A Probabilistic View of Datalog Parallelization

Sérgio Lifschitz* Victor Vianu†

Abstract

We explore an approach to developing Datalog parallelization strategies that aims at good *expected* rather than worst-case performance. To illustrate, we consider a very simple parallelization strategy that applies to all Datalog programs. We prove that this has very good expected performance under equal distribution of inputs. This is done using an extension of 0-1 laws adapted to this context. The analysis is confirmed by experimental results on randomly generated data.

1 Introduction

The performance requirements of databases for advanced applications, and the increased availability of cheap parallel processing, have naturally lend great importance to the development of parallel processing techniques for databases. Much of the existing research in this direction has focused on parallelization of Datalog queries. In this paper we investigate parallel processing of Datalog from a probabilistic viewpoint. In contrast to existing work, we propose to guide the design and evaluation of parallelization strategies by *expected* performance. We evaluate the expected performance of parallelization strategies using *0-1 laws*, a powerful tool from logic. We illustrate this approach to design and analysis using a very simple pure parallelization technique, applicable to all Datalog programs. Rather surprisingly, we show that this simple strategy has *provably* very good expected performance. This is backed up with experimental results.

Most existing approaches to parallel processing of Datalog fall into a few broad categories. In the *pure parallelization* approach, the data (or rule instantiations) are partitioned among a fixed number of sites according to some criteria. Then the Datalog program is run in parallel without communication among the sites. The final result consists of the union of the results at each site. Programs for which this algorithm is complete (i.e., computes the

*PUC-Rio Informática, Rua Marquês de São Vicente 225, Rio de Janeiro, RJ 22453-900, Brasil, lifschit@inf.puc-rio.br. This work was done while the author was affiliated with the Ecole Nationale Supérieure des Télécommunications (ENST), Paris, France.

†Univ. of California at San Diego, CSE 0114, La Jolla, CA 92093-0114, vianu@cs.ucsd.edu. Work performed in part while visiting ENST, and supported in part by the NSF under grant IRI-9221268.

entire result) are called *sharable* [26] or, in the case of disjoint results at each site, *decomposable.* Sufficient syntactic conditions for decomposability have been provided in [9, 28, 21]. The property of being decomposable was shown undecidable in [27].

The alternative to the pure parallelization approach is to allow *communication* among the sites. Among methods using communication, we can identify two different approaches. One is the so called rule instantiation partitioning strategy [12, 27, 29], which is a pure bottom-up method that parallelizes the evaluation by partitioning the rule instantiations of a program among the sites, such that each site evaluates the same original program but with less data. Since communication ensures completeness, there is no longer a need for restricting programs to ensure decomposability, as in the pure parallelization approach.

Another possibility is the so called rule (or clause) decomposition, where the rules of a program are evaluated in a top-down style, employing *sideways information passing* to focus on relevant facts [24, 14], sometimes considering a pipelined strategy [5]. Such methods are useful primarily when the query goal contains constants. They usually require an initial compilation phase. In the presence of constants in the query, bottom-up optimization techniques, like the Magic Sets rewriting method, seem to yield better results. A survey of these strategies can be found in [22].

The general paradigm in the approaches described above is to propose heuristics for parallelization, such that the resulting parallel algorithms are always *complete.* Completeness means that the union of the results from each site at the end of the algorithm provides the correct result. Much of the effort and cost in these approaches goes into guaranteeing completeness *in all cases.* As discussed above, this may include inter-site communication during evaluation, pre-processing, or syntactic restrictions on the programs.

Here we explore a somewhat different approach. We argue that the price paid for guaranteeing completeness in all cases is too high and affects adversely the *average* performance. Instead, we propose approaching the issue of completeness from a probabilistic point of view. To illustrate this idea, we examine a parallelization strategy which is extremely straightforward, works on all programs, uses no communication, but which no longer guarantees completeness in all cases. In those cases where incompleteness occurs, a sequential "completion" phase at the end of the parallel processing phase is required. In the worst case, no gain is obtained from the parallelization, since the "completion" portion of the algorithm may be as costly as a purely sequential algorithm. The key to the approach is to show that the cases where incompleteness occurs are rare. Indeed, we prove that for almost all inputs (which is given a precise probabilistic meaning), the result of the parallel phase is complete and the overall performance of the algorithm is much better than the monosite algorithm.

The probabilistic results on the behavior of our parallelization algorithm are obtained using an elegant tool from logic: *0-1 laws* [10]. These establish a connection between logical languages and the probability that properties expressible in these languages are true. Thus, the fact that a property of data is expressible in a language that has a 0-1 law provides information on the probability that the property is true. To use these results, we show that "good" behavior of our parallelization algorithm is a property of data that can be expressed by a sentence that is asymptotically true, in a language having a 0-1 law. This yields the results on the good performance of the algorithm on almost all inputs.

So far, the primary use of 0-1 laws in databases has been as a tool for understanding the expressive power of query languages. For example, this is used to show that the parity of a set is not expressible in languages like FO [10], *fixpoint*, and *while* [18]. The average complexity of various query languages has been investigated using 0-1 laws in [1]. This also included a probabilistic analysis of certain query optimization algorithms, for which heuristics were then developed in [3]. To our knowledge, the present work is the first to use 0-1 laws in the analysis of concrete parallelization strategies.

The probabilistic results described above should be viewed simply as guiding information towards the design of parallelization algorithms. Indeed, they make the important assumption that no information is available about the database, so all instances are equally likely. For example, this means that if the input is a graph (binary relation), the probability that there is an edge between two nodes is $1/2$. This is clearly not true for many applications. Data is often more sparse, or has known structure. The results extend to some of these situations, such as some sparsity assumptions. For example, they remain true for edge probability $1/c$ where c is any constant, and even when the probability is a small function of the number n of nodes, such as $n^{-1/\log\log n}$. However, in applications where more information is available on the structure of data, the pure parallelization algorithm presented here should be considered just as a starting point for the development of algorithms more adapted to that particular application.

Thus, the main contribution of the present paper lies not as much in a specific algorithm, but rather in the idea of guiding the design of parallelization strategies by expected rather than worst-case behavior in the context of known characteristics of data. The algorithm and results presented here illustrate the "limit" situation where all inputs are considered equally likely. The experimental results provide insight into the actual performance of pure parallelization for various types of inputs. Several Datalog queries on graphs are considered. As suggested by the analytical results, in the case of randomly generated graphs the behavior of the algorithm is excellent. Inputs with pre-defined structure, such as acyclic graphs and complete binary trees, are also considered. As expected, the behavior of the algorithm is less impressive for these cases. For acyclic graphs, the pure parallelization algorithm is nonetheless still faster than the sequential algorithm. For complete binary trees however, the pure parallelization algorithm performs worse than the sequential algorithm. This confirms our expectation that the algorithm needs to be adapted to the specific structure of data, in applications where such structure is known.

The paper is organized as follows. We begin with a preliminary section describing some database languages to be used, the notion of asymptotic probability, and 0-1 laws. The following two sections present the pure parallelization algorithm and the analytical results on its performance; this requires developing a variation of 0-1 laws adapted to this context. After providing performance results for all Datalog programs, we identify a large class of programs for which extremely good performance can be shown. Section 6 provides our experimental results, and the last section contains concluding remarks.

2 Background

We briefly review in this section several query languages referred to in the development, as well as some basic definitions and results related to 0-1 laws. We assume familiarity with the basic terminology and notation of relational databases, and with the first-order queries, FO, expressed by the relational calculus and algebra (see [22, 2]).

2.1 Some query languages

There are many useful queries that FO cannot express, such as the transitive closure of a graph. Numerous extensions of FO with recursion have been proposed. Most of them converge towards two central classes of queries: *fixpoint* and *while* [6, 7]. These can be defined in various ways: by adding fixpoint operators to FO, looping constructs to relational algebra, or by extensions of Datalog. We briefly review here the definition of the language Datalog, and of *fixpoint* and *while* using looping constructs.

While and Fixpoint. *While* extends FO with recursion. It provides relation variables, FO* statements of the form $R := \varphi$, and a looping construct *while* φ *do* where φ is an FO condition. *Fixpoint* is the same as *while* except the semantics of assignment is cumulative (i.e., an assignment $R := \varphi$ *adds* φ to the current content of R). This guarantees termination of *fixpoint* programs in PTIME, whereas *while* programs require PSPACE.

Datalog. A Datalog program is a set of rules of the form

$$A_0(t_0) \leftarrow A_1(t_1), \ldots, A_n(t_n)$$

where each A_i is a relation, t_i are tuples of variables of appropriate arities, and each variable in t_0 occurs in some t_i, $i > 0$. In a Datalog rule as above, $A_0(t_0)$ is called the *head* and $A_1(t_1), \ldots, A_n(t_n)$ the *body* of the rule. The relations occurring in heads of rules are the *intensional* (idb) relations, and the others are *extensional* (edb) relations. For example, the following Datalog program defines a binary relation T containing the transitive closure of a graph G:

$$\begin{array}{rcl} T(x,y) & \leftarrow & G(x,y) \\ T(x,y) & \leftarrow & T(x,z), G(z,y). \end{array}$$

Here, G is extensional and T intensional. It is usually assumed that one of the intensional predicates is designated as the *answer*. So, a Datalog program defines a query from instances over its edb relations to instances of the answer idb predicate. For simplicity of exposition, we often consider in this paper programs with one binary edb relation G (defining the edges of a graph) and binary answer.

The answer to a Datalog program can be computed by "firing" the rules for as long as new tuples can be inferred. A rule firing consists of finding all substitutions of variables by constants such that the body of the rule is satisfied, and adding the corresponding heads of the rule to the database. A *stage* consists of one firing of all the rules.

Replacing the variables by constants in a rule yields an *instantiation* of the rule. If the body of the instantiation is satisfied then the instantiation is *applicable*. An instantiation whose head has not yet been inferred is *potentially productive*. An instantiation that is both potentially productive and applicable is called *productive*. If P is a Datalog program and $\mathbf{I}$ an input database, $P(\mathbf{I})$ denotes the result of applying P to $\mathbf{I}$.

It is sometimes useful to consider the *expansions* of a Datalog program, obtained by unfolding the program all the way down to edb predicates, starting from the answer predicate, some fixed number of times[1]. Expansions are of the form $\exists\vec{u}\gamma$, where γ is a conjunction of atoms of edb predicates or equality. When the program is understood, we denote by Γ_i, $i \geq 0$, the set of expansions obtained by unfolding the program i times, and $\Gamma = \bigcup_{i\geq 0} \Gamma_i$. We describe Γ_i more precisely for binary answer predicates (this extends without difficulty to arbitrary arity). If the binary answer predicate is *answer*, first fix two variables x, y; Γ_i consists of the formulas $\exists\vec{u}([A_0 \wedge] A_1 \wedge \ldots \wedge A_n)$ with free variables x, y, where:

- A_i, $1 \leq i \leq n$, are atoms of edb predicates and $A_1, \ldots, A_n$ is obtained from $answer(x, y)$ by an SLD-derivation[2] of length i. Furthermore, we assume without loss of generality that x, y are not renamed in the derivation.
- A_0 is omitted if x, y occur in $A_1, \ldots, A_n$, and A_0 is $x = y$ if one of x or y does not occur in $A_1, \ldots, A_n$.

Similarly, expansions can be considered with respect to any of the idb predicates of the program.

For the transitive closure program above, $G(x, y) \in \Gamma_1$, $\exists z(G(x, z) \wedge G(z, y)) \in \Gamma_2$, etc. A Datalog program is equivalent to the infinitary disjunction of all formulas in Γ.

While Datalog provides recursion, it defines only monotonic queries. That is, if $\mathbf{I} \subseteq \mathbf{J}$ then $P(\mathbf{I}) \subseteq P(\mathbf{J})$. Thus, FO and Datalog are incomparable. Datalog is subsumed by the *fixpoint* (and therefore *while*) queries.

2.2 0-1 laws

0-1 laws provide an elegant connection between logical languages and the probability that properties expressible in the languages are true. Consider any property σ of inputs. Let $\mu_n(\sigma)$ be the fraction of inputs with n constants which satisfy σ. We say that σ *is almost surely true (or holds almost everywhere)* iff

$$\lim_{n\to\infty} \mu_n(\sigma) = 1.$$

A language L is said to have a 0-1 law if for all properties σ expressible in L, either σ is almost surely true or almost surely false. Surprisingly, it turns out that most of the

[1] In addition to the edb predicates, expansions may use equalities among the free variables.

[2] See [2] for a brief description of SLD resolution specialized to the Datalog framework.

commonly used query languages have a 0-1 law. This includes FO [10], Datalog, the *fixpoint* and *while* queries [18]. For example, take the property that a graph contains a cycle of length 3. Clearly, this property is expressible in FO. Thus, it is almost surely true or almost surely false. In this case, the property is almost surely true. For several languages, there are algorithms to determine whether a sentence in the language is almost surely true, or if it is almost surely false (e.g., see [13, 17]).

An important class of FO sentences that are almost surely true are the so-called *extension axioms*. For simplicity, we describe extension axioms for directed graphs (whose edges are provided by a binary relation G). This can be extended to inputs of arbitrary arity. Let k be fixed. The k-ary extension axioms, denoted $\mathcal{A}_k$, are sentences stating that any subgraph of k nodes can be extended by an additional node in all possible ways. More precisely, for each k, $\mathcal{A}_k$ contains all sentences of the form

$$\forall x_1 ... \forall x_k((\bigwedge_{i \neq j}(x_i \neq x_j)) \Rightarrow \exists y(\bigwedge_i(x_i \neq y) \wedge connections(x_1, ..., x_k; y)))$$

where $connections(x_1, ..., x_k; y)$ is a conjunction of literals containing, for each x_i, one of $G(x_i, y)$ or $\neg G(x_i, y)$, and one of $G(y, x_i)$ or $\neg G(y, x_i)$. For example, for $k = 3$, one of the 4^3 sentences of the form $connections(x_1, x_2, x_3; y)$ is

$$G(x_1, y) \wedge \neg G(y, x_1) \wedge \neg G(x_2, y) \wedge \neg G(y, x_2) \wedge G(x_3, y) \wedge G(y, x_3).$$

There is a close connection between extension axioms and the expansions of Datalog programs. Let $\exists \vec{u} \gamma$ be one expansion of a Datalog program with binary answer, and let x, y be its free variables. Consider the sentence $\forall x \forall y \exists \vec{u} \gamma$. If all atoms in γ are literals containing at least one variable in $\vec{u}$, then this sentence is implied by a finite set of extension axioms, and is therefore almost surely true. In such case, it follows that the Datalog program produces the complete graph on almost all inputs. This is the case for the transitive closure program above, and for many other programs. However, there are Datalog programs for which this property does not hold, because all its expansions contain some atoms involving just x, y. For example, such is the program

$$\begin{array}{lcl} T(x, y) & \leftarrow & G(x, y), G(x, x), G(y, y) \\ T(x, y) & \leftarrow & T(x, z), T(z, y) \end{array}$$

Indeed, all expansions contain the atoms $G(x, x)$ and $G(y, y)$. This program does not produce the complete graph on almost all inputs.

3 Pure Parallelization

Let the *arity* of a Datalog program P, denoted *arity(P)*, be the maximum arity of a relation used in P. The input to the parallelization algorithm consists of a graph $\mathbf{I}$, a Datalog program P of arity a, and a positive integer k, $k > a$. The algorithm uses C_k^a (k choose a) sites denoted s_A for each $A \subseteq \{1, \ldots k\}$, $|A| = a$, in addition to a central site. The algorithm consists of two phases:

- **Parallel phase.** The input database $\mathbf{I}$ resides originally at the central site.
  1. distribute $\mathbf{I}$ among the sites as follows: first, construct an arbitrary ordering $\leq$ of the nodes in $\mathbf{I}$; then, all arcs of $\mathbf{I}$ among nodes x such that $x \equiv i \ mod \ k$ for $i \in A$ are sent to site s_A. (The *mod* predicate applies to the rank of the node in the ordering $\leq$.)
  2. execute P in parallel at all sites s_A.
  3. collect the results from all s_A at the central site.
- **Completion phase.** This continues the computation of P at the central site starting from the results collected from the s_A:

  repeat until no new facts are inferred

  1. Compute all potentially productive instantiations of rules in P (i.e., all instantiations whose heads produce facts not present in the partial result).
  2. Add the heads of all applicable instantiations generated in (1) to the partial result.

Note that many commonly arising programs, such as those for transitive closure and same generation queries, have arity 2. In this case, the number of sites used is $\mathrm{C}_k^2 = k(k-1)/2$.

In the parallel phase, the partitioning of instantiations is based on the *mod* function; this is one way of obtaining a nice, balanced distribution of the data among sites. However, this is not crucial: other functions, and a variety of other partitioning strategies, could be used as well.

In order to evaluate the performance of the above algorithm, we use a common cost model that counts the number of rule instantiations generated by the algorithm (whether applicable/productive or not). In the analysis, we assume evaluation of the program by forward chaining up to a fixpoint, in both the sequential and the local evaluations at the sites. We assume that all instantiations are produced at each stage in the bottom up evaluation. Various optimizations of this algorithm could be factored in, but we do not focus on this aspect here. The cost of the parallel phase is the maximum of the costs at each site. The cost of the completion phase is the cost of computing the complete answer at the central site starting from the results collected from the sites. The total cost of the parallel algorithm with program P and input $\mathbf{I}$ is the sum of the costs of the two phases, denoted $\textit{parallel-cost}_{P,k}(\mathbf{I})$. For comparison purposes, we also consider the purely sequential algorithm which computes the result using just the central site, as in the completion phase above, but with no parallel phase. The cost of the sequential algorithm with program P and input $\mathbf{I}$ is denoted $\textit{sequential-cost}_P(\mathbf{I})$. The cost measure counting instantiations is a rough abstraction that ignores certain costs, such as:

1. distribution of edges to the sites, and pooling of the partial results to the central site;

2. the cost of computing the potentially productive instantiations in the completion phase of the algorithm.

Item (1) is a communication cost, incomparable to the other costs taken into account, and which can be easily factored in, in the context of a particular machine or architecture. The cost measure we use here is in line with previous investigations related to Datalog, such as [23, 4], where the cost measure is also based on rule instantiations.

4 An extension of 0-1 laws

To evaluate probabilistically the behavior of our algorithm, we use 0-1 laws. Strictly speaking, we are interested in the asymptotic probability that a given *run* of our algorithm will have certain properties. Note that, if an algorithm is deterministic (there is only one run for each given input) then a property of runs can be viewed as a property of the inputs generating the runs, and we can use the usual framework of 0-1 laws. For nondeterministic algorithms (like ours), there is a distinction between properties of runs and properties of inputs, because each input generates several runs. Among those runs, some may have the desired property, whereas others might not. Thus, we need a variation in the definition of asymptotic probability that counts runs rather than structures. (Not to worry: we will soon reduce counting runs to counting structures.) Let $\mathcal{A}$ be a nondeterministic algorithm which generates a finite number of runs for each input. Let P be a property of runs, and $\mu_n^{\mathcal{A}}(P)$ be the fraction of runs on inputs of size n that satisfy P.

Definition 4.1 *Property P is* almost surely true *for Algorithm $\mathcal{A}$ iff*

$$lim_{n\to\infty}\mu_n^{\mathcal{A}}(P) = 1$$

and P is almost surely false *iff*

$$lim_{n\to\infty}\mu_n^{\mathcal{A}}(P) = 0.$$

Let us consider now the pure parallelization algorithm, with a graph (binary relation) G as input. Note that the nondeterminism occurs in step (1), and consists of guessing an ordering of the nodes of G. This extends the given graph to an ordered structure. Once the order is constructed, the remainder of the run is uniquely determined. Using this, we can reduce properties of runs of our algorithm to properties of structures. We use the given structure extended with an ordering. Let G be the input schema and $G_{\leq}$ be an extension of G with a binary relation $\leq$ interpreted as a total order on the active domain of G. Let P be a property of runs of the pure parallelization algorithm over G and $P_{\leq}$ the property of ordered structures over $G_{\leq}$ defined by: $\mathbf{I}$ has property $P_{\leq}$ iff the run of the pure parallelization algorithm that produces $\mathbf{I}$ from $\mathbf{I}|G$ has property P. The following is now obvious:

Fact: Pure parallelization has property P almost surely iff $P_{\leq}$ holds almost surely on ordered structures over $G_{\leq}$.

We will use the above to show that pure parallelization has certain desirable properties almost surely. One way to show that pure parallelization has a property P of interest will be to describe property $P_{\leq}$ in a language that has a 0-1 law on ordered structures over $G_{\leq}$. This guarantees that the behavior almost always or almost never occurs. Finally, it must be determined which of the two holds.

Finding an appropriate language that allows expressing properties of interest to us while at the same time having a 0-1 law on ordered structures over $G_{\leq}$ has to be done with care, since it is well known that most languages which have a 0-1 law on arbitrary structures do not have a 0-1 law on ordered structures. We next consider this issue.

Consider the property that the first phase of pure parallelization is complete when applied to some query φ. To express this property we need to use, besides the input, predicates stating that a node belongs to site s_A. This involves the use of the *modulo* predicate. Given a domain ordered by $\leq$ and $k > 1$, we use k predicates $mod_i, 0 \leq i < k$ defined by: $mod_i(x)$ holds iff the rank of x wrt $\leq$ is i modulo k. Stating that x is in s_A is then written, in a language allowing disjunction, as: $\vee_{i \in A} mod_i(x)$.

Let k be fixed and let L be any of the query languages discussed so far. Then $L\tilde{+}\tilde{mod}$ denotes L extended with the predicates mod_i, $0 \leq i < k$. More precisely, let $\mathbf{R}_{\leq}$ be a database schema including a binary relation $\leq$ (interpreted as an order). A query φ over $\mathbf{R}_{\leq}$ in the language $L + mod$ is a query in L using relations in $\mathbf{R}_{\leq}$ *other than* $\leq$, and mod_i, $0 \leq i < k$. Note the following important facts:

(1) All queries in $L + mod$ operate on ordered structures; and,

(2) a query in $L + mod$ over $\mathbf{R}_{\leq}$ cannot use the ordering $\leq$ directly; however, $\leq$ is used indirectly via the *mod* predicates.

It is easily seen that, if φ is expressible in L, then the answer to φ at site s_A can be defined by some φ_A in $L + mod$. Since the statements of interest to us make use of the *mod* predicates, the 0-1 laws need to be extended to languages using these predicates in addition to database relations.

Let σ be a sentence in $L + mod$, over a database schema $\mathbf{R}_{\leq}$. Extending the 0-1 laws from L to $L + mod$ does not pose any serious problem, although it does require reconstructing the proof of the 0-1 laws for FO and *while* (see [11]) in the presence of *mod*. In particular, the extension axioms now involve the mod_i unary predicates, which act much like a labeling of nodes. Since all languages L discussed here are subsumed by *while*, it is sufficient to show:

Theorem 4.2 While + mod *has a 0-1 law.*

Proof: The proof follows closely the classical proof that *while* has a 0-1 law: first show that FO has a 0-1 law, then extend this to *while*. The extension to FO + *mod* and $while\tilde{+}\tilde{mod}$ is routine; we only present here the core of the proof which consists of showing that the extension axioms using G and the *mod* predicates are almost surely true. We sketch this argument, along the lines of [2]. Let G be the input graph and $G_{\leq}$ be G extended with a total order on the its nodes. The extension axioms involve G and the new predicates $mod_i, 0 \leq i < k$. An extension axiom is of the form

$$\forall x_1 ... \forall x_m((\bigwedge_{i\neq j}(x_i \neq x_j)) \Rightarrow \exists y(\bigwedge_i(x_i \neq y) \wedge connections(x_1, ..., x_m; y) \wedge \ \ mod_h(y)))$$

where $connections(x_1, ..., x_m; y)$ is some conjunction of literals containing, for each x_i, one of $G(x_i, y)$ or $\neg G(x_i, y)$, and one of $G(y, x_i)$ or $\neg G(y, x_i)$. For example, for $m = 3$ and $k = 10$, one of the $2^6 \times 10$ extension axioms is:

$$\begin{array}{l} \forall x_1, x_2, x_3 \ (\ (x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_3 \neq x_1) \Rightarrow \\ \quad \exists y \ (x_1 \neq y \wedge x_2 \neq y \wedge x_3 \neq y \wedge \\ \quad G(x_1, y) \wedge \neg G(y, x_1) \wedge \neg G(x_2, y) \wedge \neg G(y, x_2) \wedge G(x_3, y) \wedge G(y, x_3) \wedge mod_7(y) \ ) \ ). \end{array}$$

Consider the following extension axiom σ_m:

$$\forall x_1 ... \forall x_m((\bigwedge_{i\neq j}(x_i \neq x_j)) \rightarrow \exists y(\bigwedge_i(x_i \neq y) \wedge connections(x_1, ..., x_m; y) \wedge mod_h(y))).$$

Then $\neg\sigma_m$ is the sentence

$$\begin{array}{l} \exists x_1 ... \exists x_m(\ (\bigwedge_{i\neq j}(x_i \neq x_j)) \wedge \\ \qquad \qquad \forall y(\bigwedge_i(x_i \neq y) \rightarrow \neg(connections(x_1, ..., x_m; y) \wedge mod_h(y)))). \end{array}$$

We will show the following property on the probability that an instance with n constants *does not* satisfy σ_m:

$$(\dagger) \ \mu_n(\neg\sigma_m) \leq n \cdot (n-1) \cdot ... \cdot (n-m) \cdot [1 - \frac{1}{2^{2m}}(\frac{1}{k} - \frac{1}{n})]^{n-m}$$

Since

$$\lim_{n\rightarrow\infty} n \cdot (n-1) \cdot ... \cdot (n-m) \cdot [1 - \frac{1}{2^{2m}}(\frac{1}{k} - \frac{1}{n})]^{n-m} = 0,$$

it follows that $\lim_{n\rightarrow\infty} \mu_n(\neg\sigma_m) = 0$, so $\neg\sigma_m$ is almost surely false, and σ_m is almost surely true.

Let N be the number of instances over G with elements in $\{1, ..., n\}$. Then the number of instances over $G_{\leq}$ with constants in $\{1, ..., n\}$ where $\leq$ is a total order on these elements is $N \cdot n!$.

To prove ($\dagger$), observe the following:

1. For some fixed distinct $a_1, ..., a_m, b$ in $\{1, \ldots, n\}$, the number of instances $\mathbf{I}$ over $G_\leq$ satisfying some fixed literal in $connections(a_1, \ldots, a_m; b)$ is $\frac{1}{2} \cdot N \cdot n!$.

2. For some fixed distinct $a_1, ..., a_m, b$ in $\{1, \ldots, n\}$, the number of instances $\mathbf{I}$ over $G_\leq$ satisfying $mod_h(b)$ is at least

$$N \cdot (\frac{n}{k} - 1) \cdot (n-1)! =$$

$$N \cdot n! \cdot (\frac{1}{k} - \frac{1}{n}).$$

3. For some fixed distinct $a_1, ..., a_m, b$ in $\{1, \ldots, n\}$, the number of $\mathbf{I}$ satisfying $connections(a_1, \ldots, a_m; b) \wedge mod_h(b)$ is at least

$$N \cdot n! \cdot \frac{1}{2^{2m}} \cdot (\frac{1}{k} - \frac{1}{n})$$

(since there are $2m$ literals in *connections*).

4. The number of $\mathbf{I}$ *not* satisfying $connections(a_1, \ldots, a_m; b)$ is therefore at most

$$N \cdot n! - N \cdot n! \cdot \frac{1}{2^{2m}} \cdot (\frac{1}{k} - \frac{1}{n}) =$$

$$N \cdot n! \cdot [1 - \frac{1}{2^{2m}} \cdot (\frac{1}{k} - \frac{1}{n})]$$

5. For some fixed $a_1, ..., a_m$ in $\{1, \ldots, n\}$, the number of $\mathbf{I}$ satisfying

$$\forall y (\bigwedge_i (a_i \neq y) \to \neg connections(a_1, ..., a_m; y))$$

is at most

$$N \cdot n! \cdot [1 - \frac{1}{2^{2m}} \cdot (\frac{1}{k} - \frac{1}{n})]^{n-m}$$

(since there are $(n-m)$ ways of picking b distinct from $a_1, ..., a_m$).

6. the number of $\mathbf{I}$ satisfying $\neg\sigma_m$ is thus at most

$$N \cdot n! \cdot n \cdot (n-1) \cdot \ldots \cdot (n-m) \cdot [1 - \frac{1}{2^{2m}} \cdot (\frac{1}{k} - \frac{1}{n})]^{n-m}$$

(from the choices of $a_1, ..., a_m$).

Hence (†) is proven. □

Note that the restriction that queries in $L + mod$ may not use the ordering $\leq$ is essential. Indeed, it is known that the 0-1 law no longer holds for any of the languages considered here if this restriction is removed. For example, since *fixpoint* expresses all the PTIME queries on ordered databases [15, 25], it expresses in particular many queries (such as parity) that do not have a 0-1 law.

5 Expected behavior

We are interested in several aspects of the behavior of the pure parallelization algorithm outlined above. Clearly, its success relies crucially on the assumption that the first, parallel phase, generally produces almost the entire result. Therefore, it is of interest to evaluate the chances that the first phase is complete. In all cases it must be tested whether the parallel phase yielded the complete result, and if not the sequential completion phase must be applied. We are interested in the *total* cost of the pure parallelization algorithm, which includes the test of completeness and the sequential completion phase. We compare this cost to the cost of the purely sequential algorithm.

5.1 Completeness of the parallelization phase

We now return to Datalog and the pure parallelization algorithm. Consider the property of completeness of the parallelization phase of our algorithm, with C_k^a sites, where $a = arity(P)$. For a given Datalog query $\varphi(\vec{x})$, this can be stated as a property of data by the following sentence, which we denote by $complete_k(\varphi)$:

$$\forall \vec{x}[\varphi(\vec{x}) \leftrightarrow \bigvee_{s_A} \varphi_A(\vec{x})].$$

Note that for any language L included in *while*, $complete_k(\varphi)$ is a sentence in the language *while + mod*. Since *while + mod* has a 0-1 law, it follows that $complete_k(\varphi)$ is almost surely true or almost surely false. It turns out that $complete_k(\varphi)$ is in fact almost surely true for all Datalog queries φ, as shown next.

Theorem 5.1 *For each Datalog query φ,* $\mathrm{complete}_k(\varphi)$ *is almost surely true.*

Proof: Let P be a Datalog program expressing the query φ. Again, we assume the only edb predicate is a binary relation G. We make no assumption on the arity of the answer. The monotonicity of Datalog implies that, for all instances $\mathbf{I}$ over $\{G, \leq\}$,

$$\forall \vec{x}[\varphi(\vec{x}) \leftarrow \bigvee_{s_A} \varphi_A(\vec{x})].$$

Thus, it is sufficient to show that

$$\forall \vec{x}[\varphi(\vec{x}) \rightarrow \bigvee_{s_A} \varphi_A(\vec{x})]$$

is almost surely true. Consider the set of expansions Γ of P (see Section 2). Recall that expansions are of the form $\exists \vec{u}\gamma$, where γ is a conjunction of atoms of edb predicates (here G) or equalities among free variables. Let α be the conjunction of all atoms in γ whose variables are all free (α is **true** if there are no such atoms), and β the conjunction of the remainder of the atoms in γ. Then $\exists\vec{u}\gamma$ is equivalent to $\alpha \wedge \exists\vec{u}\beta$. Clearly, there are finitely many nonequivalent α occurring in the expansions in Γ, say $\alpha_1, \ldots, \alpha_m$. For each α_r, pick *one* β_r such that $\alpha_r \wedge \exists\vec{u}\beta_r \in \Gamma$. Let

$$\gamma_{ir} = \forall\vec{x}\exists\vec{u}(mod_i(\vec{u}) \wedge \beta_r), \quad 1 \leq i \leq k, \ 1 \leq r \leq m$$

where $mod_i(\vec{u})$ is the conjunction of atoms $mod_i(z)$ for z occurring in $\vec{u}$, and $\vec{x}$ are the variables free in β_r. Note that each γ_{ir} is almost surely true; indeed, γ_{ir} is implied by a finite set of extension axioms over $\{G, mod_i \mid 0 \leq i < k\}$. So, the conjunction of the γ_{ir} is almost surely true. Let $\mathbf{I}$ be an instance over $\{G, \leq\}$ satisfying this conjunction. Clearly, it is sufficient to show that

$$\forall\vec{x}[\varphi(\vec{x}) \rightarrow \bigvee_{s_A} \varphi_A(\vec{x})]$$

holds on $\mathbf{I}$. Suppose $\vec{a} \in \varphi(\mathbf{I})$. Let $\vec{a} = \langle a_1, \ldots, a_h \rangle$. There must exist α_r such that $\mathbf{I}$ satisfies $\alpha_r(\vec{a})$. Also, there exist $i_1, \ldots, i_h$ such that $\mathbf{I}$ satisfies $mod_{i_j}(a_j)$, $1 \leq j \leq h$. Let A be such that $\{i_1, \ldots, i_h\} \subseteq A$, and let $l \in A$. Clearly, $\mathbf{I}$ satisfies

$$\alpha_r(\vec{a}) \wedge \wedge_{j=1}^{h} mod_{i_j}(a_j) \wedge \gamma_{lr}.$$

This implies that $\mathbf{I}$ satisfies $\varphi_A(\vec{a})$, so $\mathbf{I}$ satisfies $\bigvee_{s_A} \varphi_A(\vec{a})$. $\Box$

Remark: Let P be a Datalog program of arity a, whose input is a graph and whose output has arity o, where $o \leq a$. The pure parallelization algorithm as described earlier uses C_k^a sites. Theorem 5.1 shows that the parallel phase of that algorithm almost surely yields the complete answer to the query. Moreover, it is easily seen that in fact the algorithm almost surely yields the complete result for all of the idb predicates, not just the one designated as answer. Suppose now that the algorithm uses just C_k^o sites instead of C_k^a sites. The completeness result remains true, but only with respect to the answer idb predicate. For example, if P is a program whose input and output are both graphs, the parallel phase of the algorithm using C_k^2 sites almost surely produces the complete answer.

Thus, Theorem 5.1 shows that for all Datalog programs, it is almost surely true that the parallelization phase of the algorithm provides the complete answer to the query. However, this fact in itself is not sufficient to guarantee the possibility of developing an algorithm which is efficient *overall*, since it must also provide the test of completeness, and carry out the second completion phase if the partial answer is not complete. In the case of Datalog this is facilitated by the monotonicity of Datalog queries, which implies that:

- the parallelization phase always produces a subset of the answer, and
- the answer can be obtained from the output of the first phase by continuing the firing of rules up to a fixpoint.

The above does not generally hold for languages beyond Datalog, including the Datalog$^\neg$ languages. Thus, other, possibly less efficient means, must be used in the completion phase for those languages.

5.2 Cost analysis

We next evaluate probabilistically the overall cost of the pure parallelization algorithm compared to the sequential algorithm, for Datalog queries. Recall that the costs of the two algorithms for program P and input $\mathbf{I}$ were defined earlier, and denoted $\textit{sequential-cost}_P(\mathbf{I})$ and $\textit{parallel-cost}_{P,k}(\mathbf{I})$ for the sequential and parallel algorithms with C_k^a sites ($a = \textit{arity}(P)$), respectively. For a Datalog program P and input $\mathbf{I}$, let $st_P(\mathbf{I})$ be the number of stages in the sequential evaluation of P on $\mathbf{I}$. We assume the evaluation stops when two consecutive stages yield the same result; so, in all cases $st_P(\mathbf{I}) \geq 2$.

Theorem 5.2 *Let P be a Datalog program of arity a and v be the minimum number of variables in a rule of P. Then it is almost surely true that*

$$\text{parallel-cost}_{P,k}(\mathbf{I}) \leq [(a/k)^v + 1/st_P(\mathbf{I})] \times \text{sequential-cost}_P(\mathbf{I}).$$

Proof: The core of the proof is to show that the following are almost surely true simultaneously:

(1) the parallel phase produces the complete result (Theorem 5.1); and,

(2) $st_{P,s}(\mathbf{I}) = st_P(\mathbf{I})$, where $st_{P,s}(\mathbf{I})$ is the number of stages in the evaluation of P at site s on input $\mathbf{I}$.

Indeed, let P be a Datalog program as in the statement, and suppose (1) $\wedge$ (2) is almost surely true. Let $\mathbf{I}$ be an input satisfying (1) $\wedge$ (2). We will show that

$$\textit{parallel-cost}_{P,k}(\mathbf{I}) \leq [(a/k)^v + 1/st_P(\mathbf{I})] \times \textit{sequential-cost}_P(\mathbf{I}).$$

We first need the following notation. Suppose P has r rules, and rule i has v_i variables, $1 \leq i \leq r$. Suppose $\mathbf{I}$ has n elements. Note that each site s_A has $(a \cdot n)/k$ elements. For each site s, let $st_{P,s}(\mathbf{I})$ be the number of stages in the evaluation of P at site s on input $\mathbf{I}$. The cost of the pure parallelization algorithm applied to P and $\mathbf{I}$ is then computed as follows. The cost at each site is

$$st_{P,s}(\mathbf{I}) \sum_{i=1}^{r} (\frac{a \cdot n}{k})^{v_i}.$$

The cost of the parallel phase is the maximum of these costs over all sites. Consider the cost of the completion phase. By (1), the parallel phase of the algorithm produces the complete answer. Thus, the completion phase terminates after one iteration, needed to verify that no additional facts can be inferred. The cost of that iteration is

$$\sum_{i=1}^{r} n^{v_i}.$$

Notice that, by (2), $st_{P,s}(\mathbf{I}) = st_P(\mathbf{I})$, so

$$\textit{parallel-cost}_{P,k}(\mathbf{I}) = st_P(\mathbf{I}) \sum_{i=1}^{r} (\frac{a \cdot n}{k})^{v_i} \quad + \quad \sum_{i=1}^{r} n^{v_i}.$$

The cost $\textit{sequential-cost}_P(\mathbf{I})$ of the sequential algorithm is

$$st_P(\mathbf{I}) \sum_{i=1}^{r} n^{v_i}.$$

Thus,

$$\begin{aligned}
&\frac{\textit{parallel-cost}_{P,k}(\mathbf{I})}{\textit{sequential-cost}_P(\mathbf{I})} \\
\leq \quad & \frac{st_P(\mathbf{I}) \sum_{i=1}^{r} (\frac{a \cdot n}{k})^{v_i} \ + \ \sum_{i=1}^{r} n^{v_i}}{st_P(\mathbf{I}) \sum_{i=1}^{r} n^{v_i}} \\
= \quad & \frac{\sum_{i=1}^{r} (\frac{a}{k})^{v_i} n^{v_i}}{\sum_{i=1}^{r} n^{v_i}} + \frac{1}{st_P(\mathbf{I})} \\
\leq \quad & \frac{(\frac{a}{k})^{v} \sum_{i=1}^{r} n^{v_i}}{\sum_{i=1}^{r} n^{v_i}} + \frac{1}{st_P(\mathbf{I})} \\
= \quad & (\tfrac{a}{k})^{v} + \frac{1}{st_P(\mathbf{I})},
\end{aligned}$$

which proves the statement of the theorem.

It remains to prove that (1) $\wedge$ (2) is true. We know from Theorem 5.1 that (1) is almost surely true. Since sentences which are almost surely true are closed under conjunction, it is sufficient to show that (2) is almost surely true. This is shown along the same lines as the proof of Theorem 5.1. Consider a site s_A, and an idb predicate R. Consider the set of expansions Γ of P relative to R (i.e., taking R to be the answer relation). For each $\exists \vec{u} \gamma \in \Gamma$, let α be the conjunction of all atoms in γ whose variables are all free (α is **true** if there are no such atoms), and β the conjunction of the remainder of the atoms in γ. Then $\exists \vec{u} \gamma$ is equivalent to $\alpha \wedge \exists \vec{u} \beta$. Clearly, there are finitely many nonequivalent α occurring in the expansions in Γ, say $\alpha_1, \ldots, \alpha_m$. For each α_i, let $st_P(\alpha_i)$ be the smallest j such that $\alpha_i \wedge \exists \vec{u} \beta$ is in Γ_j for some β, and pick one β_i such that

$$\alpha_i \wedge \exists \vec{u} \beta_i \in \Gamma_{st_P(\alpha_i)}.$$

Next, let

$$\gamma_i = \forall \vec{x} \ \exists \vec{u} \beta_i$$

where $\vec{x}$ are the free variables of $\exists \vec{u} \beta_i$, and

$$\gamma_{ij} = \forall \vec{x} \ \exists \vec{u} (mod_j(\vec{u}) \wedge \beta_i), \quad 1 \leq i \leq k, \ 1 \leq r \leq m,$$

where $mod_j(\vec{u})$ is the conjunction of atoms $mod_j(z)$ for z occurring in $\vec{u}$. Note that both γ_i and γ_{ij} are almost surely true. Thus, their finite conjunction (over all i and j) is almost surely true. Let $\mathbf{I}$ be an instance satisfying that conjunction. We will prove that (2) holds for $\mathbf{I}$. Let $R(t)$ be a ground literal (using elements from $\mathbf{I}$), inferred at site s_A. Since $R(t)$ is inferred, t must satisfy at least one α of $\alpha_1, \ldots, \alpha_m$. Let j be the smallest i such that t satisfies some α and $st_P(\alpha) = i$. Let α_h be such that t satisfies α_h and $st_P(\alpha_h) = j$. We claim that $R(t)$ is inferred (for the first time) at stage j in both the sequential evaluation of P and in the evaluation of P at site s_A on input $\mathbf{I}$. Indeed, $R(t)$ is inferred at stage j at site s_A because $\mathbf{I}$ satisfies $\alpha_h(t) \wedge \gamma_{hj}$ for $j \in A$; it is not inferred earlier than stage j because $\mathbf{I}$ does not satisfy $\alpha(t)$ for any $\alpha \wedge \exists \vec{u} \beta$ belonging to Γ_l for $l < j$. Similarly, $R(t)$ is inferred at stage j in the sequential evaluation of P because $\mathbf{I}$ satisfies $\alpha_h(t) \wedge \gamma_h$, and is not inferred earlier than stage j because, as before, $\mathbf{I}$ does not satisfy $\alpha(t)$ for any $\alpha \wedge \exists \vec{u} \beta$ belonging to Γ_l for $l < j$. It follows that it takes equally many stages to reach a fixpoint in the sequential evaluation of P and in the evaluation of P at a given site. So, $st_{P,s}(\mathbf{I}) = st_P(\mathbf{I})$ and (2) holds. $\Box$

Note that the above cost ratio depends critically on the value of $st_P(\mathbf{I})$. For its minimum value (2), the ratio is larger than 0.5, while for high values it approaches the constant $(a/k)^v$. Fortunately, there is a large class of programs, that includes most commonly arising ones, for which the expected ratio is precisely $(a/k)^v$. We next present this class of programs.

Recall the notion of *expansion* of a Datalog program, and the notation relating to it (Section 2). We call a Datalog program P *constraint-free* if it has some expansion $\exists \vec{u} \gamma$ such that each atom in γ is a literal containing at least one variable in $\vec{u}$ (in particular, there are no equality atoms). Recall that such programs were considered at the end of Section 2. Most Datalog programs considered in this paper, including the usual transitive closure and same-generation programs, are constraint-free. The program at the end of Section 2 is not (all answers $T(x, y)$ are subject to the "constraints" $G(x, x), G(y, y)$).

It turns out that it can be effectively tested if a given Datalog program P is constraint-free, as outlined next[3]. Let $\mathcal{V}$ be the set of substitutions of variables in P into $\{x, y, @\}$ such that $\nu(x) = x$ and $\nu(y) = y$. A valuation ν is extended to atoms by $\nu(R(t)) = R(\nu(t))$. Let $\mathbf{G}_P$ be the context-free grammar obtained from P as follows. There are no terminals, and its nonterminals are all literals $\nu(R(t))$ where $R(t)$ is a literal occurring in P and $\nu \in \mathcal{V}$. (Intuitively, @ tracks the existentially quantified variables in the expansions, and x, y track the free variables.) The start symbol is $T(x, y)$ where T is the (binary) answer predicate. For each rule

$$A_0(t_0) \leftarrow A_1(t_1), \ldots, A_n(t_n)$$

in P, $\mathbf{G}_P$ contains all productions

$$\nu[A_0(t_0) \rightarrow A_1(t_1), \ldots, A_n(t_n)]$$

where $\nu \in \mathcal{V}$. Finally, let $\mathbf{G}_{P,\epsilon}$ be $\mathbf{G}_P$ augmented with all productions $R(t) \rightarrow \epsilon$ where R is an edb predicate and t a tuple over $\{x, y, @\}$ containing at least one occurrence of @. We will show that P is constraint-free iff $\epsilon \in L(\mathbf{G}_{P,\epsilon})$. We first need the following definition and lemma.

[3]For simplicity, we describe the construction and prove the results for the case when both input and answer are binary.

Definition 5.3 *Let P be a program and A a goal. An SLD*-derivation of a goal $A_1, \ldots, A_m$ from A is defined as an SLD-derivation with the following restriction on the mgu θ used at each step in the derivation: (i) θ is the identity on the variables occurring in A, and (ii) if V is the set of variables in the current goal, $\theta(V) \subseteq V$.*

Note that (i) says that the variables in A are essentially treated as constants, while (ii) is a syntactic condition that can be assumed without any loss of generality.

Lemma 5.1 (i) If there is an SLD*-derivation of $A_1, \ldots, A_m$ from $T(x, y)$ with respect to P then $\mathbf{G}_P$ derives $\nu(A_1 \ldots A_m)$ for every $\nu \in \mathcal{V}$.
(ii) If $\mathbf{G}_P$ derives $A_1 \ldots A_m$ then there is an SLD*-derivation of a goal $A'_1, \ldots, A'_m$ from $T(x, y)$ with respect to P such that for some $\nu \in \mathcal{V}$ $A_i = \nu(A'_i)$, $1 \leq i \leq m$.

Proof: We prove (i) by induction on the length of SLD*-derivations. For the basis, suppose that there is an SLD*-derivation of length zero of $A_1 \ldots A_m$ from $T(x, y)$, i.e. $A_1 \ldots A_m = T(x, y)$. Clearly, $\nu(T(x, y)) = T(x, y)$ is derived in $\mathbf{G}_P$ for every $\nu \in \mathcal{V}$ by a derivation of length zero.

For the induction step, suppose (i) holds for SLD*-derivations of length at most i, $i \geq 0$, and consider a goal obtained by an SLD*-derivation of length $i + 1$. Such a goal has the form

$$\theta(A_1), \ldots, \theta(A_{j-1})\theta(B_1), \ldots, \theta(B_h), \theta(A_{j+1}), \ldots, \theta(A_m),$$

where

- $A_1, \ldots, A_j, \ldots, A_m$ is derived from $T(x, y)$ by an SLD*-derivation of length i,
- $A'_j \leftarrow B_1, \ldots, B_h$ is a rule in P, and
- $\theta = mgu(A_j, A'_j)$.

By the induction hypothesis, $\mathbf{G}_P$ derives $\nu(A_1) \ldots \nu(A_j) \ldots \nu(A_m)$ for every $\nu \in \mathcal{V}$. We have to show that $\mathbf{G}_P$ also derives

$$\nu(\theta(A_1)) \ldots \nu(\theta(A_{j-1}))\nu(\theta(B_1)) \ldots \nu(\theta(B_h))\nu(\theta(A_{j+1})) \ldots \nu(\theta(A_m))$$

for every $\nu \in \mathcal{V}$. Since, as we assumed, $\theta(\{x, y\}) = \{x, y\}$, it is easily seen that $\nu \circ \theta \in \mathcal{V}$. It follows that $\mathbf{G}_P$ derives

$$\nu(\theta(A_1)) \ldots \nu(\theta(A_j)) \ldots \nu(\theta(A_m)).$$

Also,

$$\nu(\theta(A'_j)) \rightarrow \nu(\theta(B_1)), \ldots, \nu(\theta(B_h))$$

is a production in $\mathbf{G}_P$. Since $\theta(A'_j) = \theta(A_j)$,

$$\nu(\theta(A_j)) \rightarrow \nu(\theta(B_1)), \ldots, \nu(\theta(B_h))$$

is a production, and $\mathbf{G}_P$ derives

$$\nu(\theta(A_1))\ldots\nu(\theta(A_{j-1}))\nu(\theta(B_1))\ldots\nu(\theta(B_h))\nu(\theta(A_{j+1}))\ldots\nu(\theta(A_m)).$$

This completes the induction.

Part (ii) is also proven by induction, on the length of derivations in $\mathbf{G}_P$. For the basis, suppose $\mathbf{G}_P$ derives $A_1\ldots A_m$ by a derivation of length zero. Then $A_1\ldots A_m = T(x,y)$. Since $T(x,y)$ is derived from $T(x,y)$ by an SLD*-derivation of length zero, the statement holds.

For the induction step, suppose the statement holds for words derived in $\mathbf{G}_P$ by derivations of length i, $i \geq 0$, and consider a word derived in $\mathbf{G}_P$ by a derivation of length $i+1$. Then the word is of the form

$$A_1\ldots A_{j-1}B_1\ldots B_h A_{j+1}\ldots A_m,$$

where

- $A_1\ldots A_j\ldots A_m$ is derived in $\mathbf{G}_P$ by a derivation of length i, and
- $A_j \to B_1\ldots B_h$ is a production in $\mathbf{G}_P$.

By the induction hypothesis, there exists an SLD*-derivation of $A'_1,\ldots,A'_j,\ldots A'_m$ from $T(x,y)$ such that $\nu'(A'_l) = A_l$, $1 \leq l \leq m$, for some $\nu' \in \mathcal{V}$. Since $A_j \to B_1\ldots B_h$ is a production in $\mathbf{G}_P$, there exists a rule $A''_j \leftarrow B'_1,\ldots,B'_h$ such that $\nu''(A''_j) = A_j, \nu''(B'_1) = B_1,\ldots,\nu''(B'_h) = B_h$ for some $\nu'' \in \mathcal{V}$. Since $\nu''(A''_j) = A_j$ and $\nu'(A'_j) = A_j$, A'_j and A''_j are literals over the same predicate. We will show that A'_j and A''_j have an mgu θ which is the identity on $\{x,y\}$. We first need the following notation: Suppose $A'_j = R(t')$, $A''_j = R(t'')$, and $A_j = R(t)$. Consider the following equivalence relations on the coordinates of R:

- $a \approx b$ iff $t(a) = t(b)$;
- $a \approx' b$ iff $t'(a) = t'(b)$; and
- $a \approx'' b$ iff $t''(a) = t''(b)$.

Since $\nu'(R(t')) = R(t)$, $\approx' \subseteq \approx$. Similarly, $\approx'' \subseteq \approx$, so $(\approx' \cup \approx'')^* \subseteq \approx$, where * denotes the reflexive-transitive closure of a relation.

Let $\equiv$ denote the equivalence relation $(\approx' \cup \approx'')^*$. From the definition of mgu, it follows that A'_j and A''_j have an mgu which is the identity on $\{x,y\}$ iff there are no coordinates i,j of R such that $i \equiv j$ and $u(i) = x$, $v(j) = y$ for $u,v \in \{t',t''\}$. Suppose towards a contradiction that there exists coordinates i,j such that $i \equiv j$ and $t'(i) = x$, $t''(j) = y$ (the other three cases are similar). Since $\nu'(t') = t$ for $\nu' \in \mathcal{V}$, $t(i) = x$. Similarly, $t(j) = y$. But $\equiv\, \subseteq\, \approx$, so $i \approx j$ and $t(i) = t(j)$. This is a contradiction.

Let θ be an mgu of A'_j and A''_j which is the identity on $\{x, y\}$, whose existence we have just proven. As one more piece of notation, let $\approx_\theta$ be the equivalence relation on coordinates of R defined by $a \approx_\theta b$ iff $\theta(t')(a) = \theta(t')(b)$. By the definition of mgu, $\approx_\theta$ coincides with $\equiv$.

Thus, there is an SLD*-derivation for

$$\theta(A'_1), \ldots, \theta(A'_{j-1}), \theta(B'_1), \ldots, \theta(B'_h), \theta(A'_{j+1}), \ldots, \theta(A'_m).$$

Now we define $\mu \in \mathcal{V}$ as follows:

(a) for each variable z occuring in the set of variables of

$$\theta(A'_1), \ldots, \theta(A'_{j-1}), \theta(A'_{j+1}), \ldots \theta(A'_m),$$

$\mu(z) = \nu'(v)$ where $z = \theta(v)$ (we will have to show that this is well defined).

(b) For the other variables z (occurring in some $\theta(B'_l)$ and not in the above set) $\mu(z) = \nu''(z)$.

We claim that:

1. μ is well defined; and,
2. $\mu(\theta(A'_l)) = A_l, 1 \leq l \leq m, l \neq j$, and $\mu(\theta(B'_l)) = B_l$, $1 \leq l \leq h$.

This would conclude the induction and the proof of the lemma. To see (1), note that it is enough to show that μ is well defined for the variables covered under (a), that is: if v, w are variables occurring in $A'_1, \ldots, A'_j, \ldots A'_m$ and $\theta(v) = \theta(w)$ then $\nu'(v) = \nu'(w)$. Clearly, if $\theta(v) = \theta(w)$ then v and w must occur in A'_j. Recall that $A'_j = R(t')$, $A''_j = R(t'')$, and $A_j = R(t)$. There exist coordinates a, b of R such that $t'(a) = v$, $t'(b) = w$. Since $\theta(v) = \theta(w)$, $\theta(t')(a) = \theta(t')(b)$ so $a \approx_\theta b$. By the earlier observations, $a \approx b$, so $t(a) = t(b)$. Since $\nu'(t') = t$, it follows that $\nu'(v) = \nu'(w)$. Thus, μ is well defined.

For (2), note that $\mu(\theta(A'_l)) = A_l$, $1 \leq l \leq m$ follows immediately from the definition of μ and the fact that $\nu'(A'_l) = A_l, 1 \leq l \leq m$. The equalities $\mu(\theta(B'_l)) = B_l$, $1 \leq l \leq h$, follow from the fact (shown along the same lines as (1) above) that if $\theta(v) = z$ for some variable v in $B_1, \ldots, B_h$ which is moved by θ, then $\nu''(v) = \nu'(z)$. $\square$

We can now show:

Theorem 5.4 *A Datalog program P is constraint-free iff $\epsilon \in L(\mathbf{G}_{P,\epsilon})$.*

Proof: Suppose P is a constraint-free Datalog program. As before, we assume for simplicity that P's input and answer relations have arity 2. Then there exists an expansion

$\exists\vec{u}\gamma$ of P with 2 free variables, where each atom in γ is a literal containing a quantified variable. Let $A_1 \ldots A_m$ be the sequence of conjuncts in γ. Then there exists an SLD*-derivation of $A_1, \ldots, A_m$ from $T(x, y)$ with respect to P. Let ν be the substitution defined by $\nu(x) = x, \nu(y) = y$, and $\nu(z) = @$ for each $z \notin \{x, y\}$. By Lemma 5.1, $\mathbf{G}_P$ derives $\nu(A_1) \ldots \nu(A_m)$. Since each $\nu(A_i)$ contains @, $\mathbf{G}_{P,\epsilon}$ derives ϵ.

Conversely, suppose $\mathbf{G}_{P,\epsilon}$ derives ϵ. Without loss of generality, we can assume that the ϵ productions are applied last in the derivation. Thus, $\mathbf{G}_P$ generates $A_1 \ldots A_m$ such that each A_i contains @. By Lemma 5.1, there is an SLD*-derivation of a goal $A'_1 \ldots A'_m$ from $T(x, y)$ and a valuation $\nu \in \mathcal{V}$ such that $\nu(A'_i) = A_i$, $1 \leq i \leq m$. Since each A_i contains @, each A'_i must contain a variable other than x, y. Thus, P has the expansion $\exists\vec{u}(A'_1 \ldots A'_m)$, where $\vec{u}$ consists of the variables different from x, y, and P is constraint-free. □

Note that the property of constraint-freeness of a Datalog program P can be tested in time polynomial in the size of $\mathbf{G}_{P,\epsilon}$. The size of $\mathbf{G}_{P,\epsilon}$ is larger than the size of P by a factor of 3^v where v is the maximum number of variables in a rule of P.

One can now show the following:

Theorem 5.5 *Let P be a constraint-free Datalog program of arity a and v the minimum number of variables in a rule of P. Then it is almost surely true that*

$$\text{parallel-cost}_{P,k}(\mathbf{I}) \leq (a/k)^v \times \text{sequential-cost}_P(\mathbf{I}).$$

Proof: Let P be a constraint-free Datalog program with binary answer. Since P is constraint-free, there exists an expansion $\exists\vec{u}\gamma$ where all atoms in γ are literals containing at least one variable in $\vec{u}$, and x, y are free in γ. Then $\forall x \forall y \exists\vec{u}\gamma$ is almost surely true. This means that P almost surely produces as answer the *complete graph.* Since in all cases the parallelization phase produces the complete answer, it is almost surely true that the parallelization phase produces the complete answer which is in turn the complete graph. Thus, it is almost surely true that there are no potentially productive instantiations at the end of the parallel phase, and the cost of the completion phase is zero. This yields the result from the same analysis as in the proof of Theorem 5.2. □

The expected speedup obtained for constraint-free programs is proportional to some power of the number of sites. Thus, for most programs the expected performance of the parallel algorithm is excellent compared to the sequential one. For example, suppose P is the standard transitive closure program ($v = 2$) and $k = 10$. Then the number of sites is 45 and it can be expected that the parallel cost is 1/25 of the sequential cost.

Since the above is an *asymptotic* result, the input has to be larger than a certain size before the behavior comes close to that described in the theorem. It is natural to ask whether the behavior actually holds for sizes of practical interest. The answer is positive: the experimental results of Section 6 show that the behavior actually matches the asymptotic behavior even with relatively few nodes (e.g., 300). Furthermore, the experimental results are often substantially better than that forecast by the analysis, which suggests that the analysis is quite conservative. In the case of transitive closure, if we ignore the first rule

$$T(x, y) \leftarrow G(x, y)$$

which is only used for initialization of T, the ratio between the parallel and sequential costs is $1/125$ for the same parameters as in the previous paragraph but with $v = 3$. This in fact matches closely the experimental results.

Remark: Theorem 5.5 shows that in many cases, the ratio between the costs of the parallel and sequential algorithms for Datalog evaluation is almost surely much smaller than $1/s$, where s is the number of sites used. This may appear strange to eyes used to classical worst-case analysis of parallel and sequential algorithms. Indeed, it suggests that the sequential algorithm can be improved simply by simulating sequentially the parallel algorithm. However, this is not contradictory if we consider the following: (i) we never assumed that the sequential algorithm was optimal (indeed, it is not), and (ii) the sequential algorithm obtained by simulation of the parallel algorithm is *almost surely* better than the given one, but there is no improvement in the worst case (to the contrary, the worst case gets worse). The idea that the sequential simulation of pure parallelization can be used to improve the *asymptotic* performance of sequential algorithms is thus a valid one worth exploring.

The probabilistic analysis above assumes that all database instances are equally likely. In the example of a single binary relation, this is equivalent to assuming probability $1/2$ for each edge. Consequently, the relation will almost always contain about half of all pairs. Relations are usually more sparse than this. Fortunately, the results hold for probability measures which allow to model at least some sparse graphs. For example, they continue to hold with edge probability $1/c$ where c is any constant, and even $n^{-1/\log\log n}$ where n is the number of nodes in the graph. Experimental results for some sparse graphs are also described in [19].

6 Experimental Results

In order to study the practical behavior of pure parallelization, we have implemented the parallel algorithm and a sequential algorithm based on semi-naive evaluation. We compared the performance of the parallel algorithm to that of the sequential evaluation.

The goal of our experiments is to show the behavior of pure parallelization for several queries, and a variety of database inputs and also to understand the interplay of the main parameters: input size, sparsity of the input, and number of sites. In particular, we wish to compare the real performance of the algorithm to the limit behavior given by the theoretical results. For example, how many nodes must the input have before the behavior begins to approach the promised expected one? And, how is this affected by sparsity of the input and the number of sites?

For the purpose of comparison, the cost measure in both parallel and sequential cases is that described in the previous section: we count all rule instantiations generated by the algorithms. The experiments involve several parameters regarding queries, inputs, and processing sites. We begin with the queries.

Three different queries are considered, mostly the same as in [4], except that here, all queries are unbounded. The first is the classical linear transitive closure query:

$$\begin{aligned} Q_1: \quad Ltc(x,y) &\leftarrow \quad Ltc(x,z), G(z,y) \\ Ltc(x,y) &\leftarrow \quad G(x,y). \end{aligned}$$

The second query is the nonlinear (quadratic) version of transitive closure, which uses double recursion:

$$\begin{aligned} Q_2: \quad Qtc(x,y) &\leftarrow \quad Qtc(x,z), Qtc(z,y) \\ Qtc(x,y) &\leftarrow \quad G(x,y). \end{aligned}$$

Finally, the third and last query chosen represents the "same-generation" query:

$$\begin{aligned} Q_3: \quad Sg(x,y) &\leftarrow \quad G(w,x), Sg(w,z), G(z,y) \\ Sg(x,x) &\leftarrow \quad NodesofG(x). \end{aligned}$$

where the binary relational predicate G corresponds to the input for all Q_1, Q_2 and Q_3 queries and $NodesofG$, used on the exit rule of Q_3, is a relational table of nodes (constants) present in G.

The database inputs used in the experiments are randomly generated binary relations, with various degrees of structure:

- no restriction,
- acyclic, and
- complete binary trees.

In each case, we considered inputs ranging from highly connected to sparse ones. Specifically, we generated graphs with 200 and 300 nodes and edge probabilities ranging from 1/5 to 1/100 for each edge. These yield input binary relations G ranging from about 300 through 18,000 tuples, with up to 90,000 tuples in the answer.

With respect to the parallel architecture, we simulated a shared-nothing scheme, with a variable number of processing nodes. In our case, this is fixed by the chosen partitioning strategy that distributes tuples among the sites, defined by the value of k. Recall that the number of sites is C^2_k. We have chosen $k = 4$, 7, 10 and 14, which yields 6, 21, 45 and 91 parallel sites, respectively.

In the following tables, the input column shows the input graph considered, with its number of nodes and edge probability rate. In each case, the tables provide the following information:

- percentage of the complete answer produced by the parallel phase; and,
- ratio of the *total* cost of the parallel algorithm, and the cost of the sequential algorithm.

For example, consider Table 1, the table for the linear transitive closure query on random graphs. For input 200-1/20 (meaning 200 nodes with edge probability 1/20) and k =7 (i.e., 21 sites), the parallel phase generates 87% of the total answer; and, the total parallel cost is 0.0929 of the sequential cost. (The percentage of correct answer produced by the parallel phase for the linear and nonlinear transitive closure programs are, of course, the same, as they compute the same query; however, the ratios of parallel and sequential cost are generally different, since these depend also on the programs.)

Table 1: Random graphs

Linear Transitive Closure Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
200-1/100	12%	4%	3%	2%	0.5724	0.5306	0.5586	0.5380
200-1/20	98%	87%	58%	33%	0.2528	0.0929	0.1393	0.1980
200-1/10	100%	99%	96%	88%	0.1666	0.0590	0.0320	0.0510
200-1/5	100%	100%	100%	99%	0.1250	0.0325	0.0106	0.0071
300-1/100	38%	6%	3%	2%	0.4578	0.3777	0.3827	0.3987
300-1/20	99%	97%	91%	70%	0.1564	0.0585	0.0448	0.0839
300-1/10	100%	100%	99%	97%	0.2500	0.0588	0.0261	0.0238
300-1/5	100%	100%	100%	99%	0.1250	0.0235	0.0106	0.0042
Nonlinear Transitive Closure Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
200-1/100	12%	4%	3%	2%	0.5987	0.5679	0.5829	0.5880
200-1/20	98%	87%	58%	33%	0.1704	0.0829	0.1529	0.2334
200-1/10	100%	99%	96%	88%	0.1250	0.0346	0.0213	0.0431
200-1/5	100%	100%	100%	99%	0.1250	0.0243	0.0080	0.0051
300-1/100	38%	6%	3%	2%	0.3947	0.4614	0.4924	0.5305
300-1/20	99%	97%	91%	70%	0.1669	0.0466	0.0411	0.1024
300-1/10	100%	100%	99%	97%	0.1250	0.0314	0.0121	0.0122
300-1/5	100%	100%	100%	99%	0.1250	0.0235	0.0080	0.0032
Same Generation Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
200-1/100	10%	2%	1%	1%	0.5632	0.5287	0.5392	0.5447
200-1/20	98%	87%	58%	30%	0.1079	0.0578	0.1454	0.2332
200-1/10	100%	99%	96%	88%	0.0625	0.0139	0.0134	0.0389
200-1/5	100%	100%	100%	99%	0.0833	0.0070	0.0016	0.0016
300-1/100	37%	4%	2%	1%	0.2779	0.3435	0.3686	0.3881
300-1/20	99%	97%	91%	70%	0.0836	0.0209	0.0316	0.0997
300-1/10	100%	100%	99%	97%	0.0833	0.0090	0.0035	0.0088
300-1/5	100%	100%	100%	99%	0.0833	0.0067	0.0016	0.0005

Recall that the total cost of the parallel algorithm includes not only the parallel phase, but also the sequential completion phase.

In general, even if the fraction of the correct answer produced in the parallel phase is very good, this does not necessarily imply that the ratio of the cost of the parallel algorithm and the cost of the sequential algorithm is as good. This is due to the completion phase. In some cases however, one can stop before the completion phase; one such case is when the query asks whether $\langle a, b\rangle$ is in the transitive closure, and $\langle a, b\rangle$ has already been obtained in the partial answer from the parallel phase.

Let us interpret the results obtained. We generally observe the common-sense interplay between the parameters:

- the more nodes, the better the behavior (i.e. the closer to the asymptotic behavior stated by Theorem 5.5);
- the sparser the graph the worst the behavior;
- large numbers of sites result in data fragmentation; so, good behavior emerges starting with larger numbers of nodes and denser graphs than if fewer sites are used. On the other hand, the improvement is *eventually* more dramatic (since, by Theorem 5.5, the speedup obtained is asymptotically proportional to some power of the number of sites).

Overall, the results show that the behavior of pure parallelization for randomly generated graphs is excellent. In particular, note that the theoretical expected performance is sometimes reached, even with the relatively few nodes considered here. For example, see non-linear transitive closure with 300 nodes and edge probability 1/5; the parallel cost with 10 sites is 0.008 of the sequential one, i.e. exactly the 1/125 ratio given by the theoretical result (Theorem 5.5).

As more assumptions are imposed on input structure, the behavior of pure parallelization degrades. For acyclic graphs it remains nonetheless very good, but far less so than for arbitrary graphs, as shown in Table 2. For instance, the best behavior obtained is a speed-up by a factor of about 2. And, the parallel algorithm behaves worse than the sequential one in a few cases (sparse graphs with edge probability 1/100).

Things get worse for highly structured inputs. Table 3 shows the behavior of the algorithm on complete binary trees. Here, the only parameter of the input is the height of the tree, ranging from height 8 (511 nodes) to 12 (8191 nodes).

As seen in Table 3 pure parallelization behaves worse than the sequential algorithm in most of the cases considered. Indeed, the parallel phase produces only a small fraction of the result, and the completion phase in these cases is as costly as the sequential evaluation.

These results only confirm our expectations at the outset: pure parallelization represents just a skeleton, which works very well in the limit situation of arbitrary inputs. But, it must be tailored to specific applications when information on the structure of data is available.

Table 2: Random acyclic graphs

Linear Transitive Closure Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
200-1/100	82%	76%	67%	69%	0.8462	1.0106	0.9958	0.9934
200-1/20	56%	31%	20%	17%	0.9486	0.8403	0.8358	0.8341
200-1/10	76%	48%	42%	29%	0.6920	0.6050	0.6748	0.6795
200-1/5	92%	80%	71%	58%	0.7030	0.5984	0.5854	0.5926
300-1/100	65%	49%	43%	42%	1.0954	1.0029	0.9926	0.9888
300-1/20	64%	37%	25%	20%	0.7804	0.7557	0.7519	0.7516
300-1/10	85%	66%	50%	38%	0.6936	0.6501	0.6345	0.5542
300-1/5	94%	86%	78%	70%	0.6840	0.4836	0.5606	0.5628
Nonlinear Transitive Closure Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
200-1/100	82%	76%	67%	69%	1.1282	1.0175	0.9977	0.9947
200-1/20	56%	31%	20%	17%	0.9373	0.8587	0.8564	0.8566
200-1/10	76%	48%	42%	29%	0.7804	0.7125	0.7042	0.7178
200-1/5	92%	80%	71%	58%	0.6788	0.5936	0.5894	0.6003
300-1/100	65%	49%	43%	42%	1.1176	1.0076	0.9941	0.9901
300-1/20	64%	37%	25%	20%	0.7173	0.7752	0.7763	0.7791
300-1/10	85%	66%	50%	38%	0.6080	0.5217	0.5256	0.5330
300-1/5	94%	86%	78%	70%	0.6649	0.5733	0.5691	0.5745
Same Generation Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
200-1/100	85%	78%	68%	70%	0.7086	0.9974	0.6630	0.9923
200-1/20	46%	22%	13%	10%	0.9120	0.8700	0.8719	0.8739
200-1/10	69%	36%	29%	17%	0.7343	0.7103	0.7110	0.7205
200-1/5	89%	71%	59%	42%	0.6502	0.6015	0.6102	0.6288
300-1/100	69%	52%	46%	44%	1.0420	0.9964	0.9925	0.9916
300-1/20	55%	27%	16%	11%	0.7155	0.7885	0.7923	0.7954
300-1/10	80%	56%	38%	26%	0.6850	0.5461	0.6560	0.5630
300-1/5	92%	80%	68%	57%	0.6306	0.5753	0.5830	0.5960

Table 3: Complete binary trees

Linear Transitive Closure Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
h8-511	33%	22%	18%	18%	1.1085	1.0127	0.9962	0.9941
h9-1023	29%	19%	16%	16%	1.1350	1.0159	0.9999	0.9977
h10-2047	26%	17%	14%	13%	1.0085	1.0150	1.0012	0.9993
h11-4095	22%	15%	13%	12%	1.0094	1.0153	1.0028	1.0001
h12-8191	21%	14%	12%	11%	1.1046	1.0155	1.0034	1.0005
Nonlinear Transitive Closure Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
h8-511	33%	21%	18%	18%	1.1344	1.0265	1.0026	0.9966
h9-1023	29%	19%	16%	16%	0.9364	1.0175	1.0021	0.9983
h10-2047	27%	17%	14%	13%	1.1264	1.0204	1.0037	1.0006
h11-4095	22%	15%	13%	12%	1.1343	1.0185	1.0062	1.0010
h12-8191	21%	14%	12%	11%	1.1264	1.0197	1.0069	1.0019
Same Generation Query								
Input	Solutions Par/Complete (%)				Costs Par/Seq			
	k = 4	k = 7	k = 10	k = 14	k = 4	k = 7	k = 10	k = 14
h8-511	44%	32%	29%	28%	1.0539	1.0010	0.9954	0.9947
h9-1023	40%	30%	27%	26%	1.0697	1.0033	0.9982	0.9974
h10-2047	36%	27%	23%	22%	1.0527	1.0041	0.9993	0.9986
h11-4095	32%	25%	23%	21%	1.0573	1.0046	1.0001	0.9993
h12-8191	29%	22%	20%	19%	1.0539	1.0045	1.0004	0.9997

There are many ways to tune pure parallelization to specific situations, such as complete binary trees. Developing such specific algorithms is beyond the scope of this paper. Clearly, there is a wide variety of possible approaches to further tailoring the parallelization algorithm to these special cases.

The implementation supporting the above experiments was done over a network of SUN-SPARC workstations using RDL//C [8], a parallel rule-based language on top of a relational DBMS (*SABRINA v7.2*), an extension of RDL/C [16], which supports both declarative production rules and procedural programming based on C code. Relations are transferred from each of the parallel sites to a central site (that can be one of the parallel sites) by dumping relations onto files using a common file system managed by the network. This can be easily done since there is no interprocessor communications during the computations.

7 Conclusion

We investigated certain aspects of parallelization strategies for Datalog queries, from a probabilistic viewpoint. The novel ideas of the paper are:

- guiding the design and evaluation of parallelization strategies by *expected* performance; and,
- the use of a powerful tool from logic, 0-1 laws, in the evaluation of the expected behavior of parallelization strategies.

To illustrate the points above, we considered pure parallelization, a particularly simple algorithm that works very well in the limit situation when no information is known *a priori* about the input. The pure parallelization algorithm has the advantage of working for *all* Datalog programs. Using 0-1 laws, we showed that the expected performance of pure parallelization is excellent on arbitrary, random inputs. This was backed up by the experimental results on randomly generated data, provided in Section 6. As more assumptions are imposed on input structure, the behavior of pure parallelization degrades. For acyclic graphs it remains nonetheless very good, but far less so than for arbitrary graphs. For highly structured data, like complete binary trees, there is little or no gain from the parallelization. As expected, in such cases pure parallelization needs to be tailored further to the specific situation in order to be practical.

References

[1] S. Abiteboul, K. Compton and V. Vianu: "Queries are easier than you thought (probably)", *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1992, pp 23–32.

[2] S. Abiteboul, R. Hull and V. Vianu: *Foundations of Databases*, Addison-Wesley, 1995.

[3] S. Abiteboul and A. Van Gelder: "Optimizing Active Databases using the Split Technique", *Proc. Intl. Conf. on Database Theory*, 1992, pp 171–187.

[4] F. Bancilhon and R. Ramakrishnan: "Performance Evaluation of Data Intensive Logic Programs", *Foundations of Deductive Databases and Logic Programming, Ed J. Minker*, 1988, pp 439–517.

[5] D.A. Bell, J. Shao and M.E.C. Hull: "A Pipelined Strategy for Processing Recursive Queries in Parallel", *Data and Knowledge Engineering*, 6(5), 1991, pp 367–391.

[6] A.K. Chandra: "Programming Primitives for Database Languages", *Proc. ACM Symp on Principles of Programming Languages*, 1981, pp 50–62.

[7] A.K. Chandra and D. Harel: "Structure and Complexity of Relational Queries", *J. Computer and System Sciences*, 25(1), 1982, pp 99–128.

[8] J.-P. Cheiney, G. Kiernan and C. de Maindreville: "A Database Rule Language Compiler Supporting Parallelism", *Proc. Intl. Symp. on Database Systems for Advanced Applications*, 1993, pp 279–286.

[9] S.R. Cohen and O. Wolfson: "Why a Single Parallelization Strategy is not Enough in Knowledge Bases", *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1989, pp 200–216.

[10] R. Fagin: "Monadic Generalized Spectra", *Z. Math. Logik* 21, 1975, pp 89–96.

[11] R. Fagin: "Finite-Model Theory: a Personal Perspective", *Proc. Int'l. Conf. on Database Theory*, 1990, pp 3–24.

[12] S. Ganguly, A. Silberschatz and S. Tsur: "A Framework for the Parallel Processing of Datalog Queries", *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, 1990, pp 143–152.

[13] E. Grandjean: "Complexity of the First-order Theory of Almost all Structures", *Information and Control*, 52, 1983, pp 180–204.

[14] G. Hulin: "Parallel Processing of Recursive Queries in Distributed Archictectures", *Proc. Intl. Conf. on Very Large Data Bases*, 1989, pp 87–96.

[15] N. Immerman: "Relational queries computable in polynomial time", *Inf. and Control*, 68:86–104, 1986.

[16] G. Kiernan and C. de Maindreville: "A Rule Language Compiler for SQL Database Servers", *Data and Knowledge Engineering*, 7(2), 1991, pp 89–113.

[17] P.G. Kolaitis and M.Y. Vardi: "The Decision Problem for the Probabilities of Higher-Order Properties", *Proc. IEEE Symp. on Logic in Computer Science*, 1987, pp 425–435.

[18] P.G. Kolaitis and M.Y. Vardi: "0-1 Laws for Infinitary Logics", *Proc. IEEE Symp. on Logic in Computer Science*, 1990, pp 156–167.

[19] S. Lifschitz: "Stratégies d'évaluation parallèle de requêtes Datalog récursives" (in French), Ph.D. Thesis, Ecole Nationale Supérieure des Télécommunications, Paris, 1994.

[20] R. Rado: "Universal Graphs and Universal Functions", *Acta Arith.*, 9, 1964, pp 331–340.

[21] J. Seib and G. Lausen: "Parallelizing Datalog Programs by Generalized Pivoting", *Proc. ACM Symp. on Principles of Database Systems*, 1991, pp 78–87.

[22] J.D. Ullman: *Principles of Database and Knowledge-Base Systems*, Volumes I and II, Computer Science Press, 1989.

[23] J.D. Ullman: "Bottom-up beats Top-down for Datalog", *Proc. ACM Symp. on Principles of Database Systems*, 1989, pp 140–149.

[24] A. Van Gelder: "A Message Passing Framework for Logical Query Evaluation", *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, 1986, pp 155–165.

[25] M. Y. Vardi: "The complexity of relational query languages", *Proc. ACM SIGACT Symp. on the Theory of Computing*, 1982, pp. 137–146.

[26] O. Wolfson: "Sharing the Load of Logic-Programming Evaluation", *Proc. Intl. Symp. on Databases in Parallel and Distributed Systems*, 1988, pp 46–55.

[27] O. Wolfson and A. Ozeri: "A New Paradigm for Parallel and Distributed Rule-Processing", *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, 1990, pp 133–142.

[28] O. Wolfson and A. Silberschatz: "Distributed Processing of Logic Programming", *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, 1988, pp 329–336.

[29] W. Zhang, K. Wang and S-C. Chau: "Data Partition: a Practical Parallel Evaluation of Datalog Programs", *Proc. Intl. Conf. on Parallel and Distributed Information Systems*, 1991, pp 98–105.