PARALLEL EVALUATION OF DATALOG PROGRAMS BY LOAD SHARING*

OURI WOLFSON

▷ We propose a method of parallelizing the evaluation of data-intensive Datalog programs. The method is distinguished by the fact that it is *pure*, i.e., does not require interprocessor communication, or synchronization overhead. The method cannot be used to parallelize every Datalog program, but we syntactically characterize several classes of Datalog programs that are *sharable*, i.e., programs to which the method can be applied. We also provide a characterization of a class of *nonsharable* programs, and demonstrate that sharability is a fundamental notion that is independent of the syntactic parallelization method proposed in this paper. This notion is related to bottom-up evaluation (we propose a formal characterization of this type of control-strategies) and to program classification. ◁

1. INTRODUCTION

1.1. Background

In recent years, the emphasis in database research has shifted to knowledge base systems [34]. A knowledge base is a database augmented with a set of Horn-clause rules (the logic program). The rules enable inference of information that is not explicitly stored in the database, and, because of its declarative style, Logic Programming is easier and more natural for specifying inferences in many problem domains. The main difficulty in knowledge base implementation turns out to be the performance of query processing. The reason is, that in order to answer a query the logic program has to be "evaluated." This means that the relevant information which is implied by the

*A preliminary version of this paper appears in the International Symposium on Databases in Parallel and Distributed Systems, December 1988. This research was supported in part by NSF grant IRI-90-03341.
Address correspondence to Ouri Wolfson, Dept. of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60680.
Accepted December 1990.

655 Avenue of the Americas, New York, NY 10010 0743-1066/92/$05.00

knowledge base should be actively inferenced. Given a large knowledge base, logic program evaluation can be a very lengthy process. This performance difficulty is evidenced by the extensive research published recently on optimization of logic program evaluation. Surprisingly little work has been done in terms of obtaining speedup by parallelism. We shall refer to this an other relevant work in subsection 1.4, but next we discuss the overhead in parallelization.

Assume an environment with multiple processors, which either communicate by message passing or have common memory. Parallel computation, on either type of architecture, usually involves an overhead required for synchronization and communication among the processors. Synchronization overhead occurs, for example, when one processor has to wait for intermediate results from another processor, or, when it waits to enter a critical section. Communication overhead occurs in a message passing architecture when processing incoming or outgoing messages. For parallel computation by hundreds of processors, the above overhead causes the thrashing phenomenon. This means that increasing the number of processors beyond a certain limit causes a decrease, rather than an increase, in performance (see, for example, [15]). Some problems are amenable to *pure* parallelization, i.e., parallelization that does not incur any communication and synchronization overhead. Therefore, it is important to investigate if pure parallelization is possible, and if so, how it should be done.

1.2. The Method

We propose a method of pure parallelization of the evaluation of data-intensive logic programs, of the type employed in knowledge base systems [33]. The proposed method is based on the data-reduction paradigm, introduced in [41]. The paradigm is to algorithmically create, for a given logic program, P, other logic programs called restricted versions of P, and to assign to each processor a different restricted version. Each processor evaluates its restricted version using as input a local copy of the database. At the end, the union of outputs comprises the output of the original program (completeness). Therefore, if these outputs are sent to the same device or stored in the same file, the result is equivalent to a single-processor evaluation. A restricted version of P is a logic program obtained by appending hash-function predicates to the body of some rules of P. A hash function maps each instantiation of the rule to a processor that becomes "responsible" for the instantiation. Instantiation-partitioning results in less output, and thus smaller relations to manipulate at each processor (hence the name data-reduction).

In this paper, we concentrate on pure data-reduction, namely on the method called load sharing. A program has a load-sharing scheme if its complete output can be produced by restricted versions evaluated in an independent fashion, i.e., without incurring a communication overhead among the processors. The next example demonstrates load sharing.

Example 1. Throughout this paper, we assume familiarity with a subset of the language Prolog, called Datalog (see [22]). Assume that a database has three relations UP, FLAT, and DOWN, which represent a directed graph with three types of arcs. The logic program below defines a tuple (a, b) to be in the intentional relation, S, if and

only if there is a path from a to b having k UP arcs, one FLAT arc, and k DOWN arcs, for some nonnegative integer k.

$$S(x, y) :- UP(x, w), S(w, z), DOWN(z, y)$$

$$S(x, y) :- FLAT(x, y)$$

The above program is called in [23] the *canonical strongly linear* (*csl*) program.

Given processors $\{0, \ldots, r-1\}$, we propose that they share the load of evaluating the relation S, as follows. Processor i executes the *csl* program, with the predicate $i = x \bmod r$ appended to the body of the second rule of the program.[1] In other words, processor i computes (independently of the other processors) the tuples (a, b) for which the path goes through a FLAT arc (c, d) with $i = c \bmod r$.[2] It is intuitively clear that for a large random graph, each one of the processors generates less tuples.

To demonstrate the time saved for a specific input the csl program, consider the extensional database relations of Figure 1. UP consists of the tuples $(i, i+1)$ for $i = 1, \ldots, 4$, FLAT consists of the tuples $(i, 6)$ for $i = 1, \ldots, 5$ and DOWN consists of the tuples $(i, i+1)$ for $i = 6, \ldots, 9$. The set NEW, defined below, consists of the tuples of S that are not in FLAT.

$$NEW = \{(4,7), (3,7), (2,7), (1,7),$$
$$(3,8), (2,8), (1,8),$$
$$(2,9), (1,9),$$
$$(1,10)\}$$

Assume that S is computed by the naive evaluation method (see [3]). It assigns FLAT to S and then iteratively adds to S the tuples in the (projection of) *UP join S join DOWN*. Then in the first iteration, a single processor evaluating csl performs the join of a four-tuples relation (UP), with a five-tuples relation (S), with a four-tuples relation (DOWN). In the second iteration, the relations UP, S, DOWN are of sizes 4, 9, 4, respectively (first row of the set NEW has been added to S); third iteration 4, 12, 4 (second row has been added); fourth iteration 4, 14, 4; fifth and last iteration 4, 15, 4.

However, if two processors share the load by having processor i execute the csl program with the predicate $i = x \bmod 2$ added to the nonrecursive rule, then the arcs (1, 6), (3, 6), (5, 6) will be assigned to processor 1, and the rest to processor 0. The maximal computation burden is placed on processor 1, performing five iterations with

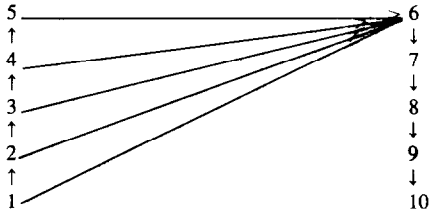

FIGURE 1. Sample input to the csl program.

[1] i = (x + y) mod r works as well

[2] This works for character-strings as well, since the binary representation can be regarded as a natural number

relations of sizes 4,3,4, 4,5,4, 4,7,4, 4,8,4, 4,9,4. Due to the smaller S-relation at each iteration, a significant time saving compared to the single processor case occurs. Processor 0 has a lower computation burden than processor 1, and completes even faster. If there are five processors instead of two a greater time saving results. In this case the maximum burden is placed on processor 0, performing five iterations, with relations of sizes 4,1,4, 4,2,4, 4,3,4, 4,4,4, 4,5,4, respectively.

Similar observations can be made if the evaluation is semi-naive [3] or relational top-down [35], rather than naive. □

1.3. Main Results

The main objective of this paper is to determine which programs are sharable, i.e., can be evaluated by the load-sharing scheme described in subsection 1.2. Specifically, we formally define what it means for a program to be sharable and explore the syntactic characterization of sharable programs. We characterize a large subclass of all the linear programs (i.e., programs in which each rule has at most one intentional predicate in the body) that is sharable. In the class of simple chain programs (originally defined in [36] and redefined in subsection 4.3), we also characterize a subclass that is sharable. Additionally, we define the potential-speedup measure for a load-sharing scheme. It is the "best-case" speedup, as measured in terms of the output size. We show that all programs in the sharable classes we characterize have a load-sharing scheme with an optimum potential-speedup. Then we provide a necessary syntactic condition for a program to be sharable. Several well-known programs do not satisfy the condition (e.g., path systems, introduced in [12]).

Well, maybe when work sharing cannot be obtained by appending hash-function predicates to the original Datalog program, some other method, with similar parallelization properties, will work. The answer is negative for bottom-up type of methods, and positive for others. To show this we extend the definition of sharability by restricted versions of the original program, to sharability by *general algorithms* for logic-program evaluation. In other words, we do not restrict a parallel algorithm to consist of independent evaluations of restricted versions, although we still insist that it is pure. Then we discover that the programs that are not sharable by restricted versions are also not sharable by bottom-up algorithms. This demonstrates that sharability is a fundamental property of the coupling of some logic-program classes with bottom-up control, and the property is independent of the syntactic parallelization scheme that we propose.

1.4. Other Relevant Work

The efficient evaluation of intentional database relations, defined by means of recursive logic programs, has recently emerged as a very active area of research [5, 20, 34]. Two main methods of improving performance have received most of the attention. One is selection propagation, and the other is parallel evaluation.

Selection propagation speeds-up the evaluation by using constants passed as parameters to the database query processor, thus reducing the number of relevant input-database tuples. This usually necessitates a rewriting of the logic program that defines the intentional relation. The best known rewriting algorithms are "magic sets" [4, 6], and "magic templates" [27].

Parallel evaluation use multiple cooperating processors for the purpose of speed-up. Most efforts in this area have been devoted to characterization of the logic programs that belong to the NC complexity class [1, 12, 20, 36]. If a program is in NC, it means that it can be evaluated very fast, given a polynomial (in the number of input-database tuples) number of processors; they have to communicate extensively, usually through common memory. If the number of processors is constant (as we assume), then the NC-type of evaluation algorithms can be adapted, by assigning the work of multiple processors to a single processor. However, it turns out that which multiple processors are assigned to a single one, i.e., how the pieces of work are grouped, is very important as far as overhead (particularly if the processors do not share memory). The present paper studies the issue of work sharing with zero overhead.

The evaluation strategies for Datalog programs usually amount to iteratively performing one or more join operations, then adding the newly generated tuples to the intentional relations, until a fixed point is reached. Another way of utilizing a constant number of processors for the evaluation is to parallelize relational algebra operators, particularly the join operation (e.g., [8, 37]). However, if so, then in order to assure that all output tuples are generated, at each iteration, each processor would have to exchange its newly generated tuples with the newly generated tuples of every other processor. This procedure involves a lot of message passing, or synchronization in accessing common memory. Generally, the use of hash-functions for partitioning the data or the work of evaluating relational algebra operations has been employed in the past (e.g., [7, 9]). The novelty of our approach is the logic-program analysis to determine whether or not hash-based parallelism with zero amount of communication is possible.

Another data-reduction method of parallelizing the evaluation without synchronization was introduced in [42]. It resembles the one we proposed above, except for an important difference. The method requires that each new tuple generated in the evaluation process is computed by a *unique* processor. The purpose is to partition (rather than share, as in our method) the evaluation load. But, consequently, the [42] method is applicable only to a very restricted class of logic programs, called *decomposable*. For example, in the class of simple chain programs [1, 36], only the regular ones are decomposable. Therefore, the csl program of example 1 is not decomposable. Intuitively, the reason for this is that since there may be more than one path between a and b it is not guaranteed that each tuple is computed by a unique processor. For instance, in the sample input of example 1, if, in addition to the listed tuples, the tuple (2, 9) is also in FLAT, then the tuple S(2, 9) is computed by both processors 0 and 1. Decomposable sirups were completely characterized syntactically in [11], and in [41] we proved that it is undecidable to determine whether or not an arbitrary Datalog program is decomposable. Two notions related to load sharing were introduced in [16]. The first classifies the programs for which *there exist* inputs, such that the Datalog program can be evaluated in parallel with zero overhead and disjoint outputs. (In contrast, the decomposability notion requires that an evaluation with these properties exists *for every* input.) The second classifies the programs for which the output can be obtained as union of the outputs of the program on all fixed-size subsets of the input. In contrast, load sharing does not make the all-fixed-size-subsets requirement.

A variant of data reduction, named "copy and constrain," was also proposed, independently, in the production-system literature [32], and its merit was demonstrated experimentally using OPS5 [26]. However, the issue addressed in this paper, namely

classification of programs that are (are not) amenable to pure data-reduction, has not been discussed.

Another comment concerns the work on parallel and concurrent versions of Prolog (see [14, 30]). There has been a lot of research on the subject, but because of the fundamental difference between the tuple-oriented processing in programming languages, and the set-oriented processing in knowledge-bases, the possible cross-fertilization between the two is not clear. The reason for set-orientation is that database applications are data intensive, and they usually looks for *all* answers to a query. A way of viewing the difference is that parallelizing tuple-oriented processing usually employs control- (or agenda-, in [10] terminology) parallelism, whereas parallelizing set-oriented processing usually employs data- (or result-) parallelism.

Finally, Ramakrishnan proposes in [28] an interesting measure for comparing the inherent parallelism of various evaluation methods that are based on *sips*. Exploiting this parallelism, and the overhead in doing so, are not discussed. We, on the other hand, emphasize these aspects, and in this sense, the approaches seem complementary. More work is necessary to reap the benefits of both of them simultaneously.

1.5. Paper Organization

The rest of this paper is organized as follows. In section 2 we provide the preliminaries, and in section 3 we define the concepts of a load-sharing scheme, and it potential speedup. In section 4 we characterize classes of programs that have a load-sharing scheme, and in section 5 we prove that a whole class of single rule programs cannot have a load-sharing scheme. In section 6 we define the theory that enables extension of the results to parallel computation by general algorithms. In Section 7 we conclude and discuss future work.

2. PRELIMINARIES

A *literal* is a predicate symbol followed by a list of arguments. An *atom* is a literal with a constant or a variable in each argument position. A *constant* is any nonnegative integer. The other arguments of an atom are the *variables*. An *R-atom* is an atom having R as the predicate symbol. A *rule* consists of an atom, Q designated as the *head*, and a set of one or more atoms, $Q^1, \ldots, Q^k$, designated as the *body*. Such a rule is denoted $Q{:-}\, Q^1, \ldots, Q^k$, which should be read "Q if Q^1 and Q^2, and, ..., and Q^k." A rule, or an atom, or a set of atoms, is an *entity*. If an entity has a constant in each argument position, then it is a *ground* entity. A ground atom is also called a *fact*. A *substitution* applied to an entity, is the replacement of each variable in the entity by another variable, or by a constant. It is denoted $x1/y1, x2/y2, \ldots, xn/yn$ indicating that xi is replaced by yi. A substitution is *ground* if each variable is replaced by a constant. A ground substitution applied to a rule in an *instantiation* of the rule. An instantiation is *one-to-one* if each variable is mapped to a distinct constant.

A Datalog program, or a program for short, is a finite set of rules who predicate symbols are divided into two disjoint subsets: the *extensional* predicates and the *intentional* predicates. The extensional predicates are distinguished by the fact that they do not appear in any head of a rule. For a predicate symbol R, a finite set of

R-ground-atoms is a *relation* for R. A *database* for P is a relation for each predicate of P. An *input* to P is a relation for each extensional predicate. Then the *output* of P given an input I is denoted $O(P, I)$ and consists of all the intentional facts that have a derivation tree. A *derivation tree* for a fact, a, is a rooted directed finite tree with facts as nodes; it has a as the root, each leaf is an atom of the input I, and for each internal node, b, that has children $b_1, \ldots, b_k$, there is an instantiated rule of P with b as the head, and $b_1, \ldots, b_k$ as the body. We say that b and its children *represent* the instantiated rule in the derivation tree.

For each rule, a variable that appears in the head is called a *distinguished* variable. For simplicity we assume that each rule of a program is *range restricted*, i.e., every distinguished variable also appears in the body of the rule. Additionally, we assume that none of the rules of a program has constants. Our main results hold even if constants are allowed in some argument positions, and we shall point this out throughout the paper.

An *evaluable predicate* is a predicate of the form $e_1 \theta e_2$, where e_1 is an arithmetic expression involving some subset of $\{+, -, *,$ modulo$\}$; the same for e_2. The predicate symbol, θ, is an arithmetic comparison operator (i.e., $<$, $>$, $\leq$, $\geq$, $=$, $\neq$). A rule, say *ra*, is an *restricted version* of some rule r, and *ra* have exactly the same variables, and r can be obtained by omitting zero or more evaluable predicates from the body of *ra*. In other words, *ra* is r with some evaluable predicates added to the body, and the arguments of these evaluable predicates are variables of r, or constants (note that in the evaluable predicates, in contrast to the other predicates, constants are allowed). For example, if r is:

$$S(x, y, z) :- S(w, x, y), A(w, z)$$

then one possible *ra* rule is

$$S(x, y, z) :- S(w, x, y), A(w, z), x - y = 5$$

A program P_i is a *restricted version* of program P if each one of its rules is a restricted version of some rule of P. P_i may have more than one restricted version of a rule r of P. To continue the above example, if P has the rule r, then P_i may have the rule *ra* as well as the rule *ra'*:

$$S(x, y, z) :- S(w, x, y), A(w, z), x - y = 6$$

Throughout this paper, only a restricted version of a program (but not the program) may have evaluated predicates, and they are used as hash functions that map each rule-instantiation to a processor. The input of a restricted version is defined as before. The output is also defined as before, with the following exception. If an instantiated rule is represented in the derivation tree, then the instantiation must satisfy[1] the evaluable predicates of the rule. In other words, instantiations for a restricted version of a rule are disregarded, if they do not satisfy the additional evaluable predicates. Intuitively, this is the reason we are interested in restricted versions. Their evaluation is faster since it necessitates performing only a fraction of the instantiations (the others are disallowed by the hash functions), and this fraction of the instantiations also considers a smaller database.

[1] For example, the substitution x/14, y/8 satisfies the evaluable predicate x − y = 6, whereas the substitution x/13, y/9 does not.

A predicate Q in a program P *directly derives* a predicate R if it occurs in the body of a rule whose head is a R-atom. The predicate Q is *recursive* if (Q, Q) is in the nonreflexive transitive closure of the "directly derives" relation. A program is *recursive* if it has a recursive predicate. The predicate Q *derives* predicate R if (Q, R) is in the reflexive transitive closure of the "directly derives" relation (particularly, every predicate derives itself). A rule is *recursive* if the predicate in its head derives some predicate in its body.

3. LOAD SHARING SCHEMES

In this section, we define and discuss the concept of a load-sharing scheme. Then we define and discuss the notion of potential speedup of a load-sharing scheme. In subsequent sections, we determine that all the load-sharing schemes discussed in this paper have the maximum potential speedup.

Assume that P is a program, and $P_1, \ldots, P_r$ are restricted versions of P. The set $D = \{P_1, \ldots, P_r\}$ is a *load-sharing scheme* for evaluating P, if the following two conditions hold:

1. For each input I to the programs $P, P_1, \ldots, P_r, \bigcup_i O(P_i, I) \supseteq O(P, I)$ (completeness).
2. There is an input I_0, such that $O,(P, I_0) \supset O(P_i, I_0)$ for each i (nontriviality).

If the program P has a load sharing scheme, then we say that P is *sharable*.

In order to intuitively explain the above definition, we assume that each processor has an restricted version of the program P, and the whole database, i.e., the set of input base relations, is replicated at each one of r processors. Alternatively, the database may reside in common memory.

The completeness requirement in the definition, is that no output atom is lost by evaluating all the P_i's, rather than P. Although the requirement is for inclusion in one direction only, the fact that $\bigcup_i O(P_i, I)$ does not contain any output atoms which are not in $O(P, I)$ is implied by the fact that each P_i is a restricted version of P. Thus, by using multiple processors and taking the union of the outputs, the exact output of P is obtained.

The nontriviality requirement says that for some input, I_0, the output of each P_i is smaller than the output of P. If, along the lines suggested in [5, Section 4], the load of evaluating an intentional relation is measured in terms of the number of new tuples generated in the process, then the evaluation by the load-sharing scheme completes sooner for the input I_0. The very permissive form of the nontriviality requirement, namely that time saving occurs for "some" input, has two independent reasons, each interesting in its own right. First, even for this permissive form, some single-rule-programs do not have a load-sharing scheme, thus strengthening the negative results. Second, for the classes of programs shown sharable in this paper, there is an infinite number of inputs that satisfy a stronger condition than nontriviality. For each one of them, the output-production load is evenly partitioned among the processors. This will be shown using the potential-speedup notion. Furthermore, although we use modulo throughout this paper, any hash function can be used instead.

Finally, observe that the combination of completeness and nontriviality forces the size of any load-sharing scheme to be bigger than one.

Given a program P, the set of restricted versions $\{P_1, \ldots, P_r\}$ is a *load-decomposing scheme* if it is a load-sharing scheme and an additional requirement, called lack-of-duplication, is satisfied. Lack-of-duplication states that for each input I to $P_1, \ldots, P_r$, each pair of distinct outputs, $O(P_i, I)$ and $O(P_j, I)$ are disjoint. In other words, two restricted versions do not duplicate one another's work, by computing the same output atom. Load decomposition was defined in [42]. In this type of parallelization, the total work of all the processors is equal to the work done by one processor (see [11]), using a serial evaluation method. However, as explained in the introduction, it is applicable to a much more restricted class of programs.

Assume now that there is an algorithm which given P, and the set of restricted versions, $D = \{P_1, \ldots, P_r\}$, determines whether D constitutes a load sharing scheme for evaluating T in P. Then we could solve the following problem, polynomial solvability, which is undecidable based on results in [18, 24]. Given a polynomial $p(x_1, \ldots, x_{13})$ in 13 variables, with integer coefficients, are there natural numbers $\alpha_1, \ldots, \alpha_{13}$ such that $p(\alpha_1, \ldots, \alpha_{13}) = 0$. For a given polynomial, say $p_0(x_1, \ldots, x_{13})$, consider the program $S(x_1, \ldots, x_{13}) :- B(x_1, \ldots, x_{13})$ and restricted versions of it:

$$S(x_1, \ldots, x_{13}) :- B(x_1, \ldots, x_{13}), p_0(x_1, \ldots, x_{13}) = 0$$

and

$$S(x_1, \ldots, x_{13}) :- B(x_1, \ldots, x_{13}), p_0(x_1, \ldots, x_{13}) \neq 0.$$

Completeness is obviously satisfied, and nontriviality is satisfied if an only if $p_0(x_1, \ldots, x_n)$ has a solution in the natural numbers. Therefore,

Proposition 1. For a given program, P, and a set of restricted versions, $D = \{P_1, \ldots, P_r\}$, it is undecidable to determine whether D is a load-sharing scheme for evaluating P.

A related problem is that of determining for a given program P, whether there exists a load-sharing scheme for evaluating it (assuming that we restrict attention to computable evaluable predicates). We conjecture that this problem is also undecidable.

Next we define the notion of the potential speedup. Let P be a program, and $D = \{P_1, \ldots, P_r\}$ a load-sharing scheme for evaluating it. The *potential speedup* of D, denoted $Ps(D)$, is the maximal number M for which the following condition is satisfied. For every integer n and every ϵ, there is an input I for which $|O(P, I)| > n$, and $|O(P, I)| / \max_i |O(P_i, I)| \geq M - \epsilon$. Intuitively, the potential speedup is the number to which the ratio $|O(P, I)| / \max_i |O(P_i, I)|$ can come arbitrarily close, for an input I which is arbitrarily large. The definition is somewhat complicated since there are load-sharing schemes (the ones discussed in subsection 4.3) for which the potential speedup cannot be achieved, but to which the ratio can come arbitrarily close. Note that the fact that D is a load-sharing scheme implies that $1 \leq Ps(D) \leq r$.

The potential speedup means that for each one in an infinite set of inputs, the output of each P_i is at least $Ps(D)$ times smaller than the output of P; also, this output reduction occurs for arbitrarily large outputs. When the load to evaluate P is measured in terms of new ground atoms generated in the evaluation process, $Ps(D)$ is the ratio between the load evaluating P with one processor, and the maximum load of a processor of the scheme. Although we defined the potential speedup based on some

infinite set of inputs, for the load sharing schemes that we are discussing in this paper, it is intuitive that time saving can be achieved for the "average input." The reason is that each load-sharing scheme discussed in this paper is obtained by adding the evaluable predicate $i = (x_1 + , \ldots , + x_k) \bmod r$ to some of the rules, where $x_1, \ldots, x_k$ are distinguished variables. For an input that is distributed evenly across a range of natural numbers, this reduces the number of newly generated tuples at each processor.

Finally, note that the potential-speed notion can be similarly defined for D being any parallel evaluation algorithm, not necessarily a load-sharing scheme. However, a moment of reflection will reveal that the more communication and synchronization overhead the algorithm incurs, the less accurate the output-size measure becomes.

4. SHARABLE PROGRAMS

4.1. Pivoting Single Rule Programs

A *single rule program* (see [13]), or a *sirup* for short, is a Datalog program that has a single intentional predicate, denoted S in this paper. The program consists of two rules. A nonrecursive rule,

$$S(x1, \ldots , xn) :- B(x1, \ldots , xn).$$

where the xi's are distinct variables; and one other, possibly recursive, rule in which the predicate symbol B does not appear.

Assume that R is a set of atoms, with each atom having a variable in each argument position. The set R is *pivoting* if there is a subset d of argument positions, such that in the positions of d:

1. the same variable appear (possibly in a different order) in all atoms of R, and
2. each variable appears the same number of times in all atoms of R.

A member of d is called a *pivot*. Note that a variable which appears in a pivot may also appear in a nonpivot position. The recursive rule of a sirup is *pivoting* if all the occurrences of the recursive predicate in the rule constitute a pivoting set. For example, the rule

$$S(w, x, x, y, z) :- S(u, y, x, x, w), S(v, x, y, x, w), A(u, v, z)$$

is pivoting, with argument positions 2, 3, and 4 of S being the pivots.

Theorem 1. If the recursive rule of a sirup, P, is pivoting, then P has a load-sharing scheme of any size. The potential-speedup equals the size of the scheme.

PROOF. Assume that argument positions $i_1, \ldots, i_k$ of S are the pivots. To obtain a scheme of size r, consider restricted version P_j of P which has the same recursive rule as P, and a nonrecursive rule

$$S(x1, \ldots , xn) :- B(x1, \ldots , xn), \qquad j = (xi_1 + xi_2 + , \ldots , + xi_k) \bmod r$$

for $j = 0, \ldots , r - 1$. It is easy to see that $D = \{P_0, \ldots , P_{r-1}\}$ is a load-sharing scheme. Completeness is quite straightforward and has been shown in [42, Theorem 2]. Intuitively, it results from the fact that, since the sirup is pivoting, all the S-facts in any

derivation tree have the same constants in the pivot positions, possibly in a different order. Completeness results from the sum being a commutative function (in fact, any commutative function can be used to hash the instantiations to processors). Based on the observation about derivation-trees, lack-of-duplication has also been shown in [42], namely, that for every input the outputs of the restricted versions are pairwise disjoint. A potential-speedup of r, which obviously also implies nontriviality, can be demonstrated using as input a prefix of the following sequence:

$$\{B(1, \ldots, 1, q, 1, \ldots, 1) \mid q \geq r, \text{ and } q \text{ appearing in position } i_1\}$$

In other words, this is the infinite sequence of B-atoms which in all positions, except the i_1-th have a 1; in position i_1, the first member of the sequence has r, the second has $r+1$, the third has r + 2, etc. Let n be arbitrarily large integer. Take the input I to be the first $n \cdot r$ members of the sequence. Then, the instantiation of the nonrecursive rule of each restricted version adds n ground-atoms to the output. Note also that an instantiation of the recursive rule can add some atoms to the output, only if the body of the rule does not contain any extensional-predicate atoms. In this case, for each new ground-atom obtained by instantiation of the recursive rule in one restricted version, there is a new atom obtained in any other restricted version. Therefore, for the input I, the same-size output is generated by each restricted version. Thus, by completeness and lack-of-duplication, the ratio $|O(P, I)|/\max_i |O(P_i, I)|$ equals exactly r. □

Theorem 1 holds even if the sirup is allowed to have constants in nonpivot positions.

4.2. Linear Programs

In this subsection, we discuss linear programs. A program is *linear* if the body of each rule contains at most one intentional predicate. Let A and C be sets of atoms without constants. We say that A *subsumes* C if there is a subset C of C, and a substitution, s, such that C is obtained by applying s to A. Note that if A does not subsume C, then for a one-to-one instantiation of C, there is no instantiation of A, such that the instantiated A is a subset of the instantiated C. A rule of a program P is an *exit* rule if its body consists of extensional predicates only. An exit rule, r_e, is *distinct*, if there is no other rule r of P for which the following condition is satisfied: the set of extensional-predicate atoms in the body of r subsumes the set of atoms in the body of r_e. In other words, r_e is not distinct if its body is subsumed by the set of extensional-predicate atoms in the body of another rule. Note the exit rule of the *csl* program of Example 1 is distinct. A linear program is *distinct* if it has a distinct exit rule.

Theorem 2. If P is a distinct linear program, then there is a load-sharing scheme of any size, for evaluating P. The potential-speedup equals the size of the scheme.

PROOF. Assume without loss of generality that the first variable of the atom in the head of each exit rule is x. To obtain a scheme of size r, let restricted version P_j of P be obtained by adding the predicate $j = x \bmod r$ to each exit rule, for $j = 0, \ldots, r-1$ (all the other rules stay the same).

To show completeness, assume that for some input, I, an atom, say a, is in the output of P. Consider a derivation tree, t, for a. It is easy to see that exactly one instantiated

exit rule is represented in the tree t. The reason for this is the following. Since P is linear, each node has at most one internal (nonleaf) son, implying that there exists a path, p, from the root to an internal node, which goes through all the internal nodes. If there are two instantiated exit rules, then consider the internal nodes, d and e, corresponding to the heads of these rules. All the sons of d are leaves, and so are all the sons of e, and, consequently, none of these two internal nodes is an ancestor of the other. This contradicts the existence of p. Therefore, consider the single instantiated exit rule in t. In the instantiation, x is substituted by some constant, say n. Let $l = n \bmod r$. Then t is a derivation tree for a in P_l. Thus, a is in at least one $O(P_i, I)$. (It may be in more then one output if a can be obtained by a derivation tree with a different instantiation of an exit rule.)

Now we shall show a potential-speedup of r, and thus nontriviality. Specifically, we shall describe how to obtain an arbitrarily large input, I, of P, for which the output is arbitrarily large, such that each one of the r restricted versions outputs the same number of tuples; furthermore, the outputs are pairwise disjoint. Thus, by completeness, the ratio $|O(P, I)|/\max_i |O(P_i, I)|$ equals exactly r.

Denote the distinct exit rule of P by r_e. For an arbitrary integer, n, the input I is the union of $n \cdot r$ sets of facts. Each set S_{ij}, $1 \le i \le n$, $1 \le j \le r$, consists of the body of some one-to-one instantiation, I_{ij}, of r_e. We require that in each instantiation I_{ij}, the variable x is mapped to a constant, c, such that $c \bmod r = j$. Additionally, we require that the set of $n \cdot r$ instantiations is one-to-one, namely each constant is used (i.e., mapped-to) by at most one instantiation. (For example, if instantiation I_{23} maps some variable, y, into 2001, then no other instantiation maps a variable into 2001.) There clearly exists such a set of instantiations.

Having defined I, note that, because of subsumption, for each i, in $O(P_i, I)$ there are only facts whose derivation tree represents instantiations of r_e. Since x is a distinguished variable of r_e, $O(P_i, I)$ and $O(P_j, I)$ must be disjoint for each pair of restricted versions, P_i and P_j.

Finally, we have to show that all the output-sets are of equal cardinality. For this purpose we will define a one-to-one, onto, mapping, f, from $O(P_i, I)$ to $O(P_j, I)$ for each pair of restricted versions, P_i and P_j. Let us number the atoms in the body of r_e by $0, 1, \ldots, m-1$. Then each input fact can be labeled. Fact I_{pqk} is obtained from the k'th atom by instantiation I_{pq}. Since the whose set of instantiations is one-to-one, there cannot be a fact with two different labels. A fact in $O(P_i, I)$ having a derivation tree, t, is mapped by f into the fact in $O(P_j, I)$ having a derivation tree, t', that is obtained from t as follows. Replace each input fact labeled I_{hil} by the input fact labeled I_{hjl}, and replace each input fact labeled I_{hjl} by the input fact labeled I_{hil}. Clearly, f is a bijection. □

If the program P of Theorem 2 is not distinct, then the proof given does not work. For example, consider the linear program

$r1$: $S(x, y) :- B(x, z), B(y, w), S(v, y)$

$r2$: $S(x, y) :- B(x, z), B(y, w)$

If *odd* x is added to the nonrecursive rule in restricted version P_1, and *even*(x) in restricted version P_0, then nontriviality is not satisfied. For a proof of this fact, assume that for some input, I, the ground atom $S(o, a)$ is in $O(P_1, I)$ but not in $O(P_0, I)$, and the ground atom $S(e, b)$ is in $O(P_0, I)$ but not in $O(P_1, I)$. Therefore, the atoms

$B(o, c)$, $B(e, d)$, $B(a, g)$, $B(b, f)$, for some constants c, d, f, g are in the input I. By instantiation of $r2$, the atom $S(o, b)$ is in $O(P_1, I)$. Then,

$$S(e, b) :- B(e, d), B(b, f), S(o, b)$$

is an instantiation of $r1$ which derives $S(e, b)$ in P_1. This contradicts our assumption that $S(e, b)$ is not in $O(P_1, I)$. Note that we have not shown that the above program does not have a load sharing scheme, just that the nontriviality proof of Theorem 2 does not apply to it. However, the completeness part of the proof works for any linear program. Note also that the proof of Theorem 2 holds even if we allow constants in the distinct linear program (but not in the exit rules).

Let us conclude this subsection with a practical example (taken from [25]) of a distinct linear program. Suppose that there is an extensional relation, PERFECTOR(x, y), of people, x, and products, y, such that y is perfect for x. Similarly, there is an extensional relation, IDOL(x, y), of people, x, and their idols, y, and an extensional relation, CHEAPER(x, y), of pairs of products, such that x is cheaper than y. Suppose that a person buys a product if it is perfect for her/him, or if their idol buys its, or if it is cheaper than another product that the person buys. The following program defines recursively the predicate BUYS(x, y) of people, x, and the products they buy, y.

$$BUYS(x, y) :- IDOL(x, w), BUYS(w, y)$$

$$BUYS(x, y) :- BUYS(x, z), CHEAPER(y, z)$$

$$BUYS(x, y) :- PERFECTOR(x, y)$$

4.3. *Weakly Regular Programs*

A simple chain program is a recursive sirup in which (a) all the predicates are binary, (b) the argument positions in the left hand side of the recursive rule have distinct variables, and these variable appear in the first argument position of the first atom in the body, and in the last argument position of the last atom, respectively, (c) all the argument positions in the body of the recursive rule have distinct variables, except that the first argument position of the second atom has the same variable as the last argument position of the first atom, the first argument position of the third atom has the same variable as the last argument position of the second atom, etc. For example, the following is a simple chain program

$$S(x, y) :- A(x, z_1), S(z_1, z_2), S(z_2, z_3), C(z_3, z_4), D(z_4, y)$$

$$S(x, y) :- B(x, y).$$

(A, B, C, D are extensional predicates).

Two sirups are *equivalent* if they produce the same output, for any given input (in the literature equivalence of two programs is defined with respect to an intentional predicate, e.g., [31], but since a sirup has a single intentional predicate the definitions coincide). A simple chain program is *regular* if the recursive rule has exactly one occurrence of S in the body, and it is leftmost or rightmost. A simple chain program is *weakly regular* if the leftmost (or rightmost) predicate symbol in the body of the recursive rule is S, and by replacing all other S predicate-symbols in the body of the

recursive rule, by B, an equivalent program is obtained. For example the program P^0,

$$S(x, y) :- S(x, z_1), A(z_1, z_2), S(z_2, y)$$
$$S(x, y) :- B(x, y)$$

is weakly regular since it is equivalent to the following program (see [36] for the equivalence proof):

$$S(x, y) :- S(x, z_1), A(z_1, z_2)B(z_2, y)$$
$$S(x, y) :- B(x, y)$$

Similarly, any sirup, denote it P^1, of the form

$$S(x, y) :- S(x, z_1), S(z_1, z_2), \ldots, S(z_n, y)$$
$$S(x, y) :- B(x, y)$$

is weakly regular, since it is equivalent to the program

$$S(x, y) :- S(x, z_1), B(z_1, z_2), \ldots, B(z_n, y)$$
$$S(x, y) :- B(x, y)$$

In particular, note that the nonlinear transitive closure,

$$S(x, y) :- S(x, z), S(z, y)$$
$$S(x, y) :- B(x, y)$$

is a weakly regular simple chain program.

Although a weakly regular chain program can be rewritten as a regular program, which is decomposable, this may not be desirable for performance reasons. In [42] it has been shown that among the simple chain programs, only the regular ones are decomposable. In other words, programs such as P^0 and P^1 are not decomposable. Here we show that the class of weakly regular simple chain programs are sharable.

Theorem 3. A weakly regular simple chain program has a load-sharing scheme of any size. The potential-speedup is the size of the scheme.

PROOF. Assume that x is the leftmost variable of the head of the recursive rule. To obtain a load-sharing scheme of size r, create a restricted version P_i, by adding to the recursive rule the predicate $i = x \bmod r$, for $i = 0, \ldots, r-1$.

Completeness: Assume that for some input I, the ground atom $S(a, b)$ is in S. Denote by P' the regular program obtained from P by replacing all S predicate symbols, except the leftmost one, by B. Since P and P' are equivalent, $S(a, b)$ is in the output of P' for the input I. Consider a derivation tree T for $S(a, b)$ in P'. Observe that every occurrence of an S-atom in T is of the type $S(a, n)$ for some constant n. We replace in T each $B(c, d)$ node (except the single one whose father is $S(c, d)$, representing the instantiated exit rule) by the subtree $S(c, d) \rightarrow B(c, d)$, representing an instantiation of the exit rule. The resulting tree, T, is a derivation tree for $S(a, b)$ in P. Assume that $a \bmod r = i$. Then T is a derivation tree for $S(a, b)$ in P_i, for the following reason. By construction, each S-atom in T is either obtained by an instantiation of the recursive rule that replaces x by a (thus satisfies the evaluable predicate of P_i), or is obtained by an instantiation of the exit rule, that is unrestricted by an evaluable predicate.

Nontriviality and speedup: Consider the input I consisting of the following tuples, in each extensional predicate relation of P.

$$\{(i, i+1) \mid i = 1, \ldots, N-1\} \cup \{(i, i) \mid i = 1, \ldots, N\}$$

Then it can be shown that the output of P consists of all the tuples (i, j) such that $i \leq j$ (easier to see by considering P' rather than P). Therefore,

$$|O(P, I)| = \frac{N(N+1)}{2}, \quad \text{and} \quad O(P, I)/\max_i O(P_i, I) \to_{N \to \infty} r \qquad \square$$

REMARK 1. The above theorem is proven by adding to the *recursive* rule in restricted version P_i, the predicate $x \bmod r = i$. In this respect it is different than previous proofs, where the predicate was added to the nonrecursive rule.

5. NONSHARABLE PROGRAMS

In this section, we demonstrate that not every program has a load-sharing scheme. Specifically, we provide a necessary condition for a sirup to have a load-sharing scheme. It turns out that some famous sirups do not satisfy the condition. An example is the first P-complete problem, *path-systems* [12]. The input is a set of triples, the hyperedges, and an initial set of "marked" nodes. The problem is to mark all additional nodes, according to the following rule. If there is a hyperedge of which two nodes are marked then the third node is marked as well. The sirup P, for the problem is the following:

$$S(x) :- S(y), S(z), H(x, y, z)$$
$$S(x) :- B(x)$$

Intuitively, that reason that path systems cannot have a load sharing scheme is as follows. Assume that a load sharing scheme, $\{P_1, P_2\}$ exists, and for some input, I, restricted version P_1 produces $S(b)$ but does not produce S(a), and restricted version P_2 produces $S(a)$ but does not produce S(b). Then if we add to I the fact $H(n, a, b)$, for a new constant n, then $S(n)$ cannot be produced by P_1 nor by P_1, although it is in the output of P. This idea is formalized in Theorem 4, and extended to a class of programs and to an arbitrary number of restricted versions.

Another example of a sirup without a load-sharing scheme, is a variant of path systems called the *blue blooded frenchman* [13]:

$$BBF(x) :- BBF(m), BBF(f), MOTHER(x, m), FATHER(x, f)$$
$$BBF(x) :- FRENCH(x)$$

Some other sirups that have not been defined previously, as far as we known, and do not have a load-sharing scheme, are the following (the exit rule is obvious, thus its specification is omitted):

$$S(x, u) :- H_1(x, y, u), H_2(x, z, w), S(y, u), S(z, w)$$
$$S(x, u) :- H(x, y, z, u, w), S(y, u), S(z, w)$$
$$S(x) :- H_0(x, w), H_1(w, y), H_2(w, z), S(y), S(z)$$
$$S(x, w, y) :- UP(x, t, u), S(t, u, v), FLAT(v, w, z),$$
$$\quad S(z, r, s), DOWN(r, s, x)$$

$$S(x, y) :- H_0(x, w, z), H_1(u, v, y), S(w, z), S(u, v)$$

$$S(x, y) :- H_0(w_0, w, z), H_1(u, v, u_0), H_2(x, w_0), H_3(u_0, y), S(w, z), S(u, v)$$

What do the above sirups have in common? This is what the next definitions establish.

Given a sirup P, denote by $A(P)$ the set of atoms in the body of the recursive rule, and by $V(P)$ the set of variables in $A(P)$. Let $R(P) = \{x \mid x$ is in $V(P)$, and x appears in some S-atom of $A(P)\}$. Let the *extensional graph* of P, denoted $G(P)$, be an undirected graph defined as follows. Its set of nodes is $V(P) - R(P)$, in other words, variables which do not appear in any S-atom in the body of the recursive rule. For two distinct nodes of $G(P)$, x and y, the edge $x - y$ is in the graph if and only if there is an extensional-predicate atom, A, in the body of the recursive rule such that x and y are variables of A. The sirup P is called *propagating* if the following requirements are satisfied.

1. Except for the S-atoms, there are no two atoms of $A(P)$ that have the same predicate symbol.
2. There are at least two S-atoms in $A(P)$, and the S-atoms in $A(P)$ have pairwise disjoint variables, and none of them has repeated variables.
3. Each extensional predicate atom in $A(P)$ has a variable that is not in $R(P)$, and each variable in $R(P)$ appears in some extensional predicate atom.
4. The graph $G(P)$ has a distinguished variable in each one of its connected components.

It is easy to verify that *path systems* and the other sirups that have been discussed in this section are propagating.

Theorem 4. A propagating sirup is not sharable.

PROOF. Let P be a propagating sirup, and r be its recursive rule. Assume that $\{P_1, \ldots, P_q\}$ is a load-sharing scheme for P. Denote by S_j the output of P_j. By nontriviality, there is an input I, for which each S_i is a proper subset of S. Assume that there are m atoms in the set $\cup_{i=1}^{r}(S - S_i)$, and denote them $\{S(\bar{c}_1), \ldots, S(\bar{c}_m)\}$. Next we show how to construct another input, I', for which we shall later show that completeness cannot be satisfied.

The input I', is obtained algorithmically as follows. First I' is initialized to I, then the sets of atoms $H_1, \ldots, H_{m-1}$ are added. Next we explain how to obtain H_1, and then how to obtain H_{i+1} from H_i, for $i = 2, \ldots, m-2$.

Construction of H_1: The set H_1 is constructed in such a way that $S(\bar{c}_1)$ and $S(\bar{c}_2)$ derive a new ground atom, $S(\bar{d}_1)$. It consists of the extensional predicate ground atoms in the following instantiation, ρ_1, of r. In ρ_1 the variables of some S-atom, S', in the body of r are substituted to obtain $S(\bar{c}_1)$, and the variables of all other S-atoms in the body are substituted such that all these atoms instantiated to $S(\bar{c}_2)$. By requirement 2 in the definition of a propagating sirup, such a substitution is possible, regardless of $\bar{c}_1$ and $\bar{c}_2$. Denote by $z_1, \ldots, z_t$ the uninstantiated, as of yet, variables in the body or r. The instantiation ρ_1 is complete by instantiating z_j to e_j^1, for $j = 1, \ldots, t$, such that: edge e_j^1 is different than every other e_k^1, and each e_j^1 is not in I' constructed thus far. For the instantiation ρ_1 denote the atom at the head of r by $S(\bar{d}_1)$.

Construction of H_i *for* $2 \le i \le m-1$: H_i is constructed such that $S(\bar{c}_{i+1})$ and $S(\bar{d}_{i-1})$ derive a new ground atom, $s(\bar{d}_i)$. The construction is similar to the construc-

tion of H_1, except that $S(\bar{c}_{i+1})$ and $S(\bar{d}_{i-1})$ are used instead of $S(\bar{c}_1)$ and $S(\bar{c}_2)$, respectively. Also, the instantiation is denoted by ρ_i, and is completed by instantiating each z_j to e_j^i for $j = 1, \ldots, t$. The e_j^i's are distinct constants, which do not exist in I' constructed so far.

This completes the construction of I'. The atoms in the set $D = \{S(\bar{d}_1), \ldots, S(\bar{d}_{m-1})\}$ are in the output S of P, given the input I'. The reason for this is that $S(\bar{c}_1), \ldots, S(\bar{c}_m)$ are in the output of P given I, and I has been extended to I', such that from $S(\bar{c}_1), \ldots, S(\bar{c}_m)$ the set D can be derived. We shall show that the load sharing scheme cannot satisfy the completeness condition for I'. Particularly, we shall show that the whole set D cannot be derived. The heart of the proof is the following lemma.

Lemma 1. For the input I', the atom $S(\bar{d}_1)$ is in some output S_j, only if, for the input I, $S(\bar{c}_1)$ and $S(\bar{c}_2)$ are both in the output S_j.

PROOF. First we will prove that $S(\bar{c}_1)$ and $S(\bar{c}_2)$ are in the output S_j, if P_j is given the input I', and then we will prove that they are in S_j, if P_j is given the input I. We shall refer to the constants which are in the atoms of I' but are not in the atoms of I (i.e., the e_j^i's) as '*new constants*'.

Assume that $S(\bar{d}_1)$ is in S_j, for the input I'. By requirement 4 in the "propagating" definition, the graph $G(P)$ has at least one distinguished variable which does not appears in any S-atom. Such a variable, by construction of H_1, is instantiated to a new constant. Therefore, one of the constants in $\bar{d}_1$ is new. Note that in I' there are no new constants in the B-atoms, since the B-atoms are the same in I and I'. Therefore, $S(\bar{d}_1)$ is obtained in S_j by an instantiation of the recursive rule.

In the proof of this lemma, we shall use three claims.

Claim 1. Let η be an instantiation of r, such that the instantiated rule is represented in some derivation tree, given the input I'. Suppose that $S(\bar{d}_k)$ is in the head of the instantiated rule. Then if ρ_k instantiates a variable of $R(P)$, say y, to the constant c, then η also instantiates y to c.

PROOF. The variable y is not a node in $G(P)$, by definition of the extensional graph. However, by requirement 3 in the "propagating" definition, there is a node of $G(P)$, say variable x, and there is an extensional predicate atom, A, such that x and y are both arguments of A. By requirement 4, there is a path in $G(P)$ from x to some distinguished variable, say z. Denote the path p: $z = z_{i_1}, \ldots, z_{i_q} = x$. Note that z_{i_1} is distinguished, and both instantiations of r, ρ_k and η, produce the head $S(\bar{d}_k)$. Therefore η, instantiates z_{i_1} to $e_{i_1}^k$. Since there is an edge in $G(P)$ between z_{i_1} and z_{i_2} there is in r an extensional predicate atom, $C(\ldots z_{i_1}, \ldots, z_{i_2} \ldots)$. Based on the way H_i was defined and based on requirement 1, it is easy to see that in I' there is a unique C-ground atom which has the constant $e_{i_1}^k$ in the position corresponding to z_{i_1}. Furthermore, in that C-ground-atom, in the position corresponding to z_{i_2}, appears the constant $e_{i_2}^k$. Therefore, η, as ρ_k, instantiates z_{i_2} to $e_{i_2}^k$. The above argument can be continued inductively on the length of the path p, to show that x is instantiated to $e_{i_q}^k$ by η. The variables x and y are both arguments of the extensional predicate atom A, and again by requirement 1 it is easy to see that ρ_k and η instantiate y to the same constant. $\square_{Claim\ 1}$.

Therefore, if for the input I', the atom $S(\bar{d}_1)$ is in the output S_j, then $S(\bar{c}_1)$ and $S(\bar{c}_2)$ must be in the output S_j. To complete the proof of Lemma 1, left to show is that $S(\bar{c}_1)$ and $S(\bar{c}_2)$ are in the output S_j for the input I.

Claim 2. Let η be an instantiation of r, such that the instantiated rule is represented in some derivation tree, given the input I'. Suppose that the extensional predicate atom $C(\bar{f})$ is in the body of the instantiated rule. Assume that $C(\bar{f})$ has a new constant, i.e., $C(\bar{f})$ belongs to some H_i. Then the S-ground-atom in the head has a new constant.

PROOF. By requirement 3 in "propagating" definition, the C-atom in the body of r has a variable, w, which is not in $R(P)$. By construction of H_i, the variable w is instantiated by ρ_i to some new constant, e_j^i. Denote by p the path from w to a distinguished variable, z, in $G(P)$ (by requirement 4 there is such). Consider also the variable v, which succeeds w on the path p. By definition of $G(P)$, there is an extensional predicate atom, $D(\ldots w, \ldots, v \ldots)$, and v does not belong to $R(P)$. By requirement 1, in I' there is a unique D-ground-atom which has the e_j^i in the position corresponding to w. Furthermore, that D-ground has a new symbol in the position corresponding to v. Therefore, η must substitute a new constant for v. This argument can be continued inductively on the length of the path p, to show that the distinguished variable z is substituted for a new constant by η. $\square_{Claim\ 2}$.

Claim 3. Let η be an instantiation of r, such that the instantiated rule is represented in some derivation tree, given the input I'. Suppose that in the body of the instantiated rule there is an S-atom with a new constant. Then the S-ground-atom in the head has a new constant.

PROOF. The substitution η must replace some variable of $R(P)$ by a new constant. Assume that x is the variable of the S-atom, which is replaced by a new constant. According to requirement 3 in "propagating" definition, the variable x also appears in an extensional predicated atom in r. By Claim 2, the proof is completed. $\square_{Claim\ 3}$.

To complete the proof of Lemma 1, consider a derivation tree, T, for $S(\bar{c}_1)$, given the input I'. By Claims 2 and 3, it is easy to see that if any new constant appears in a derivation tree for $S(c_1)$, then it is "propagated" up to $S(\bar{c}_1)$. Since in $\bar{c}_1$ there do not exist any new constants, than a new constant does not appear in the derivation tree T. By the construction of the H_i's and by requirement 3, it can be seen that every atom of $I' - I$ has a new constant. Therefore, T is a derivation tree for $S(\bar{c}_1)$ given the input I. Similarly it can be shown that $S(\bar{c}_2)$ can be derived given the input I. $\square_{Lemma\ 1}$.

Now we shall show by induction on n, that if $S(\bar{d}_n)$ is in the output S_j, given the input I', then $S(\bar{c}_1), \ldots, S(\bar{c}_{n+1})$ are in the output S_j, given I. If so, then theorem 3 follows, because completeness cannot hold for the input I'. The reason is that there is no S_j that contains the whole set $\{S(\bar{c}_1), \ldots, S(\bar{c}_m)\}$ for the input I, and, therefore, $S(\bar{d}_{m-1})$ cannot be derived, given I'.

Lemma 1 provides the basis for the induction of n. The inductive step is very similar to the proof of Lemma 1, and argues as follows. If $S(\bar{d}_n)$ is in S_j for the input I', then by Claim 1, both $S(\bar{d}_{n-1})$ and $S(\bar{c}_{n+1})$ are in S_j. By claims 2 and 3, $S(\bar{c}_{n+1})$ is in S_j for the input I. By the inductive assumption, $S(\bar{c}_1), \ldots, S(\bar{c}_n)$ are in S_j for the input I. $\square_{Theorem\ 3}$.

Let us observe that the first two requirements of the "propagating" definition are not, by themselves, sufficient for nonexistence of a load-sharing scheme. They are satisfied in the following sirup, although, as shown in subsection 4.3, it does have a load-sharing scheme.

$$S(x, y) :- S(x, z_1), A(z_1, z_2)S(z_2, y)$$

$$S(x, y) :- B(x, y).$$

Next we observe that there exist also programs with constants that do not have a load-sharing scheme. For example, the following program, which is a simple variation of path systems, does not do so (the proof is as in Theorem 4):

$$S(5, x) :- S(5, y), S(5, z), H(x, y, z)$$

$$S(5, x) :- B(x)$$

6. EXTENSION TO PARALLELIZATION BY BOTTOM-UP ALGORITHMS

The purpose of this section is to extend the previous results for bottom-up algorithms for the evaluation of logic programs. The algorithms do not necessarily evaluate a restricted version of the original program. In other words, consider the nonsharable program path systems. Is it possible to purely parallelize the work of evaluating it? Although independent evaluation of restricted versions does not work, it is conceivable that some other purely parallel method does work. For example, maybe if we allow a larger class of evaluate predicates (e.g., log, sine) in a restricted version, then path systems would be sharable. To determine whether path-systems and other nonsharable programs can be purely parallelized, we define an algorithmic load sharing scheme of a program as a set of algorithms, each working independently of the others. Furthermore, we distinguish between bottom-up algorithmic load-sharing schemes, and other ones. When considering sharability in this broader context, namely, algorithmic-sharability, we show that path systems is top-down algorithmically sharable, but not bottom-up algorithmically sharable. In other words, path-systems cannot be evaluated in parallel by multiple bottom-up algorithms that do not need to communicate, such that each algorithm computes less.

Let P be a Datalog program. For an input I to P, a *partial computation of* P, denoted $c(I)$, is a sequence of ground atoms. The predicate symbols in $c(I)$ are taken from P, and the sequence $c(I)$ satisfies the following two conditions. First, each extensional predicate atom in $c(I)$ must be in I. Second, every intentional predicate atom, a, in $c(I)$, is in $O(P, I)$, and is preceded by all atoms of some derivation tree of a. The sequence $c(I)$ corresponds to the order in which the output of P is evaluated, and is called a "partial" computation, since not all ground atoms of $O(P, I)$ have to be in $c(I)$.

An *algorithm*, A, *for partial computation of* P is a function that maps each input, I, into a partial computation of P, denoted $A(I)$. The algorithm A does not communicate with other algorithms for producing $A(I)$, since it is required to produce all the atoms of some derivation tree of a, before being able to produce a.

Let $D = \{A_1, \ldots, A_r\}$ be a set of algorithms for partial computation of P. The set D is *complete* for an input, I, if for each ground-atom, $a \in O(P, I)$, there is an

algorithm $A_j \in D$, such that $a \in A_j(I)$. Algorithm $A_j \in D$ *computes less* for some input I, there is some atom $b \in O(P, I)$, which is not in $A_j(I)$.

Input I is a *time-saving input* for the set of algorithms, D if each algorithm computes less for I. The set D is an *algorithmic load-sharing scheme* for P, if it is complete for every input, and has at least one time-saving input. It is easy to see that this generalizes the definition of load sharing by restricted versions (given in section 3), based on the following proposition.

Proposition 2. Assume that a program P has a load sharing scheme consisting of the set $D = \{P_1, \ldots, P_r\}$ *of restricted versions. Then P has an algorithmic load sharing scheme,* $D' = \{A_1, \ldots, A_r\}$.

PROOF. Define algorithm A_i to work as follows. For each input I it first outputs all the atoms of I in some arbitrary order, then it evaluates P_i, outputting the intentional fact as they are computed. It is easy to see that $D' = \{A_1, \ldots, A_r\}$ is a load-sharing scheme for the program P, as follows. Completeness is obviously satisfied, since D is a load-sharing scheme. Additionally, each algorithm A_j computes less for any input that satisfies the nontriviality condition for D. Since there is at least one such input for D, the set D' has a time-saving input. □

A program that has an algorithmic load-sharing scheme is called *algorithmically sharable*. Now observe that a program may be algorithmically sharable, although it is not sharable. For example, consider the following scheme for evaluating path systems:

$S(x) :- B(x)$

$S(x) :- H(x, y, z), S(y), S(z)$

Suppose that two processors work top-down, using, say, the Prolog method. Processor one assumes responsibility for producing the $S(x)$ tuples in which x is a constant in the database, and is odd, and processor two assumes responsibility for producing the $S(x)$ tuples in which x is in the database, and is even. Each processor works independently, and attempts to "prove," one by one, the facts in its responsibility domain. For proof, the rules are considered in a top-to-bottom order, and the atoms in the body of each rule are considered in a left-to-right order. If a processor discovers in the course of proving a fact in its responsibility domain, that it has to prove a fact, f, then it does so regardless of whether or not f is in its responsibility domain. So, for example, if in the course of attempting to prove $S(3)$ processor one has the subgoal $S(2)$, then it proves it, even though $S(2)$ is the other processor's responsibility. It is easy to see that this scheme is complete and has a time-saving input (e.g., an input in which the relation H does not contain any triple having both, an odd and an even constant).

Now let $D = \{A_1, \ldots, A_r\}$ be a set of algorithms for partial computation of P, and let I_0 be an input to P. We say that A_j *bottom-up-evaluates* I_0, if for each fact b that is not in $A_j(I_0)$, and for each set of input atoms, Z, the following requirement is satisfied:

(noncontribution) If for the input $I_0 \cup Z$ there is no derivation tree of b, which contains an atom of Z, then $b \notin A_j(I_0 \cup Z)$.

The noncontribution requirement simply says that if the atom b is not in $A_j(I_0)$, and if the set Z does not "contribute" to the derivation of b (i.e., there is no derivation tree which contains an atom Z), then b is also not in $A_j(I_0 \cup Z)$. The naive and semi-naive evaluation methods clearly satisfy this requirement.

Note that at top-down, a la Prolog, type of algorithm does not satisfy the noncontribution requirement, since Z may "trigger" the derivation of an atom without actually contributing to this derivation. For example, suppose that I is an input to path systems, in which the relation H does not contain any triple having both an odd and an even constant. Furthermore, suppose that $S(2)$ is in the output. Then processor 1 will derive $S(2)$ for $I \cup \{H(1,2,3)\}$ although $\{H(1,2,3)\}$ is not in the derivative tree of $S(2)$.

An algorithmic load sharing scheme, D, is *bottom-up* if every input is bottom-up evaluated by each algorithm in D. A program is *bottom-up* algorithmically-sharable if it has a bottom-up algorithmic load sharing scheme.

Theorem 5. The sirup path systems is not bottom-up algorithmically-sharable.

PROOF. Denote the sirup path systems by P. Assume by way of contradiction that $D = \{A_1, \ldots, A_r\}$ is a bottom-up algorithmic load sharing scheme for P. Consider a time-saving input, I_0. Observe that S is the only intentional predicate in P. Thus for each A_j, since it computes less for I_0, there is at least one $S(c_j)$ in $O(P, I_0)$, which is not in $A_j(I_0)$. In other words, there is not any $A_j(I_0)$ that contains all $S(c_i)$'s. Let

$$C = \{S(\bar{c}) \mid S(\bar{c}) \in O(P, I_0), \text{ and for some } A_j,\ S(\bar{c}) \notin A_j(I_0)\}.$$

Denote $C = \{S(\bar{c}_1), \ldots, S(\bar{c}_m)\}$. Let $d_1, \ldots, d_{m-1}$ be $m-1$ constants, none of which is in I_0. Denote $I_1 = I_0 \cup H$ where

$$H = \{H(d_1, c_1, c_2), H(d_2, d_1, c_3), H(d_3, d_2, c_4), \ldots, H(d_{m-1}, d_{m-2}, c_m)\}.$$

Obviously, each $S(d_i)$ for $i = 1, \ldots m-1$ is in $O(P, I_1)$. Observe that there cannot be a derivation tree for $S(c_i)$ in I_1, which has as a node a ground atom of H, say H_1. If there is such, then the father of H_1 in the tree is some $S(d_j)$. Then, $S(d_j)$ must have a brother in the derivation tree which is another member of H, say H_2, and therefore the father of $S(d_j)$ in the derivation tree must be $S(d_{j+1})$. Proceeding inductively it can be shown that the root of the derivation tree for $S(c_i)$ is some $S(d_k)$, obviously a contradiction. Therefore, since I_0 is bottom-up evaluated by each A_j, there is not any $A_j(I_1)$ which contains the whole set C.

Now, we show by induction on i, that if $S(d_i)$ is in $A_j(I_1)$ for some A_j, then $S(c_1), \ldots, S(c_{i+1})$ are all in $A_j(I_1)$. For the basis, note that each derivation tree for $S(d_1)$ must have $S(c_1)$, $S(c_2)$, and $H(d_1, c_1, c_2)$, as the sons of the root. Therefore, since $A_j(I_1)$ is a partial computation of P, there sons must precede $S(d_1)$ in $A_j(I_1)$. For the inductive step, note that each derivation tree for $S(d_i)$ must have $S(d_{i-1})$, $S(c_{i+1})$, and $H(d_i, d_{i-1}, c_{i+1})$, as the sons of the root. Therefore, if $S(d_i)$ is in $A_j(I_1)$, then $S(d_{i-1})$, $S(c_{i+1})$ are both also in $A_j(I_1)$. Since $S(d_{i-1})$ is in $A_j(I_1)$, by inductive assumption, $S(c_1), \ldots, S(c_i)$ are also there. This completes the inductive proof. Then completeness of the load-sharing scheme for I_1 is violated for the following reason. The atom $S(d_{m-1})$ cannot be in any $A_j(I_1)$, because, as established, there is not any $A_j(I_1)$, which contains all $S(c_i)$'s. □

Similarly, it can be shown that any propagating sirup is not bottom-up algorithmically sharable.

7. CONCLUSION AND FUTURE WORK

In this paper, we examined pure parallelization of data intensive Datalog programs. The method proposed is independent parallelization by restricted versions, where each restricted version generates part of the output. It has been shown that some important classes of programs (for example, most linear ones) can be parallelized this way, i.e., are sharable, whereas other classes cannot. Evaluation of the sharable programs discussed in this paper can be spread among an arbitrary number of processors. Furthermore, the potential speedup of such a parallelization is optimal, i.e., equal to the number of processors participating in the parallelization scheme. We have shown that the class of propagating sirups are not sharable, and, moreover, they cannot be evaluated in parallel by an set of parallel algorithms that work bottom-up, and do not incur a communication overhead.

Load sharing is applicable in conjunction with rewriting methods that propagate constants from the query, such as Magic Sets [4, 6]. For example, consider the following program called "same generation":

$$SG(x, x) :- H(x)$$

$$SG(x, y) :- PARENT(x, xp), PARENT(y, yp), SG(xp, yp)$$

The source-to-source transformed program produced by Magic Sets in response to the query, $SG(a, ?)$, is the following:

$$MAGIC(xp) :- MAGIC(x), PARENT(x, xp)$$

$$MAGIC(a)$$

$$SG(x, x) :- H(x)$$

$$SG(x, y) :- MAGIC(xp), PARENT(x, xp),$$

$$\quad PARENT(y, yp), SG(xp, yp)$$

By appending the predicate $i = x \bmod r$ to the exit rule, $SG(x, x) :- H(x)$, of the transformed program, we obtain a restricted version of a load-sharing scheme.

It seems that load sharing is also applicable in conjunction with more advanced source-to-source program transformation methods, such as Magic-Templates [27]. In response to a query to the *csl* program, the method may produce, depending on the choice of *sips*, the following program:

$$S(x, y) :- MAGIC(x, y), UP(x, w), S(w, z), DOWN(z, y)$$

$$S(x, y) :- MAGIC(x, y), FLAT(x, y)$$

$$MAGIC(w, z) :- MAGIC(x, y), UP(x, w)$$

$$MAGIC(a, z)$$

In this case, we can obtain a restricted version of a load-sharing scheme by appending the predicate $i = x \bmod r$ to the seed, MAGIC(a, z).

As far as future research is concerned, an obvious direction is to extend the class of programs for which it can be algorithmically determined, by examining the syntax of the program, whether or not the program has a load-sharing scheme. The scope of the investigation should be broadened to include programs with function symbols and negation, such as those written in the language LDL (see [40]). It should also be investigated whether constants in a program increase the possibilities of pure-parallelization, as conjectured in [21], or the opposite occurs. Inspired by the work in [16], we would also like to examine sharability of a program with respect to an input. Maybe for programs that are not sharable, for some inputs that can be very efficiently identified, the benefits of sharability are achievable. We would also like to determine how to minimize communication among the processors when it cannot be avoided. In other words, for parallelizing nonsharable programs communication is necessary. How should this communication be minimized? The study of pure parallelization proves helpful for answering this question, as demonstrated in [11, 17]. Finally, we intend to study the enhancement of load sharing with some interesting parallelization ideas that appeared in the literature [2, 19, 29, 38].

The author thanks Nissim Francez and Oded Shmeli for helpful discussions and comments, and the referees for numerous suggestions and comments that helped improve this paper significantly.

REFERENCES

1. Afrati, F., and Papadimitriou, C. H., The Parallel Complexity of Simple Chain Queries, in: *Proceedings of the 6th ACM Symposium on PODS*, 1987.
2. Agrawal, R., Jagadish, H. V., Direct Algorithms for Computing the Transitive Closure of Database Relations, in: *Proceedings of the 13th VLDB Conference*, 1987.
3. Bancilhon, F., Naive Evaluation of Recursively Defined Relations, in: M. Brodie and J. Mylopoulos (eds.), *On Knowledge Base Management Systems—Integrating Database and AI Systems*, Springer-Verlag, New York, 1985.
4. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., Magic Sets and Other Strange Ways to Implement Logic Programs, in: *Proceedings of the 5th ACM Symposium on PODS*, 1986.
5. Bancilhon, F., and Ramakrishnan, R., An Amateur's Introduction to Recursive Query Processings, in: *Proceedings of the SIGMOD Conference*, 1986.
6. Beeri, C., Ramakrishnan, R., The Power of Magic, in: *Proceedings of the 6th ACM Symposium on PODS*, 1987.
7. J. Bergsten et al., A Parallel Database Accelerator, in: *PARLE '89, Springer-Verlag Lecture Notes in CS Series, no. 365*, Springer-Verlag, New York, 1989.
8. Bitton, D., Boral, H., DeWitt, D. J., and Wilkinson, W. K., Parallel Algorithms for the Execution of Relational Database Operations, *ACM TODS*, 8 (1983).
9. Boral, H., et al., Prototyping Bubba, a Highly Parallel Database Systems, *IEEE Trans. Data Knowledge Engrg.* 2 (1990).
10. Carriero, N., and Gelernter, D., How to Write Parallel Programs: A Guide to the Perplexed, *ACM Comput. Surveys*, 21:323–357 (1989).
11. Cohen, S., and Wolfson, O., Why a Single Parallelization Strategy is not Enough in Knowledge Bases, *8th ACM Symposium on PODS*, 1989.

12. Cook, S. A., An Observation on Time-Storage Trade-off, *J. Comput. System. Sci.* 9:308–316 (1974).
13. Cosmadakis, S. S., and Kanellakis, P. C., Parallel Evaluation of Recursive Rule Queries, in: *Proceedings of the 5th ACM Symposium on PODS*, 1986.
14. DeGroot, D., and Lindstrom, G., eds. Logic Programming—Functions Relations and Equations, Prentice Hall, Englewood Cliffs, NJ, 1986.
15. Dias, D. M., Iyer, B. R., Yu, P. S., On Coupling Many Small Systems for Transaction Processing, Research Report RC11722, IBM T. J. Watson Research Center, Yorktown Heights, NY.
16. Dong, G., On Distributed Processibility of Datalog Queries by Decomposing Databases, in: *Proceedings of the ACM-SIGMOD Conference*, 1989.
17. Ganguly, S., Silberschatz, A., and Tsur, S., A Framework for the Parallel Processing of Queries, Manuscript, Computer Science, Department, University of Texas at Austin, Austin, Texas, 1989.
18. Hilbert, D., Mathematische Problems, *Bull. Am. Math. Soc.* 8:437–479 (1901).
19. Houtsma, M. W., Apers, P. M. G., and Ceri, S., Parallel Computation of Transitive Closure Queries on Fragmented Databases, Technical Report INF-88-56, University of Twente, Twente, Holland, 1988.
20. Kanellakis, P. C., Logic Programming and Parallel Complexity, in: *Proceedings of ICDT '86, International Conference on Database Theory, Springer-Verlag Lecture Notes in CS Series, no. 243*, Springer-Verlag, New York, 1986.
21. Lozinskii, E., Private communication, 1988.
22. Maier, D., and Warren, D. S., Computing with Logic: Introduction to Logic Programming, Benjamin-Cummings Publishing Co., Menlo Park, California, 1987.
23. Marchetti-Spaccamela, A., Pelaggi, A., Sacca, D., Worst Case Complexity Analysis of Methods for Logic Program Implementation, in: *Proceedings of the 6th ACM Symposium on PODS*, 1987.
24. Matijasevic, Y., and Robinson, J., Reduction of an Arbitrary Diophantine Equation to One in 13 Unknowns, *Acta Arith.* 27:521–553 (1975).
25. Naughton, J. F., Compiling Separable Recursions, in: *Proceedings of the ACM-SIGMOD Conference*, 1988.
26. Pasik, A. J., A Methodology for Programming Production Systems and its Implications on Parallelism, Ph.D. Thesis, Columbia University, New York, 1989.
27. Ramakrishnan, R., Magic Templates: A Spellbinding Approach to Logic Programs, in: *Proceedings of the International Conference on Logic Programming*, 1988.
28. Ramakrishnan, R., Parallelism in Logic Programs, Technicaal Report 892, University of Wisconsin, Madison, Wisconsin, Science Department 1989.
29. Raschid, L., Sellis, T., and Lin, C. C., Exploiting Concurrency in a DBMS Implementation of Production Systems, in: *Proceedings of the International Symposium on Databases in Distributed and Parallel Systems*, 1989.
30. Shapiro, E. Y., *Concurrent Prolog, Collected Papers, Vols. 1 and 2*, MIT Press, Cambridge, Massachusetts, 1987.
31. Shmueli, O., Decidability and Expressiveness Aspects of Logic Queries, in: *Proceedings of the 6th ACM Symposium on PODS*, 1987.
32. Stolfo, S. J., Miranker, D. P., and Mills, R., A simple processing scheme to extract and load balance implicit parallelism in the concurrent match of production rules, in: *Proceedings of the AFIPS Symposium on Fifth Generation Computing*, 1985.
33. Ullman, J. D., Implementation of Logical Query Languages for Databases, *ACM TODS* 10:289–321 (1985).
34. Ullman, J. D., Database Theory: Past and Future, in: *Proceedings of the 6th ACM Symposium on PODS*, 1987.
35. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Vol. 2*, Computer Science Press, New York, 1989.

36. Ullman, J. D., and Van Gelder, A., Parallel Complexity of Logic Programs, *Algorithmica*, 3:5-42 (1988).
37. Valduriez, P., and Gardarin, G., Join and Semijoin Algorithms for a Multiprocessor Database Machine, *ACM TODS* 9 (1984).
38. Valduriez, and Khoshafian, S., Parallel evaluation of the Transitive Closure of a Database Relation, *Int. J. Parallel Programming*, 17 Feb. (1988).
39. Van Emden, M. H., and Kowalski, R. A., The Semantics of Predicate Logic as a Programming Language, *JACM*, 23:733-742 (1976).
40. Tsur, S., and Zaniolo, C., LDL: A Logic Based Data Language, in: *Proceedings of the 12th VLDB Conference*, 1986.
41. Wolfson, O., and Ozeri, A., A New Paradigm for Parallel and Distributed Rule-Processing, in: *Proceedings of the ACM-SIGMOD 1990, International Conference on Management of Data*, 1990.
42. Wolfson, O., and Siberschatz, A., Distributed Processing of Logic Programs, in: *Proceedings of the ACM-SIGMOD Conference*, 1988.