PARALLEL BOTTOM-UP EVALUATION OF DATALOG PROGRAMS BY LOAD SHARING

by

Ouri Wolfson

Technical Report #564

June 1989
PARALLEL BOTTOM-UP EVALUATION OF DATALOG PROGRAMS
BY LOAD SHARING

Ouri Wolfson
Computer Science Department
The Technion - Israel Institute of Technology
Haifa 32000, Israel
wolfson@techsel.bitnet

ABSTRACT

We propose a method of parallelizing bottom-up-evaluation of logic 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 logic program, but we syntactically characterize several classes of logic programs to which the method can be applied (e.g., the class of linear single rule programs). We also provide a characterization for a class of programs to which the parallelization method cannot be applied, and show that in general, determining applicability is undecidable. Then, we generalize the concepts beyond the proposed method, and lay the foundations for a theory of logic programs and parallelization. It demonstrates that the logic programs have an interesting hierarchical class-structure, with respect to pure parallelization.

1. This research was supported by Technion V. P. R. Fund - Elron-Elbit Electronic research fund.
1. INTRODUCTION

1.1 Background

In recent years, the emphasis in database research has shifted to knowledge base systems ([U3]). A knowledge base is a database augmented with a set of Horn-clause rules (the logic program). The rules enable inference of information which 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 knowledge base, should be actively inferred. 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 and other relevant work in subsection 1.3, 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, [DIY]). Some problems are amenable to pure parallelization i.e., parallelization which 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 bottom-up-evaluation of logic programs. This is the evaluation strategy usually employed in knowledge base systems, when computing relations defined by logic programs ([U1]). The proposed method is to algorithmically create, for a given logic program, \( P \), rewritten versions of \( P \), and
assign to each processor a different version. Each processor evaluates its version, without communicating with the other processors, and 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. The next example demonstrates our method.

Example 1: Throughout this paper we assume familiarity with a subset of the language PROLOG, called DATA-LOG (see [MW]). 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 \(k\).

\[
S(x,y) :\neg \text{UP}(x,w) \land S(w,z) \land \text{DOWN}(z,y) \\
S(x,y) :\neg \text{FLAT}(x,y)
\]

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

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

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

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

Assume that \(S\) is computed by the naive evaluation method (see [B]). It assigns FLAT to \(S\) and then iteratively adds

\[
i = (x+y) \mod r \quad \text{works as well}
\]

\[
\text{** This works for character-strings as well, since the binary representation can be regarded as a natural number}
\]
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 4-tuples relation ($UP$), with a 5-tuples relation ($S$), with a 4-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 $i = x \mod 2$ 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 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 ([B]) rather than naive.

1.3 Other Relevant Work

The efficient bottom-up-evaluation of intentional database relations, defined by means of recursive logic programs, has recently emerged as a very active area of research ([U2], [BR], [K]). Two main methods of improving
performance have received most of the attention. One is selection propagation, and the other is parallel evaluation.

Selection propagation reduces the number of relevant input-database tuples, by using constants passed as parameters to the database query processor. This usually necessitates a rewriting of the logic program which defines the intentional relation. The best known rewriting algorithm for this purpose is "magic sets" (see [BMSU]).

Parallel evaluation uses multiple cooperating processors, to reduce the overall evaluation time form start to finish. Most efforts in this area have been devoted to characterization of the logic programs which belong to the NC complexity class ([UV], [K], [AP]). If a program is in NC, it means that its intentional relations 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. Unfortunately, this research means very little as far as utilizing a constant number of processors, particularly if they do not share common memory (e.g. a hypercube multiprocessor system* having 1024 nodes).

The bottom-up-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 multiple processors, for the evaluation, is to parallelize relational algebra operators, particularly the join operation (e.g. [BBDW, VG]). For this, even a constant number of processors can be used. However, if so, 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.

Presently, the only other method of parallelizing bottom-up evaluation, without synchronization, is the one introduced in [WS]. 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 [WS] method is applicable only to a very restricted class of logic programs, called decomposable. For example, in the class of simple chain programs ([UV], [AP]) only to the regular ones are decomposable. Therefore, the alg 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

* see [H]
One last comment regarding comparison with relevant literature concerns the work on parallel and concurrent versions of PROLOG (see [S]). There has been a lot of research on the subject, but because of the fundamental difference between bottom-up (or forward chaining) and top-down (or backward chaining) evaluation of logic programs, this research is not applicable to our problem. Specifically, parallelization methods for PROLOG, which takes the top-down approach are not applicable in database query processing, which employs bottom-up. The reason for bottom-up is that database applications are data intensive, and they usually look for all answers to a query.

1.4 Objectives of this Paper

The first main objective of this paper is to determine which programs are sharable, i.e., which programs can be evaluated by the load sharing method described in subsection 1.2. Specifically, we formally define what it means for a program to be sharable, and explore the syntactic characterization of programs which are sharable. 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) which is sharable. In the class of simple chain programs, defined in [UV], we also characterize a subclass which is sharable. Then we provide a necessary syntactic condition for a program to be sharable. Several well known programs do not satisfy this condition (e.g. path systems, introduced in [C]).

The second main objective of this paper is to examine how fundamental are the sharability and decomposability concepts. Maybe the work sharing, or partitioning, cannot be obtained by adding evaluable predicates to the original DATALOG program, but some other similar method will work. Therefore, we extend the definitions to include sharability and decomposability with respect to general algorithms for logic-program evaluation. We determine that the class of sharable programs is properly contained in the class of all DATALOG programs, and, in turn, the class of decomposable programs is properly contained in the class of sharable programs. This classification requires establishing the basis for a new theory of logic programs and parallelization, and proving impossibility results.

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, its potential speedup, and prove undecidability of load sharing. In section 4 we characterize classes of programs which 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 which
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 natural number. 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 conjunction of one or more atoms, denoted \( Q^1, \ldots, Q^k \), 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, \ldots, and \( Q^k \)." A rule or an atom is an entity. If an entity has a constant in each argument position, then it is a ground entity. A substitution applied to an entity, is the replacement of each variable in the entity by another variable, or a constant. It is denoted \( x_1/y_1, x_2/y_2, \ldots, x_n/y_n \) indicating that \( x_i \) is replaced by \( y_i \). A substitution is ground if the replacement of each variable is by a constant. A ground substitution applied to a rule is an instantiation of the rule. When we talk about an instantiation we refer either to the ground rule, or to the substitution; which reference, will be clear from the context.

A DATALOG program, or a program for short, is a finite set of rules whose 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 a relation for each intentional predicate of \( P \). Notice that the input and output are disjoint sets of ground atoms. The output of \( P \) given input \( I \), is obtained by the following procedure, called bottom up evaluation (BUE).

BUE1. Start with an initial database consisting of the ground atoms in \( I \).

BUE2. If there is an instantiation of a rule of \( P \) such that all the ground atoms in the body are

in the database generated so far, and the one in the head is not, then:

add to the database the ground atom in the head of the instantiated rule, and reexecute BUE2.

BUE3. Stop.

This procedure is guaranteed to terminate, and produce a finite output for any given \( P \) and \( I \) (VEK)). The output is
unique, in the sense that any order in which bottom up evaluation adds the atoms to the database will produce the same output.

Let ground atom \( a \) be in the output for \( P \), and let \( s \) be a minimal sequence of iterations of BUE2 for deriving \( a \). To \( s \) corresponds a derivation tree for \( a \); it is a rooted directed tree with ground atoms as nodes, and \( a \) as the root. Node \( b \) has children \( b_1, \ldots, b_k \), if and only if \( b :- b_1, \ldots, b_k \) is an instantiation in \( s \).

For some rule, a variable which 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.

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 \( \{ +, -, \times, \text{modulo}\} \); the same for \( e_2 \). The predicate symbol, \( \theta \), is an arithmetic comparison operator (i.e. \( <, >, \leq, \geq, =, \neq \) ). A rule, say \( ra \), is an adorned copy of some rule \( r \), if \( r \) 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. 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 \( P_1 \) is a adorned copy of program \( P \) if each one of its rules is a adorned copy of some rule of \( P \). Note that \( P_1 \) may have more than one adorned copy of a rule \( r \) of \( P \). To continue the above example, if \( P \) has the rule \( r \), then \( P_1 \) may have the rule \( re \) as well as the rule \( re' \):

\[
S(x,y,z) :- S(w,x,y), A(w,z), x-y=6
\]

Throughout this paper, only adorned copy of a program may have evaluable predicates. The input of a program with evaluable predicates, i.e. a adorned copy, is defined as before. The output is also defined as before, except that BUE2 also verifies that the substitution satisfies* the evaluable predicates in the ground rule; only then the atom in the head is added to the database and BUE2 is reexecuted. In other words, in considering instantiations for a adorned copy of a rule, the procedure bottom up evaluation disregards database atoms which do not satisfy the additional evaluable predicates. Intuitively, this is the reason we are interested in adorned copies. Since they work with

* 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 smaller database, their evaluation is faster.

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. \( Q \) is recursive if \( (Q,Q) \) is in the nonreflexive transitive closure of the "directly derives" relation. 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 program is recursive if it has a recursive predicate. A rule is recursive if the predicate in its head transitively derives some predicate in its body.

3. LOAD SHARING SCHEMES

In this section we define and discuss the concept of a load sharing scheme, and establish that it is, in general, undecidable. Then the notion of the potential speedup of a load sharing scheme is defined. All the load sharing schemes discussed in this paper have the maximum potential speedup.

Assume that \( P \) is a program, and \( P_1, ..., P_r \) are adorned copies of \( P \). The set \( D = \{P_1, ..., P_r\} \) is a load sharing scheme for evaluating \( P \), if the following two conditions hold:

1. For each input \( I \) to the programs \( P, P_1, ..., P_r \), \( \bigcup_i O(P_i,I) \supseteq O(P,I) \) (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 adorned copy 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 memory common to all the processors.

The completeness requirement in the definition, is that no output atoms are 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 adorned copy 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 [BR, 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, at least for the classes of programs discussed in this paper, if a program is sharable, then it has an infinite number of inputs which satisfy the nontriviality condition. Furthermore, the sharable programs have load sharing scheme in which some relation is partitioned among the adorned copies by the modulo predicate. Thus for the "average" input, each adorned copy works with smaller relations.

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 adorned copies \( \{P_1, \ldots, P_r\} \) 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, \( O(P_1, I) \) and \( O(P_r, I) \), are disjoint. In other words, two adorned copies do not duplicate one another's work, by computing the same output atom. Load decomposition was defined in [WS], and we shall come back to it in section 6.

Next we show that load sharing is undecidable. Assume that there is an algorithm which given \( P \), and the set of adorned copies, \( D = \{P_1, \ldots, P_r\} \), 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 [MR], [Hi]. 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

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

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 and 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 adorned copies, \( D = \{P_1, \ldots, P_r\} \), it is undecidable to determine whether \( D \) is a load sharing scheme for evaluating \( P \).
There is a related problem, that of determining for a given program $P$, whether there exists a load sharing scheme for evaluating it. 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 $\epsilon$, there is an input $I$ for which $|O(P, I)| > n$, and $\frac{|O(P, I)|}{\max_i |O(P_i, I)|} \geq M - \epsilon$. Intuitively, the potential speedup is the number to which the ratio $\frac{|O(P, I)|}{\max_i |O(P_i, I)|}$ 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 of evaluating $P$ with one processor, and the maximum load of a processor, when the sharing scheme is used. 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 $i = (x_1 + \ldots + x_a) \mod r$ to some of the rules, where $x_1, \ldots, x_a$ are distinguished variables. For an input which is distributed evenly across a range of natural numbers, this reduces the number of newly generated tuples at each processor.

4. SHARABLE PROGRAMS

4.1 Pivoting Single Rule Programs

A single rule program (see [CK]), or a sirup for short, is a DATALOG program which has a single intentional predicate, denoted $S$ in this paper. The program consists of two rules. A nonrecursive rule,

$$S(x_1, \ldots, x_n) : - B(x_1, \ldots, x_n),$$

where the $x_i$'s are distinct variables; and one other, possibly recursive, rule in which the predicate symbol $B$ does
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 variables 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 or may not 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

$$S(w,x,y,z) \leftarrow S(u,y,x,x,w), S(v,x,y,x,w), A(u,v,z)$$

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 adorned copy $P_j$ of $P$ which has the same recursive rule as $P$, and a nonrecursive rule

$$S(x_1, \ldots, x_n) \leftarrow B(x_1, \ldots, x_n), j = (x_1 + x_2 + \ldots + x_k) \mod r$$

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 [WS, Theorem 2]. In fact, lack-of-duplication has also been shown, namely, for every input, the outputs of the adorned copies 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:

{ $B(1, \ldots, 1, q, 1, \ldots, 1)$ | $q \geq r$, and $q$ 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 an 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 adorned copy 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 adorned copy, there is a new atom obtained in any other adorned copy. Therefore, for the input $I$, the same-size output is generated by each adorned copy. Thus, by completeness and lack-of-duplication, the ratio $|O(P,J)| / \max_i |O(P_i,J)|$ equals exactly $r$. \[]
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. A rule of a program $P$ is an exit rule if its body consists of extensional predicates only. An exit rule $r_e$ with extensional predicate symbols $B_1, \ldots, B_t$ is distinct, if there is no other rule $r$ of $P$ for which the following condition is satisfied: in the body of $r$ there are some extensional predicates, and every such extensional predicate belongs to the set $\{B_1, \ldots, B_t\}$. In other words, $r_e$ is not distinct if there is another rule of $P$ in which a subset of the $B_i$'s appears, but no other extensional predicate does. Note that the exit rule of a sirip is distinct. Particularly, the exit rule in the $csl$ program of Example 1 is distinct, because FLAT does not appear in the recursive rule. A linear program is distinct if it has a distinct 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 adorned copy $P_j$ of $P$ be obtained by adding the predicate $j = x \mod r$ to each exit rule, for $j=0, \ldots, r-1$ (all the other rules stay the same).

To show completeness, assume that for an input, say $I$, an atom, say $a$, is in the output of $P$. Consider a derivation tree, $t$, for $a$. It is easy to see by induction on the height of $t$ that in the tree exactly one instantiation of an exit rule is represented (one node which has only extensional predicate sons). In that instantiation, $x$ is substituted by some constant, say $n$. Let $l = n \mod r$. Then $t$ is a derivation tree for $a$ in $P_l$. Thus, $a$ is in at least one $O(P_j, I)$. (It may be in more than one output if $a$ can be obtained by a derivation tree with a different instantiation of an exit rule.)

Now assume that the distinct exit rule in $P$ is:

$$R(x, \ldots) :- B_1(\ldots), B_2(\ldots), \ldots, B_k(\ldots)$$

A potential-speedup of $r$, and thus nontriviality, can be demonstrated by using as input the set of atoms

$$\{B_1(i, \ldots, i), B_2(i, \ldots, i), \ldots, B_k(i, \ldots, i) \mid 1 \leq i \leq N\}$$

for a large enough $N$. Specifically, for an arbitrarily large integer, $n$, the input is the above set, with $N = n \cdot r$. Then the instantiation of the distinct rule of each adorned copy, $P_i$, adds $n$ $R$-ground-atoms to $O(P_i, I)$. Also, note that the instantiation of another rule can add atoms to the output only if the body of that rule does not contain any extensional predicate atoms (for example, a rule may have the form $T(x, y, z) :- R(x, y, z)$). Furthermore, for each new ground-atom obtained by such an instantiation in $P_i$, there is a new atom obtained by an instantiation of the same rule in each one of the other adorned copies. Therefore, the outputs of the
adorned copies are of equal-size. Moreover, it is easy to see that these outputs are pairwise disjoint. Thus, by completeness, the ratio \( \frac{O(P,J)}{\max_{i} O(P,J)} \) equals exactly \( r \).

In [WS] we have shown that a linear sirup without repeated variables in the recursive predicate, is not decomposable if it is not pivoting. In contrast, observe that theorem 2 implies that every linear sirup has a load-sharing scheme of any size.

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

1. \( S(x,y) \leftarrow B(x,z), B(y,w), S(v,y) \)
2. \( S(x,y) \leftarrow B(x,z), B(y,w) \)

If \( \text{odd}(x) \) is added to the nonrecursive rule in adorned copy \( P_1 \), and \( \text{even}(x) \) in adorned copy \( P_0 \), 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 \( S_1 \) but not in \( S_0 \), and the ground atom \( S(e,b) \) is in \( S_0 \) but not in \( S_1 \). 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 \( S_1 \). Then,

\( S(e,b) \leftarrow B(e,d), B(b,f), S(o,b) \)

is an instantiation of \( r1 \) which adds \( S(e,b) \) to \( S_1 \). This contradicts our assumption that \( S(e,b) \) is not in \( S_1 \). 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.

Let us conclude this subsection with two practical examples (taken from [NJ]) of distinct linear programs. The first assumes that there is an extensional relation, \( \text{PERFECTFOR}(x,y) \), of people, \( x \), and products, \( y \), such that \( y \) is perfect for \( x \). Similarly, there are extensional relations, \( \text{FRIEND}(x,y) \) and \( \text{IDOL}(x,y) \), of people, \( x \), and their friends and idols, \( y \). Suppose that a person buys a product if it is perfect for her/him, or if their friend or idol buys it. The following program defines recursively the predicate \( \text{BUYS}(x,y) \) of people, \( x \), and the products they buy, \( y \).

\[
\begin{align*}
\text{BUYS}(x,y) & :\leftarrow \text{FRIEND}(x,w), \text{BUYS}(w,y) \\
\text{BUYS}(x,y) & :\leftarrow \text{IDOL}(x,w), \text{BUYS}(w,y) \\
\text{BUYS}(x,y) & :\leftarrow \text{PERFECTFOR}(x,y)
\end{align*}
\]

For the second program, assume that a person no longer buys a product if their friend buys it, but if it is cheaper than another product that the person buys. Then the predicate \( \text{BUYS} \) is defined as follows.
BUYS(x,y):- IDOL(x,w), BUYS(w,y)

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

BUYS(x,y):- PERFECTFOR(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 variables 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 programs are equivalent if they produce the same output, for any given input. 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 \([UV]\) for the equivalence proof)

\[ S(x, y) : - S(x, z_1), A(z_1, z_2)B(z_2, y) \]

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

\[ S(x, y) : - S(x, z_1), S(z_1, z_2), \ldots, S(z_n, y) \]

is weakly regular, since it is equivalent to the program

\[ S(x, y) : - S(x, z_1), B(z_1, z_2), \ldots, B(z_n, y) \]
In particular, note that the nonlinear transitive closure,
\[ S(x,y) :- B(x,y) \]
is a weakly regular simple chain program.

Although a weakly regular simple chain program can be rewritten as a regular program, which is decomposable, this may not be desirable for performance reasons. In [WS] 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 adorned copy \( P_i \), by adding to the recursive rule the predicate \( i = x \mod r \), 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 \( n \). Furthermore, we can replace all \( B \) predicate symbols (except the one corresponding to the instantiation of the exit rule) by \( S \), and we shall obtain a derivation tree, \( T' \), for \( S(a,b) \) in \( P \). Assume that \( a \mod r = i \). Then it is easy to see that \( T' \) is a derivation tree for \( S(a,b) \) in \( P_i \). (Observe that there may be a node \( B(c,d) \) in the tree, such that \( i \not\equiv c \mod r \); therefore, when creating adorned copies, the evaluable predicate cannot be added to the nonrecursive rule.)

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

\[
\{(i,i+1) | i = 1, \ldots, N-1\} \cup \{(i,i) | i = 1, \ldots, N\}
\]

Then it can be shown that the output of \( P \) consists of all tuples \( (i,j) \) such that \( i \leq j \) (easier to see by considering \( P' \) rather than \( P \)). Therefore, \( |O(P,I)| = \frac{N(N+1)}{2} \), and \( O(P,I)/\max_i O(P_i,I) \rightarrow r \). □

**Remark 1:** The above theorem is proven by adding to the recursive rule in adorned copy \( P_i \), the predicate
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 ([C]). 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 for the problem is the following:

\[
S(x):- S(y), S(z), H(x, y, z)
\]

\[
S(x):- B(x)
\]

Another example of a sirup without a load sharing scheme, is a variant of path systems called the blue blooded frenchman ([CK]):

\[
BBF(x):= BBF(m), BBF(f), MOTHER(x, m), FATHER(x, f)
\]

\[
BBF(x):= FRENCH(x)
\]

Some other variations which have not been defined previously, as far as we know, are (nonrecursive rule obvious, thus specification omitted):

\[
S(x, u):= H_1(x, y, u), H_2(x, z, w), S(y, u), S(z, w)
\]

\[
S(x, u):= H(x, y, z, w), S(y, u), S(z, w)
\]

\[
S(x):= H_0(x, w), H_1(w, y), H_2(w, z), S(y), S(z).
\]

\[
S(x, w, y):= UP(x, t, u), S(t, u, v), FLAT(v, w, z), S(z, r, s), DOWN(r, s, x)
\]

\[
S(x, y):= H_0(x, w, z), H_1(u, v, y), S(w, z), S(u, v).
\]

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 \( R(P) = \{ x \mid x \text{ is in } V(P), \text{ and } x \text{ appears in some } S\text{-atom of } A(P) \} \). 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 $A(P)$ which 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 which 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 the output of $P_j$ by $S_j$. By nontriviality, there is an input $I$, for which each $S_j$ is a proper subset of $S$.

Assume that there are $m$ atoms in the set $\bigcup_{i=1}^{r}(S - S_i)$, and denote them $\{S(C_1), \ldots, S(C_m)\}$. 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 such that $S(C_1)$ and $S(C_2)$ derive a new ground atom, $S(\overline{d}_1)$. It consists of the extensional predicate ground atoms in the following instantiation, $\rho_1$, of $r$. In $\rho_1$ the variables of some $S$-atom, $S'$, in the body of $r$ are substituted to obtain $S(\overline{C}_1)$, and the variables of all other $S$-atoms in the body are substituted such that all these atoms instantiated to $S(\overline{C}_2)$. By requirement 2 in the definition of a propagating sirup, such a substitution is possible, regardless of $\overline{C}_1$ and $\overline{C}_2$. Denote by $z_1, \ldots, z_t$ the uninstantiated, as of yet, variables in the body or $r$. The instantiation $\rho_1$ is completed by instantiating $z_j$ to $e_j$, for $j=1, \ldots, t$, such that: each $e_j$ is different than every other $e_j'$, and each $e_j$ is not in $I'$ constructed thus far. For the instantiation $\rho_1$ denote the atom at the head of $r$ by $S(\overline{d}_1)$. 


Construction of $H_{i}$ for $2 \leq i \leq m-1$: $H_{i}$ is constructed such that $S(\overline{c}_{i+1})$ and $S(\overline{d}_{i-1})$ derive a new ground atom, $S(\overline{d}_{i})$. The construction is similar to the construction of $H_{1}$, except that $S(\overline{c}_{i+1})$ and $S(\overline{d}_{i-1})$ are used instead of $S(\overline{c}_{1})$ and $S(\overline{c}_{2})$, respectively. Also, the instantiation is denoted by $\rho_{i}$, and is completed by instantiating each $z_{j}$ to $e_{j}^{i}$ for $j=1,...,t$. The $e_{j}^{i}$'s are distinct constants, which do not exist in $I$ constructed thus far.

This completes the construction of $I'$. The atoms in the set $D = \{S(\overline{d}_{1}),...,S(\overline{d}_{m-1})\}$ are in the output $S$ of $P$, given the input $I$. The reason for this is that $S(\overline{c}_{1})'''''S(\overline{c}_{m})$ are in the output of $P$ given $I$, and $I$ has been extended to $I'$, such that from $S(\overline{c}_{1}),...,S(\overline{c}_{m})$ 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 $S(\overline{d}_{1})$ is in some output $S_{j}$, only if, for the input $I$, $S(\overline{c}_{1})$ and $S(\overline{c}_{2})$ are both in the output $S_{j}$. 
Proof: First we will prove that $S(\overline{c}_{1})$ and $S(\overline{c}_{2})$ 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 $S(\overline{d}_{1})$ is in $S_{j}$, for the input $I'$. By requirement 4 in "propagating" definition, the graph $G(P)$ has at least one distinguished variable which does not appear in any $S$-atom. Such a variable, by construction of $H_{1}$, is instantiated to a new constant. Therefore, one of the constants in $\overline{d}_{1}$ 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, $S(\overline{d}_{1})$ 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 $\eta$ be an instantiation of the recursive rule $r$, that produces some $S(\overline{d}_{k})$ in the head. Then if $\rho_{k}$ instantiates a variable of $R(P)$, say $y$, to the constant $c$, then $\eta$ 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 "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 $p: z=z_{1},...,z_{n}=x$. Note that $z_{i}$ is distinguished, and both instantiations of $r$, $\rho_{k}$ and $\eta$, produce the head $S(\overline{d}_{k})$. Therefore, $\eta$ instantiates $z_{i}$ to $e_{j}^{i}$. Since there is an edge in $G(P)$ between $z_{i}$ and $z_{k}$ there is in $r$ an extensional predicate atom, $C(...z_{i},...,z_{k})$. 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 \( e_l^k \) in the position corresponding to \( z_i \). Furthermore, in that \( C \)-ground-atom, in the position corresponding to \( z_{i_2} \), appears the constant \( e_l^{k_2} \). Therefore \( \eta \), as \( \rho_k \), instantiates \( z_{i_2} \) to \( e_l^{k_2} \). The above argument can be continued inductively on the length of the path \( p \), to show that \( x \) is instantiated to \( e_l^k \) by \( \eta \). 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 \( \rho_k \) and \( \eta \) instantiate \( x \) to the same constant. □Claim 1.

Therefore, if for the input \( I' \), the atom \( S(c_1) \) is in the output \( S_j \), then \( S(c_1) \) and \( S(c_2) \) must be in the output \( S_j \). To complete the proof of Lemma 1, left to show is that \( S(c_1) \) and \( S(c_2) \) are in the output \( S_j \) for the input \( I \).

**Claim 2:** Let \( \eta \) be an instantiation of \( r \) which has extensional predicate atom, \( C(\bar{f}) \), in the body of the ground rule. Assume that \( C(\bar{f}) \) has a new constant, i.e., \( C(\bar{f}) \) belongs to some \( H_l \). 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 \( H_l \), the variable \( w \) is instantiated by \( \rho_l \) to some new constant, \( e_l \). Consider the path, say \( p \), in \( G(P) \), from \( w \) to a distinguished variable, \( z \) (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, \( D(...w,...,v...) \), and \( v \) does not belong to \( R(P) \). By requirement 1, in \( I' \) there is a unique \( D \)-ground-atom which has \( e_l^k \) in the position corresponding to \( w \). Furthermore, that \( D \)-ground has a new symbol in the position corresponding to \( v \). Therefore, \( \eta \) 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 \( \eta \). □Claim 2.

**Claim 3:** Let \( \eta \) be an instantiation of \( r \), which has in the body of the ground rule an \( S \)-atom with a new constant. Then the \( S \)-ground-atom in the head has a new constant.

**Proof:** The substitution \( \eta \) 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 predicate atom in \( r \). By Claim 2, the proof is complete. □Claim 3.

To complete the proof of Lemma 1, consider a derivation tree, \( T \), for \( S(c_1) \), 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 \( S(c_1) \). Since in \( c_1 \) there do not exist any new constants, then a new constant does not appear in the derivation tree \( T \). By the construction of the \( H_l \)'s, and by requirement 3, it can be seen that every atom of \( I' \sim I \) has a new constant. Therefore, \( T \) is a derivation tree for \( S(c_1) \) given the input \( I \). Similarly it can be shown that \( S(c_2) \) can be derived...
given the input \( I \). □

Now we shall show by induction on \( n \), that if \( S(\vec{d}_1), \ldots, S(\vec{d}_n) \) are in the output \( S_j \) given the input \( I' \), then \( S(\vec{c}_1), \ldots, S(\vec{c}_{n+1}) \) 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 \( S_j \) which contains the whole set \( \{S(\vec{c}_1), \ldots, S(\vec{c}_n)\} \) for the input \( I \), and therefore, \( S(\vec{d}_{m-1}) \) cannot be derived, given \( I' \).

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

□

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.

\[
S(x,y) \leftarrow S(x,z_1), A(z_1,z_2)S(z_2,y)
\]

\[S(x,y) \leftarrow B(x,y)\]

6. EXTENSION TO PARALLELIZATION BY GENERAL ALGORITHMS

The purpose of this section is to extend the previous results to algorithms for the evaluation of logic programs.

The algorithms do not necessarily evaluate an adorned copy 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 bottom-up-evaluation of adorned copies does not work, it is conceivable that some other, similar, method does work. For example, maybe if we allow a larger class of evaluable predicates (e.g. log, sine) in an adorned copy, then path systems would be sharable. Or, possibly some direct* algorithm can be purely parallelized? To determine whether path-systems, and other nonsharable programs, can be purely parallelized, we define a load sharing scheme of a program, as a set of algorithms, and consider sharability in this broader context. We prove that path systems is not sharable, i.e., cannot be evaluated in parallel by multiple algorithms that do not need to exchange information, such that each algorithm computes less. Furthermore, we prove that the class of sharable programs properly contains the

* In [AJ] terminology, a direct algorithm is one which evaluates a program by using problem-specific techniques which do not necessarily work for every recursive program.
class of decomposable programs. These are sharable programs which can be evaluated in parallel by algorithms that compute pairwise disjoint sets of facts. Therefore, the concepts of sharability and decomposability are fundamental, and apply to the programs, independently of the pure parallelization method that we proposed.

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 $O(P,J)$, and is preceded by all atoms of some derivation tree of $a$. The sequence $c(I)$ represents the order in which the output of $P$ is evaluated, and is called a "partial" computation, since not all ground atoms of $O(P,J)$ have to be in $c(I)$.

An algorithm, $A$, for partial computation of $P$ is a function which maps each input, $I$, into a partial computation of $P$, denoted $A(I)$. We assume that communication among a large number of algorithms is prohibitively expensive. 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 $D = \{A_1, \ldots, A_n\}$ be a set of algorithms for partial computation of $P$. The set $D$ is complete for an input, $I$, if for each ground-atom, $a \in O(P,J)$, 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 $I$, if there is some atom $b \in O(P,J)$, which is not in $A_j(I)$. Suppose now that $A_j \in D$ computes less for some input $I_0$. We say that $A_j$ bottom-up-evaluates $I_0$ if for each $b \notin A_j(I_0)$, and for each set of input atoms, $Z$, the following two conditions are satisfied:

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

2. (contribution) Assume that $b \in A_j(I_0 \cup Z)$ and for some set of input atoms $Z'$, in the input $I_0 \cup Z \cup Z'$ there is a derivation tree of $b$ which contains an atom of $Z'$. Then, $b \in A_j(I_0 \cup Z \cup Z')$.

The noncontribution requirement is simply that if the atom $b$ is not in $A_j(I_0)$, and 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 contribution requirement is the following. If $Z$ does contribute to the derivation of $b$, and its addition to $I_0$ induces the production of $b$ by $A_j$, then further addition of a set of input atoms, $Z'$, which contributes to the derivation of $b$, cannot suppress the production of $b$. Note that if $A_j$ is monotonic, then the contribution requirement is satisfied, but
the converse is not necessarily true. The reason is that the requirement does not say anything about the production of b, if Z does not contribute to its derivation; then Z may suppress the production of b. The reason for the term "bottom-up-evaluates" is that a "top-down", i.e. a la Prolog, type of algorithm does not satisfy any one of the above requirements. Also note that for the sake of generality we defined "bottom-up-evaluation" with respect to a specific input. It is conceivable that an algorithm bottom-up-evaluates one input (which contains the constant 25, for example) and does not do so for another input.

Input I is a time-saving input for the set of algorithms D, if each algorithm computes less for I, and bottom-up-evaluates it. The set D is a load sharing scheme for P if it is complete for every input, and has at least one time-saving input. A program which has load sharing scheme of algorithms, is called sharable. It is easy to see, that this definition of load sharing by algorithms, generalizes the definition of load sharing by adorned copies (given in section 3), based on the following proposition.

Proposition 2: Assume that a program P has a load sharing scheme of adorned copies, D = {P, ... , P}. Then P has a load sharing scheme of algorithms, D' = {A,...,A,}.

Proof: Define algorithm A, to work as follows. For each input I it first outputs all the atoms of I in some arbitrary order, then it executes the procedure BUE on P, outputting the intentional predicate atoms as they are evaluated. It is easy to see that D' = {A,...,A,} 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, computes less, and bottom-up-evaluates each input which satisfies the nontriviality condition for D. Since there is at least one such input for D, the set D' has a time-saving input. []

Theorem 5: The sirup path systems (defined in section 5) is not sharable.

Proof: Denote the sirup path systems by P. Assume that D = {A,...,A,} is a load sharing scheme for P. Consider a time-saving input, I. Observe that S is the only intentional predicate in P. Thus for each A, since it computes less for I, there is at least one S(c) in O(P,I), which is not in A/(I). In other words, there is not any A/(I) which contains all S(c)'s. Let

C = (S(C)1 S(C) ∈ O(P,I), and for some A, S(C) ∈ A/(I))

Denote C = {S(C),... ,S(C)}. Let d1,...,dm−1 be m−1 constants, none of which is in I. Denote I = I ∪ H where

H = (H(d1,c1,c2),H(d2,d1,c3),H(d3,d2,c4),...,H(dm−1,dm−2,cm))

Obviously, each S(d) for i = 1,...,m−1 is in O(P,I). Observe that there cannot be a derivation tree for S(c) in I,
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 \( P \), these 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. []

Actually, it can be shown that any propagating sirup is not sharable.

Next we consider decomposability of a program, \( P \). Let \( D = \{A_1, \ldots, A_r\} \) be a set of algorithms. The set \( D \) is a \textit{load-decomposing scheme} of \( P \), if it is a load sharing scheme, and in addition the following condition, called \textit{lack-of-duplication}, is satisfied. For each input \( I \), and every pair of algorithms \( A_i \) and \( A_j \), for \( j \neq i \), there is no intentional predicate atom which belongs to both, \( A_i(I) \) and \( A_j(I) \). A program which has a load decomposing scheme of data driven algorithms, is called \textit{decomposable}. Obviously, if the program \( P \) has a load decomposing scheme of adorned copies, then it has a load decomposing scheme of algorithms. It is easy to see, and has been proven in [WS], that the transitive closure, defined by the following program, is decomposable.

\[
S(x, y) :- S(x, z), A(z, y)
\]

\[
S(x, y) :- A(x, y)
\]

Decomposability is achieved simply by creating two adorned copies, with the evaluable predicates \textit{even} \((x)\) and \textit{odd} \((x)\) added to the exit rules. In fact, in [WS] it has been proven that every pivoting sirup has a load decomposing scheme of adorned copies. Similarly to proposition 2, it is easy to prove that if a program has a load decomposing scheme of adorned copies, then it has a load decomposing scheme of algorithms.

By proposition 2 the canonical strongly linear program is sharable. Contrast this with the following.
Theorem 6: The canonical strongly linear program (see section 1) is not decomposable.

Proof: Denote the csl program by $P$. Assume that the set of algorithms $D = \{A_1,\ldots,A_r\}$ is a load decomposing scheme for $P$. By definition, there is a time-saving input $I_0$. Since each algorithm computes less for $I_0$, there is an algorithm, say $A_k$, and an atom, say $S(a,b)$ in $O(P,I_0)$, such that $S(a,b)$ is not in $A_k(I_0)$. By completeness $S(a,b)$ is in the output of some algorithm, say $A_i(I_0)$. Again, since $A_j$ computes less for $I_0$, there is another atom $S(c,d)$ of $O(P,I_0)$, such that $S(c,d)$ is not in $A_j(I_0)$. By completeness, $S(c,d)$ is in some $A_j(I_0)$, and by lack of duplication $S(a,b)$ is not in $A_j(I_0)$. Therefore, $S(a,b)$ and $S(c,d)$ are two distinct atoms, and $A_i$ and $A_j$ are two distinct algorithms of $D$, such that $S(a,b)$ is in $A_i(I_0)$ but not in $A_k(I_0)$, for $h \neq i$; and $S(c,d)$ is in $A_j(I_0)$ but not in $A_k(I_0)$, for $h \neq j$. Now let $m$ and $n$ be two distinct constants which do not exist in $I_0$. Denote $Z = \{UP(m,a), DOWN(b,n)\}$. Observe that in $I_0 \cup Z$ there is no derivation tree for $S(a,b)$ which includes an atom of $Z$ (since in $I_0$ there is no atom $UP(x,m)$). Thus, since each algorithm bottom up evaluates $I_0$, the atom $S(a,b)$ is not in any $A_h(I_0 \cup Z)$, for $h \neq i$. By completeness, $S(a,b)$ must be in $A_i(I_0 \cup Z)$. Observe also that $S(m,n)$ is in $O(P,I_0 \cup Z)$. Any derivation tree for $S(m,n)$ has $S(a,b)$ as a son of the root, therefore, by completeness, $S(m,n)$ must be in $A_i(I_0 \cup Z)$. Next, denote $Z = \{UP(m,c), DOWN(d,n)\}$. As has been argued for $I_0 \cup Z$ it can be argued for $I_0 \cup Z'$, that $S(m,n)$ must be in $A_j(I_0 \cup Z')$. Now consider $I_0 \cup Z \cup Z'$. Since both $A_i$ and $A_j$ bottom up evaluate $I_0$, the atom $S(m,n)$ belongs to both, $A_i(I_0 \cup Z \cup Z')$ and $A_j(I_0 \cup Z \cup Z')$. But this contradicts lack of duplication for $I_0 \cup Z \cup Z'$. 

Therefore, there is a hierarchy of logic programs, depicted in figure 2. The decomposable programs are most amenable to pure parallelization. A representative of this class is the transitive closure. Next in the hierarchy, is the class of sharable programs. A representative of this class is the canonical strongly linear program. Finally, the class of nonsharable programs, of which path systems is a representative, is not amenable to pure parallelization.

7. CONCLUSION AND FUTURE WORK

In this paper we examined pure parallelization of data intensive DATALOG programs. The method proposed is parallelization by adorned copies. It has been shown that some important classes of programs (for example the linear ones) can be parallelized this way, i.e., are sharable, while other classes cannot. Evaluation of the sharable programs can be spread among an arbitrary number of processors. Furthermore, the potential speedup of such a parallelization is optimal, specifically, the potential speedup is equal to the number of processors participating in the
parallelization scheme. Finally, we examined pure parallelization by general algorithms rather than adorned copies, and demonstrated a hierarchy of three classes of DATALOG programs, each more amenable to pure parallelization than the class above it.

As far as future research is concerned, an obvious direction is to extend the class of programs which have load sharing schemes, and the class for which we can prove nonexistence of such a 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 [TZ]). Another research direction, is to quantify the average (rather than potential) speedup achieved by pure parallelization. Finally, we would like to determine how to minimize communication among the processors, when it cannot be avoided. In other words, for parallelizing nonshareable programs communication is necessary. How should this communication be minimized? We conjecture that the study of pure parallelization will also prove helpful for answering this question.
Acknowledgement

The author thanks Nissim Francez and Oded Shmueli for helpful discussions and comments.

References


