Position Paper: Logic Programming for Parallel Irregular Applications

Jeremiah Willcock and Andrew Lumsdaine
CREST, Indiana University
Need for Parallel Graph Algorithms

- Graph algorithms important to computer science
  - Breadth-first search, PageRank, shortest paths, etc.
- New applications demand large graphs
  - Social network analysis, bioinformatics
- Problems beyond capacity of single processors
  - Both memory and CPU performance limitations
  - Need parallelism
Implementing Parallel Graph Algorithms

- Algorithms difficult to implement
  - Managing parallelism hard in general
  - Many programming models
  - Different approaches for different systems, data

- Limited portability
  - Shared memory vs. distributed memory
  - Cray XMT, GPUs, other systems

- Generic programming mitigates this partially
  - C++ limited for expressing vastly different computation models in the same code
Data Structures

- Different from those in most databases
- Vertex, edge properties can often be arrays
  - Might be distributed
- Graphs in compressed sparse row, adjacency list, STINGER, specialized GPU formats, many variants
- Want to support symbolically represented graphs
  - BDD-based state machines for model checking
  - Implicit graphs such as grids for computer vision
  - de Bruijn graphs for bioinformatics
Proposal: High-level Specification

- Write algorithms in a declarative language
- Compiler retargets them to different platforms
- Code generated for various
  - Platforms
  - Graph data structures
  - Data sizes
- Semi-automatically optimize code for each platform
- Graph algorithms easier to tune
  - Configure parameters to generator
Datalog

- Declarative language for querying databases
- Subset of Prolog
  - No function symbols
  - Fewer primitives
  - Limited negation (usually)
- Allows recursion
  - Unlike SQL or relational algebra
  - Does not need barriers between recursive steps
- Programs always terminate
Datalog Example

ancestor(X, Y) :- parent(X, Y).
“For any X and Y, if X is the parent of Y then X is an ancestor of Y.”

ancestor(X, Z) :-
  ancestor(X, Y), ancestor(Y, Z).
“For any X, Y, and Z, if X is an ancestor of Y and Y is an ancestor of Z, X is an ancestor of Z.”

» Would not terminate in normal Prolog
Stratification

- Approach used for negation in Datalog
  - No cyclic dependencies containing negations
- Divide program into strongly connected components
- Negations only allowed between components

\[
\text{ancestor}(X, Y) :- \text{parent}(X, Y).
\]
\[
\text{ancestor}(X, Z) :-
  \text{ancestor}(X, Y),
  \text{ancestor}(Y, Z).
\]
\[
\text{not\_ancestor}(X, Y) :- \neg \text{ancestor}(X, Y).
\]
Need for Multi-Valued Logic

- Graph algorithms return non-Boolean results
  - Shortest paths returns optimal distances
  - Semiring often used as structure for this
- Normal Datalog predicates are either true or false
  - Using one argument to a predicate as the result does not allow updating
- Multi-valued logic generalizes predicate values
  - \( \text{dist}(X, Y) \) is valued as a number between \(-\infty\) and \(\infty\)
- Monotonicity requirements similar to stratification
  - Algorithms such as betweenness centrality need multiple passes with barriers in between
Semilattices

- Set with associative, commutative, idempotent binary operation $\sqcap$ ("meet")
- Operation defines partial order $\sqsubseteq$ on elements
  - $x \sqsubseteq y$ if and only if $x \sqcap y = x$
- Likely to relax commutativity in practice
  - Allow nondeterministic behavior in case of distance ties
Monotonicity

- Moving input of a function up or down semilattice moves function result the same direction.
- Formally, \( x \sqsubseteq y \) implies that \( f(x) \sqsubseteq f(y) \).
Lattice-Valued Datalog Example

- Compute path length from source to each vertex (pseudocode), assuming no negative-length cycles:

  Define table \( \text{dist} \) using \( \min \) as reduction operator.
  \[
  \text{dist}(S) = \text{source}(S), \ 0.
  \]
  \[
  \text{dist}(W) = g(V, W, E), (\text{dist}(V) + \text{weight}(E)).
  \]

- Aggregation done automatically
- Works because of monotonicity and distributivity, plus finite number of simple paths
Data Flow

- Source
- Map
- Join
- Join
- Union
- Dist
- Weight
- g
Lattice-Valued Datalog Stratification

- Stratification has been generalized to lattice-valued Datalog
- Non-monotonic operators treated as negations
- Same rules as before apply

For applications, might want to stratify based on values, not just syntax:

```prolog
foo(X, N+1) :- foo(Y, N), bar(X, Y).
```
Data Flow Adding Reciprocal
Explicit Aggregation

- Sums, averages, etc. not semilattice operations
  - Not idempotent
- Necessary for practical system
- Only allow these between strata
  - Compute input data, then do aggregate operator
  - Could do some operators incrementally
    - Associative, commutative, with inverses
Performance Hints

- Query optimizer may not get optimal performance
- Want ways to tune query plan, data structures, etc.
- One option is a “simple” implementation
  - Always uses a fixed query plan based on the program
  - User rearranges program to change behavior
  - Can be cumbersome to write some things in this model
    \[ \text{dist}(W) = \text{dist}(V) + (g(V, W, E), \text{weight}(E)). \]
- Can use same data structures as in hand-written code as long as query patterns are limited
- Will also need to import external data
  - Including from sources such as SQL databases
Comparison to Other Models

- Generalization of linear algebra over semirings
  - Multiple dimensions
  - Non-distributive operations (with imprecise results)
- Generalization of SQL, SPARQL, relational algebra
  - Allows recursion to be expressed directly
- Simplification of visitor-based models
  - Limits set of operations in visitors
  - Increases available parallelism
- Less limited than many graph DSLs
  - Recursion without requiring level-based iterations
Expressiveness

- Expresses simpler algorithms directly
  - “Simpler” is relative — more difficult algorithms are hard to write in any framework
- User queries are likely to be simple
  - Ad-hoc queries unlikely to use multilevel methods
  - Can still provide tuned kernels for these
Join and Fixpoint Algorithms

- Joins map directly to Datalog
  
  \[ \text{abc_path}(X, Y) \leftarrow \text{a}(X, Z), \text{b}(Z, W), \text{c}(W, Y). \]

- Compute paths using recursion
  
  \[ \text{abstar_path}(X, X). \]
  
  \[ \text{abstar_path}(X, Y) \leftarrow \text{a}(X, Z), \text{b}(Z, W), \text{abstar_path}(W, Y). \]

  \[ \text{alternating_path}(X, Y) \leftarrow \text{abstar_path}(X, Y). \]
  
  \[ \text{alternating_path}(X, Y) \leftarrow \text{b}(X, Z), \text{abstar_path}(Z, W). \]

- Might want performance hint about ordering to use
  - Somewhat similar to tag collections in Intel Concurrent Collections
Iterative Algorithms

- Some algorithms normally written level-synchronous
  - Breadth-first search
  - PageRank

- Most of these have formulations with fewer barriers
  - Often not as work-efficient as with barrier

```
pr_iter(0, V, ...).
pr_iter(N+1, V, ...) :- pr_iter(N, ..., ...).
pagerank(V, PR) :- stop_iter(N), pr_iter(N, V, PR).
```
Multi-Level Algorithms

- Most difficult class to express in this model
- Would need same features as iterative algorithms
- Maybe easier to use relational algebra directly?
  - Or linear-algebra-like operations with recursion?
Programmability

- Graph data sets likely to have few types of relationships used in a single query
  - Using many inputs at once leads to verbose code
- Can define helper relations to abstract out intermediate computations
- Datalog being first-order might be a limitation
- User experience would be needed
  - SQL is commonly used, and constructs used are similar
  - Logic programming likely to be unfamiliar to most users
- Researchers have developed specialized front-ends
Prior Work on Datalog

- Automatic parallelization
- Automatic incrementalization
- Implementation using SQL
- Implementation on binary decision diagrams (BDDs)
- Datalog for distributed systems
  - Bloom$^L$
  - StarLog
- Front-ends generating Datalog
  - Program Query Language
  - GraphQL
Conclusion

- Lattice-valued Datalog advocated as a good high-level specification for irregular algorithms
  - Aids productivity
  - Simplifies tuning
- Generate code for various platforms, data structures from a single specification
  - Plus possibly some platform-specific hints
- Hybrid approaches possible
  - Datalog plus hand-written execution strategy/query plan
  - Parts of implementation given manually
    - Might include problem- and/or platform-specific data structures
Prototype Compiler

- Translates lattice-valued Datalog to C++
- Currently targets
  - Sequential code
  - SQLite
  - Larrabee intrinsics
- Designed to target other platforms in the future
- Likely to become JIT
Compiler Structure

- Datalog code
- Recursive table equations
- Abstract stream equations
- CSP processes
- C++ code
Datalog Code (S-expressions)

(InputTable 'source (TableType `(,(VertexT)) (TypeAndReducer (VoidT) (FlatR))) (SingleEntryTable))
(ResultTable 'dist (TableType `(,(VertexT)) (TypeAndReducer (FloatT) (MinR))) (ArrayTable))
(InputTable 'g (TableType `(,(VertexT) ,(VertexT) ,(EdgeT)) (TypeAndReducer (VoidT) (FlatR))) (GraphTable))
(InputTable 'weight (TableType `(,(EdgeT)) (TypeAndReducer (FloatT) (MinR))) (ArrayTable))

((TableRef 'dist '(W))
  . ← .
  (Combine
    add-combiner
    (TableRef 'dist '(V))
  (Combine
    second-combiner
    (TableRef 'g '(V W E))
    (TableRef 'weight '(E))))))

((TableRef 'dist '(S))
  . ← .
  (Map
    (const-map-func (FloatT) (MinR) (ConstE "0.0"))
    (TableRef 'source '(S))))
Conversion to Table Equations

- Combine all rules for a single relation
- Normalize conjunctions into joins
- Insert wildcard elements and projections (aggregation of multiple results)
Recursive Table Equations
Stream Equation Generation

- Define processing of newly added/updated elements
  - Seminaïve evaluation
- Normalize joins to input stream and single relation
  - Add dummy streams as needed
- Insert explicit queues
  - Order of tuple processing matters for performance
  - Dijkstra’s algorithm uses increasing order of distances
Stream Equations

table init: combine (a, b) \rightarrow (b)

universe: map (a) \rightarrow (0.0)

universe: rearrange V W E \rightarrow W

universe: queue min

universe: insert dist

table weight: combine (a, b) \rightarrow \text{first}(a) + b

table weight: combine make_tuple

table weight: table g

universe: rearrange V \rightarrow V W E
Implementing Stream Operators

- Query optimization
- Solve for modes of tuples being passed around
  - Which elements are not filled in
- Determine best formats for streams
  - Scalar variables, SIMD, SQL tables, etc.
- Based on user-defined formats for relations

- Optimizer is currently simple
- Most platform-specific configuration will be here
Coroutine Generation

- Convert each stream operator into coroutines
- Communication by CSP channels
  - Bounded resources, simple to implement
- Data sent in channels depends on implementation
- “Real” (non-coroutine) parallelism done by data parallelism
  - Local coroutine-handling code does not need to know about it
Coroutine Structure

1. Read priority queue
2. Add unbound slots
3. Join with out edges
4. Join with edge weights
5. Project out source and edge
6. Copy
7. Update dist
8. Push into priority queue
9. Write initial value
10. Join with source vertex
11. Set distance to 0
12. Copy
Code Generation

- Coroutines interleaved statically into a single thread
- Channels implemented using variables
  - Synchronization done by code generator
- Generates control flow graph
- Convert CFG into C++ code