Synthesizing Parallel Graph Programs via Automated Planning

Dimitrios Prountzos
The University of Texas at Austin, Texas, USA
dprountz@cs.utexas.edu

Roman Manevich
Ben-Gurion University of the Negev, Israel
romanm@cs.bgu.ac.il

Keshav Pingali
The University of Texas at Austin, Texas, USA
pingali@cs.utexas.edu

Abstract

We describe a system that uses *automated planning* to synthesize correct and efficient parallel graph programs from high-level algorithmic specifications. Automated planning allows us to use constraints to declaratively encode program transformations such as scheduling, implementation selection, and insertion of synchronization. Each plan emitted by the planner satisfies all constraints simultaneously, and corresponds to a composition of these transformations. In this way, we obtain an integrated compilation approach for a very challenging problem domain. We have used this system to synthesize parallel programs for four graph problems: triangle counting, maximal independent set computation, preflow-push maxflow, and connected components. Experiments on a variety of inputs show that the synthesized implementations perform competitively with hand-written, highly-tuned code.

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

General Terms Languages, Performance, Verification

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

1. Introduction

Sparse graph computations play a central role in important application areas like web search and machine learning on big data [19]. Due to the enormous size of the data sets and the need for rapid responses to queries, these computations must be performed in parallel. Implementing correct and efficient parallel sparse graph programs is very challenging. Even on sequential machines, there are many algorithm and implementation choices for solving a given problem, and the best choice may depend on the input graph and the platform. Parallelism further complicates this problem. Irregular access patterns and dynamic dependences render traditional compile-time dependence-based techniques ineffective and require exposing and exploiting parallelism at runtime, using techniques like speculative execution [25]. This introduces additional challenges; for example, threads may need to acquire an unbounded number of locks and may need to roll back when they cannot acquire a lock.

Manually exploring this vast implementation space to find optimal solutions for a given graph algorithm and parallel platform is challenging, especially for non-expert programmers. An attractive solution to this problem is to automatically generate parallel program variants from a high-level algorithm specification, and then use search to find the best one. One notation for such a specification is Elixir [26]. Algorithms are described by graph update rules called *operators* that are applied to a graph non-deterministically until a termination condition is reached; optional scheduling hints can be provided to guide the application of these update rules[1]. The Elixir compiler translates these specifications into a high-level intermediate representation (HIR), then synthesizes a parallel low-level intermediate representation (LIR), and finally generates C++ code.

Generating efficient parallel code from this HIR requires solving three key problems: (i) finding a good schedule of HIR statements, (ii) selecting efficient implementations of HIR statements, and (iii) inserting synchronization to ensure transactional operator execution. One design strategy for the compiler is to implement separate compiler phases for each problem, but this introduces the familiar *phase ordering problem* that prevents generating high-quality code for many problems. *Integrated compilation* avoids the phase-ordering problem by combining compiler phases but it is unclear how to perform integrated compilation for the above-mentioned transformations. We describe these challenges in §2.

§3 gives an overview of our approach, which uses *automated planning* (§4) to solve these compilation problems. It encodes individual compilation tasks as constraints and can use an off-the-shelf planner to *simultaneously* solve them (§5). *This is the first integrated compilation approach for tasks such as scheduling and synchronization*, and it can be applied to other compilation problems.

We use this framework to synthesize parallel code for four challenging graph problems. To the best of our knowledge, this is the first time that parallel solutions were automatically synthesized for problems of this complexity. We automatically explore various scheduling, implementation-selection, and synchronization policies that capture algorithmic and implementation insights, such as using efficient iteration patterns based on graph data structure properties.

PLDI'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2737953

[1] This is similar to how an execution semantics can be given for λ-calculus by specifying β-reduction, which is a rewrite rule, and a scheduling order, such as normal-order reduction.

(a) ***Elixir specification***

```
1  Graph[nodes(n : Node, status : int)
2        edges(src:Node, dst:Node) ]
3  match(a) =
4  [ nodes(a, sa)
5    sa = Unmatched ∧
6    ∀ b. {edges(a, b) nodes(b, sb) :
           sb ≠ Matched }
   ]→
7  [sa := Matched;
8   map { edges(a, c) nodes(c, sc) :
      c ≠ a :
      sc := NMatched };
   ]
9  mis = foreach match
```

(b) ***misH***

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

(c) ***misV1***

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

(d) ***misV2***

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

Figure 1: (a) *MIS* Elixir specification; (b) HIR; (c) ATS-instrumented HIR; (d) Alternative ATS-instrumented variant.

In §6, we present *Aliasing tracking-based synchronization* (ATS), a novel speculative locking protocol that is synthesized by our system and provides custom synchronization for each operator. ATS can handle operators working on an unbounded neighborhood and can outperform generic runtime-based solutions for speculation.

We compare our synthesized solutions to those written by expert programmers and demonstrate promising results: our approach is always competitive with and sometimes outperforms hand-optimized implementations (§7). Related work is described in §8.

2. Generating Parallel Code: Challenges

In §2.1 we give an overview of the Elixir specification language, using maximal independent set (*MIS*) computation as an example. We use the *triangle counting* (*Triangles*) problem to illustrate other HIR aspects. In §2.2 we discuss three major problems that must be solved to produce efficient parallel code from HIR.

2.1 *MIS* and *Triangles* Examples

A *maximal independent set* of nodes in an undirected graph $G = (V, E)$ is a set $S \subseteq V$ such that a node is in S iff its immediate neighbors are not in S. Fig. 1(a) shows an Elixir program computing *MIS*. Elixir programs have three components: (i) the graph data structure declaration, (ii) the declaration of a set of operators that encode the main algorithm logic and which can be applied non-deterministically to solve the problem, and (iii) a specification of how to schedule operators to compute the solution efficiently.

For *MIS*, the graph (lines 1-2) is described by two relations, one for nodes and one for edges. Each node has a *status* attribute, which is initialized to *Unmatched*. The algorithm sets this attribute to *Matched* if the node is added to S and to *NMatched* if one of its neighbors is added to S. The *match* operator (lines 3-8) defines a graph rewrite rule. The left-hand side, called the *operator guard*, defines a predicated subgraph pattern in which node a is *Unmatched* and none of its neighbors, $\mathcal{N}(a)$, is *Matched*. When such a subgraph, dubbed as a *redex*, is detected, the right-hand side of the rewrite rule is executed, which adds a to S and updates the status of a and its neighbors appropriately. Elixir provides special looping instructions to encode operators working on an unbounded number of nodes. The $\forall$ predicate (line 6) holds if $sb \neq Matched$ is true for all $b \in \mathcal{N}(a)$. The *map* instruction (line 8) updates to *NMatched* the status of every neighbor of a distinct from a (nodes may have self-loops). Finally, a `foreach` statement (line 9) computes the initial set of redexes, and applies *match* once to each redex, in some non-deterministic order. Optionally, a scheduling tactic could be used to refine the execution order of redexes.

Fig. 1(b) shows the HIR for implementing the specification. This program expresses the execution of multiple operators according to the schedule. For *MIS* this is straightforward: the '**for** a : **nodes do**' statement (line 1) iterates over all nodes and executes *match* transactionally for each a. Fig. 2 shows another HIR program, `TrH`, that solves the *Triangles* problem in an undirected graph, where a triangle is formed by three nodes (a, b, c) each of which is connected to the other two by an edge. In this code, nodes are assumed to be numbered, and a triangle is counted only if $a<b<c$ to avoid counting the same three nodes multiple times.

```
TrH =                    TrH1 =                   TrH2 =                   TrL =
for a : nodes do         for a : nodes do         for a : nodes do         for a : nodes do
 for b : nodes do         for b : nodes do         for b : nodes do         for b : Succ(a) do
  for c : nodes do         if a < b                 if edges(a,b)            if a < b
   if edges(a,b)            if edges(a,b)            if a < b                 for c : Succ(b) do
    if edges(b,c)            for c : nodes do         for c : nodes do         if b < c
     if edges(c,a)            if b < c                 if edges(b,c)            if a < c
      if a < b                 if a < c                 if b < c                 if edges(c,a)
       if b < c                 if edges(b,c)            if a < c                 counter++
        if a < c                 if edges(c,a)            if edges(c,a)        fi fi fi odSucc fi
         counter++               counter++                counter++          odSucc od
fi fi fi fi fi od od od  fi fi fi fi od fi fi od od  fi fi fi fi od fi fi od od
```

Figure 2: Triangle counting variants.

2.2 Producing Efficient Parallel Code from HIR

Even the generation of efficient *sequential* implementations from Elixir programs is very challenging since it requires solving two difficult tasks: (i) finding a good schedule of HIR statements, and (ii) selecting efficient implementations of HIR statements (in the context of conventional compilers, analogs of these tasks are instruction scheduling and instruction selection). To generate efficient parallel code, we also have to insert synchronization to ensure that operator execution is transactional. We argue that unless all three problems are solved simultaneously, there is a phase-ordering problem that prevents the generation of efficient parallel code.

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

A simple example is an Elixir guard $p_1(a) \wedge p_2(b)$ (*e.g.*, lines 5-6 of the MIS code), which can be implemented by HIR of the form $if\ p_1(a)\ if\ p_2(b)$.... or of the form $if\ p_2(b)\ if\ p_1(a)$.... Depending on the selectivity of the predicates, one order may be more efficient than the other.

A more important scheduling opportunity arises from invariant predicates within node iterators. `TrH` and `TrH1` show an example. Since the predicates $a<b$ and **edges**(a, b) are invariant within the

'**for** c : **nodes do**' loop, they can be lifted out and the execution of the loop can be made conditional on these predicates as shown in `TrH1`. In a sparse graph, the predicate **edges**(a,b) is false for most pairs of nodes (a,b), so the optimized code is far more efficient than the original code. Even for a very dense graph, executing the c loop conditionally depending on the predicate $(a<b)$ will halve the total execution time. Note that these kinds of transformations are well beyond the capabilities of conventional loop invariant removal algorithms [1] since these algorithms only move invariant computations out of loops, and cannot make the execution of a loop dependent on the value of an invariant predicate within it.

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

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

Synchronization Producing parallel code adds extra complexity to code generation, since it is necessary to insert locking code to ensure transactional execution of the operators. Transactional execution can be achieved using synchronization protocols such as order-and-spin locking or speculative locking. We focus on *speculative locking* since it is used in existing graph frameworks [20, 25]. At a high level, correct parallel execution of *match* in *MIS* requires the following actions: (i) lock `a` and its neighbors, (ii) perform the checks on the status fields of these nodes, (iii) set these fields appropriately, and (iv) release all the locks. If a lock cannot be acquired in step (i), all currently held locks are released, and *match* is retried later. Each of these four steps can be implemented by code that touches node `a` and iterates over its neighbors; we call this the baseline version.

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

Moreover, implementing any of these variants requires bookkeeping code to keep track of locks acquired by an operator execution. An operator-agnostic generic implementation is the *stamp-and-log* strategy: each thread maintains a runtime log of locked nodes and releases the log contents when the operator execution terminates. A stamp associating each node with its current owner is used during lock acquires to detect conflicts. This strategy is used by systems that delegate concurrency management to a runtime system [25]. *However, compile-time reasoning of locks acquired along different paths in the HIR code permit the generation of synchronization code that is customized to the operator and does not need such runtime structures*.

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

The Need for an Integrated Solution How should scheduling, synchronization and implementation selection be implemented in a compiler that generates parallel code from HIR programs? A staged approach with separate compiler passes per task is easy to implement but introduces the phase-ordering problem. We illustrate this using the *Triangles* example. Starting with `TrH`, we can apply scheduling (T_S) and implementation selection (T_{IS}) in either order, using phase-local optimization heuristics. For T_S, the obvious heuristic is to nest node iterators within conditionals whenever possible; moreover, complex conditionals involving graph data (*e.g.*, **if edges**(a,b)) should be nested within scalar ones (**if** $a < b$) if possible. T_{IS} favors maximal tile usage since tiles encode efficient implementations of HIR statement sequences.

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

3. A Planning-Based Synthesis Framework

In this paper, we show that these parallel code generation problems can be formulated using constraints, and that these constraints can be solved efficiently using *planning*. A STRIPS-style planning problem [9] is specified by an initial state, a goal state, and a set

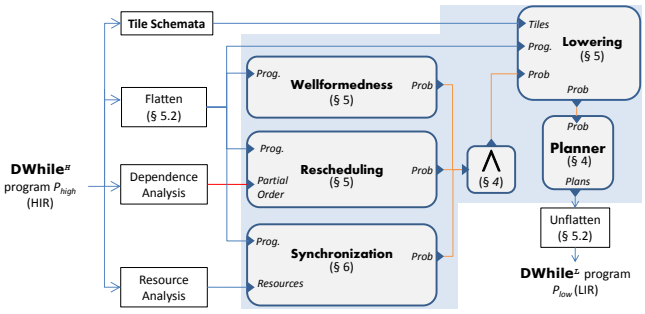

Figure 3: Framework architecture. *Prob* : planning problem.

of actions that can be used to transition from one state to another (§4 provides a detailed definition). The planning problem is to synthesize a sequence of actions that lead from the initial state to the goal state. Properties that a solution must satisfy are encoded by *temporal constraints*. There are two main advantages to this approach. **Integration:** searching for solutions that *simultaneously* solve all constraints avoids the phase-ordering problem and produces better code. **Engineering:** each code generation problem is defined declaratively and succinctly; different correctness and profitability concerns are seamlessly composed together, allowing easy construction and experimentation.

Fig. 3 shows our system, which is parameterized by: (i) the HIR description; (ii) a dependence analysis; (iii) a resource analysis; and (iv) a tile schema for each statement type in the low-level language. At a high level, the system works as follows. The HIR program P_{high} is fed to several planning problem construction units, and each unit emits a planning problem related to a different code generation problem. Individual problems are then combined to define a single composite planning problem that is fed to a planner, which emits the LIR program. One detail is that since planners deal with sequences rather than nested structures, the HIR program is *flattened* by producing an in-order representation of its abstract syntax tree, and this sequence is actually the input to the planning problem construction units. Although our planning-based transformations are rewrites on sequences of actions, they can alternatively be reinterpreted in a more traditional form as tree rewrites on a structured IR. At the other end, the planner produces an in-order representation of the LIR program, which is *unflattened* to produce the actual LIR program.

Our current system has the following planning problem construction units for tasks related to parallel code generation for graph programs: (i) ensuring that the output program is syntactically and semantically correct (*Wellformedness*), (ii) is equivalent to the input program (*Rescheduling*), (iii) is properly synchronized (*Synchronization*), and (iv) is implemented by statements in the low-level language (*Lowering*). Since rescheduling must respect dependences, it takes the results of a *dependence analysis* as input. The *Resource Analysis* module extracts all accesses to shared resources in the program, so that these can be synchronized properly; for our code generation problem, these are accesses to shared-memory variables.

MIS Synthesis as a Planning Problem We now explain the flow though some of the modules in Fig. 3 by showing how to derive `misV1` and `misV2` from `misH`.

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

The ATS planning problem augments $\mathcal{U}$ with a set $\mathcal{L}$ of lock acquisition and lock release statements. $\mathcal{L}$ contains multiple instruction variants for each node to be locked, one per history of previous lock acquisitions. For example, in `misH` node a can be locked before $\mathcal{N}(a)$ or after it. Hence, $\mathcal{L}$ includes '**acq** a **ctx** $\emptyset$' and '**acq** a **ctx** $\mathcal{N}(a)$'. The encoding of ATS planning actions enables only valid combinations in plans, and allows locking customization to specific schedules. For example, in `misV1` the combination is **acq** a **ctx** $\emptyset$, **acq** $\mathcal{N}(a)$ **ctx** a. Such statements encode all the information necessary to perform conflict detection and which locks to release in case of aborts. Temporal constraints enforce global correctness properties of ATS. For example, to encode two-phase locking, which guarantees serializable execution, we require that all locks happen before unlocks. Additionally, to guarantee operator *cautiousness* [25], which enables transactional execution without storing rollback information, we require all locks to occur before shared state updates. ATS is a very good example of the value that planning adds to the field of program transformations. Here, the planning system does not merely find a permutation that reorders statements subject to the partial order dictated by program dependences but *it synthesizes the right sequence of actions that constitute a custom version of a locking protocol for a specific program*.

The input HIR instantiates a set of tile-schemas. Lowering adapts the planning problem to also use tile-related actions. Solutions to the new planning problem encode low-level programs. For example,'$tile[\forall(b)$ **with** rs_1 **ctx** $rs_2] \triangleq$ **acq** rs_1 **ctx** $rs_2, \forall(b)$' combines locking and the $\forall$ evaluation in `misV2` (line 5) .

4. Planning with Temporally Extended Goals

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

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

Action Notation. We lift negation to sets of fluents by writing $\neg S$ for $\{\neg f \mid f \in S\}$. We denote actions by $\langle I \rangle\, o\, \langle O \rangle$ where o is an identifier; $I = I^t \cup \neg I^f$ and $O = O^t \cup \neg O^f$; and I^t, O^t, I^f, and O^f are sets of fluents. Thus, $Pre^t(o) = I^t$, $Pre^f(o) = \neg I^f$, $Add(o) = O^t$, and $Del(o) = \neg O^f$.

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

Plans. A *plan* π is a sequence of actions $o_1, \ldots, o_k$[2] such that $[\![o_k]\!] \circ \ldots \circ [\![o_1]\!]\ \mathcal{I} \supseteq \mathcal{G}$. We write $Plans(P)$ for the set of plans of a planning problem P. For a given plan length, it is possible to efficiently encode the planning problem as a propositional formula, which can be handed to a SAT solver. Shortest plans can be found by searching for plans of increased length [18].

Planning with Temporally Extended Goals. Temporal goals specify the conditions that plans must satisfy. We express such conditions in (a subset of) linear temporal logic (LTL), whose models are the sequences of states generated by the corresponding plans, starting from the initial state. Planning problems may be extended by a set of temporal goals, specified by LTL formulas. In §5 and §6, we define temporal goals via the following operators:

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

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

Inverse Homomorphism. A *macro action* is a sequence of actions, written as $m = o_1; \ldots; o_k$. A sequence of actions can be composed into a single action: $[\![m]\!] = [\![o_k]\!] \circ \ldots \circ [\![o_1]\!]$. We write $\overline{m} = o_1, \ldots, o_k$ for the corresponding sequence of actions identifiers. Given a planning problem P and set of macro actions M over $\mathcal{A}^P$, we wish to obtain another planning problem P_M such that $Plans(P_M) = \{m_1, \ldots, m_k \mid \text{for } i \in [1,k] : m_i \in M \text{ and } \overline{m_1}, \ldots, \overline{m_k} \in Plans(P)\}$. That is, the language $Plans(P_M)$ is induced by the inverse homomorphism h^{-1} where $h(m) \triangleq \overline{m}$. This language can be obtained by transforming the planning problem appropriately, which we denote by $InvHom(P, M) = P_M$.

5. Formal Synthesis Framework

This section shows how compilation tasks other than the insertion of synchronization can be formulated as planning problems. Synchronization is treated in §6.

5.1 The Parametric Language DWhile

We now describe the class of data-intensive programming languages targeted by our synthesis framework. It is parameterized by: (i) type of atomic state-changing statements, (ii) Boolean expressions, and (iii) data range expressions. We call the languages obtained by instantiating these parameters with the grammar shown in Fig. 4(a) DWhile languages. Programs may refer to global variables, defined externally, via update statements. We

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

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

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

Production	Semantic Rules
S ::= **for** $x : r$ **do**^{L} B_1 **od**^{L}	if $x \in bnd(S)$ then error, if $vars(r) \nsubseteq bnd(S)$ then error, $bnd(B_1) = bnd(S) \cup \{x\}$.
| **while** b **do**^{L} B_2 **od**^{L}	if $vars(b) \nsubseteq bnd(S)$ then error, $bnd(B_2) = bnd(S)$.
| **if** b^L B_t **else**^{L} **skip**^{L} **fi**^{L}	if $vars(b) \nsubseteq bnd(S)$ then error, $bnd(B_t) = bnd(S)$.
B ::= S	$bnd(S) = bnd(B)$.
| A	$bnd(A) = bnd(B)$.
A ::= $AtomUpd$	$bnd(AtomUpd) = bnd(A)$.
| **acq** rs_1 **ctx** rs_2^L; A_1	if $vars(rs_1, rs_2) \nsubseteq bnd(A)$ then error. $bnd(A_1) = bnd(A)$.
| **if** b^L A_t **else**^{L} R **exit**^{L} **fi**^{L}	if $vars(b) \nsubseteq bnd(A)$ then error, $bnd(A_t) = bnd(R) = bnd(A)$.
| **for** $x : r$ **do**^{L} A_b **od**^{L}	if $\neg val(r)$ then error, if $vars(r) \nsubseteq bnd(A)$ then error, if $x \in bnd(A)$ then error, $bnd(A_b) = bnd(A) \cup \{x\}$.
R ::= ϵ	
| **rel** rs^L	if $vars(rs) \nsubseteq bnd(R)$ then error.
$AtomUpd$ ::= Upd; R; **commit**	$bnd(Upd) = bnd(R) = bnd(AtomUpd)$.
Upd ::= upd^L	if $vars(upd) \nsubseteq bnd(Upd)$ then error.
| **acq** rs_1 **ctx** rs_2^L	if $vars(rs_1, rs_2) \nsubseteq bnd(Upd)$ then error.
| Upd_1; Upd_2	$bnd(Upd_1) = bnd(Upd_2) = bnd(Upd)$.

(a)

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

$$
\begin{array}{rcl}
U & ::= & upd^L \mid \textbf{commit}^L \mid \textbf{if}\ b^L \mid \textbf{else}^L \mid \textbf{exit}^L \mid \textbf{fi}^L \\
 & \mid & \textbf{while}\ b\ \textbf{do}^L \mid \textbf{for}\ x : r\ \textbf{do}^L \mid \textbf{od}^L \\
 & \mid & \textbf{acq}\ rs_1\ \textbf{ctx}\ rs_2^L \mid \textbf{rel}\ rs^L \mid \textbf{skip}^L \\
F & ::= & U \mid U\ F
\end{array}
$$

(b)

Figure 4: (a) AG for DWhile, and (b) A regular grammar for flat (labelled) DWhile.

use attribute grammars (AG for short) to impart semantic conditions to our parametric language and define functions. A statement **for** $x : r$ **do** S **od** has a dual role: (i) iterating over a range of data values (defined by r), and (ii) introducing a scope in which the local immutable variable x is bound and initialized to the single-value range r. We use the predicate $val(r)$ to test whether a range expression denotes a single value. A statement **var** $x := r; S$ introduces a scoped local variable and is syntactic sugar for **if** $\neg$**empty**(r) **for** $x : r$ **do** S **od fi**. The AG ensures that: (i) local variables are only accessed within their scope, never hiding other variables; (ii) there exists at most one atomic section[3]; and (iii) updates appear only in the inner-most nesting level[4]. In the sequel, we fix a high-level language, DWhile^H, and a low-level language, DWhile^L. The synchronization-related statements **acq** rs_1 **ctx** rs_2 and **rel** rs, which are explained in §6, do not appear in the input program. We slightly abuse notation by conflating meta variables and their terminals/non-terminals. Finally, we abbreviate "**if** b S **else** N **fi**" (N is either **exit** or **skip**) by "**if** b S **fi**".

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

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

5.2 Flattening and Unflattening DWhile Programs

Our synthesis technique operates over a deconstructed form of DWhile programs, which we call *flat programs*, defined by the regular grammar in Fig. 4(b). Flat programs consist of a sequence of *units*. The function *flat* : DWhile $\rightarrow F$ labels each unit in the input program and returns them in order, taking care to associate the same label with units matching a given HIR statement: $\{\textbf{if}\ b^L, \textbf{else}^L, \textbf{exit}^L, \textbf{fi}^L\}$, $\{\textbf{if}\ b^L, \textbf{else}^L, \textbf{skip}^L, \textbf{fi}^L\}$, $\{\textbf{while}\ b\ \textbf{do}^L, \textbf{od}^L\}$, and $\{\textbf{for}\ x : r\ \textbf{do}^L, \textbf{od}^L\}$.

For the remainder of this section, we fix a labelled high-level program $S \in \text{DWhile}^H$ and $F^\flat = \mathit{flat}(S)$.

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

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

5.3 Encoding Wellformedness by a Planning Problem

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

Lemma 1. *Fig. 5 defines the planning problem WF(S) whose plans are all wellformed permutations of flat(S):*

$$\mathit{Plans}(\mathit{WF}(S)) = \{F^{\flat\prime} \mid F^{\flat\prime} \approx^\Pi \mathit{flat}(S), F^{\flat\prime}\ \textit{is wellformed}\}\ . \quad (1)$$

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

5.4 Encoding Rescheduling by a Planning Problem

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

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

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

$$\mathcal{G}^{\mathit{Depend}(S)} \triangleq \{u_i \sqsubset u_j \mid i \preceq j \in \mathit{Dependences}(S), i \neq j\}\ . \quad (2)$$

Lemma 2. *Define Equiv(S) = WF$(S) \wedge \mathcal{G}^{\mathit{Depend}(S)}$ to be WF(S) extended with the dependence analysis goals. The plans of Equiv(S) are all flat programs that can be unflattened to a dependence-*

Attribute	Value Type
$\mathit{prn}(\cdot)$	$\wp(U^{\mathit{Label}} \times U^{\mathit{Label}})$ (synthesized)
$\mathit{ite}(\cdot)$	$\wp(U^{\mathit{Label}} \times U^{\mathit{Label}} \times U^{\mathit{Label}})$ (synthesized)
$\mathit{bndF}(\cdot)$	$\wp(\mathcal{B}[\mathit{Var}])$ (synthesized)
$\mathit{bndA}(\cdot)$	$\wp(\mathcal{A})$ (synthesized)

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

(a)

$\mathcal{F}^{\mathit{WF}(S)}$	=	$\{\mathit{flat}(S)\} \cup \mathit{bndF}(S)$
$\mathcal{A}^{\mathit{WF}(S)}$	=	$\{\langle\rangle\ u\ \langle \mathit{only}(u)\rangle\ \mid u \in \mathit{flat}(S)\} \wedge \mathit{bndA}(S)$
$\mathcal{I}^{\mathit{WF}(S)}$	=	$\emptyset$
$\mathcal{G}^{\mathit{WF}(S)}$	=	$\bigcup_{i=1}^{5} \mathcal{G}_i^{\mathit{WF}(S)}$
$\mathcal{G}_1^{\mathit{WF}(S)}$	=	$\{\mathsf{F}!\, u \mid u \in \{\mathit{flat}(S)\}\}$
$\mathcal{G}_2^{\mathit{WF}(S)}$	=	$\{po \sqsubset pc \mid (po, pc) \in \mathit{prn}(S)\}$
$\mathcal{G}_3^{\mathit{WF}(S)}$	=	$\{i \sqsubset e \sqsubset f \mid (i, e, f) \in \mathit{ite}(S)\}$
$\mathcal{G}_4^{\mathit{WF}(S)}$	=	$\{p \otimes p' \mid p, p' \in \mathit{prn}(S), p \neq p'\}$
$\mathcal{G}_5^{\mathit{WF}(S)}$	=	$\{(i \sqsubset po \sqsubset e \Rightarrow pc \sqsubset e) \mid$ $(po, pc) \in \mathit{prn}(S), (i, e, f) \in \mathit{ite}(S)\}$.

(b)

Figure 5: (a) AG for computing delimiters, fluents for tracking bound variables, and actions for tracking sets of bound variables over units. To avoid clutter, we handle productions of similar form together by letting the meta non-terminals N, N_1, N_2 stand for potions of the right-hand sides of productions in Fig. 4(a). (b) planning problem for wellformedness. We write $\{\mathit{flat}(S)\}$ for the set of units in the sequence $\mathit{flat}(S)$ and $\mathcal{B}[\mathit{vars}(e)]$ for the set $\{\mathcal{B}[z] \mid$ variable z appears in $e\}$.

equivalent program:

$$\mathit{Plans}(\mathit{Equiv}(S)) = \{F^{\flat\prime} \mid S \approx^{\preceq} \mathit{unflat}(F^{\flat\prime})\} \ . \tag{3}$$

5.5 Encoding Lowering by a Planning Problem

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

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

Multi Tiles. Tiles implementing loops and conditions usually cover only one delimiter of the loop or condition statement. To cover the remaining delimiters, we couple them with additional tiles. These tiles appear in tandem, covering all of the delimiters of a high-level statement (2 for loops and up to 3 for conditionals). We call these *multi tiles*.

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

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

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

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

$$\mathit{Lower}_M(S^H) \triangleq \mathit{InvHom}(\mathit{Equiv}(S^H), M) \ . \tag{4}$$

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

6. Planning-Based Synchronization

We now turn to parallel DWhile programs and the problem of synchronizing them both correctly and efficiently. Let $[\![r]\!]\sigma$ mean the set of objects denoted by range expression r in a program state σ. To support parallelism, we assume that the outermost loop takes parallel semantics.[5] We have that

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

where $[\![r]\!]\sigma = \{v_1, \ldots, v_k\}$ and $[\![S(x := v_i)]\!]$ denotes the **atomic** execution of the loop body in the context where x is bound to v_i. The tasks $S(x := v_i)$ execute in parallel while ensuring serializability. When a task aborts due to a conflict, it is re-executed. General techniques for synchronizing arbitrary programs usually rely on variants of transactional memory [12, 25]. We consider these approaches as a reference for comparison. We define an efficient speculative lock-based synchronization technique (§6.1), dubbed ATS, that can achieve better performance than the reference synchronization techniques. Finally, we define a planning problem (§6.2) to automatically insert synchronization statements that realize ATS.

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

Resources and Resource Expressions. We assume an analysis that computes for each unit u the sets of expressions $\mathit{rd}(u)$ and $\mathit{wt}(u)$, which denote the (overapproximation of) shared runtime objects that it may directly access for reading and writing, respectively. We define $\mathit{res}(u) = \mathit{rd}(u) \cup \mathit{wt}(u)$. For example, $\mathit{res}(mis) = \{\mathrm{a}, \mathcal{N}(\mathrm{a})\}$ for `misH` in Fig. 1(b).

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

6.1 Alias Tracking-Based Synchronization (ATS)

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

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

"**acq** rs_1 **ctx** rs_2" attempts to acquire the objects in $[\![\mathit{rs}_1]\!]$ where rs_2 denotes the set of objects already owned by the current task, which we call *context*. If $[\![\mathit{rs}_1]\!]\sigma \subseteq [\![\mathit{rs}_2]\!]\sigma$ the operation succeeds; otherwise, if locking rs_1 fails then the objects denoted by rs_2 and the portion of rs_1 successfully acquired are unlocked and the task is re-executed.

We say that a DWhile program is correctly synchronized by ATS when all shared data structures are linearizable and the following conditions hold: ($C1$) **Isolation:** each resource is acquired before accessed; ($C2$) **Two-phase locking:** all unlocks happen after all locks; ($C3$) **Release:** all locks are released when execution aborts or commits; ($C4$) **Cautious:** all locks are acquired before any updates occur, which ensures that on abort, no rollback operations are required to restore state to the one at the beginning of the task; and ($C5$) **Locks tracking:** "**acq** rs_1 **ctx** rs_2" statements execute with the correct context. $C1$–3 ensure serializability. In Fig. 1(c), `misV1` is correctly synchronized by ATS[6]. It is obtained from `misH` by applying instrumentation, as explained next.

6.2 Instrumenting DWhile for ATS via Planning

Fig. 6 defines the planning problem $\mathit{Synch}(S)$ used for instrumenting DWhile programs, which ensures that $C1$–5 hold on every execution path.

Encoding a Lockset Analysis. Let $\mathit{res}(S)$ denote the set of all resources in S. We encode a flow-sensitive dataflow may-analysis, which tracks the set of acquired resources via the powerset lattice $\langle \wp(\mathit{res}(S)), \subseteq, \cup, \cap, \emptyset, \mathit{res}(S) \rangle$. We encode lattice elements by the set of fluents $\{\mathit{locked}[r] \mid r \in \mathit{res}(S)\}$ and define the shorthand

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

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

Figure 6: ATS problem. To avoid clutter, set formers don't specify that: $L \in \mathit{Label}, u \in \mathit{flat}(S), \mathit{rs}, \mathit{rs}_2 \subseteq \mathit{res}(S), \mathit{rs}_1 \subseteq \mathit{res}(S) \setminus \{\}$.

notation $\mathcal{D}[\mathit{rs}] \triangleq \mathit{locked}[\mathit{rs}] \cup \neg\mathit{locked}[\mathit{res}(S) \setminus \mathit{rs}]$ to refer to a specific lattice element in action pre-/postconditions. The definition of $\mathcal{A}_2$ requires that expressions rs_1 have not been acquired and in such a case adds the correct dataflow facts to the postcondition. $\mathcal{A}_3$ ensures that a **rel** statement releases all locks. To capture the control flow of conditions, $\mathcal{A}_4$ records the lattice element upon entry to the condition and $\mathcal{A}_5$ re-establishes that element in the else branch.

Notice that since update statements may only appear before the commit statement (by a control dependence), then $\mathcal{G}_3$ restricts lock statements to appear before **commit**. Together with $\mathcal{G}_4$, they restrict lock statements to appear inside the atomic section.

We ensure $C1$ via the fluents $\{\mathit{read}[r], \mathit{write}[r] \mid r \in \mathit{res}(S)\}$ and $\mathcal{G}_1$. We ensure $C2$ by $\mathcal{G}_2$. We ensure $C3$ by $\mathcal{A}_6$ — all locks should be released as a precondition to committing or exiting a condition via the else branch (which is where a **exit** unit would appear). We ensure $C4$ by $\mathcal{G}_3$. Finally, we ensure $C5$ by having plans that choose **acq** rs_1 **ctx** rs_2 statements that satisfy the lockset analysis defined above.

7. Experimental Evaluation

To demonstrate the competitiveness of our approach, we used our system to generate a large number of parallel program variants for four complex graph problems: maximal independent set, triangle counting, preflow-push, and connected components. We instantiate the planning framework with a SATPLAN-based planner that we implemented, which is parametric on the used SAT solver (in our experiments we use the Sat4J solver). We note that our planning formulation is generic and agnostic to the planning algorithm used.

7.1 Implementation and Experimental Details

For each problem we consider an implementation space consisting of different Elixir schedules capturing algorithmic insights, and different plans capturing different implementation-level insights. The Elixir scheduling language allows programmers to express both the *static* and *dynamic* components of the schedule. Intuitively, Elixir generates a (parallel) loop iterating over the contents of a worklist W containing the redexes that remain to be executed. The dynamic component of the schedule denotes the order in which the we iterate over the contents of W. The static component corresponds to a hard-coded schedule of multiple operator instances that are executed on each loop iteration, starting from an initial (potentially partial) redex. Each iteration may conditionally schedule new redexes. Different plans correspond to different implementations of this loop iteration. Elixir relies on a host runtime to provide a parallel loop construct. The synthesized code also assumes there is a graph data structure that supports a generic API with methods such as '**for** b **:** `Succ` (a) **do**'. In our experiments we used parallel loops, graphs, and work-lists from the Galois runtime. The graph implementations are variants of the Compressed Sparse Row (CSR) format. Galois provides its own synchronization but we disabled this feature and used ATS in the synthesized code.

We performed our experiments on a 40-core machine with Intel Xeon E7-4860 hyper-threaded processors, with 24MB of L3 cache and 128GB of main memory. The operating system is Scientific Linux 6.3 and the compiler is GCC 4.8.1 (-O2). We used four kinds of graphs: (i) the DIMACS USA road network (24M nodes and 58M edges), (ii) the *wikipedia-2007* graph (3.5M nodes and 8M edges), (iii) the *rmat24* graph (a=0.5,b=c=0.1,d=0.3), which is a synthetic scale-free graph (16M nodes,268M edges), and (iv) a number of random graphs dubbed *randX* (2^X nodes,4×2^X edges). Since all our input problems are defined over undirected graphs we preprocess the inputs to add symmetric edges, in case they are not already present in the graph. We also removed multi-edges (otherwise we would not even have a graph).

7.2 Evaluation Methodology

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

Elixir is the first system that automatically synthesizes efficient parallel implementations for the complex domain of sparse graph problems, permitting it to optimize implementations for a given input. To demonstrate its effectiveness in generating efficient parallel implementations and enabling input adaptivity, we present an in-depth performance analysis using the following methodology. For each problem we generate a large number of variants and then use search to find the best-performing implementations for a number of interesting inputs. We then compare these best-performing solutions against hand-optimized implementations by expert programmers. We present graphs showing the runtime distribution of Elixir solutions, as well as graphs comparing their performance against the manual implementations. In order to improve clarity of exposition in the comparison graphs we normalize runtimes against the fastest single-thread runtime across all implementations (manual and synthesized) and we report speedups over that baseline. All reported times are the medians of five runs and baseline times are presented in figure captions.

The hand-optimized solutions we compare against are implemented on top of the state-of-the-art Galois and Cilk frameworks. While we cannot know what would be the fastest hand-tuned implementation for the problems we study, we believe that the solutions we compare against are very competitive. The interested reader is referred to [28] for a recent study by Intel, which compares existing graph frameworks and shows that Galois performs competitively

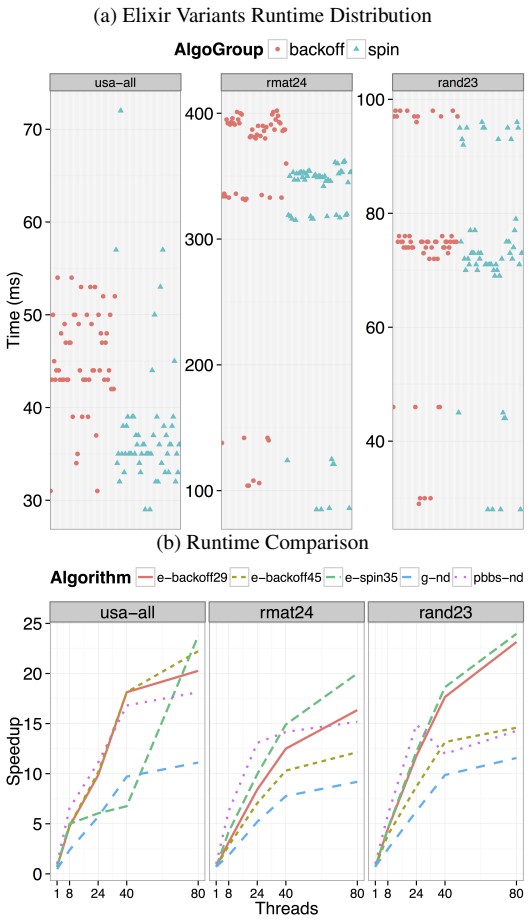

Figure 7: *MIS* variants runtime-distribution and comparison with hand-written codes. Base-time (ms): 689 (usa-all), 1700 (rmat24), 671 (rand23).

against other frameworks as well as hand-written expert implementations. Similarly, [24] compares Galois favorably with Ligra [32] and GraphLab [20].

Additionally, we note that some of the solutions we compare against use customized, elaborate synchronization requiring expert parallel-programming skills. For example, the hand-written connected-components solutions on top of Galois, as well as the *MIS* Cilk variant, use specialized lock-free synchronization, which is not automatically supported by Galois.

7.3 Maximal Independent Set

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

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

Conflict resolution: The *spin* strategy keeps trying to dispatch the same work-item wi until it succeeds; the *back-off* strategy chooses a different work-item and inserts wi in a special abort queue where it is processed by a special thread to guarantee forward progress (the default Galois policy).

The product of all these choices gives us a set of variants that use different ways to grow and shrink the atomic section corresponding to the *match* operator, and to resolve conflicts. Fig. 7a presents the distribution of the best runtimes for all Elixir variants on three input graphs. In Fig. 7a we cluster variants in two families, based on the conflict resolution policy. Fig. 7b presents comparisons of the best Elixir variants for each input and family against hand-written implementations. For performance comparisons, we used the Galois program `g-nd`, which employs the `match` operator but uses the default transactional execution scheme of the Galois system implemented with *stamp-and-log* within the Galois data structure library; conflicts are resolved using back-off. Additionally, we used *pbbs-nd*, the non-deterministic parallelization from the Problem Based Benchmark Suite (PBBS) [4]. *pbbs-nd* is a lock-free parallelization using the Cilk runtime (compiled with ICC 13.1). In Fig. 7b the name *e-spin*$\langle i \rangle$ (*e-backoff*$\langle i \rangle$) denotes the i-th spin-based (backoff-based) Elixir variant. The key observations are as follows.

First, for all inputs, spinning is a better conflict resolution policy than back-off, the default Galois policy. The best synthesized spin version, *spin-35* outperforms the Galois version by more than a factor of 2. This shows the advantages of customizing synchronization policies to applications. Additionally, as shown in Fig. 7a, even among Elixir variants spin-based variants tend to be better performing than backoff-based variants. One could expect that backoff is always better since it prevents live-locks from happening. However, depending on the sparsity of the input graphs and the operator structure, the probability of live-locks may be small. In that case, it may be more beneficial to simply retry dispatching the same redex, instead of paying the overhead of an always-on runtime mechanism that prevents livelocks.

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

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

7.4 Triangle Counting

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

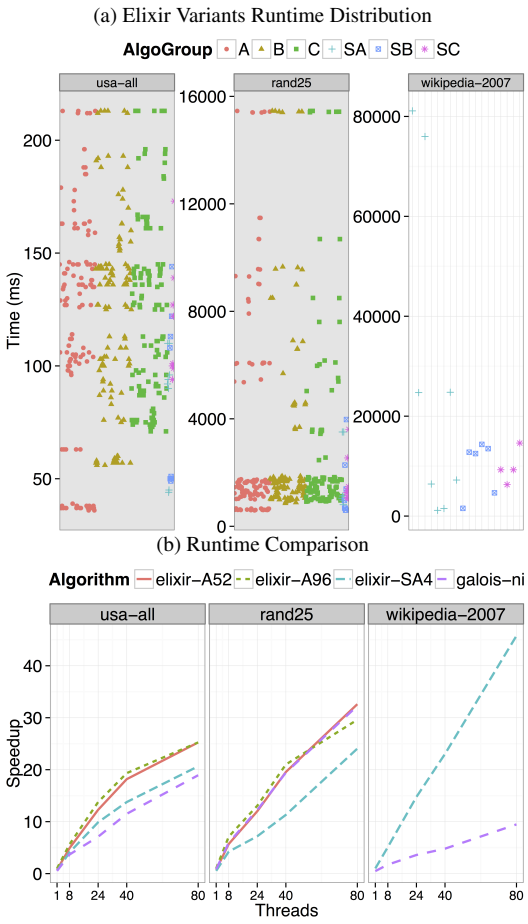

Figure 8: *Triangles* variants runtime-distribution and comparison with hand-written code. Base-time (ms): 909 (usa-all), 19367 (rand25), 52590 (wikipedia-2007).

sorted in increasing order. Communicating this information to our system allows the planner to use tiles encoding specialized strategies of iterating over the neighbors of each node.

These two design parameters give us six different algorithm families: *A, B, C, SA, SB, SC*, with the first letter denoting the starting node of the blocked operator, and the conditional prefix S denoting whether sorted property is taken into consideration or not. For each family, our planner enumerates a number of solutions that correspond to the different schedules of evaluating the operator. In total, we have 384 variants. Fig. 8a presents the distribution of best runtimes for all the synthesized Elixir variants on three input graphs. Fig. 8b compares the best performing variants against a version of the *node-iterator* algorithm [29] (`galois-ni`) implemented using the Galois system. The key observations are:

First, no single variant performs best for all input graphs. In order to get the best performing solution for each input we need to customize the implementation to properties of the input graph. The relative performance of Elixir variants in Fig. 8a shows that the best performing variants for the road network and random graphs use simple implementations of iterating over the node neighbors. Since the average node degree is small and uniform, a simple strategy of iterating through all neighbors can incur less overhead than a more elaborate strategy that tries to find the right set of neighbors to start iterating over. This effect is more obvious in the road network graph, while in the random graph the differences between the different families are smaller. When we move to the scale-free Wikipedia graph however, which has few nodes with very high connectivity, the more elaborate iteration pattern is essential in getting performance. For this input, we do not report times for variants in the A, B, C families because their running time (80 threads) exceeded a timeout of 300 seconds.

Second, we note that the schedule of evaluating the operator constraints greatly affects performance. For example, as we can see in Fig. 8a variants of the A, SA families outperform Elixir variants in SB across all examined inputs. Moreover, there is variation even within the same family. Studying the Wikipedia graph results reveals that the best Elixir variant in the SB family, which takes 1.5 seconds, iterates over all neighbors a of b, and for each a examines all potential c's. A sub-family that first iterates over all neighbors c, and for each c iterate again over the neighbors of b to find an appropriate a gives a best runtime of 12.8 seconds. Similar performance variations exist in the SA family, with the best variant being more than 4 times faster than `galois-ni` on Wikipedia (5.5 seconds), while being slower on the random graph.

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

7.5 Connected Components

The classic formulation [14] applies the *hook* and *compress* operators non-deterministically up to a fixpoint. A popular scheduling heuristic, which is adapted both in the PRAM literature [31] and in mutlicore implementations [6], is to alternate *hook* and *compress* rounds, selecting the number of applied operators in each round heuristically. We generate 3200 variants by considering different operator schedules yielding different mixes of operators per round, and different plans for each schedule.

Fig. 9 presents a comparison of the best variants for each input against two manually parallelized solutions in Galois. The first, *g-uf*, is based on the union-find-based algorithm [7] (union and find are merely *hook*/*compress* schedules). The second solution, *g-l*, is a parallelization of the label-propagation algorithm, as implemented in the Ligra framework [32]. The *g-l* solution relies on an implementation of the Ligra API on top of Galois, which performs competitively with the original Ligra implementation [24]. Both solutions use lock-free synchronization and represent a level of performance that cannot be obtained (automatically) by systems such as Galois, but instead requires expert parallel programming knowledge.

In all cases, the synthetic variants perform competitively with the hand-written implementations. For the road-network graph, *g-uf* takes 59% of the runtime of the best Elixir variant, and both are roughly two orders of magnitude faster than *g-l*. In the other cases, *g-l* outperforms *g-uf* and the best Elixir variant is faster than *g-l* (takes 95% of the *g-l* time on *rmat24* and 67% on *rand23*). For lack of space we do not present analytical plots of the distribution of best runtimes for the Elixir variants. The best runtime difference between the best and worst variants is at most $\times 3.3$ for *rmat24*, $\times 2$ for *usa-all*, and $\times 3.5$ for *rand23*.

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

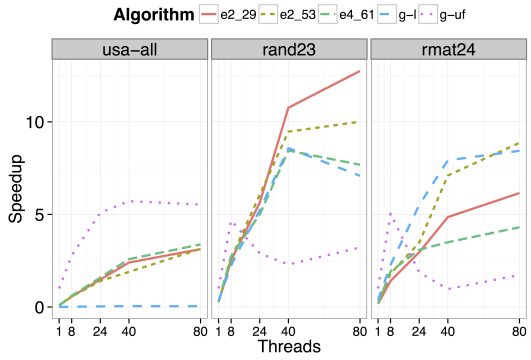

Figure 9: Elixir CC variants comparison with hand-written codes. Base-time (ms): 2007 (usa-all), 2282 (rand23), 7813 (rmat24).

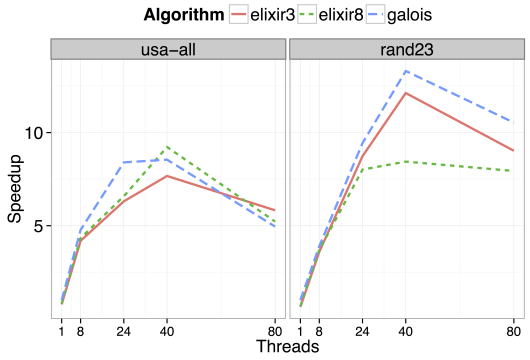

Figure 10: Elixir preflow-push variants comparison with Galois implementation. Base-time (ms): 8921 (usa-all), 15836 (rand23).

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

7.6 Preflow-Push

The preflow-push algorithm is an efficient solution to the maximum-flow problem. The main algorithm kernel non-deterministically applies the *push* and *relabel* operators until reaching a fixpoint. We guide Elixir to generate solutions closely matching the static schedule of the *discharge* kernel in the *relabel-to-front* algorithm [7]. This schedule considers a node a and alternates between pushing flow to all its neighbors b and relabeling a. In addition, we use a worklist with *fifo* policy (chunk-size: 32) to dynamically schedule new redexes. For this schedule, we select the first 50 variants returned by the planner. We compare their performance against a hand-written solution on top of Galois, which uses the same worklist policy and a roughly similar static operator schedule. Fig. 10 shows the performance of the best-performing Elixir solutions for each input and the Galois code. We observe that for both inputs Elixir variants are competitive with the hand-written code. On the *usa-all* graph *elixir8* is the fastest and its runtime is roughly 93% of the time taken by the hand-written code. On the *rand23* graph the Galois code takes roughly 91% of the time taken by *elixir3* and 63% of the time taken by *elixir8*. For lack of space we do not present analytical plots of the best runtimes distribution of the Elixir variants. We note however, that the time of the best Elixir variant is 60% of the time of the worst Elixir variant for the *rand23* graph and 72% for the *usa-all* graph.

The three solutions differ primarily in the synchronization implementation. The Elixir variants use ATS to synchronize individual operators participating in the schedule, while the Galois variant considers the entire static schedule as a single composite transaction synchronized by *stamp-and-log*. The static schedule constitutes a cautious composite operator [25] (each individual operator is also cautious). Consequently, no state rollback is necessary to in the case a transaction aborts. A notable difference between the two Elixir variants is the synchronization of *relabel*. This operator checks a predicate over a node and its neighbors, similar to *match* in *MIS*. In *elixir8* the locking of neighbors is performed incrementally as each neighbor is visited, while in *elixir3* locking and predicate evaluation are not fused. Elixir currently does not support synthesizing solutions that emulate the synchronization of the Galois variant, which shares locks across operators. This requires encoding a more complete dataflow analysis in our planning framework, and is a subject of future work.

8. Related Work

Synthesis. [13] synthesizes graph programs from logical problem specifications at a much higher level of abstraction than Elixir. That work is not concerned with concurrency. [10] synthesizes *loop-free* programs of component compositions using SMT-based reasoning. [23] generates parallel tree traversals for attribute grammar evaluation. Our work could be used synergistically to parallelize individual traversals. [2] explores SIMD loop synthesis by extracting the equivalence relation from the loop body and using it as specification to synthesize the parallel loop. Our technique can synthesize code with loops but is less ambitious in the sense that it lowers from a high to a low-level program rather than synthesizing from a pre-post specification. Other work focusing on concurrency is Sketching [33] and Paraglide [35]. Their goal is to start from a (possibly partial) sequential implementation 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 solutions [36]. Our plans encode correct programs by construction. Not all plans encode the tightest synchronization to optimize different aspects of the computation. [15] uses planning to generate straight-line code of library API calls, and uses programmer-compiler interaction to prune undesirable compositions. Our work handles more general programs and instrumentation transformations, but this work also focuses on information flow between planner generated and host application code. [11] synthesizes concurrent data-structures from relational specifications by generating a set of plans and choosing the most profitable ones.

Compiler Techniques. Guard encapsulation [3] describes transformations similar in spirit to the ones captured by the tiles we use. Several papers have tackled the phase-ordering problem by using Lightweight Modular Staging and rewrite rules to optimize programs [27, 34]. Equality preserving rewrite rules used by Tate et al. cannot support synchronization synthesis with global constraints (e.g. cautiousness). ILP-based techniques have been used to generate embedded processor code for basic blocks and for software pipelining [8].

Superoptimization. Superoptimizers find optimal straight-line machine code sequences. [21] exhaustively enumerates programs of increasing length or cost, which limits applicability to small sequences. [30] uses MCMC sampling as a search technique and achieves better scaling. Denali [16, 17] uses SAT-based constraints

and equality-preserving rewritings to find an optimal loop-free sequence for a guarded multi-assignment input program. Compared to our solution, none of these approaches handle more general programs involving conditionals and loops. Denali does not handle dependences and does not consider instrumentation transformations.

Synchronization. Several papers have used static analysis to automatically insert locking code such as order-and-spin locks [22] and pessimistic, multi-grain locks [5]. The ATS speculative locking is more difficult to implement but is better for some applications (see §7).

9. Conclusion and Future Work

This paper presented a methodology based on automated planning for synthesizing correct and efficient parallel graph programs from high-level specifications. Planning provides an elegant and concise way of defining program transformations and composing them to deliver efficient implementations. We experimented with four problems and generated code competitive with state-of-the-art hand-written implementations. Our results show that it is indeed possible to provide programming abstractions allowing high-productivity and good performance for this complex problem domain.

Our current work is a first step towards the goal of automatically and efficiently producing high-performance input-customized solutions for graph analytics problems. Our synthesis techniques allow programmers to automatically generate competitive parallel solutions and enable for the first time the use of search techniques to find the best one for a specific input. Currently, however, we simply exhaustively search among the existing implementations to find the best one — a technique that is obviously non-scalable. The next challenge to address is given a new input, to effectively search the implementation space to find the best one. It is possible that machine-learning techniques like automatic clustering and classification might help; we plan to examine such approaches in future work.

Finally, we believe that planning-based synthesis can be useful in other problem domains and other compilation-related tasks; we plan to explore these possibilities in future work.

Acknowledgements

We thank our shepherd Michelle Strout, the reviewers, Noam Rinetzky, Josh Berdine, and all members of the Galois group for their useful feedback.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.

[2] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to SIMD loop synthesis. PPoPP '13, 2013.

[3] A. J. C. Bik and H. A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *ICS*, 1993.

[4] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. PPoPP '12, 2012.

[5] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*. ACM, 2008. ISBN 978-1-59593-860-2.

[6] G. Cong, G. Almasi, and V. Saraswat. Fast pgas connected components algorithms. PGAS '09. ACM, 2009.

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

[8] M. Eriksson and C. Kessler. Integrated code generation for loops. *ACM Trans. Embed. Comput. Syst.*, 11S(1), June 2012.

[9] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, 1971.

[10] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. PLDI '11, 2011.

[11] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[12] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*. ACM, 2008.

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

[14] J. F. JaJa. *An introduction to parallel algorithms*. Addison Wesley, 1992.

[15] T. A. Johnson and R. Eigenmann. Context-sensitive domain-independent algorithm composition and selection. PLDI '06, 2006.

[16] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[17] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28(6), 2006.

[18] H. A. Kautz, B. Selman, et al. Planning as satisfiability. *ECAI*, 1992.

[19] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? WWW '10, 2010.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[21] H. Massalin. Superoptimizer: A look at the smallest program. In *ASPLOS*, 1987.

[22] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*. ACM, 2006.

[23] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Parallel schedule synthesis for attribute grammars. PPoPP '13, 2013.

[24] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[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. Prountzos, R. Manevich, and K. Pingali. Elixir: A system for synthesizing concurrent graph programs. OOPSLA, 2012.

[27] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL*, 2013.

[28] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. SIGMOD, 2014.

[29] T. Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Universität Karlsruhe, 2007.

[30] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. ASPLOS '13, 2013.

[31] Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[32] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. PPoPP '13, 2013.

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

[34] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. In *POPL*, 2009.

[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.