator execution.

2.3.1 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 Fig. 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 bd 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 $db + w_{bc} \geq dc$.

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 Fig. 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 priority 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.

2.3.2 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.

3. The Elixir Graph Programming Language

In this section, we formalize our language whose grammar is shown in Fig. 3. Technically, a graph program defines graph transformations, or *actions*, that may be used within an application. A graph program 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 only read. They may be initialized by the larger application before invoking an action. 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 alter the graph structure. Actions compose statements and name them. They compile to C++ functions that take a single graph reference argument.

3.1 Graphs and Patterns

Let the *Attrs* denote a finite set of *attributes*. An attribute denotes a subtype of one of the following types: the set of numeric values *Nums* (integers and reals), graph nodes *Nodes* and sets of graph nodes $\wp(\mathit{Nodes})$. Let $\mathit{Vals} \overset{\text{def}}{=} \mathit{Nums} \cup \mathit{Nodes} \cup \wp(\mathit{Nodes})$ stand for the union of those types.

Definition 3.1 (Graph)**.** [1] *A graph* $G = (V^G, E^G, \mathit{Att}^G)$ *where* $V^G \subset \mathit{Nodes}$ *are the graph nodes,* $E^G \subseteq V^G \times V^G$ *are the graph edges, and* $\mathit{Att}^G : ((\mathit{Attrs} \times V^G) \to \mathit{Vals}) \cup ((\mathit{Attrs} \times V^G \times V^G) \to \mathit{Vals})$ *associates values with nodes and edges. We denote the set of all graphs by Graph.*

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

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) **]** $\to$ **[** 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	::=	statExp dynSched
opsExp	::=	opid | opid **or** opsExp
statExp	::=	opsExp 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 3: Elixir language grammar (EBNF). The notation e? means that e is optional.

Definition 3.2 (Pattern)**.** *A pattern* $P = (V^P, E^P, \mathit{Att}^P)$ *is a connected graph over variables. Specifically,* $V^P \subset \mathit{Vars}$ *are the pattern nodes,* $E^P \subseteq V^P \times V^P$ *are the pattern edges, and* $\mathit{Att}^P : (\mathit{Attrs} \times V^P) \to \mathit{Vars} \cup (\mathit{Attrs} \times V^P \times V^P) \to \mathit{Vars}$ *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 3.3 (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* $\mathit{Match} : \mathit{Vars} \to \mathit{Nodes}$*.*

We extend a matching $\mu : V^P \to V^G$ to evaluate attribute variables $\mu : \mathit{Vars} \to \mathit{Vals}$ as follows. For every attribute a, pattern nodes $y, z \in V^P$, and attribute variable x, we define:

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

Lastly, we extend μ to evaluate expressions over the variables of a pattern by structural induction over the natural definitions of the sub-expression types defined in Fig. 3.

3.2 Operators

We denote an operator 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 $Upd^{op} : V^R \to Exprs$ 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 the 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 $\mu(Gd) = \text{True}$. In such a case, μ induces the subgraph $D = \mu(R)$, which we call a *redex* and define by:

$$\begin{aligned} V^D &\overset{\text{def}}{=} \{\mu(x) \mid x \in V^R\} \\ E^D &\overset{\text{def}}{=} \{(\mu(x), \mu(y)) \mid (x, y) \in E^R\} \\ Att^D &\overset{\text{def}}{=} \{(a, Att^G(a, u)), (b, Att^G(b, v, w)) \mid \\ &\qquad a, b \in Attrs,\ u \in V^D, (v, w) \in E^D\} \ . \end{aligned}$$

We define

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

where $D = \mu(R^{op})$ and the node and edge attributes in D are updated using the expressions in Upd^{op}:

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

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

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.

3.3 Semantics of Unordered Statements

Unordered statements have the form '`iterate` *unordExp*' or '`foreach` *unordExp*' where *unordExp* uses the **or** operator, which we will refer to as disjunction, to combine expressions of the form

$$opsExp >> statExp >> dynSched \ .$$

Intuitively, a disjunction represents alternative graph transformations.

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

The expression *statExp*, 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*>>*statExp* 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 paper we present the semantics under the simplifying assumption that *statExp* is empty. For the full technical treatment, the reader is referred to [27].

3.3.1 Preliminaries

Definition 3.4 (Active Element). *An active element, denoted by elem$\langle op, \mu\rangle$, pairs an operator op with a matching $\mu \in$ Match. Intuitively, it means that op is applied on μ. We denote the set of all active elements by $\mathcal{A}$.*

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

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

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

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

We lift the operation to disjunctions:

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

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 $V^R \setminus \overline{v}$ to 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

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

We use $\textsc{Expand}$ to implement the `group` static scheduling term and to implement $\textsc{Delta}$.

3.3.2 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 a operator and applying that operator to the redex. We now describe how `metric`, `approx metric`, `LIFO`, and `FIFO` scheduling terms define a quasi order over active elements by associating them with priorities, as they are created. An execution of `iterate` results with a (possibly infinite) sequence $G = G_0, \ldots, G_k, \ldots$ where each index represents the application of one operator. Let $elem\langle op, \mu\rangle$ be an active element created at G_i. For an arithmetic expression a, the terms `metric` a and `approx metric` a associate the priority $\mu(a)$ evaluated at G_i and quasi order $p \leq_{\texttt{metric}\,a} p'$ if $p \leq p'$. The terms `LIFO` and `FIFO` associate the priority i and quasi order $p \geq_{\texttt{LIFO}} p'$ if $p \leq p'$ and $p \leq_{\texttt{FIFO}} p'$ if $p \leq p'$.

We denote the priority given by term t to an active element v constructed at graph G_i by $p(t, v, i)$. We define the priority given by an expression $d = t_1 \texttt{>>} \ldots \texttt{>>} t_k \in$ *dynSched* by $p(d, v, i) = \langle p(t_1, v, i), \ldots, p(t_k, v, i)\rangle$ and the lexicographic quasi order, which is also defined for vectors of different lengths. A prioritized active element $elem\langle op, \mu, p\rangle$ 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 $p_v \leq p_w$

We define the type of prioritized worklists by $\mathcal{W}^{\mathcal{P}} \overset{\text{def}}{=} \mathcal{A}^{\mathcal{P}*}$. We say that a prioritized worklist $\omega = e_1, \ldots, e_k \in \mathcal{W}^{\mathcal{P}}$ is ordered according to a dynamic scheduling expression $d \in$ *dynSched*, if for every $1 \leq i \leq j \leq k$, we have that $e_i \leq e_j$, i.e., ω starts with the lowest priority element. We define $\text{PRIORITY}[\![d]\!]\,\omega = \omega'$ if ω' is a permutation of ω preserving the quasi order induced by $d \in$ *dynSched*. We define the following scheduler-related operations for a dynamic scheduling expression *exp*:

$$\begin{array}{lcl}
\text{EMPTY} & \overset{\text{def}}{=} & \epsilon \\
\text{POP}\,\omega & \overset{\text{def}}{=} & (head(\omega), tail(\omega)) \\
\text{MERGE}[\![exp]\!]\,(\omega, \delta) & \overset{\text{def}}{=} & \text{PRIORITY}[\![exp]\!]\,(\omega \cup \delta) \\
\text{INIT}[\![exp]\!]\,G & \overset{\text{def}}{=} & \text{MERGE}[\![exp]\!]\,(\epsilon, \text{RDX}[\![op_{1..k}]\!]G)
\end{array}$$

3.3.3 Iteratively Executing Operators

We define the set of program states as $\Sigma \overset{\text{def}}{=} Graph \cup Graph \times Wl$. The meaning of statements is given in terms of a transition relation having one of the following forms:

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 Fig. 4. The semantics induced by the transition relation yields (possibly infinite) sequences of states $\sigma_1, \ldots, \sigma_k, \ldots$. A correct concurrent implementation gives the illusion that each transition occurs atomically, even though the executions of different transitions may interleave.

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

$\langle \texttt{iterate}\ exp, G\rangle \Longrightarrow \langle \texttt{iterate}\ exp, G + Wl\rangle$ if
$Wl = \text{INIT}[\![exp]\!]\,G$

$\texttt{iterate}^{\text{step}}$ executes an operator

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

$$\begin{array}{lcl}
(elem\langle op, \mu, p\rangle, Wl') & = & \text{POP}\ Wl \\
G' & = & [\![op]\!](G, \mu) \\
\Delta & = & \text{DELTA}[\![op, op_{1..k}]\!]\,(G, \mu) \\
Wl'' & = & \text{MERGE}[\![exp]\!]\,(Wl', \Delta)
\end{array}$$

$\texttt{iterate}^{\text{done}}$ returns the graph when no more operators can be scheduled

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

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

$\texttt{foreach}^{\text{step}}$ executes an operator

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

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

Figure 4: An operational semantics for Elixir statements.

3.4 Semantics of Ordered Statements

Ordered statements have the form

$$\texttt{iterate}\ opsExp \texttt{>>} statExp \texttt{>>} \texttt{metric}\ exp\ .$$

The static scheduling expression *statExp* 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 **or** $\ldots$ **or** op_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 have parallel threads process all work in bucket B_i before moving to B_{i+1}. This implements the so-called “level-by-level” parallel execution strategy.

3.5 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 made a change to the redex, i.e., $Att'^D = Att^D$. This requires comparing the result of the operator to a backup copy of the redex 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 3.5 (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 reasoning procedure, as explained in Sec. 4.3.

Our compiler checks for strong guards at compile-time and signals an error to the programmer otherwise (see details in Sec. 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.

4. Synthesis

In this section, we explain how to emit code to implement Elixir statements. We use the notation $\mathsf{Code}(e)$ for the code fragment implementing the mathematical expression e in a high-level imperative language.

This section is organized as follows. First, we discuss our assumptions regarding the implementation language. Sec. 4.1 describes the synthesis of operator-related procedures. Sec. 4.2 describes the synthesis of the EXPAND operation, which is used to synthesize RDX and as a building block in synthesizing DELTA. Sec. 4.3 describes the synthesis of DELTA via automatic reasoning. Sec. 4.4 puts together the elements needed to synthesize unordered statements. Finally, Sec. 4.5 describes the synthesis of ordered statements.

Implementation Language and Notational Conventions

We assume the language contains 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 by 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 **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. In defining procedures, we will use the notation $f[\mathit{statArgs}](\mathit{dynArgs})$ 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: $Att'^G = Att^G(a, n) \mapsto d$, and $\mathsf{set}(b, v_n, v_m, d_b)$ updates the value of the attribute b on the edge (n, m) to d': $Att'^G = Att^G(b, n, m) \mapsto d'$.
- $\mathsf{edge}(v_n, v_m)$ checks whether $(n, m) \in E^G$.
- $\mathsf{succs}(v_n)$ and $\mathsf{preds}(v_n)$ return (iterators to) the sets of nodes $\{s \mid (n, s) \in E^G\}$ and $\{p \mid (p, n) \in E^G\}$, respectively.
- nodes returns (an iterator to) the set of graph nodes V^G.

In addition, we require that the graph data structure be linearizable[2].

4.1 Synthesizing Atomic Operator Application

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

[2] In practice, our graph implementation is optimized for non-morphing actions. We rely on the locks acquired by the synthesized code to correctly synchronize concurrent accesses to graph attributes.

```
def apply[$op$](mu : record[$n_{1..k}$]) =
  static for $i = 1..k$ {$n_i$ := mu[$n_i$]}
  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 checkShape[$op$](mu) ∧ 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 $\mu(s_j, d_j)$
  static for $i = 1..k$
    static for $j = 1..k$
      if $n_i = n_j$ // Check if $\mu$ 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($(G,\mu)Gd^{op}$)

Figure 5: Operator-related procedures.

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'.

Fig. 5 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 uses synchronization to ensure atomicity. The procedure 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, $\text{lk}_{1..k}$) 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.

```
def apply[relaxEdge](mu : record[a, b]) =
  a := mu[a]; b := mu[b];
  lk_1 := a; lk_2 := b;
  if lock_less(lk_2, lk_1) // inline sort
    swap(lk_1, lk_2);
  lock(lk_1); lock (lk_2);
  ad := get(dist, a); bd := get(dist, b);
  w := get(wt, a, b);
  if ad + w < bd // test guard
    set(dist, b, ad + w);
  unlock(lk_1); unlock (lk_2);
```

Figure 6: Code($[\![relaxEdge]\!]$).

Fig. 6 shows the code we emit for Code($[\![relaxEdge]\!]$).

4.2 Synthesizing Expand

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 now develop a procedure that accepts a matching $\mu \in D \to V^G$, where D 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, but it is more efficient to choose an order that enables scanning the edges incident to nodes that are already bound. The alternative way requires scanning the entire set of graph nodes for each unbound pattern node and checking whether it is a neighbor of some bound node, which is too inefficient. We represent an efficient order by a permutation of $v_{m+1..k}$, $u_{m+1..k}$, and by an auxiliary sequence $T(R, v_{m+1..k}) = (u_{m+1}, w_{m+1}, dir_{m+1}), \ldots, (u_k, w_k, dir_k)$ where each tuple defines the connection between an unbound node u_j and a previously bound node w_j and the direction of the edge between them — forward for false and reverse otherwise. More formally, for every $j = m+1..k$ we have that $w_j \in v_{1..j}$ and if $dir_j =$ false then $(u_j, w_j) \in E^R$ and otherwise $(w_j, u_j) \in E^R$.

The procedure expand, shown in Fig. 7, 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, dir_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 by testing a matching with checkShape and checkGuard, respectively.

We use expand to define Code(Rdx$[\![op]\!](G,\mu)$) in Fig. 8.

```
def expand[$op, v_{1..m}, T$ : record[$n_{m+1..k}$]]
          (mu : record[$n_{1..k}$],
           $f$ : record[$n_{1..k}$] $\Rightarrow$ unit) =
 mu$'$ := record[$n_{1..k}$] // 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[$i, code$] =
  [$s_i, d_i, dir_i$] := $T$[$i$]
  if $dir_i$ = true
    for $s \in$ succs(mu[$v_s$])
      mu$'$[$v_i$] := $s$
      $code$ // inline $code$
  else // $d = in$
    for $p \in$ preds(mu[$v_d$])
      mu$'$[$v_i$] := $p$
      $code$ // inline $code$
```

Figure 7: Code for computing $\textsc{Expand}[\![op, v_{1..m}]\!](G, \mu)$ and applying a function f to each matching.

```
def redexes[$op$]($f$ : record[$n_{1..k}$] $\Rightarrow$ unit) =
  for $v \in$ 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) $\wedge$ checkGuard[$op$](mu)
      $f$(mu)
```

Figure 8: Code for computing $\textsc{Rdx}[\![op]\!](G, \mu)$ and applying a function f to each redex.

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 matching μ, and how to emit the corresponding code.

The definition of Delta given in Sec. 3 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, as we explain next.

For the rest of this subsection, we will associate matchings with the corresponding patterns 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:

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

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 $\rho(x) = y$ if $\mu_{R'}(x) = \mu_R(y)$. We call the function ρ the *influence function* of R and R' and denote the domain of ρ by ρ_{dom}. Two matchings $\mu^1_{R'}$ and $\mu^2_{R'}$ are equivalent if they induce the same influence function ρ. We denote the equivalence class of an influence function by $[\rho]$. We can compute the class $[\rho]$ by

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

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

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

The first overapproximation we obtain is

$$\textsc{Delta}^1[\![op, op']\!]\,(G, \mu_R) \stackrel{\text{def}}{=} \bigcup_{\rho \in infs(op, op')} \textsc{Expand}[\![op', \rho_{dom}]\!](G, shift(\mu_R, \rho))$$

An obvious way to obtain a tighter approximation still is to filter out matchings not satisfying the shape and value constraints of ρ.

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. 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 $\textsc{Delta}^2[\![op, op']\!]\,(G, \mu_R)$ via useful influence functions.

$$\textsc{Delta}^2[\![op, op']\!]\,(G, \mu_R) \stackrel{\text{def}}{=} \bigcup_{\rho \in useInfs(op, op')} \textsc{Expand}[\![op', \rho_{dom}]\!](G, shift(\mu_R, \rho)) \ .$$

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 the nodes of R' with R and rename node attribute variables in R' to the variables used in the corresponding nodes of R, and similarly renamed edges attributes for identified edges.

Example 4.1 (Influence Patterns)**.** *Fig. 9 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 showing both names.*

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, which has the following form:

```
assume  (Guard)
assume  !(Guard')  // comment out if identity pattern
update(C)
assert  !(Guard')
```

Intuitively, the program constructs the following verification condition: (i) if the guard of R, `Guard`, holds (first `assume`); and (ii) the guard of R', `Guard'`, does not hold (second `assume`); and (iii) the updates assign new values; then (iv) the guard R' does not hold for the updated values. Proving the verification condition 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 Boogie [6] and Z3 [12]) asking it to prove the last assertion. This amounts to checking satisfiability of a propositional (single conjunction in fact) 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 condition, we remove the corresponding influence function. If the verifier is unable to prove the condition or a timeout is reached, we conservatively consider the function as useful.

Example 4.2 (Query Programs)**.** *Fig. 9 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.**

```
def delta2[$op, op', \rho_{1..m}$](mu : record[$n_{1..k}$],
                                   f : record[$n_{1..k}$] ⇒ unit) =
  static for $i = 1..m$
    // mu' = shift(mu)
    mu' = record[$n_{1..k}$]
    for $j = 1..k$
      mu'[inv_rho[$j$]] = $mu[j]$
    Code(EXPAND[[$op', \rho_{i\,dom}$]]$(G, \mu')$)

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

Figure 10: Code for computing $\text{DELTA}^2[\![op, op']\!]\,(G, \mu)$ and applying a function f to each matching.

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

Fig. 10 shows the code we emit for DELTA^2. We represent influence functions by appropriate records and supply an inverse function inv_rho for every influence function rho.

4.3.1 Optimizations

Our compiler applies a few useful optimizations, which are not shown in the procedures above.

***Reducing Overheads in* checkShape.** The procedure expand uses the auxiliary data structure T to compute potential matchings. In doing so it is checking a portion of the shape constraint — the edges of the redex pattern that are included in T. The compiler omits checking these edges. Often, T includes all of the pattern edges; in such a case we specialize checkShape to only check the one-to-one condition.

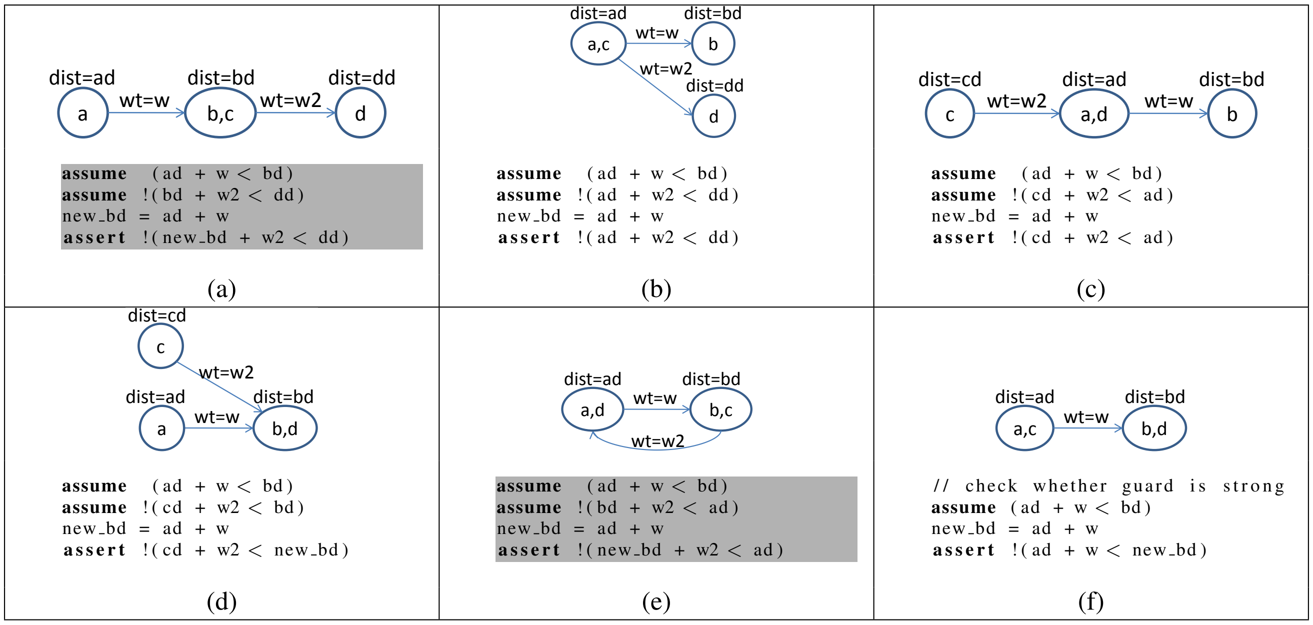
Reusing Potential Matchings. In cases when two operators have the same shape, expand reuses the matchings it computes for both of them.

4.4 Synthesizing Unordered Statements

We implement the operational semantics defined in Sec. 3.3.3 by utilizing the Galois system runtime, which enables one to: (i) automatically construct a concurrent worklist from a dynamic scheduling expression, and (i) 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 $\text{DELTA}^2[\![op, op']\!]\,(G, \mu)$, which inserts the found elements to the worklist for further processing.

4.5 Synthesizing Ordered Statements

Certain algorithms, such as [5, 19], have additional properties that enable optimizations over the baseline ordered parallelization scheme discussed in Sec. 3.4. 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



```
assume   (ad + w < bd)
assume  !(bd + w2 < dd)
new_bd = ad + w
assert  !(new_bd + w2 < dd)
```

(a)

```
assume   (ad + w < bd)
assume  !(ad + w2 < dd)
new_bd = ad + w
assert  !(ad + w2 < dd)
```

(b)

```
assume   (ad + w < bd)
assume  !(cd + w2 < ad)
new_bd = ad + w
assert  !(cd + w2 < ad)
```

(c)

```
assume   (ad + w < bd)
assume  !(cd + w2 < bd)
new_bd = ad + w
assert  !(cd + w2 < new_bd)
```

(d)

```
assume   (ad + w < bd)
assume  !(bd + w2 < ad)
new_bd = ad + w
assert  !(new_bd + w2 < ad)
```

(e)

```
// check whether guard is strong
assume (ad + w < bd)
new_bd = ad + w
assert !(ad + w < new_bd)
```

(f)

Figure 9: Influence patterns and corresponding query programs for *relaxEdge*. (b), (c), (d), and (f) are spurious patterns.

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.

4.5.1 Automating the Optimizations

We now discuss how we use automated reasoning to enable the above optimizations. What we have to show is that if the priority of active elements at the current level has some arbitrary value k, then all new active elements have the same priority $k + s$, where $s \geq l$. 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 through the BFS example. In the case of BFS the worklist delta consists only of a case similar to that of Fig. 9(a) with all weights equal to one. The query program we construct is shown in Fig. 11. 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.

```
assume (ad == k)
assume (s == C_i)
assume (ad + 1 < bd)
new_bd = ad + 1
assume (cd == new_bd)
assume (cd + 1 < dd)
assert (ad == k & new_bd == k + s)
```

Figure 11: Query program to enable leveled worklist optimization. `C_i` stands for a heuristically guessed value of s.

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 1: Dimensions explored by our synthesized algorithms.

5. Design Space Exploration

Elixir makes it possible to automatically generate a large number of program variants for solving an irregular problem like SSSP, and evaluate which one performs best on a given input and architecture. Tab. 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 [20, 23]. 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 `CL` 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.

6. Empirical Evaluation

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`). 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 [18]. In the `BFS` comparison, we use a hand-parallelized code from Leiserson and Schardl [19], and for `BC`, we use a hand-parallelized code from Bader and Madduri [5]. 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. It uses graph data structures from the Galois library but all synchronization is done by the code generated by Elixir, so the synchronization code built into Galois graphs is not used. The Galois graph classes use a standard graph API, and it is straightforward to use a different graph class if this is desirable. Implementations of standard collections such as sets and vectors are taken from the C++ standard library. 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 [3]. The generator is based on the Recursive MATrix (R-MAT) scale-free graph generation algorithm [10]. 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`.

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 2: Dimensions explored by our synthetic `SSSP` variants.

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

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 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.

6.1 Single-Source Shortest Path

We synthesize both ordered and unordered versions of `SSSP`. In Tab. 2, we present the range of explored values in each dimension for the synthetic `SSSP` variants. In Tab. 3, we present the combinations that lead to the three best performing asynchronous `SSSP` variants (`v50`, `v62`, `v63`) and the best performing delta-stepping variant (`dsv7`). In Fig. 12a and Fig. 12b 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 [22]. 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 Fig. 12c we present the runtime distribution of all synthetic `SSSP` variants on the FLA network. 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.

6.2 Breadth-First Search

We experiment with both ordered and unordered versions of the `BFS`. In Tab. 4 and Tab. 5, we present the range of explored values for the synthetic `BFS` variants and the combinations that give the best performance, respectively. In Fig. 13, we present a runtime comparison between the 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 [19], 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.

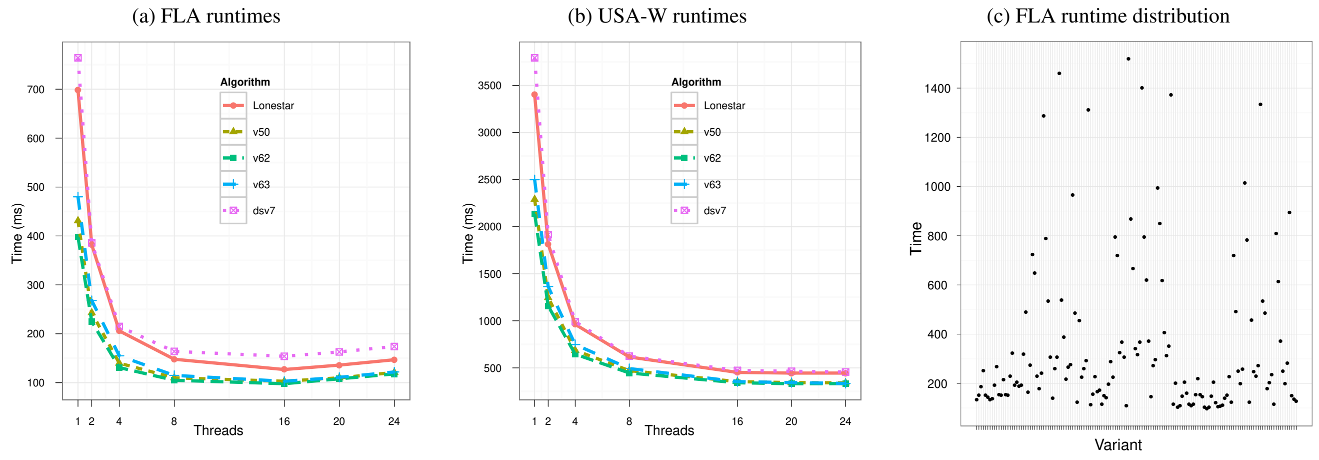


Figure 12: Runtime comparison of SSSP algorithms on FLA,USA-W inputs and runtime distribution of synthetic variants on FLA input.

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.

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: Dimensions explored by our synthetic BFS variants.

For level-by-level algorithms, there is also a spectrum of interesting choices for the worklist implementation. Elixir can deduce that BFS under the metric ad scheduling policy can have only two simultaneously active priority levels, as we discussed in Sec. 4.5. 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).

6.3 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 of graph

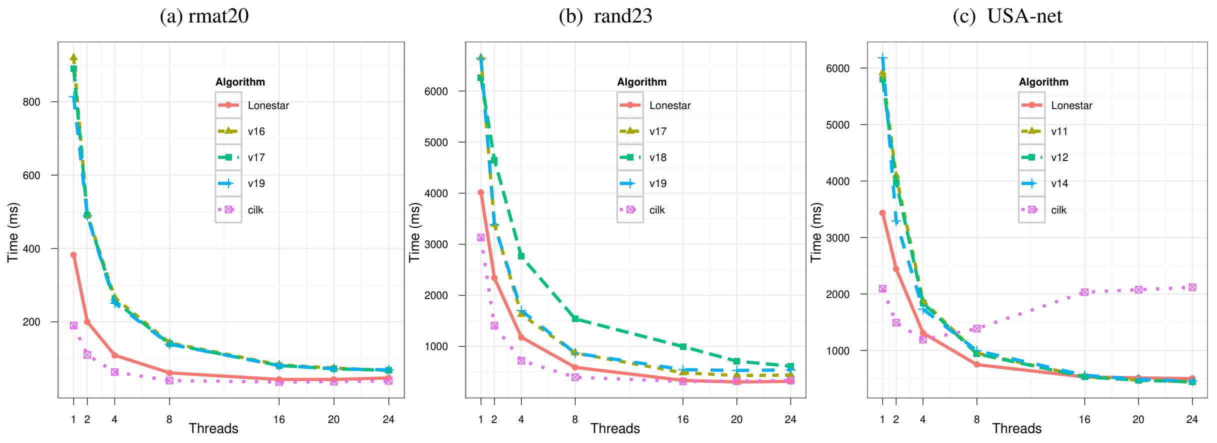


Figure 13: Runtime comparison of BFS algorithms.

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

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

Table 6: Dimensions explored by the forward and backward phase in our synthetic `BC` variants.

nodes. The betweenness score of a node u is the percentage 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 [7]. 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. [5]) 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 good approximation of betweenness values for real-world graphs [4]).

In Tab. 6 and Tab. 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 Fig. 14 we present a runtime comparison between the three best performing `BC` variants, and a hand-written, OpenMP parallel `BC` implementation by Bader and Madduri [5], which is publicly available in the SSCA benchmark suite [3]. 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.

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 7: Chosen values and priority functions for `BC` variants (✓denotes `ALL`, $\times$ denotes `NONE`, `L` denotes `LOCAL`). For the backward phase there is a fixed range of values for most parameters (see Tab. 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.

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 straight-forward 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 $p(u)$ 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.

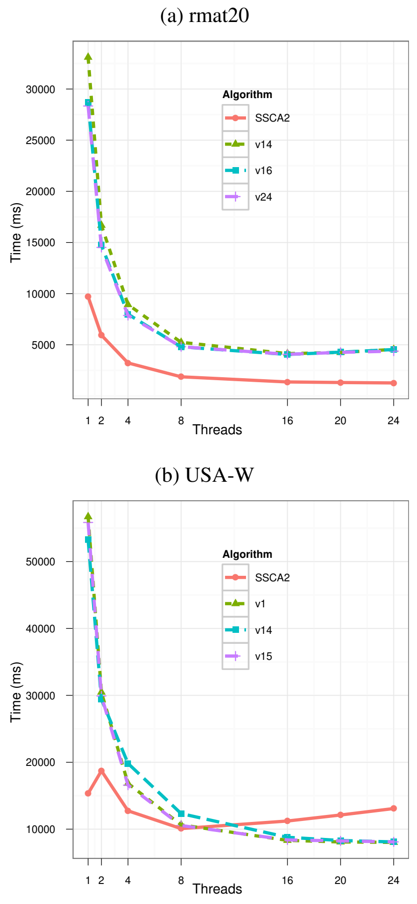


Figure 14: Runtime comparison of BC algorithms.

7. Related Work

We discuss related work in program synthesis, term and graph rewriting, and finite-differencing.

Synthesis Systems: The SPIRAL system uses recursive mathematical formulas to generate divide-and-conquer implementations of linear transforms [29]. Divide-and-conquer is used in the Pochoir compiler [34], which generates code for finite-difference computations, given a finite-difference stencil, and in the synthesis of dynamic programming algorithms [28]. 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. Green-Marl [16] 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 this paper 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 automatically. Green-Marl provides support for nested parallelism, which Elixir currently does not support. In [23] 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.

Another line of work focuses on synthesis from logic specifications [17, 33]. The user writes a logical formula and a system synthesizes a program from that. These specifications are at a much higher level of abstraction than in Elixir. Another line of work that focuses on concurrency is Sketching [32] and Paraglide [35]. 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 [36].

Term and Graph Rewriting: Term and graph rewriting [30] are well-established research areas. Systems such as GrGen [14], PROGRES [31] and Graph Programming (GP) [26] 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 [8, 15]. In a similar spirit, systems that are based on dataflow constraints are trying to efficiently perform incremental computations using runtime techniques [13]. 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.

Finite-differencing: Finite differencing [24] has been used to automatically derive efficient data structures and algorithms from high level specifications [9, 21]. This work is not focused on parallelism. Differencing can be used to come up with incremental versions of fixpoint computations [9]. 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.

8. Conclusion and Future Work

In this paper we present Elixir, a system that is the first step towards synthesizing high performance, parallel implementations of graph algorithms. Elixir starts from a high-level specification with two main components: (i) a set of operators that describe how to solve a particular problem, and (ii) a specification of how to schedule these operators to produce an efficient solution. Elixir synthesizes efficient parallel implementations with guaranteed absence of concurrency bugs, such as data-races and deadlocks. Using Elixir, we automatically enumerated and synthesized a large number of solutions for interesting graph problems and showed that our solutions perform competitively against highly tuned hand-parallelized implementations. This shows the potential of our solution for improving the practice of parallel programming in the complex field of irregular graph algorithms.

As mentioned in the introduction, there are two main restrictions in the supported specifications. First, Elixir supports only operators for which neighborhoods contain a fixed number of nodes and edges. Second, Elixir does not support mutations on the graph structure. We believe Elixir can be extended to handle such algorithms, but we leave this for future work.

Another interesting open question is how to integrate Elixir specifications within the context of a larger project that mixes other code fragments with a basic graph algorithm code. Currently Elixir allows the user to insert inside an operator fragments of uninterpreted C++ code. This way, application specific logic can be embedded into the algorithmic kernel easily, under the assumption that the uninterpreted code fragment does not affect the behavior of the graph kernel. Assessing the effectiveness of the above solution and checking that consistency is preserved by transitions to the uninterpreted mode is the subject of future work.

Acknowledgments

We would like to thank the anonymous referees and Noam Rinetzky for their helpful comments, and to Xin Sui for many useful discussions.

References

[1] 9th DIMACS Implementation Challenge. `http://www.dis.uniroma1.it/~challenge9/download.shtml`, 2009.

[2] Galois system. `http://iss.ices.utexas.edu/?p=projects/galois`, 2011.

[3] D. Bader., J. Gilbert, J. Kepner, and K. Madduri. Hpcs scalable synthetic compact applications graph analysis (SSCA2) benchmark v2.2, 2007. `http://www.graphanalysis.org/benchmark/`.

[4] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, 2007.

[5] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, 2006.

[6] 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.

[7] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[8] 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.

[9] J. Cai and R. Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11(3), 1989.

[10] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *In SIAM Data Mining*, 2004.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[12] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[13] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *OOPSLA*, 2011.

[14] R. Gei, G. Batz, D. Grund, S. Hack, and A. Szalkowski. Grgen: A fast spo-based graph rewriting tool. In *Graph Transformations*, 2006.

[15] A. H. Ghamarian, A. Jalali, and A. Rensink. Incremental pattern matching in graph-based state space exploration. In *GraBaTs*, 2010.

[16] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, 2012.

[17] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, 2010.

[18] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[19] 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.

[20] A. Lenharth, D. Nguyen, and K. Pingali. Priority queues are not good concurrent priority schedulers. Technical Report TR-11-39, UT Austin, Nov 2011.

[21] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM TOPLAS*, 31(6), 2009.

[22] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[23] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS*, 2011.

[24] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM TOPLAS*, 4(3):402–454, 1982.

[25] 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.

[26] D. Plump. The graph programming language GP. In *CAI*, 2009.

[27] D. Prountzos, R. Manevich, and K. Pingali. Elixir: A system for synthesizing concurrent graph programs. Technical Report TR-12-27, UT Austin, Aug 2012.

[28] Y. Pu, R. Bodik, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *OOPSLA*, 2011.

[29] 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.

[30] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., 1997.

[31] 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.

[32] A. Solar-Lezama, C.G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[33] S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *POPL*, 2010.

[34] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *SPAA*, 2011.

[35] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, 2008.

[36] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

```
1  Graph [ nodes(node : Node dist  : int
2                sigma : double delta  : double
3                nsuccs : int preds : set[Node],
4                bc : double bcapprox : double)
5          edges(src : Node dst : Node dist : int)
6  ]
7
8  source : Node
9
10 // Shortest path rule.
11 SP = [ nodes(node a, dist ad)
12        nodes(node b, dist bd, sigma sigb, preds pb, nsuccs nsb)
13        edges(src a, dst b)
14        (bd > ad + 1) ] →
15      [ bd   = ad + 1
16        sigb = 0
17        nsb  = 0
18      ]
19
20 // Record predecessor rule.
21 RP = [ nodes(node a, dist ad, sigma sa, nsuccs nsa)
22        nodes(node b, dist bd, sigma sb, preds pb)
23        edges(src a, dst b, dist ed)
24        (bd == ad + 1) & (ed != ad) ] →
25      [ sb  = sb + sa
26        pb  = pb + a
27        nsa = nsa + 1
28        ed  = ad
29      ]
30
31 // Update BC rule.
32 updBC = [ nodes(node a, nsuccs nsa, delta dela, sigma sa)
33           nodes(node b, nsuccs nsb, preds pb, bc bbc,
34                 bcapprox bbca, delta delb, sigma sb)
35           edges(src a, dst b)
36           (nsb == 0 & a in pb) ] →
37         [ nsa  = nsa - 1
38           dela = dela + sa / sb ∗ (1 + delb)
39           bbc  = bbc - bbca + delb
40           bbca = delb
41           pb = pb - a
42         ]
43
44 backwardInv :
         ∀a: Node, b: Node: : a ∈ preds(b) ⟹ ¬(b ∈ preds(a))
45 Forward = iterate (SP or RP) ≫ metric ad ≫ fuse ≫ group b
46 Backward = iterate {backwardInv} updBC ≫ group a
47 main = Forward; Backward
```

Figure 15: Elixir program for betweenness centrality.

A. Betweenness Centrality

Fig. 15 shows an Elixir program for solving the betweenness-centrality problem.

The Elixir language allows a programmer to specify invariants (in first-order logic) and use them to annotate actions, as shown in lines 44 and 45. (We avoided discussing annotations until this point to simplify the presentation of statements.) Our compiler adds these invariants as constraints to the query programs to further optimize the inference of useful influence patterns. Additionally, we could use these invariants to optimize the redexes procedure in order to avoid scanning the entire graph to find redexes. In all of our benchmarks this optimization would reduce the search for redexes to just those including the source node.