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. 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 operation 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

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

1This 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.
In §6, we present Alias 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 perform generic runtime-based solutions for speculation.

We compare our synthesized solutions to those written by expert programmers and show 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 \( S \) if 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 \textit{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, \( N(a) \), is Matched. When such a subgraph, dubbed a \textit{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 \texttt{forall} predicate (line 6) holds if \( sb \neq \text{Matched} \) is true for all \( b \in N(a) \). The \texttt{map} instruction (line 8) updates to NMatched the status of every neighbor of \( a \) distinct from \( a \) (nodes may have self-loops). Finally, a \texttt{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 \( a : \text{nodes} \)' statement (line 1) iterates over all nodes and executes match transactionally for each \( a \). Fig. 2 shows another HIR program, \( T_{\text{tri}} \), 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.

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, conjuncts, disjuncts, 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 \( p_1(a) \land p_2(b) \) (e.g., lines 5-6 of the MIS code), which can be implemented by HIR of the form \( i f \ p_1(a) \land p_2(b) \ldots \) or of the form \( i f \ p_2(b) \) \( i f \ p_1(a) \ldots \). 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. \( T_{\text{tri}} \) and \( T_{\text{tri1}} \) show an example. Since the predicates \( a < b \) and \( \text{edges}(a, b) \) 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 \texttt{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.

\textbf{Implementation Selection} It may be possible to improve perfor-
mane 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 \texttt{for b : nodes do if
edges(a,b)...} can be implemented more efficiently by the code
\texttt{for b : Succ(a) do...}, where \texttt{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 \texttt{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,
TrL

\texttt{edges}(a,b)\texttt{is} \texttt{false} \texttt{for} most pairs of nodes \((a,b)\), \texttt{so} the
\texttt{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.

\textbf{Implementation Selection} It may be possible to improve perform-
ance 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 \texttt{for b : nodes do if
\texttt{edges}(a,b)...} can be implemented more efficiently by the code
\texttt{for b : Succ(a) do...}, where \texttt{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 \texttt{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 \[\texttt{first}, \texttt{last}\] and checking whether \(a < b\)
for each \(b\), we can use a custom iterator \texttt{for b : sortSucc(a) do}
that initially performs binary search to find the first element \(b_{\texttt{first}} : a < b_{\texttt{first}}\), and then linearly scans all nodes in \([b_{\texttt{first}}, \texttt{last}]\), which
definitely satisfy this constraint.

\textbf{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 specula-
tive locking since it is used in existing graph frameworks [20, 25].
At a high level, correct parallel execution of match in MRS 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 appro-
priately, and \(iv\) release all the locks. If a lock cannot be acquired
in step \((i)\), all currently held locks are released, and \texttt{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 \texttt{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 perform-
ing any checks might perform better since it reduces wasted com-
putation because of aborts. Which version performs better there-
fore may depend on the graph structure, the thread count, etc. Simi-
lar choices arise in steps \((iii)\) and \((iv)\). Fusing the status updates
with lock releases results in tighter atomic sections and fewer con-
flicts potentially, whereas the baseline version may permit the use
of vector store instructions. Synchronization therefore introduces
new scheduling opportunities.

Moreover, implementing any of these variants requires book-
keeping code to keep track of locks acquired by an operator execu-
tion. An operator-agnostic generic implementation is the \texttt{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 syn-
chronization 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 informa-
tion 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. \texttt{misVi} is an ATS
synchronized version of \texttt{miVi}. In line \(5\), \(N(a)\) is locked in a con-
text where only \(a\) is locked (\texttt{ctx a}). We need to perform a \texttt{acq(b)}
only for \(b \in N(a)\) such that \(b \neq a\). This is because \(a\), which is
already locked, may be aliased to \(b\) — elements of \(N(a)\) are not
aliased to one-another, so no further checks are needed. If \texttt{acq(b)}
fails, then the thread definitely does not own \(b\) and a conflict oc-
curs. 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 \texttt{ctx a}. If it’s false, we simply release
\(a\) (line 11) and terminate operator execution. Statically computing
this information allows simply emitting an ‘\texttt{rel a}’, obviating the
need for a runtime log.

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

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

\section{A Planning-Based Synthesis Framework}

In this paper, we show that these parallel code generation prob-
lems can be formulated using constraints, and that these constraints
can be solved efficiently using \texttt{planning}. A STRIPS-style planning
problem [9] is specified by an initial state, a goal state, and a set
of actions that can be used to transition from one state to another (§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\text{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) 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 through 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 U is fed to the problem construction units. Different misH schedules correspond to permutations of 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 ∈ U corresponding to ‘if a = Unmatched’ and ‘status(a) := Matched’, respectively. We encode the control dependence between i, j using the temporal constraint ¬j 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 U with a set L of lock acquisition and lock release statements. 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 N(a) or after it. Hence, L includes ‘acq a cxt 0’ and ‘acq a cxt 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 cxt 0, acq N(a) cxt 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[ν(b)] with rs1, cxt rs2 ∠ acq rs1, cxt rs2, ν(b)’ combines locking and the ∠ 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 = (F^P, I^P, A^P, G^P) where F^P is a set of propositional facts, called fluents; I^P ⊆ F^P represents the initial state; A^P represents the set of actions (defined next); and G^P ⊆ F^P represents the goal. In the sequel, we shall drop the superscript P when it is obvious from the context. An action o ∈ A is represented by an identifier Id(o) and four sets of propositions called the Add, Del, Pre^1, and Pre^2. Add describes the fluents that o makes true, Del, the fluents that o makes false, Pre^1 (Pre^2), 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 ¬S for \{ f | f ∈ S \}. We denote actions by (I) o(\{O(O (O(O^I, O^I, O^I, O^I).

)}), where o is an identifier; I = I^I ∪ ¬I^I and O = O^I ∪ ¬O^I; and I^I, O^I, I^I, and O^I are sets of fluents. Thus, Pre^1(o) = I^I, Pre^2(o) = ¬I^I, Add(o) = O^I, and Del(o) = ¬O^I.
States and State Transformers. A state $\sigma \subseteq \mathcal{F}$ represents a truth-assignment $\sigma^o : \mathcal{F} \rightarrow \{0,1\}$ such that $\sigma^o(p) = 1$ if and only if $p \in \sigma$. An action $a \in A$ denotes a partial state transformer $\lceil a \rceil : \Sigma \rightarrow \Sigma$ such that $[a]_0 = \sigma^o$ if $Pre^l(\sigma) \subseteq \sigma$, $\sigma' \cap Pre^l(\sigma) = \emptyset$, and $\sigma' = (\sigma \setminus Del(a) \cup Addl(a))$ hold.

Plans. A plan $\pi$ is a sequence of actions $a_1, \ldots, a_k$ such that $[a_1]_0 \ldots \circ [a_k]_I \sqsupseteq \mathcal{G}$. 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 $\S$ 5 and $\S$ 6, we define temporal goals via the following operators:

$$\begin{align*}
\text{Formula} & \quad \text{Description} \\
F & \equiv \varphi \quad \text{occurs at least once.} \\
\varphi_1 \lor \varphi_2 & \equiv \varphi_1 \text{ or } \varphi_2 \text{ occurs exactly once.} \\
\varphi_1 \land \varphi_2 & \equiv \varphi_1 \text{ and } \varphi_2 \text{ occurs at least until } \varphi_2. \\
\varphi_1 \land \varphi_2 & \equiv \varphi_1 \text{ occurs at least once, or } \varphi_2 \text{ always occurs.} \\
\varphi_1 \lor \varphi_2 & \equiv \varphi_1 \text{ or } \varphi_2 \text{ occurs at least until } \varphi_2. \\
\varphi_1 \land \varphi_2 & \equiv \text{Balanced parentheses.} \\
\varphi_1 \lor \varphi_2 & \equiv (a,a') \otimes (b,b') = (a \in a' \cup b \in b') \lor (b \in b' \cap a \in a'). \\
\end{align*}$$

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: $Plans(P \land Q) \supseteq Plans(P) \cap Plans(Q)$.

Inverse Homomorphism. A macro action is a sequence of actions, written as $m = a_1 \ldots a_k$. A sequence of actions can be composed into a single action: $[m]_0 = [a_k]_0 \circ \cdots \circ [a_1]_0$. We write $\overline{m} = a_1 \ldots a_k$ for the corresponding sequence of actions identifiers. Given a planning problem $P$ and set of macro actions $M$ over $\mathcal{F}$, we wish to obtain another planning problem $P_M$ such that $Plans(P_M) = \{ m_1, \ldots, m_k \}$ for $i \in [1,k] : m_i \in M$ 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 $h(m) = \overline{m}$. 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 $\S$ 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

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

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 syntactically sugar for $for \ x : r \ do \ S \ od \ fi$. 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; and (iii) updates appear only in the inner-most nesting level. In the sequel, we fix a high-level language, DWhile$^H$, and a low-level language, DWhile$^L$. The synchronization-related statements $acq \ rs \ do \ rel \ rs \ od$ and $rel \ rs$, which are explained in $\S$ 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 $b \ S \ else \ N \ fi”$ (where $N$ is either $exit$ or $skip$) by “if $b \ S \ fi”$.

Footnotes:

1. In practice, we will be interested in the sequence of action identifiers.

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

3. 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 → L 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: {if bₗ, else bₗ', exit bₗ';}; {if bₗ, else bₗ', skip bₗ'}}, {while b do l, od l}, and {for x : r do l, od l}.

For the remainder of this section, we fix a labelled high-level program S ∈ DWhile and F₀ = flat(S).

We invert flattening by defining the (pseudo-inverse) function unflat : F → DWhile such that flat(unflat(F₀)) = F₀. unflat can be implemented by an LALR(1) parser operating on the sequence of tokens that are the units of F₀.

Permuting Flat Programs. We are now interested in defining planning problems that encode the constraints between F₀ and the output of the planner F₀ such that F₀ is a permutation of F₀ that represents an equivalent-meaning program. Let F₀ be the sequence of units u₁, ..., uₙ. We write F₀(i) = uᵢ, and |F₀| = n for the length of F₀. We abbreviate [1, ..., n] by 1..n. A permutation Π : 1..n → 1..n induces a transformation Π : F → F over flat programs, defined as Π(F₀) = F₀(Π(1)) = F₀(Π(n)). We define the equivalence relation F₁ ≈ F₂ if and only if F₁ is a permutation of F₂.

5.3 Encoding Wellformedness by a Planning Problem

We say that a flat program F₀ is wellformed if there exists a program S ∈ DWhile such that flat(S) = F₀.

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

\[ \text{Plans}(WF(S)) = \{ F₀′ \mid F₀′ ≈ WF(\text{flat}(S)), F₀′ \text{ is wellformed} \} \]

To ensure syntactic wellformedness, the planning problem WF contains a fluent u and the action \( {\{ u \}} \) per unit u in the original program where \( \{ \) only \( (u) \} \) per unit u in the original program where only \( \{ u \} \) represents the output program. The first goal establishes that the output program is a permutation of the input program. We refer to units of the form if bₗ, while b do lₗ, and for x : r do lₗ as opening delimiters and units of the form fiₗ as closing delimiters, as they immediately precede and succeed compound statements. Units of the form else bₗ 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 nesting scopes. To ensure semantic wellformedness, we use fluents of the form B[x] per variable appearing in a for statement. A for x : r do lₗ unit adds a B[x] fluent and the corresponding odₗ 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 Plans(WF(S)) are wellformed permutations of flat(S), however, they do not necessarily preserve the semantics of S. We add goals to ensure that the semantics is preserved.

Let \( S ⊆ 1..n \times 1..n \) be a partial order over 1..n. We say that a permutation \( \Pi : 1..n → 1..n \) is monotone w.r.t. ≤, written \( \Pi ≤ \), if \( i ≤ j \) implies \( \Pi(i) ≤ \Pi(j) \). We say that a partial order \( S ⊆ 1..n \times 1..n \) is dependence preserving if every monotone permutation \( \Pi ≤ \) induces an equivalent program: \( [S] = [\text{unflat}(\Pi≤(\text{flat}(S)))] \).

A dependence preserving partial order \( ≤ \) induces an equivalence relation among programs: S and S' are dependence-equivalent, written \( S ≅ S' \), if there exists a monotone permutation \( \Pi ≤ : 1..n → 1..n \) such that \( S = [\text{unflat}(\Pi≤(\text{flat}(S)))] \).

We say that Dependences : DWhile × N × N is a dependence analysis if Dependences(S) is a dependence preserving partial order for every S ∈ DWhile. 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:

\[ G(\text{Depend}(S)) \triangleq \{ u \subseteq v \mid i ≤ j \in \text{Dependences}(S), i \neq j \} \]

![Figure 5](image-url)

**Lemma 2.** Define \( \text{Equiv}(S) = WF(S) \land G(\text{Depend}(S)) \) to be \( WF(S) \) extended with the dependence analysis goals. The plans of \( \text{Equiv}(S) \) are all flat programs that can be un flattened to a dependence-equivalent.
equivalent program:

\[
\text{Plans(Equiv}(S)) = \{ F^{S'} \mid S \preceq \unflat(F^{S'}) \}. \tag{3}
\]

5.5 Encoding Lowering by a Planning Problem

Tiles and Tilings. Let FDWhile^H and FDWhile^L denote the flat languages corresponding to DWhile^H and DWhile^L, respectively. A tile associates a sequence of (high-level) units from FDWhile^H with a (low-level) unit from FDWhile^L, written as \( tile(lu) = hu_1, \ldots, hu_k \). We say that a flat low-level program \( F^L = lu_1, \ldots, lu_m \) is a tiling of the flat high-level program \( F^H = tile(F^L) = tile(lu_1), \ldots, tile(lu_m) \). We also say that \( lu \) covers \( tile(lu) \) 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 \( S^H \in \text{DWhile}^H \), a set of tiles \( M \), and a dependence analysis \( \text{Dependences} \), find a program \( S^L \in \text{DWhile}^L \) such that there exists a dependence-equivalent program \( S^H \preceq S^H \) and \( \flat(S^L) \) is a tiling of \( \flat(S^H) \) where \( \preceq = \text{Dependences}(S^H) \).

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

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

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

The tile above is coupled with the following tile, which is used as a closing delimiter: \( \text{tile(} \text{oldSucc)} = fi, od. \)

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

\[
\text{Lower}_{M}(S^H) \equiv \text{InvHom(Equiv}(S^H), M \). \tag{4}
\]

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

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 \( \sigma \). To support parallelism, we assume that the outermost loop takes parallel semantics. We have that

\[
\text{for } x : r \text{ do } S(\text{od}) = S([x := v_1]) \ldots [S(x := v_k)]\sigma
\]

where \([r]\sigma = \{v_1, \ldots, v_k\}\) and \([S(x := v_i)]\sigma\) 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.

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

Stable Ranges. We further assume that the range expressions appearing in an DWhile program are invariant for each update statement \( \text{upd} \) appearing in it: \([r]\sigma = [r] \circ [\text{upd}]\sigma \). 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 \( obj.acq = 1 \) 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 \( rs: [r]\sigma \subseteq [rs]\sigma \). If so, the object is owned by the current task. To achieve this form of synchronization we introduce the following instrumentation statements.

“\( \text{rel } rs \)” releases the objects in \([rs]\), by setting \( acq = 0 \), 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.

“\( \text{acq } rs1 \text{ ctx } rs2 \)” attempts to acquire the objects in \([rs1]\) where \( rs2 \) denotes the set of objects already owned by the current task, which we call context. If \([rs1]\sigma \subseteq [rs2]\sigma \) the operation succeeds; otherwise, if locking \( rs1 \) fails then the objects denoted by \( rs2 \) and the portion of \( rs1 \) 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 locks 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: “\( \text{acq } rs1 \text{ ctx } rs2 \)” statements execute with the correct context. C1–3 ensure serializability. In Fig. 1(c), \( \text{misV1} \) is correctly synchronized by ATS\footnote{\text{For simplicity, we removed the exit and commit statements.}}. It is obtained from \( \text{misV} \) by applying instrumentation, as explained next.

6.2 Instrumenting DWhile for ATS via Planning

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

Encoding a Lockset Analysis. Let \( \text{res}(S) \) denote the set of all resources in \( S \). We encode a flow-sensitive dataflow analysis, which tracks the set of acquired resources via the powerset lattice \( \{ \text{res}(S), \subseteq, \cup, \cap, \emptyset, \text{res}(S) \} \). We encode lattice elements by the set of flunes \{locked\}| r | \text{res}(S)\} and define the shorthand

\[\]
generates a (parallel) loop 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 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 "rmt24" graph (a=0.5, b=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-degree graphs like road networks but perform poorly for low-degree 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.
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 $N(a)$: The incremental strategy fuses the evaluation of the $V$ predicate with locking $N(a)$, whereas one-shot locks $N(a)$ 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 $w_i$ until it succeeds; the back-off strategy chooses a different work-item and inserts $w_i$ 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 $q$-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$(i)$ ($e$-backoff$(j)$) 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 $N$matched, 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 Előír 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 Előír schedules, we use the $group$ tactic to create a "blocked" composite operator that co-schedules multiple instances of $count$. For example, the schedule `$count \gg group a,b` starts from a specific node $a$ and considers all possible bindings for $b$ and $c$. Alternatively, we can start from $b$ or $c$ and perform similar blocked explorations. Second, we consider input graphs where the neighbors of each node are
sorted in increasing order. Communicating this information to our system allows the planner to use tiles encoding specialized strategies of iterating over the neighbors of each node. These two design parameters give us six different algorithm families: A, B, C, SA, SB, SC, with the first letter denoting the starting node of the blocked operator, and the conditional prefix S denoting whether sorted property is taken into consideration or not. For each family, our planner enumerates a number of solutions that correspond to the different schedules of evaluating the operator. In total, we have 384 variants. 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 to get 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 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 multicore 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 rand25). 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 \(3.3 \) for rmat24, \(2\) for usa-all, and \(3.5\) for rand25.

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.
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 Elxir 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 Elxir variant is 60% of the time of the worst Elxir variant for the rand23 graph and 72% for the usa-all graph.

The three solutions differ primarily in the synchronization implementation. The Elxir 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 [28] (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 Elxir 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. Elxir currently does not support synthe-

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

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

sizing 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 Elxir. 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 Paradigm [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].

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 dependencies 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


