Using Logica to Teach About Parallel Complexity

Evgeny Skvortsov[1], David S. Warren[2], Ojaswa Garg[1], Yilin Xia[3], Shawn Bowers[4] and Bertram Ludäscher[3]

[1]Google LLC, USA
[2]Stony Brook University, USA
[3]University of Illinois, Urbana-Champaign, USA
[4]Gonzaga University, USA

Abstract

Understanding parallel complexity is increasingly important in the era of multicore processors and distributed computation. Yet parallel complexity is rarely taught at the undergraduate level, in part because it has lacked an accessible programming model. We show that Logica, an open-source logic programming language, makes parallel complexity practical and teachable. In Logica, the parallel time of a computation corresponds to the recursion depth of the program, and the total work corresponds to the size of the computation across all steps. We illustrate this through a series of short, readable programs, which consists of transitive closure, shortest paths, connected components, edit distance, matrix exponentiation, and PageRank, each expressing a parallel algorithm in a few lines. We also discuss the connection to tabling in XSB Prolog. Several of our examples can be expressed via lattice answer subsumption, and we present an experimental `logica_table` library for SWI-Prolog that enables broader use of aggregation in tabling.

Keywords

parallel complexity, logic programming, Logica, Prolog, NC, teaching

1. Introduction

Parallel complexity — the study of how computation time can be reduced by using multiple processors — is fundamental to modern computer science. Every laptop today has multiple cores; every cloud computation is distributed. Understanding what can and cannot be parallelized is no longer a pure theoretical question.

Yet parallel complexity is not broadly taught at the undergraduate level. The standard complexity curriculum covers P, NP, and PSPACE, giving students tools to reason about sequential time. But it skips the parallel dimension almost entirely. Most students graduate knowing how to ask "can this be solved in polynomial time?" but not "can this be efficiently parallelized?"

We believe the gap is pedagogical, not substantive. The key concepts — parallel time, total work, the trade-off between them — are intuitive and important. What has been missing is an accessible programming model that supports intuitive reasoning and experimentation.

It is a well-known result from descriptive complexity theory that parallel complexity classes can be characterized via iterative Datalog evaluation [4, 12]. The recursion depth of a bottom-up evaluation corresponds directly to parallel time. However, in standard Datalog syntax that the theory directly refers to, this characterization remains a theoretical observation. A classical Datalog program for transitive closure is elegant, however for anything more interesting the programs are very hard to read and are rarely spelled out. On the other hand, the programs one can write in conventional programming languages like C++ or Java are too verbose to serve as compelling illustrations of parallel algorithms.

We propose that Logica fills this gap. Logica is an open-source logic programming language for data analysis that extends Datalog with aggregation, functions, and modules while compiling to SQL for execution. Its aggregation operators — `Min=`, `Max=`, `+=` — along with set-theoretic deduplication via `distinct`, allow parallel algorithms to be expressed concisely and readably. The structure of a Logica program — its recursion depth and the work done across all steps — makes parallel time and total work natural to understand and easy to reason about.

PEG 2026: Prolog Education Workshop



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

CEUR-WS.org/Vol-4311/PEG_regular_10.pdf

A student can write two versions of the same program — one with linear recursion depth, one with logarithmic — run both, and reason about the trade-off between parallel time and total work. This is parallel complexity, made tangible.

The rest of the paper is organized as follows. Section 2 provides background on parallel time and work, and introduces Logica. Section 3 develops the core pedagogical approach through transitive closure, showing how students can reason about recursion depth as parallel time, and introduces the complexity class NC. Section 4 presents a gallery of examples — shortest paths, connected components, edit distance, matrix exponentiation, and PageRank — each illustrating a different aspect of parallel complexity in a few lines of Logica. Section 5 discusses the connection to XSB Prolog's tabled evaluation and presents `logica_table`, an experimental library for semigroup aggregation in SWI-Prolog. Section 6 concludes.

2. Background

2.1. Parallel Time and Work

A parallel computation can be characterized by two measures: *parallel time* (also called *depth* — the number of steps, where each step may use arbitrarily many processors) and *work* (the total number of operations across all processors).

A good parallel algorithm minimizes depth while keeping work manageable. The ideal is a *work-efficient* parallel algorithm: one whose work matches the best sequential algorithm, but whose depth is polylogarithmic.

2.2. Recursion Depth as Parallel Time

The connection between recursion depth and parallel complexity is established through what is known as *naive evaluation* of logic programs. A program evaluated bottom-up computes a sequence of tables:

$$T_0, T_1, T_2, \ldots, T_k$$

where each T_{i+1} is obtained by applying all rules to T_i once, and k is the number of iterations until a termination condition (for instance fixed point) is reached.

The key insight is that a typical logic rule built out of predicate calls connected by conjunction and disjunction can be efficiently parallelized, that is evaluated in constant parallel time [13]. Fortunately parallelization remains possible when we extend classical logic programming with abelian semigroup aggregation as Logica does, since aggregation over a set of size n can be computed in $O(\log n)$ parallel time via a balanced reduction tree[4]. And therefore we have

- **Parallel time** = number of iterations = recursion depth
- **Work** = total size of computation across all iterations

It is worth noting that Prolog and Datalog implementations often prefer *semi-naive evaluation*[1], which avoids redundant computation by only considering new facts at each step. Logica uses naive evaluation, which recomputes the full table at each iteration. This can often increase total work, but the clean level-by-level structure gives a direct correspondence between recursion depth and parallel time, making it easy to reason about the asymptotics of parallel algorithms and to see the trade-offs between parallel speed and total work. For many of our examples, naive evaluation achieves the theoretically optimal parallel depth. A work-efficient parallelization — one matching both the optimal depth and the optimal total work — often cannot be expressed via naive evaluation alone. For teaching parallel complexity, however, this is good level of abstraction, and we use it to reason about parallel complexity in the following sections.

2.3. Logica

Logica [9, 8] is an open-source logic programming language for data analysis. Programs are compiled to SQL and evaluated bottom-up by the underlying engine (DuckDB, BigQuery, PostgreSQL, or SQLite).

Logica extends standard logic programming with *abelian semigroup aggregation*: aggregation defined by an associative and commutative binary operation. Operators such as `Min=`, `Max=`, `+=` appear directly in rule heads, giving concise expression to optimization, counting, and summation. Logica's aggregation mode is similar to XSB Prolog's answer subsumption [11], which supports *lattice* aggregation — semigroup aggregation with the additional requirement of idempotency. We discuss this connection in detail in Section 5.

Beyond aggregation, Logica provides functions, modules, and integration with SQL databases, making it practical for both education and data analysis. Execution stages are displayed during program evaluation, allowing students to observe the progression of computation.

3. Teaching to Reason About Parallel Complexity

We introduce the core ideas through the simplest possible example: transitive closure.

3.1. Two Programs for Transitive Closure

Consider a directed graph — for instance, a family tree where edges represent parent-child relationships. Computing the ancestor relation means finding all pairs of people connected by a chain of parent-child links. This is transitive closure, and a student usually encounters it in a first course on logic programming.

Given edges via predicate `E(a, b)`, we can compute all connected pairs with Logica as follows.

```
TC(a, b) distinct :- E(a, b);
TC(a, c) distinct :- TC(a, b), E(b, c);
```

Here we note Logica's convention of using `PascalCase` for predicates and `snake_case` for variables, using semicolon rather than period to denote end of the rule. Also note that Logica uses keyword `distinct` to declare predicates that are sets rather than multisets. As a language design choice, `distinct` is annotated at the rule level, however all rules defining a predicate must be consistently annotated.

In practice we often encounter graphs of bounded degree, so we focus our analysis on the case where degree of vertex is bounded by a constant b. So let us assume we are running this program for a graph with n edges of total diameter d. At each iteration, paths grow by one edge. Thus the number of iterations equals the diameter d of the graph. Then the work of each iteration in the worst case is $O(n^2)$ (joining TC, which is of size up to n^2, with the edge table E, each pair from TC matching to at most b edges). Since the longest path has length d, we need d iterations to find all paths. Total work: $O(n^2 \cdot b \cdot d)$.

Alternatively in the second rule we may be making both calls to predicate TC.

```
TC(a, b) distinct :- E(a, b);
TC(a, c) distinct :- TC(a, b), TC(b, c);
```

At each iteration, known paths are composed with known paths. The maximum path length doubles at each step. The number of iterations is $O(\log n)$. But each iteration does $O(n^4)$ work (joining TC of size n^2 with itself). Total work: $O(n^4 \log n)$.

The difference in the program text is a single edit: replacing `E(b, c)` with `TC(b, c)`. Yet this changes both the parallel time and the work.

3.2. Parametric Analysis

Just as students learn to analyze sequential time complexity parametrically, e.g. knowing for QuickSort algorithm "this algorithm runs in $O(n \log n)$", now they can analyze parallel depth parametrically. Like we have seen in the previous subsection for a graph of size n of degree bounded by b and diameter bounded by d we have:

- Single-recursion TC: recursion depth, i.e. parallel time is $O(d)$ and total work is $O(n^2 \cdot b \cdot d)$.
- Double-recursion TC: recursion depth, i.e. parallel time is $O(\log n)$ and work is $O(n^4 \log n)$.

Which is better? It depends on the graph. For a chain graph ($d = n$), single recursion has depth $O(n)$ and work $O(n^3)$, while double recursion has depth $O(\log n)$ and work $O(n^4 \log n)$. The double-recursion version is massively faster in parallel time, but at the cost of significantly more total work.

For a graph with small diameter ($d = O(\log n)$), single recursion already has logarithmic depth — and far less work. The "obvious" parallelization isn't always needed.

This kind of reasoning, in terms of parameters of the problem and possible trade-offs, is exactly what we want students to learn.

3.3. When Depth is Polylogarithmic: The Class NC

When the parallel time (recursion depth) of a program is $O(\log^k n)$ for some constant k, the program computes a problem in **NC** (Nick's Class) — the class of problems efficiently solvable in parallel.

Class NC is one of the central complexity classes. Whether P and NC are equal, that is, can every polynomial-time computation be efficiently parallelized or not, is a major open problem, on par with P vs NP in its importance for understanding the nature of computation. With Logica, NC becomes concrete. A student can ask: "Can I rewrite this linear-depth program to have logarithmic depth?" For transitive closure and shortest paths, the answer is yes. For some problems, no such rewriting is known, and finding one (or proving none exists) would resolve the NC vs P question.

This gives students a concrete way to engage with one of the deepest questions in complexity theory: not as an abstract theorem, but as a programming challenge. For example, in the previous section we showed a logarithmic-depth program for transitive closure, proving that TC is in NC.

Moreover, Logica programs are not merely illustrations: they compile to SQL and execute in parallel via the underlying engine. DuckDB utilizes multiple cores; BigQuery distributes computation across thousands of machines. Distribution does introduce synchronization overhead, but modern SQL engines are engineered to manage it efficiently. Part of the course can be dedicated specifically to explaining how the engine distributes data across machines and evaluates joins and aggregations in parallel, giving students a path to understand the lower level implementation.

4. Examples Gallery

To demonstrate Logica's usefulness in the education process we present a collection of classic parallel algorithms written in Logica. Each example is short and self-contained. For each, we discuss recursion depth (which corresponds to the parallel time) as well as total work.

4.1. All-Pairs Shortest Paths

Given a graph with edges `E(a, b)`, the all-pairs shortest path problem asks for the minimum number of edges between every pair of vertices. As with transitive closure, we can write two versions.

The single-recursion version extends paths one edge at a time:

```
D(a, b) Min= 1 :- E(a, b);
D(a, c) Min= D(a, b) + 1 :- E(b, c);
```

Here we encounter Logica's functional notation. We define distance predicate as a function `D(from_node, to_node) = distance`. It is an ordinary predicate, but we use one of the columns for value as a syntactic sugar. In this case it is especially natural to use the functional notation because for each pair of connected vertices we identify a single value, which is the distance. Note that in general in Logica functional value is just an attribute of the predicate, so we can be defining multi-valued functions.

Functional value can be defined without aggregation with a simple equality sign, or via a predicate-level aggregation, like it is done in this example. When a column is aggregated it has to be done consistently across all the rules of the predicate, similarly to how `distinct` would be applied to all or none of the rules.

The first rule says: if there is an edge from `a` to `b`, then the distance from `a` to `b` is at most 1. The second rule says: if the distance from `a` to `b` is known and there is an edge from `b` to `c`, then the distance from `a` to `c` is at most that distance plus one. The `Min=` operator keeps only the minimum over all derivations. Similarly to transitive closure we see that depth would be $O(d)$, where d is the diameter of the graph.

The double-recursion version composes pairs of known paths:

```
D(a, b) Min= 1 :- E(a, b);
D(a, c) Min= D(a, b) + D(b, c);
```

The second rule now says: if the distance from `a` to `b` and from `b` to `c` are both known, then the distance from `a` to `c` is at most their sum. At each iteration, the maximum known path length doubles — this is the logical equivalent of repeated squaring of the distance matrix. Here depth is $O(\log n)$ and thus this program proves that finding all distances is a problem from NC.

4.2. Connected Components

For an undirected graph (a graph where edges are symmetric) a connected component of a graph is a set of nodes such that you can travel between each pair of nodes. It is a natural question to ask: given a graph how many connected components does it have?

To solve the problem with Logica we note that a connected component of an undirected graph can be identified by the smallest vertex reachable from each vertex of the component. We can determine which vertex is reachable from which via computing transitive closure `TC`. We have computed it in Subsection 3.1 for a directed graph, and computing it for a simple graph can be done via replacing each undirected edge with two directed edges that go both ways of the original edge. Once `TC` has been computed we can then compute the number of connected components of a graph as follows.

```
Component(x) Min= y :- TC(x, y);
NumComponents() += 1 :- Component(x) == x;
```

The first rule assigns each vertex `x` to the component labeled by the minimum vertex `y` reachable from it. The second rule counts components: a vertex is a component representative if it is its own minimum, i.e. `Component(x) == x`. The `+=` operator sums 1 for each such representative. These two lines add only a constant to the depth — the parallel cost is dominated by TC.

4.3. Edit Distance

Edit distance is a classic dynamic programming problem. Given strings A and B of lengths n and m defined as functions like `A(i) = i_th_letter_of_a` we can compute edit distance incrementally defining a predicate `ED(len_of_prefix_a, len_of_prefix_b)`.

```
ED(0, 0) Min= 0;
ED(i, j) Min= ED(i-1, j) + 1 :- A(i);
ED(i, j) Min= ED(i, j-1) + 1 :- B(j);
ED(i, j) Min= ED(i-1, j-1) + cost :-
   cost = (if A(i-1) == B(j-1) then 0 else 1);
```

The sequential time complexity is $O(n \cdot m)$. But what is the *parallel* time?

Each cell (i, j) depends on cells $(i-1, j)$, $(i, j-1)$, and $(i-1, j-1)$. The longest chain of dependencies runs along the anti-diagonal: cells on the same anti-diagonal (where $i + j$ is constant) are independent and can be computed in parallel. There are $n + m$ anti-diagonals, so the parallel time is $O(n + m)$, so it is linear, not quadratic. Note that parallel complexity is revealed by simply writing the declarative formulation of the problem, and moreover when executed on Logica this complexity will be automatically achieved.

Writing this simple program, a student can immediately observe that an algorithm with quadratic sequential time has only *linear* parallel time. This follows directly from analyzing the recursion depth of the Logica program, applying the same parametric reasoning used for transitive closure.

4.4. Matrix Exponentiation

The previous examples use `Min` — an idempotent aggregation. Matrix multiplication requires `+=` (summation), which is not idempotent; this is an example of a program that is not natively supported by Prolog tabling, which we discuss in Section 5. This distinction matters: many parallel algorithms in linear algebra are built on matrix exponentiation.

```
@Recursive(PowerOfA, 5);
PowerOfA(i, j, 1) += A(i, j);
PowerOfA(i, j, 2*p) +=
   PowerOfA(i, k, p) * PowerOfA(k, j, p);
PowerOfA(i, j, 2*p+1) +=
   PowerOfA(i, k, p) * PowerOfA(k, j, p+1);
```

The `+=` operator performs the summation that constitutes matrix multiplication. The two recursive rules implement repeated squaring: $A^{2p} = A^p \cdot A^p$ and $A^{2p+1} = A^p \cdot A^{p+1}$.

The directive `@Recursive(PowerOfA, 5)` instructs Logica to perform 5 iterations. Since the power doubles at each step, this computes $A^{2^5} = A^{32}$ — the iteration count grows only logarithmically with the desired exponent. Logica also supports stop conditions, but a fixed iteration count is the simplest approach for the classroom: students set it to the value they need.

This is the building block for matrix inversion via the Leverrier-Faddeev algorithm (∼30 lines of Logica), determinant computation, and solving linear systems — all of which are in NC [2].

4.5. PageRank

PageRank is a more advanced example that illustrates an *iterative* parallel algorithm, where values are refined over successive rounds rather than computed in a single pass to a fixed point.

```
PR(x) += 0.15 / N() :- Node(x);
PR(x) += 0.85 * PR(y) / OutDeg(y) :- Edge(y, x);
```

The first rule gives each node a base rank of $0.15/N$, where N is the number of nodes. The second rule distributes each node's rank equally among its outgoing neighbors, scaled by the damping factor 0.85. The `+=` operator sums contributions from all incoming edges.

Like all programs in this paper, Logica evaluates PageRank by naive iteration: at each step, the old table of facts is replaced by a new table derived from the rules. Once intuition is developed, we see that in all our examples this iteration converges monotonically to the answer with respect to some order — however, this order becomes more and more subtle as we progress to more advanced examples.

For matrix exponentiation, the new facts are a superset of the old — this is set monotonicity, the most transparent case. For shortest paths, distances are updated, but the replacement is clearly "better": values can only decrease in the lattice order. For PageRank, monotonicity is the most subtle: the Euclidean distance from the current distribution to the stationary distribution decreases at each step — a consequence of how a stochastic matrix contracts along its non-dominant eigenvectors [6, 3]. Convergence is geometric: after k iterations the error is proportional to 0.85^k, so $O(\log(1/\varepsilon))$ iterations suffice for precision ε [5].

Unlike lattice monotonicity, this guarantee does not follow from the syntax of the program alone — it requires understanding the underlying linear algebra. A lecturer should introduce such examples only after students are comfortable with the simpler cases, or skip them entirely.

5. Connection with XSB Tabling

The ideas above are not exclusive to Logica. XSB Prolog's tabled evaluation with answer subsumption [11, 10] supports related functionality, and it is instructive to examine the connection.

5.1. Lattice Answer Subsumption

XSB's answer subsumption allows aggregation during tabling via lattice operations — binary operations that are associative, commutative, and *idempotent*. For our examples that use `Min=`, as well as those that rely on deduplication (`distinct`), this is a direct fit. Shortest paths with double recursion, for instance, translate naturally:

```
:- table dist(_, _, lattice(min/3)).
min(X, Y, Z) :- (X < Y -> Z = X ; Z = Y).

dist(X, Y, 1) :- edge(X, Y).
dist(X, Y, D) :- dist(X, Z, D1), dist(Z, Y, D2),
             D is D1 + D2.
```

5.2. Beyond Lattices

For examples that use summation (`+=`), the situation is more nuanced. Summation is not idempotent, so it is not a lattice operation in the strict sense. Nevertheless, XSB's `lattice(sum/3)` can correctly compute summation in cases where the data flow is acyclic — for instance, in matrix exponentiation, where the recursion flows through the exponent argument without cycles. In such cases, each contribution arrives exactly once. We ran a few tests with SWI-Prolog and the non-idempotency of summation caused no problems in our experience. However, the keyword "lattice" indicates that this usage is not quite originally intended.

For cyclic computations like PageRank, however, standard answer subsumption with summation diverges: the accumulating table grows without bound.

5.3. An Experimental Bridge: `logica_table`

To explore the possibility of bringing Logica-style semigroup aggregation to Prolog, we developed `logica_table`, an experimental library for SWI-Prolog. The library tracks individual contributions by tagging each with a unique key, and replaces — rather than accumulates — contributions from the same source when they are updated during fixpoint iteration. This effectively reduces semigroup aggregation to lattice aggregation on tagged values.

With `logica_table`, PageRank converges correctly in SWI-Prolog:

```
:- consult(logica_table).
sum(X, Y, Z) :- Z is X + Y.

:- logica_table pr(_, abelian_semigroup(sum/3)).
pr(X, V) :- n_count(C), n(X), V is 0.15 / C.
pr(Y, V) :- pr(X, PX), d(X, D), e(X, Y),
         V is PX / D * 0.85.
```

The library is at an early stage — it handles PageRank and similar iterative computations, but more complex programs may encounter subtleties related to key derivation and contribution tracking. We see this as a promising direction for future work rather than a finished tool. The library is available as part of the supplementary materials.

Note that the current implementation of XSB tabling is sequential, although certain progress towards developing a parallel version was made. YAP Prolog explored parallel tabled evaluation more completely [7].

5.4. Summary

The lattice-based examples in this paper (transitive closure, shortest paths, connected components) can be expressed directly in XSB Prolog via answer subsumption and thus Prolog can also be used in the discussions of parallel complexity. Examples requiring non-idempotent aggregation (matrix exponentiation, PageRank) work in Logica natively, and can be brought to Prolog via `logica_table` with varying degrees of maturity.

6. Conclusion

Parallel complexity is increasingly important yet rarely taught at the undergraduate level. We have shown that Logica makes it accessible through short, readable programs where parallel time corresponds directly to recursion depth.

The key pedagogical move is parametric analysis of recursion depth — the same skill students already learn for sequential complexity, applied to the parallel dimension. For transitive closure, single recursion gives depth proportional to the graph diameter, while double recursion gives logarithmic depth. For edit distance, a quadratic-time algorithm turns out to have only linear parallel time. For matrix exponentiation, repeated squaring achieves logarithmic depth with non-idempotent aggregation. In each case, a few lines of Logica express a parallel algorithm that students can analyze, compare, and reason about.

When depth is polylogarithmic, the program computes a problem in NC — one of the most important and least taught complexity classes. Where depth cannot be reduced — that is the frontier of parallel complexity theory, and students can now see exactly where that frontier lies.

We have also shown that the Prolog ecosystem supports much of this through tabled evaluation with answer subsumption. The `logica_table` library offers an experimental path toward broader aggregation support in Prolog, and we see its further development as promising future work. Logic programming — whether Logica or Prolog — is a natural home for reasoning about parallel complexity.

Acknowledgments

We thank the Prolog Education Workshop community for fostering discussions on logic programming pedagogy. We are grateful to Laura Andrea Cecchi for the discussion of the topic and encouragement to pursue this line of research.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini (Google), Claude (Anthropic), and ChatGPT (OpenAI) for grammar and spelling checks and to reword portions of the text. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] François Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer, 1986.

[2] L Csanky. Fast parallel matrix inversion algorithms. *SIAM Journal on Computing*, 5(4):618–623, 1976.

[3] Georg Frobenius. Über Matrizen aus nicht negativen Elementen. *Sitzungsberichte der Königlich Preussischen Akademie der Wissenschaften*, pages 456–477, 1912.

[4] Neil Immerman. *Descriptive Complexity*. Springer, 1999.

[5] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[6] Oskar Perron. Zur Theorie der Matrices. *Mathematische Annalen*, 64(2):248–263, 1907.

[7] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 5(1–2):161–205, 2005.

[8] E. S. Skvortsov, Y. Xia, S. Bowers, and B. Ludäscher. The Logica System: Elevating SQL Databases to Declarative Data Science Engines. In *Datalog-2.0*, volume 3801 of *CEUR Workshop Proceedings*, pages 69–73, 2024.

[9] E. S. Skvortsov, Y. Xia, and B. Ludäscher. Logica: Declarative Data Science for Mere Mortals. In *EDBT*, pages 842–845, 2024.

[10] Terrance Swift and David S Warren. Tabling with answer subsumption: Implementation, applications and performance. In *European Conference on Logics in Artificial Intelligence (JELIA)*, pages 300–312. Springer, 2010.

[11] Terrance Swift and David S Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[12] Jeffrey D Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

[13] Moshe Y Vardi. The complexity of relational query languages. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 137–146, 1982.