Compiling Data-Parallel Datalog

Thomas Gilray
University of Alabama at Birmingham
USA
gilray@uab.edu

Sidharth Kumar
University of Alabama at Birmingham
USA
sid14@uab.edu

Kristopher Micinski
Syracuse University
USA
kkmicins@syr.edu

Abstract

Datalog allows intuitive declarative specification of logical inference tasks while enjoying efficient implementation via state-of-the-art engines such as LogicBlox and Soufflé. These engines enable high-performance implementation of complex logical tasks including graph mining, program analysis, and business analytics. However, all efficient modern Datalog solvers make use of shared memory, and present inherent challenges scalability.

In this paper, we leverage recent insights in parallel relational algebra and present a methodology for constructing data-parallel deductive databases. Our approach leverages recent developments in parallelizing relational algebra to create an efficient data-parallel semantics for Datalog. Based on our methodology, we have implemented the first MPI-based data-parallel Datalog solver. Our experiments demonstrate comparable performance and improved single-node scalability versus Soufflé, a state-of-art solver.

CCS Concepts: • **Computing methodologies → Parallel programming languages**.

Keywords: Datalog, MPI, Logic programming

ACM Reference Format:
Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. 2021. Compiling Data-Parallel Datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2–3, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3446804.3446855

1 Introduction

Database systems are broadly declarative, typically supporting expressive query languages. Systems for *deductive databases* such as Datalog further allow persistent rules that specify relations defined intensionally, only in terms of other relations. Picture a database listing inventory and sales for an online business where a set of simple declarative rules are used to update an out-of-stock table or a table listing the total profit earned for each customer. In such systems, expressive reasoning can be embedded alongside ones data and used to generate sophisticated analytics on-the-fly as changes are made.

Effective declarative programming represents a long-standing dream of computing—exchanging code describing *how* to compute for code simply describing *what* to compute. Instead of requiring programmers to themselves balance the concerns of correctness, maintainability, and scalability in each task, declarative programming languages permit users to focus on the first two concerns, writing high-level, correct specifications of *what* should be computed, while allowing the underlying implementation (i.e., *how* the operational mechanics of the program work) to be extracted automatically. This puts significant pressure on the implementation strategy used for declarative languages as it must compete with hand-optimized implementations while remaining generic and effective for a wide variety of applications.

At the same time, this approach presents an opportunity: as effective means are found for parallelizing the semantics of declarative languages, the strategy will apply immediately to all analytics using these platforms. This is already the case for bottom-up (forward chaining) logic programs written in *Datalog*, and related languages with semantics that implement first-order or higher-order Horn-clause satisfiability. Such programs may be implemented using high-performance relational algebra, as is done in the state-of-the-art logic-programming language Soufflé [30]. Standard operations on relations such as selection, projection, join, and union may be used in combination to implement efficient kernels that infer new facts from available facts in a (fixed-point) loop. Fortunately, the underlying primitives used in these kernels are inherently quite data-parallel. For example, the Cartesian product of two relations, $R \times S$ may be computed by partitioning the left-hand relation R for a set of threads and then performing a local product $R_i \times S$ for each partition i, of R, in a trivially parallel manner. Unfortunately, many important inference problems must be scaled beyond single-node parallelism; e.g., state-of-the-art program analyses for Java that are implemented in Soufflé can take many

CC '21, March 2–3, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8325-7/21/03…$15.00
https://doi.org/10.1145/3446804.3446855

hours and only terminate for lower-precision tunings, targeting real-world codebases [34].

While this approach to parallelizing bottom-up logic programming is already being used to great effect on single-node systems [3], scaling the approach to many-thread clusters using MPI's inter-process communication paradigms has proven a significant challenge. Recent work has introduced techniques for managing parallel relational algebra using MPI on clusters. This has been quite tricky as both communication and decomposition of the computational workload over many nodes must be done explicitly in a dynamically balanced manner, while on shared-memory systems both these problems become nearly trivial when using efficient thread-safe data-structures—tools produced through decades of successful research.

In this paper, we present a synthesis of two ongoing threads of research: (1) an approach to compiling Datalog-like languages to relational algebra for synthesizing fast program analyzers [17], and (2) a scheme for balanced parallel relational algebra using MPI that addresses communication challenges of putting operations on relations in a loop [24]. Putting these ideas together, we are able to present a highly data-parallel deductive database engine that shows promising scaling on the DOE supercomputer Theta.

To the literature, we contribute:

1. An approach to developing scalable, data-parallel Datalog solvers using parallel relational algebra. We formalize our approach as a parallel relational algebra machine (PRAM) for which we have implemented a compiler.
2. An evaluation of kCFA, a core program analysis algorithm, showing higher performance and scalabiliy compared with Soufflé. In general, we observed varying but comparable performance and improved relative scaling in our comparison studies.

2 Background

Our approach synthesizes two foundational threads: compilation of Datalog-like inference languages to relational algebra kernels and recent success enabling the development of data-parallel relational algebra implementations. In this section, we present a summary of these developments to ground our own contributions.

2.1 Relational Algebra

Projection of a relation R restricts it to a particular set of dimensions $\alpha_0, \ldots, \alpha_j$, where $\alpha_0 < \ldots < \alpha_j$, and is written $\Pi_{\alpha_0,\ldots,\alpha_j}(R) \triangleq \{(r_{\alpha_0}, \ldots, r_{\alpha_j}) \mid (r_0, \ldots, r_k) \in R\}$. Renaming (i.e., recording) columns can be defined in several ways. We define a renaming operator, $\rho_{\alpha_i/\alpha_j}(R)$, to swap two columns, α_i and α_j where $\alpha_i < \alpha_j$—an operation that can be repeated to rename/reorder as many columns as desired.

Two relations can also be *joined* into one on a subset of columns they have in common. Join combines two relations into one, where a subset of columns are required to have matching values, and generalizes both intersection and Cartesian product operations.

To formalize natural join as an operation on such a relation, we parameterize it by the number of prefix values that must match, assumed to be the first j of each relation (if they are not, a renaming operation must come first). The join of relations R and S on the first j columns is written as $R \bowtie_j S$ and can be defined:

$$R \bowtie_j S \triangleq \{ (r_0, \ldots, r_k, s_j, \ldots, s_m) \mid (r_0, \ldots, r_k) \in R \wedge (s_0, \ldots, s_m) \in S \wedge \bigwedge_{i=0..j-1} r_i = s_i \}$$

Naturally a system of relational algebra can support a variety of additional operations or compositions of the above operations (e.g., join followed by project followed by reorder) in theory, and usually does in practice for reasons of efficiency.

2.2 Datalog

Datalog is a bottom-up logic programming language supporting a restricted logic corresponding to first-order HornSAT—the satisfiability problem for conjunctions of Horn clauses [1]. A *Horn clause* is a disjunction of atoms, all but one of which is negated: $a_0 \vee \neg a_1 \vee \ldots \vee \neg a_j$. Atoms are predicates over universally-quantified variables. By DeMorgan's laws, Horn clauses may be seen equivalently as $a_0 \vee \neg(a_1 \wedge \ldots \wedge a_j)$ or equivalently via implication as $a_0 \leftarrow a_1 \wedge \ldots \wedge a_j$.

A Datalog *program* is a set of (Horn clause) rules $P(x_0, \ldots, x_k) \leftarrow Q(y_0, \ldots, y_j) \wedge \ldots \wedge S(z_0, \ldots, z_m)$. It is common to have an "input" database of facts called the *extensional database* (EDB), consisting of an explicitly listed (extensional) set of tuples. Running a Datalog program in terms of some EDB reifies the *intensional database* (IDB), all facts (transitively) derivable via the program's rules. The following example program computes the `uncle` relation from the relations `brother` and `parent`:

```
uncle(x,u) :- parent(x,p), brother(p,u).
```

Efficient implementation strategies for Datalog would evaluate this program by first compiling to relational algebra primitives. On shared-memory systems, this would involve ordering the `parent` relation so it can be efficiently joined with `brother`, then projecting out the shared `p` column; this is written $\Pi_{1,2}(\rho_{0/1}(\texttt{parent}) \bowtie_1 \texttt{brother})$. This reduces optimization of evaluation to efficient implementation of the relational algebra primitives. We call a set of RA primitives used to implement a Datalog program a relational algebra *plan* (RA plan).

While some relations can be computed via a fixed number of RA operations in sequence, others must be computed via a fixed-point of a set of operations. For example, *transitive closure* (TC) of a relation or graph is efficiently implemented via a loop (checking whether a fixed-point has been achieved) over a set of high-performance RA operations. Consider the Datalog rules for computing the `ancestor` relation:

```
ancestor(x,p) :- parent(x,p).
ancestor(x,a) :- ancestor(x,ac), parent(ac,a).
```

The first rule represents a base case that says any parent is trivially an ancestor, and the second represents the inductive step of inferring that an ancestor `ac` for `x` and a parent `a` for `ac` implies an older ancestor `a` for `x`.

We can also view `parent` as an input graph defined as a set of edges, and view TC computation as a graph mining problem. Computing the transitive closure `ancestor`, of input graph `parent`, is a simple example of logical inference. From paths of length 0 (an empty graph) and the existence of edges in graph `parent`, we may trivially deduce the existence of paths of length $0 \ldots 1$. From paths of length $0 \ldots n$ and the original edges in graph `parent`, we may infer the existence of paths of length $0 \ldots n+1$. The function $F_{\texttt{parent}}$ below performs a single round of inference, finding paths one edge (parental relationship) longer than any found previously and exposing new inferences to be made for the next iteration of $F_{\texttt{parent}}$. When the computation reaches a fixed-point, the solution has been found as no further paths may be deduced from the available facts.

$$F_{\texttt{parent}}(\texttt{ancestor}) \triangleq \texttt{parent} \cup \Pi_{1,2}(\rho_{0/1}(\texttt{ancestor}) \bowtie_1 \texttt{parent})$$

The first rule says that any direct edge in `parent` implies a path, in `ancestor` (taking the role of the left operand of union in F_G), and the second rule says that any path (x, ac) and edge (ac, a) imply a path (x, a) (adding edges for the right operand of union in $F_{\texttt{parent}}$). Other kinds of graph mining problems, such as computing triangles or k-cliques, can also be naturally implemented as Datalog programs [41]. See Section 4 for several case studies on graph mining and program analysis applications.

Each Datalog rule may be encoded as a monotonic function F, mapping databases to databases conjoined with their immediate consequences, where a fixed-point for F is guaranteed to satisfy the corresponding rule. Once a set of functions $F_0 \ldots F_m$ are constructed (one for each rule), naïve Datalog evaluation operates by iterating the initially empty IDB and input EDB to a mutual fixed-point for $F_0 \ldots F_m$. Because the EDB includes a fixed number of atoms, and because Datalog programs cannot generate new atoms, termination is guaranteed.

In practice, most "Datalog" implementations are more expressive than this, and most permit some basic built-in operations (e.g., `x!=y`), and frequently will allow constructors for inductive data types, such as linked lists. However, while most Datalog implementations are ultimately Turing-complete, they optimize for terminating programs accelerated via fast RA operations.

2.3 Implementation via RA

So far, we have elided important optimization and implementation details in favor of focusing on Datalog's semantics. High-performance Datalog solvers employ specializations of general RA operations that combine sequences of simple operations (join, project, and rename) into one efficient combined operation. We have so far presented the so-called naïve evaluation strategy, recomputing all previously-discovered facts at every iteration. Efficient implementations employ *incrementalization*, tracking a frontier of freshly-
discovered facts to produce yet-undiscovered facts. For example, when computing transitive closure, another relation $\texttt{ancestor}_{\Delta}$ is used which only stores the longest ancestry paths—those discovered in the previous iteration. When computing paths of length n, in fixed-point iteration n, only new paths discovered in the previous iteration, paths of length $n-1$, need to be considered, as shorter paths extended with edges from `parent` necessarily yield paths which have been discovered already. This optimization is known as *semi-naïve evaluation* [1].

Using semi-naïve evaluation, each non-static relation (those that may be updated in given iteration, such as `ancestor`) is effectively partitioned into three relations: $\texttt{ancestor}_{\text{full}}$, $\texttt{ancestor}_{\Delta}$, and $\texttt{ancestor}_{\text{new}}$. $\texttt{ancestor}_{\text{full}}$ stores any facts discovered more than one iteration ago; $\texttt{ancestor}_{\Delta}$ maintains facts that were newly discovered in the previous iteration, and is joined with `parent` each iteration to discover new facts; and $\texttt{ancestor}_{\text{new}}$ holds these newly discovered facts only just learned in the current iteration. At the end of each iteration, $\texttt{ancestor}_{\Delta}$'s tuples are added to $\texttt{ancestor}_{\text{full}}$, the pointers are swapped for $\texttt{ancestor}_{\Delta}$ and $\texttt{ancestor}_{\text{new}}$, and $\texttt{ancestor}_{\text{new}}$ is truncated to prepare for the subsequent iteration.

The state of art evaluating Datalog on a single compute node is perhaps best embodied in the Soufflé engine [17–19, 30]. Soufflé systematically optimizes the RA kernels obtained from an input Datalog program, yielding a program for an abstract *Relational Algebra Machine* (RAM). Figure 1 shows a portion of the exact C++ code produced by Soufflé (v1.5.1) for the two-rule TC program (indentation

and code comments have been added by the authors to improve clarity). To compute $\rho_{0/1}($ `ancestor` $_{\Delta}) \bowtie_1$ `parent`, first the outer relation (the left-hand relation—in this case T_{Δ}) is partitioned so that Soufflé may process each on a separate thread via OpenMP (line 1 in Figure 1). For each partition, a loop iterates over all tuples in the current partition of `ancestor` $_{\Delta}$ (line 2) and computes a selection tuple, `key`, representing all tuples in `parent` that match the present tuple from `ancestor` $_{\Delta}$ in its join-columns (in this case, just the second column value, `env0[1]`). This selection tuple is then used to produce an iterator selecting only tuples in `parent` whose column-0 value matches the particular tuple `env0`'s column-1 value. Soufflé thus iterates over each $(x, y) \in$ `ancestor` $_{\Delta}$ and creates an iterator that selects all corresponding $(y, z) \in$ `parent`. Soufflé iterates over all matching tuples in `parent` (line 5), and then constructs a tuple (x, z), produced by pairing the column-0 value of the tuple from `ancestor` $_{\Delta}$, `env0[0]`, with the column-1 value of the tuple from `parent`, `env1[1]`, which is inserted into `ancestor` $_{\text{new}}$ (line 8) only if it is not already in `ancestor` $_{\text{full}}$ (line 6). Given this architecture,

```
   // Partition ancestor_Δ for a pool of OpenMP threads; iterate over parts
1  pfor(auto it = part.begin(); it<part.end();++it){
     // Iterate over each tuple, env0, of ancestor_Δ (in each partition)
2    try{for(const auto& env0 : *it) {
       // Construct an iterator selecting tuples in parent that match env0
3      const Tuple<RamDomain,2> key({{env0[1],0}});
4      auto range = rel_1_edge->equalRange_1(key,
                                READ_OP_CONTEXT(rel_1_edge_op_ctxt));
       // Iterate over matching tuples in parent
5      for(const auto& env1 : range) {
         // Has this output tuple already been discovered (is in ancestor_full)
6        if(!(rel_2_path->contains(Tuple<RamDomain,2>({{env0[0],env1[1]}}),
                                  READ_OP_CONTEXT(rel_2_path_op_ctxt)))) {
           // Construct the output tuple and insert it into T_new
7          Tuple<RamDomain,2> tuple({{static_cast<RamDomain>(env0[0]),
                                      static_cast<RamDomain>(env1[1])}});
8          rel_4_new_path->insert(tuple,
                                READ_OP_CONTEXT(rel_4_new_path_op_ctxt));
9        }
10     }
11   }} catch(std::exception &e){SignalHandler::instance()->error(e.what());}
12 }
```

Figure 1. The join in TC, as implemented by Soufflé.

Soufflé achieves good performance by using thread-safe data-structures, template specialized for common use cases, that represent each relation extensionally—explicitly storing each tuple, organized to be amenable to fast iteration, selection, and insertion. Soufflé includes a concurrent B-tree implementation [18] and a concurrent prefix-tree implementation [19] as underlying representations for relations. Soufflé does not support MPI or distributed computation of Datalog programs.

2.4 Balanced Parallel RA

Implementation of Datalog as performant RA suggests a strategy for extracting parallelism if these primitive operations themselves can be made highly data-parallel. Several lines of work have approached the challenge of developing schemes for decomposing large RA operations over many parallel threads. The double-hashing approach, with local hash-based joins and hash-based distribution of relations, is broadly the most commonly used method to distribute join operations over many nodes in a networked cluster computer [12, 13, 38]. Recently, both radix-hash join and merge-sort join have been evaluated [7] at up to 4k threads.

Another recent approach proposes algorithms for *balanced parallel relational algebra* (BPRA) adapting the representation of imbalanced relations, using a two-layered distributed hash-table to partition tuples over a fixed set of *buckets*, and, within each bucket, to a dynamic set of *subbuckets* which may vary across buckets [23]. This represents a vertical decomposition of the relation across processes that is keyed both on a set of join columns, and also on all other columns to ensure a balanced mapping of tuples to processes. Each tuple is assigned to a bucket based on a hash of its join-column values, but within each bucket, tuples are hashed on all non-join-column values, assigning them to a subbucket. Each bucket and subbucket pair unqiuely assigns those tuples to a particular MPI process. This scheme permits buckets that have more tuples to be split across multiple processes uniformly and for the number of subbuckets to increase with the tuples in that bucket. Balancing can be done periodically in both the direction of splitting buckets into more subbuckets, or of consolidating them, as needed. To distribute subbuckets to managing processes, BPRA uses a round-robin mapping scheme that requires a very small amount of additional communication, but ensures that no process manages more than one subbucket more than any other. Locally, subbuckets store tuples using B-trees (an approach also used by Soufflé, although their data structures have undergone particular engineering refinements). This carries performance advantages over the double-hashing approach's use of hash tables. Crucially, hash-tables can also lead to complications in a distributed setting where a resizing operation may delay synchronization.

Figure 2 shows a schematic diagram of the BRPA join algorithm in the context of an incrementalized TC computation. A join operation can only be performed for two *co-located relations*: two relations each keyed on their respective join columns that share a bucket decomposition (but not necessarily a subbucket decomposition for each bucket). This ensures that the join operation may be performed separately on each bucket as all matching tuples will share a logical bucket; it does not ensure that all pairs of matching tuples will share the same subbucket as tuples are assigned to subbuckets (within some bucket) based on the values of non-join columns.

The first step in a join operation is an *intra-bucket communication* phase within each bucket in which every subbucket receives all tuples for the outer relation, across all

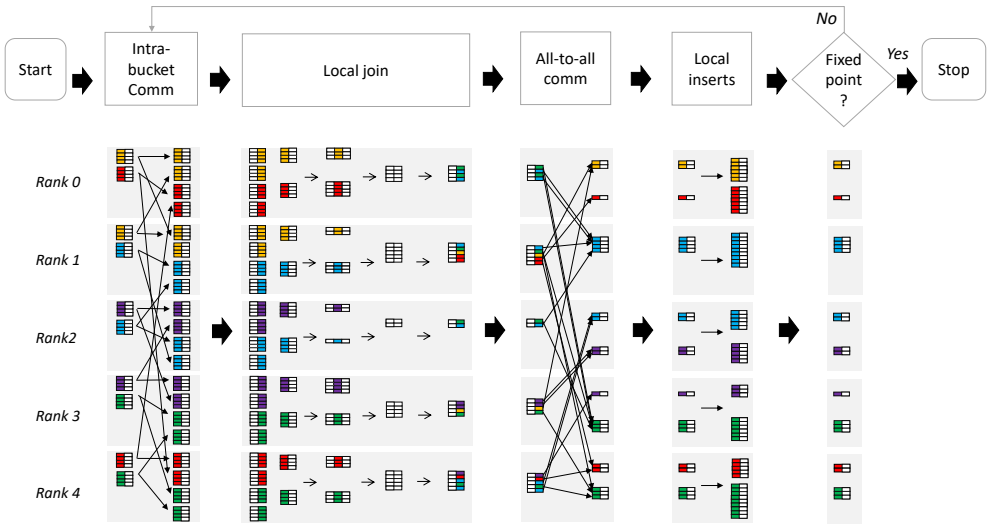

Figure 2. Shows the major phases of a BPRA join in the context of a TC computation.

subbuckets (while the inner relation only needs tuples belonging to the local subbucket). After this, a *local join* operation (corresponding to a Datalog rule, with possible projection and renaming) can be performed in every subbucket in parallel. Output tuples from these local joins may each belong to an arbitrary bucket in the output relation, so an MPI *all-to-all* communication phase shuffles the output of all joins to their managing processes (preparing them for any subsequent rules or iterations). Upon receiving output tuples from the previous join, each receiving process inserts them into the local B-tree for each applicable R_{new}. It then propagates R_{Δ} into R_{full} and R_{new} becomes R_{Δ} for the next iteration. If no new tuples have been discovered, globally, a fixed-point has been reached and iteration may halt.

Intra-bucket communication (shown on the left of Figure 2) uses MPI point-to-point communication to shuffle tuples from each subbucket of the outer relation to subbuckets of the inner-relation, which is then able to perform local, per-subbucket joins. It may seem appealing to fuse the final all-to-all communication phase among buckets with the intra-bucket communication of the subsequent iteration, sending new tuples (for R_{Δ} in the next iteration) directly to all subbuckets of the inner relation; however, doing this fusion forgoes an opportunity for per-subbucket deduplication and yields meaningful slowdowns in practice.

The local join phase proceeds in a parallel and unsynchronized fashion. Each process iterates over its subbuckets, performing a single join operation for each. Our join is implemented as a straightforward tree-based join as shown in the center of Figure 2. In this diagram, colors are used to indicate the hash value of each tuple as determined by its join-column value. Once received, the outer relation's tuples are iterated over, grouped by key values, where, for each, a lookup is performed to select a portion of the inner relation's B-tree where all tuples have a matching key value (in the case of TC computation, this is selecting the first column of `parent`). For two sets of tuples with matching join-column values, we effectively perform a Cartesian product computation, yielding one tuple for all possible pairs of outer and inner tuple.

Each output tuple has projection and renaming performed on-the-fly; in the case of TC, the prior join columns that matched are projected away. These tuples are locally deduplicated, organized, and staged for transmission to new managing subbuckets in their receiving relation. After evaluating a rule, each output tuple destined for a head-relation R belongs to R_{new} and must be hashed on its join columns; in the case of TC, this is the rightmost column of `ancestor`. Join columns are hashed to determine the bucket and non-join columns to determine the subbucket; together bucket and subbucket determine a managing process via the current round-robin mapping (stored on every process). An all-to-all communication phase (shown on the right side of Figure 2) transmits materialized joins to their new bucket-subbucket decomposition in head-relation R_{new}. The managing process for each bucket and subbucket involved is obtained from a local round-robin map and tuples are organized into buffers for MPI's `All_to_allv` synchronous communication operation. When this is invoked, all tuples are shuffled to their destination processes.

Finally, after the one synchronous communication phase per iteration, each R_{Δ} is locally propagated into R_{full}, which stores all tuples discovered more than 1 iteration ago. New tuples are checked against this R_{full} to ensure they are genuinely new facts, and are inserted into a B-tree for R_{new} on each receiving process to perform remote deduplication. At this point, the iteration ends, R_{new} becomes R_{Δ} for the subsequent iteration, and an empty R_{new} is allocated. If no new

tuples were actually discovered in the previous iteration, a fixed-point has been reached and no further iterations are needed as the database as stabilized with respect to all rules of inference.

BPRA is notable for allowing two kinds of load-imbalance to be remediated dynamically across fixed-point iterations. *Spatial load imbalance* occurs when a relation's stored tuples are mapped unevenly to processes due to key-skew or inherent imbalance in relation's distribution of tuples. *Temporal load imbalance* occurs when the number of output tuples produced varies significantly across iterations.

BPRA includes three main algorithms for adjusting the representation of a relation or RA operation to improve both these kinds of balancing. *Bucket refinement* is a dynamic check of each bucket to see if its subbuckets are significantly heavier than average. When this is detected, it triggers a refinement in which new subbuckets are allocated to support the larger number of tuples in this specific bucket. *Bucket consolidation* is the reverse and occurs only if there are a significant number of refined buckets. It consolidates buckets into fewer subbuckets when spatial imbalance has again lessened. Last, *iteration roll-over* allows particularly busy iterations to be interrupted part-way, with completed work being processed immediately via an added communication phase and with the residual workload from the iteration "rolling over" to a new iteration. This improves robustness in the face of temporal imbalance, preventing crashes at the cost of an additional sychronization phase and poorer deduplication behavior.

3 Parallel RA Machine

We develop the standard techniques for compiling Datalog-like languages to RA, and for parallelizing RA over large numbers of threads using BPRA, into a *Parallel Relational Algebra Machine* (PRAM) and its intermediate representation (IR). Soufflé's RAM is a model for deductive databases in terms of shared-memory relational algebra on a single machine. We introduce PRAM as a evolution of this concept that suits the unique challenges and opportunities of applying BPRA to logical inference tasks.

In the example of Soufflé's compiled C++ in Section 2, any k-ary join operation could be implemented as a series of k nested for loops (with partitioning among threads, efficient selecting iterators, etc). By contrast, BPRA imposes a key restriction to facilitate data-parallelism: all joined relations must use indices that have precisely homogenous join columns. This means a rule such as

```
H(x,y,z) :- B0(w,x,y), B1(y,z), B2(z).
```

must be evaluated using two sequential binary joins, since `B0` and `B1` share a different column in common than do `B1` and `B2`. While trinary or k-ary joins that are homogenous in their join columns are conceivable via BPRA, they are not common in our Datalog code. The fact that we must use binary joins, and cannot reuse indices for their prefixes [35], represent key restrictions that countervail Soufflé's approach to optimizing Datalog for shared-memory systems. However, with several key innovations, we manage and exploit the unique distributed setting of our parallel RA operations and obtain an approach that can scale well.

We implement PRAM using BPRA. BPRA's approach to parallel RA is amenable to iterated RA operations over relations with highly dynamic topologies, but it has several challenging limitations we must overcome:

- Lacking of support for heterogenous k-ary joins,
- Inability to run operations in parallel—in BPRA, each operation has its own synchronous all-to-all communication phase.
- Lack of support for tracking multiple indices of relations, adding new tuples to each index.

3.1 PRAM IR

$$
\begin{aligned}
\text{ir} \in \text{PRAM} &= \mathcal{P}(\text{SCC}) \times \mathcal{P}(\text{SCC} \times \text{SCC}) \\
\text{scc} \in \text{SCC} &= \mathcal{P}(\text{Rule}) \\
\text{rule} \in \text{Rule} &::= \texttt{(rule hclause bclause)} \\
&\quad \mid \texttt{(rule hclause bclause prim)} \\
&\quad \mid \texttt{(rule hclause bclause bclause)} \\
&\quad \mid \texttt{(arule hclause bclause)} \\
\text{hclause} \in \text{HClause} &::= \texttt{(hrel var ...}^{+}\texttt{)} \\
\text{hrel} \in \text{HRel} &::= \texttt{(rel-select name arity index)} \\
\text{bclause} \in \text{BClause} &::= \texttt{(brel var ...}^{+}\texttt{)} \\
\text{Brel} \in \text{BRel} &::= \texttt{(rel-version name arity index ver)} \\
\text{prim} \in \text{Prim} &::= \texttt{(op var ...}^{+}\texttt{)} \\
\text{name} \in \text{Name} &= \langle \textit{symbols} \rangle \\
\text{var} \in \text{Var} &= \langle \textit{symbols} \rangle \\
\text{op} \in \text{Op} &= \langle \textit{primitive operations} \rangle \\
\text{arity} \in \text{Arity} &= \mathbb{N} \\
\text{index} \in \text{Index} &= \mathbb{N}^{*} \\
\text{ver} \in \text{Version} &= \{\textbf{delta}, \textbf{full}\}
\end{aligned}
$$

Figure 3. PRAM domains used to define our core IR.

Our definition of a compiled PRAM program or IR is shown in Figure 3. A program is a directed acyclic graph among compiled strongly connected components (SCCs), of program rules, indicating which other SCCs must be run before each SCC can be run. SCCs with no incoming dependencies may be run immediately, and SCCs that contain only a single non-recursive rule are non-recursive and are not iterated to a fixed point because a single iteration always guarantees output consistent with the rule. PRAM supports four fundamental kinds of rules: a unary copy rule that propagates one relation into another, with possible

reordering and projection; i.e., `H(y,x) :- B(x,y,z).` ; a copy with primitive operation that can filter or extend tuples; i.e.,
`H(x,y) :- B(x,y), x<y.` ; a binary join rule that joins two relations with possible reordering and projection; i.e., `H(z,x) :- B0(x,y), B1(y,z).` ; and finally, an administrative copy rule between a canonical index for a relation and a non-canonical index for a relation.

Our system designates exactly one index for each relation as canonical (this may be the only index for a relation), and creates administrative rules that, acting like a unary copy rule, propagate newly discovered tuples from a canonical index to each of the non-canonical indices. The canonical index is always used in the head of a rule and our compiler generates appropriate indices and administrative copy rules. Note that an hclause is the same as a bclause except that it must be the canonical index and does not need a version tag, as newly discovered tuples are always placed in H_{new}. Body clauses are specific to either the version **delta** or **full**, an artifact of semi-naïve evaluation.

3.2 Implementation

Our implementation of a Datalog compiler and run-time is written in Racket and C++, composed of several passes:

Parsing and lexing. We support typical Datalog notation that overlaps with a substantial fragment of syntactic niceties supported by Soufflé. We track source locations explictly so we can report errors and we've included support for a few dozen common built-in operators to be attached to rules, such as the comparison `x < y` .

Organization. A first pass performs simplification and canonicalization steps. For example, we allow suggestions to be given for how to order rules as an extension, these are broken apart into a sequence of rules in this pass. Rules with multiple head clauses can also be split effectively into multiple rules with a single head-clause each. Wildcard variables, as in the clause `r(_,x)` are renamed to a unique anonymous variable; e.g., `r(_3,x)` .

Static unification. A second pass performs transitive static unification of variables that are explicitly unified by a built-in comparison, as in `p(x,x),q(y),x=y` , so equal variables across relations use the same variable name and within the same relation use a built-in to filter that relation; this example is converted to `p(x,_0),q(x),x=_0` .

Partitioning. The next pass partitions complex rules into a set of simple rules that form a linear chain of dependence on one another. For example, the rule

```
a(x,y,z) :- b(w,x), c(x,y), d(y,z).
```

is compiled into a sequence of two binary rules, inserting an internal relation representing the half-evaluated rule:

```
a(x,y,z) :- b(w,x), int(x,y,z).
int(x,y,z) :- c(x,y), d(y,z).
```

As balanced parallel relational joins must be performed on two relations with an identical selection index (to guarantee a compatible parallel decomposition of the two relations), the only rules permitted after this pass are unary copy rules that may reorder or project columns, unary built-in rules that may also perform a generative or filtering built-in operation (such as `x < y`), or a binary rule that joins two relations, projects some columns, and reorders columns for the head relation.

Complex rules that join four or more relations at once, are likewise partitioned into a linear chain of joins, each performed after the last. We also experimented with creating balanced binary trees of joins, and with various heuristics for partitioning some complex rules roughly in half, before recuring to partition each partition of body clauses. In nearly every case, we observed between $5\times$ and $25\times$ the overall tuple-load in these experiments (vs strictly linear chains of rules), representing a very substantial blow-up in intermediate relations. Intuitively, this is because in a linear chain it takes a greater number of iterations to compute the rule, more join operations between starting the rule and adding facts to the head relation, however the tuples involved are always maximally grounded and filtered by all relations taking into account so far. Using balanced binary trees of joins optimizes for lower latency, but leads to constraints apparent in the original rule being taken into account only after a much larger number of intermediate facts are materialized, at significant expense.

As the goal of our system is to exploit the massive data-parallelism of balanced parallel RA operations, we favor pipelining a longer sequence of RA operations in a linear chain, with the exception of cases where two subsets of body clauses have completely disjoint sets of variables, as in the example

```
h(x,y) :- f(a,x), g(x,b), p(c,y), q(y,d).
```

For this input, our compiler will produce two *independent* binary rules, for `f` and `g` and for `p` and `q`, followed by a single binary rule to compute the Cartesian product `int0(x),int1(y)` :

```
h(x,y) :- int0(x),int1(y).
int0(x) :- f(a,x), g(x,b).
int1(y) :- p(c,y), q(y,d).
```

As the Cartesian product is the actual semantics of our original rule, it does not represent a spurious blow-up. Note that the compiler will detect that variables such as `a` or `b` are not

required for the head, and so can be pruned from intemediate relations as well.

Selection splitting. This pass looks at the binary rules generated in the previous pass and determines the needed set of indices for each relation. This is a set of ordered subsets of their columns that may be used as a key for efficient access in a parallel join operation. If a relation is used in the body of a binary rule, it must have an index for the exact subset of columns it has in common with the other relation used in the binary rule, because we require such compatible matching indices for our parallel join algorithm.

This pass also decides which index is *canonical*, and modifies rules so that the head of every rule is the canonical index for that relation. Administrative rules are then added to propagate discovered tuples to all indices once discovered in the canonical index. These admin rules are essentially unary rules that copy tuples with possible column reordering. This makes it natural to compact the communication needed to maintain a set of indices for a relation with the normal communication used during a parallel join, in pipelined fashion. This is to say, the iteration after a tuple is discovered, it is copied to all its indices during the single communication phase shared by all rules.

Stratification. uses a modified Tarjan's algorithm [36] to compute a directed graph of strongly connected components (SCCs) in the dependency structure of rules. For example, rules that can be run upon initialization only once will appear as rules whose body relations are already available and never modified.

Incrementalization. prepares the program for seminaïve evaluation by explicitly splitting each relation into three versions, a **new**, **delta**, and **full** version as described previously. The body clauses of rules are modified so that static relations (those that do not change when this rule is evaluated) draw tuples from the **full** version, while dynamic relations must use the **delta** version. When two dynamic relations are joined, these are split into three rules that join **full** with **delta**, **delta** with **full**, and **delta** with **delta**. For complex k-ary join rules, these combinatoins blow up exponentially, so it ends up being convenient we are stuck with having already partitioned these down to chains of binary rules.

C++ emission. Finally, the incrementalized rules are written out as a driver in C++ corresponding to PRAM IR code in the previous section. That is to say, our backend's API interfaces to the IR the AST described in the last section. There is an object for encoding an administrative rule between two relations with a particular renaming and projection and another for encoding a join rule or a rule that operates on a relation using a built-in primitive to filter or extend the relation as tuples are emitted.

Run-time system. Our backend then finally interprets this PRAM IR, evaluating it efficiently in terms of our extension to the original BPRA source [24]. We have modified the library for BPRA into a more general run-time for PRAM IR. The original library was not developed to allow multiple RA rules to be evaluated in parallel, leading to some engineering challenges in terms of meta-data transfer and keeping track of all relations. We add a logical inference engine object which can be populated with any number of relations and can act generically as a primitive RDBMS and can accept a PRAM IR and set of input relations to evaluate a program and extract the final IDB.

Instead of running each RA operation in an SCC, one at a time, we modify evaluation to interleave all RA operations in a collective single operation with one shared communincation phase. Communication is non-uniform, where every process sends different amounts of data to every other process. This is typically implemented using `MPI_alltoallv`, but, in order to facilitate this non-uniform communication we first have to share the offsets and sizes of all relations and processes, so that every process has a consistent, global view. This meta-data exchange is implemented by `MPI_alltoall`. After this metadata exchange, we populate send buffers and receive buffers on every process before invoking `MPI_alltoallv`.

Unlike plain BPRA, every process performs a large batch of local joins (or copies, reorderings, projections, etc) for all given rules in the current SCC. This means that before a single comunication phase, each process will have generated output for potentially a large number of PRAM IR rules. We have introduced an crucial optimization in the form of comm-compaction where we concatenate all-to-all send buffers across all *rules* into one large buffer that can be transmitted in a single communication epoch. This step significantly cuts down communication costs, especially for compute intensive problem such as *k*CFA (see section 4) that have several rules in a single SCC and at each step of a collective fixed-point iteration for that SCC. This crucial improvement to RA that would otherwise be individually parallel, but not across multiple operations, requires ordering information to be broadcast during the epoch's fixed meta-data transfer, but appears to grant improved scalability versus Soufflé.

4 Case Studies

Graph-pattern mining (GPM) includes a rich source of core problems that highlight the expressivity of deductive databases. As a simple example we consider transitive closure (TC). We can implement TC using the following Datalog program, which iteratively computes the path relation:

```
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y), edge(y,z).
```

To compile this Datalog program to PRAM IR, we generate two initial administrative rules to set up path and prepare for the bulk of the work to be done in a subsequent SCC:

```
; Admin rule to load index for edge on column-1:
(arule
 ((rel-select edge 2 (1)) ident a1)
 ((rel-version edge 2 (1 2) delta) a1 ident))
; The first Datalog rule as a binary join rule:
(rule
 ((rel-select path 2 (1 2)) x y)
 ((rel-version edge 2 (1 2) total) x y))
; The next two rules are in an SCC together,
;    stratified to run after the previous rules.
; A rule to join path and edge:
(rule
 ((rel-select path 2 (1 2)) x z)
 ((rel-version path 2 (2) delta) y x)
((rel-version edge 2 (1) total) y z))
; Admin rule to propagate discovered paths to
; the index for path on column-2:
(arule
 ((rel-select path 2 (2)) a1 a0)
 ((rel-version path 2 (1 2) delta) a0 a1))
```

The third rule implements the iterative extension of path to compute TC. Observe that tuples are pulled from the delta version, implementing semi-naïve evaluation.

4.1 Program Analysis

Static program analysis is a key, impactful application of Datalog-like solvers that attempts to develop an accurate bounded model of program behavior based only on the program's source text. Program analyses are constructed using a variety of different theories and approaches; what these approaches share in common is the goal of obtaining sufficient precision for specific program properties while guaranteeing analysis termination, and ideally, efficiency. This central challenge of static analysis is made explicit in the methodology of *abstract interpretation* [14]. An abstract interpretation of a program evaluates its input source code in terms of imprecise or abstract values and machine components, permitting a careful loss of precision in exchange for reasonable bounds on analyis complexity.

A wide variety of abstract interpretations can be systematically engineered as Datalog programs, as has been extensively explored in the literature [17, 20, 21, 28, 42]. In particular the DOOP framework [9] for points-to analysis of Java, originally developed for LogicBlox, has been ported and optimized for Soufflé.

A class of these algorithms known as flow-analyses model the propagation of data-flow information or control-flow information through a target program [25]. Data-flow analysis (DFA) requires control-flow analysis (CFA) to obtain any reasonable precision for functional languages, for multi-paradigm languages like Java that support closures and methods associated with objects, or for structured languages with function pointers, as in C/C++: data-flow properties and control-flow properties are naturally entangled and must be simulated together to obtain a model with any reasonable precision [31, 32]. CFAs form an important foundation for analysis of most programming languages, in particular highly dynamic languages, and are often extended with additional client analyses for tracking relations among variables [5] or for verifying sophisticated contracts using abstract symbolic execution [27]. Systematic approaches to abstract interpretation of abstract-machine-based semantics [40] allow analyses to be developed from a variety of standard (concrete) abstract machines that precisely specify a language's semantics [26].

This systematic development of an abstract abstract machine from a concrete abstract machine yields is highly configurable and tunable, so it corresponds to a broad design space of analyses that strike subtly different trade offs between precision of result and complexity of analysis [16]. One classic instantiation of this framework yields *k-call-sensitive control flow analysis* (kCFA), a well-ordered hierarchy of CFAs with increasing precision and complexity as parameter k is increased. (Although complexity can both increase and decrease non-obviously as such tuning parameters are increased.) Our implementation of kCFA for the plain lambda calculus is about 40 lines in Datalog and can be easily tuned to any k to increase its degree of context sensitivity. We also wrote a worst-case input generator based on the worst-case input for kCFA discussed in [39]. This allows us to scale up either the term size, we call (t) or the context-sensitivity (k). We use this to generate sufficiently large experimental workloads, but we do not use either of these parameters to attempt weak-scaling experiments as they do not scale up the problem size in a simple way. Below is a fragment of our analysis showing how free variables are computed:

```
// Every variable is free at a reference to it
free(x, e) :- var-ref(e, x)
// At unary application a free variable for either
// subexpression is free at the call-site.
free(x, e) :-
  app(e, e0, e1),
  free(x, e0) or free(x, e1).
// At a lambda abstraction, variables free in the
// body that are not the formal parameter, are free.
free(x, e) :-
  lambda(e, y, body),
  free(x, body),
  x != y.
```

Free variable computation propagates information up the AST recusively, and forms an SCC in the compiled PRAM IR that is stratified before the primary CFA logic runs.

5 Evaluation

5.1 Dataset and HPC Platforms

We perform experiments on the Theta Supercompter at the Argonne National Lab and on a machine rented via Amazon Web Services (AWS). For AWS, each of our experiments was

run on an instance of type m5d.24xlarge consisting of 96 virtual CPUs (Intel Xeon Platinum 8000) and 384 GiB of RAM and NVMe-based SSD storage. Of these 96 virtual CPUs, we ran experiments utilizing up to either 60 processes (for MPI-based imlementation) or threads. Theta supercomputer is a Cray machine with a peak performance of 11.69 petaflops, 281,088 compute cores, 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM and 10 PiB of online disk storage. Theta uses the Dragonfly network topology and is backed by a Lustre filesystem. Theta's node is a Intel KNL 7230 which comprises of 64 physical cores.

5.2 Strong Scaling on AWS

To compare the single-node performance of our PRAM-based implementation versus Soufflé, we ran each case study using representative input data. For TC, and especially for CFA, we choose to focus on strong scaling both because it is more indicative of scaling performance and because it is difficult to accurately scale up the problem size for weak scaling. For k-CFA, the k parameter does not scale problem size up in a predictable way. We performed TC using a directed graph of the Arxiv High-Energy Physics paper citation network (consisting of 34,546 nodes and 412,578 edges) compiled by Gehrke et al. [15]. Second, we performed kCFA using a scaled-down version of the experiments. We use experiments labled t-k with a specific number of terms (t) and degree of sensitivity (k), as described in section 3.2. For our experiments here, we set term size (n) to be 100, and precision (t) to be 6.

We compiled each of our experiments using the compiler described in Section 3. We modified our compiler to generate a Soufflé program consisting of only binary joins and benchmark the results. We then used Soufflé's compiling mode to produce an optimized binary and we minimizeI/O overhead by dumping the smallest output relation. We performed each experiment three times, reporting the average of each of the runs, though runtime was roughly consistent across runs. For each of our experiments, we validated the correctness of results of Soufflé against our MPI-based implementation by checking both implementations produce the same set of tuples.

The results of our experiments are shown in Figure 4. We plot time (in seconds) along the y axis against process / thread count along the x axis. PRAM achieves better performance than Soufflé for kCFA in our benchmarks. We believe this is because Soufflé parallelizes individual joins, but does not interleave joins to perform them in parallel (further discussion is included in Section 3.2). Because kCFA contains more rules than our other benchmarks, the difference between sequential and parallel joins becomes more apparent. This demonstrates one advantage to our approach even at smaller scales.

For transitive closure computation our experiments showed modestly-decreasing runtime for Soufflé, however,

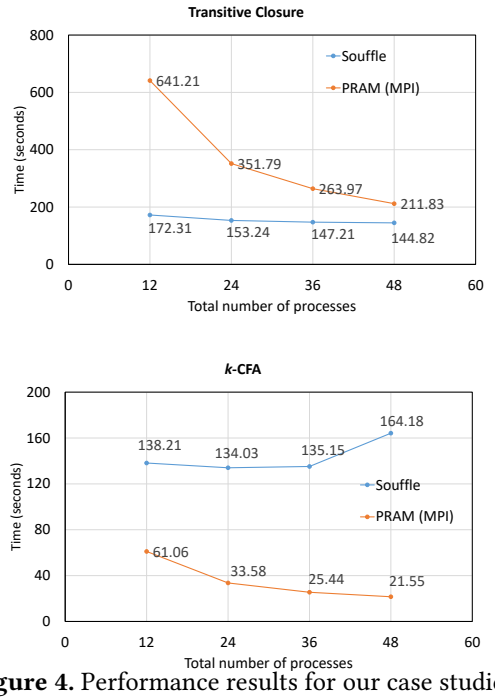

Figure 4. Performance results for our case studies

we observed poor scaling characteristic when compared to PRAM. At a high level, our results demonstrate that our system scales better than Soufflé as degree of parallelism increases but achieves worse constant factors. One key difference between PRAM and Soufflé is tuple representation. While Soufflé employs highly-optimized shared memory datastructures (discussed briefly in Section 2). PRAM uses messsage passing and an off-the-shelf B-tree implementation to communicate and represent tuples. Because of this, when lots of tuples are generated during an iteration PRAM will allocate large amounts of memory to exchange messages and represent tuples. We see underlying tuple representation as an important but orthogonal issue to our underlying scaling approach, and plan to investigate using Soufflé's datastructures in future work.

5.3 Strong Scaling on Theta

We ran two single node experiments run at 32 and 64 cores (nodes on Theta support 64 threads per node). We were unable to run experiments at smaller scale due to memory constraints. We ran a k-CFA worst-case instance with 80 terms and $k = 7$, which took 2648 iterations to complete, turning 724 EDB facts into 165,389,799 IDB facts. We observe a speed-up of 1.76× and scaling efficiency of 88% while going from 32 to 64 processes. We have also run some preliminary multi-node experiments with Theta and have observed 82% and 100% scaling efficiency on two k-CFA benchmarks at 512 threads. Overall our experiments are indicative of healthy scaling.

6 Related Work

Relational Algebra. Our work directly builds upon BPRA, parallel relational algebra primitives designed to scale on supercomputers [23, 24]. This implementation leverages the observation that joins can be distributed to a cluster via a double-hashing approach, consisting of local hash-based joins and hash-based distribution of relations. Their double-hashing approach is inspired by the earlier work of Cheiney et al. [13] and Cacace et al. [12], who describe data-parallel strategies for path computation.

Barthels et al. describe a system for distributing the radix hash join and merge-sort join algorithms [7]. Their implementation scales to 4,096 cores via MPI and reaches extremely high tuple throughput at peak load. Work by Kim et al. and Balkesen et al. demonstrates how these joins may be further accelerated via AVX/SIMD instructions [6, 22]. While this work successfully scales a single join iteration, it does not reorganize or balance tuples to allow subsequent joins, and thus does not readily enable the fixed-point computation necessary for deductive databases.

In BPRA, tuples are distributed via a two-layered distributed hash-table which multiplexes tuples onto a statically fixed set of *buckets* and dynamically-tunable set of *subbuckets* [23]. Each tuple is assigned a bucket based on the hash of its join columns; this then enables local hash-based joins. Next, all-to-all communication is performed to communicate the result of each join to its appropriate bucket and subbucket. In subsequent work they develop strategies to enable spatial and temporal load balancing of tuples across the cluster, and use these techniques to perform the largest-ever computation of transitive closure [24].

Program Analysis and Datalog. Deductive databases offer an attractive option for the implementation of large-scale program analyses as they enable declarative analysis specification alongside efficient solving via modern Datalog engines. the DOOP framework by Smaragdakis et al. poineered an elaborate context-sensitive points-to analysis for Java implemented in Datalog [10, 34]. DOOP originally used the LogicBlox Datalog engine to achieve an order of magnitude speedup compared to a predecessor hand-written points-to analysis for Java [4]. DOOP was later ported to the Soufflé Datalog engine, which enabled further scalability via Soufflé's single-node task-level parallelism [3]. While Soufflé represents the state-of-the-art analysis platform, it is fundamentally limited in that it cannot provide data parallelism, hindering it from operating beyond a single node. By contrast, our parallel relational-algebra approach can likely be scaled to clusters.

There is also some recent work on program analysis and graph processing via Apache Spark. Our approach leverages a set of specific underlying techniques (BPRA) that are developed from the ground up, atop MPI directly, to balance communication and computation in a manner Spark's communication paradigm does not allow. Spark is not appropriate for leadership class supercomputers such as Theta, which permit granular and tunable communication that we can exploit directly via MPI, leading to a noted gap in performance [37]. Spark is based on Java and optimized to leverage commodity-class nodes coordinating via standard network hardware (vs. InfiniBand on Theta). For example, work on BigDatalog [**?**] gives benchmarks for transitive closure that our system has exceeded in scale by more than a factor of 1000. As discussed in [24] section 3, BPRA specifically addresses communication issues that arise at this scale.

There are several other notable efforts in distributed and parallel program analysis that achieve scalability via application-specific task-level parallelism. For example, Aiken et al.'s Saturn program analysis system includes a distributed mode via MPI, which they anecdotally report achieves scalability [2]. Their system distributes the analysis via a worklist of function summaries and distributing work among the cluster. This approach assumes that the analysis is summarization-based and does not offer data parallelism. Similarly, there have been multiple efforts to distribute symbolic execution [11, 29, 33]. The recent Formulog system harmoniously integrates Datalog, functional programming, and constraint solving, and may provide useful inspiration for future work [8]. In contrast to these systems, our approach offers true data parallelism, enabling the entire cluster to make progress on the analysis at once rather than requiring application-specific task deliniation.

7 Conclusion

Exciting advances in high-performance Datalog solvers have enabled new frontiers in large-scale static analysis. However, current-generation Datalog solvers are fundamentally limited in parallelism. We've presented a methodology building upon emerging work in data-parallel relational algebra that allowed us to build the first MPI-based Datalog solver. Our solver is built on a novel parallel relational algebra machine, which makes several key decisions to enable implementing Datalog rules via data-parallel relational algebra. We see this as a foundational step forward in the implementation of high-performance logical-inference engines. Our benchmarks demonstrate that our approach achieves better single-node scalability than Soufflé, the state-of-the-art Datalog engine. Additionally, we observed promising initial scalability of up to 512 threads on the Theta supercomputer. In future work, we hope to build next-generations platforms for graph mining, program analysis, and other large-scale logical inference problems.

Acknowledgments

We are grateful to the ALCF Director's Discretionary program for providing us with compute hours on Theta.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc.

[2] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2007. An Overview of the Saturn Project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*. Association for Computing Machinery, New York, NY, USA, 43–48. https://doi.org/10.1145/1251535.1251543

[3] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 25–30.

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[5] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming* 72, 1-2 (2008), 3–21.

[6] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013), 85–96. https://doi.org/10.14778/2732219.2732227

[7] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 517–528.

[8] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog = Datalog + ML + SMT.

[9] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. *SIGPLAN Not.* 44, 10 (Oct. 2009), 243–262. https://doi.org/10.1145/1639949.1640108

[10] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[11] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 183–198. https://doi.org/10.1145/1966445.1966463

[12] Filippo Cacace, Stefano Ceri, and Maurice A. W. Houstma. 1991. An Overview of Parallel Strategies for Transitive Closure on Algebraic Machines. In *Proceedings of the PRISMA Workshop on Parallel Database Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 44–62.

[13] Jean-Pierre Cheiney and Christophe de Maindreville. 1990. A Parallel Strategy for Transitive Closure Using Double Hash-based Clustering. In *Proceedings of the Sixteenth International Conference on Very Large Databases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 347–358.

[14] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[15] Johannes Gehrke, Paul Ginsparg, and Jon Kleinberg. 2003. Overview of the 2003 KDD Cup. *SIGKDD Explor. Newsl.* 5, 2 (Dec. 2003), 149–151. https://doi.org/10.1145/980972.980992

[16] Thomas Gilray, Michael D. Adams, and Matthew Might. 2016. Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-flow Analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16)*. ACM, New York, NY, USA, 407–420. https://doi.org/10.1145/2951913.2951936

[17] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[18] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. [n.d.]. A Specialized B-tree for Concurrent Datalog Evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 327–339.

[19] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. Brie: A Specialized Trie for Concurrent Datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19)*. ACM, New York, NY, USA, 31–40.

[20] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. *SIGPLAN Not.* 48, 6 (June 2013), 423–434. https://doi.org/10.1145/2499370.2462191

[21] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

[22] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[23] Sidharth Kumar and Thomas Gilray. 2019. Distributed Relational Algebra at Scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE.

[24] Sidharth Kumar and Thomas Gilray. 2020. Load-Balancing Parallel Relational Algebra. In *High Performance Computing*, Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 288–308.

[25] Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM computing surveys (CSUR)* 44, 3 (2012), 1–33.

[26] Matthew Might. 2010. Abstract interpreters for free. In *International Static Analysis Symposium (SAS '10)*. Springer, 407–421.

[27] Phúc C Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2017. Soft contract verification for higher-order stateful programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 51.

[28] Wytse Oortwijn, Tom van Dijk, and Jaco van de Pol. 2017. Distributed binary decision diagrams for sbolic reachability. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM, 21–30.

[29] R. Sasnauskas, O. S. Dustmann, B. L. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski. 2011. Scalable Symbolic Execution of Distributed Systems. In *2011 31st International Conference on Distributed Computing Systems*. 333–342.

[30] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 196–206.

[31] Olin Shivers. 1988. Control Flow Analysis in Scheme. In *Proceedings of the Conference on Programming Language Design and Implementation.*

ACM, New York, NY, 164–174.

[32] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. Carnegie-Mellon University, Pittsburgh, PA.

[33] J. H. Siddiqui and S. Khurshid. 2010. ParSym: Parallel symbolic execution. In *2010 2nd International Conference on Software Technology and Engineering*, Vol. 1. V1–405–V1–409.

[34] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded (Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 245–251.

[35] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-scale Datalog Computation. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 141–153.

[36] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[37] George K Thiruvathukal, Cameron Christensen, Xiaoyong Jin, François Tessier, and Venkatram Vishwanath. 2019. A benchmarking study to evaluate apache spark on large-scale supercomputers. *arXiv preprint arXiv:1904.11812* (2019).

[38] Patrick Valduriez and Setrag Khoshafian. 1988. Parallel Evaluation of the Transitive Closure of a Database Relation. *Int. J. Parallel Program.* 17, 1 (Feb. 1988), 19–42.

[39] David Van Horn and Harry G Mairson. 2008. Deciding k CFA is complete for EXPTIME. *ACM Sigplan Notices* 43, 9 (2008), 275–282.

[40] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1863543.1863553

[41] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 763–782.

[42] John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. *SIGPLAN Not.* 39, 6 (June 2004), 131–144.