WHY SINGLE PARALLELIZATION STRATEGY IS NOT ENOUGH IN KNOWLEDGE BASES

by

Simona Cohen and Ouri Wolfson

Technical Report #523

October 1988
WHY A SINGLE PARALLELIZATION STRATEGY IS NOT ENOUGH IN KNOWLEDGE BASES

Preliminary Version

Simona Cohen
Ouri Wolfson

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

ABSTRACT

We argue that the appropriate parallelization strategy for logic-program evaluation depends on the program being evaluated. Therefore, this paper is concerned with the issues of program-classification, and parallelization-strategies. We propose five parallelization strategies that differ based on the following criteria. Their evaluation cost, the overhead of communication and synchronization among processors, and the programs to which they are applicable. In particular, we start our study with pure-parallelization, i.e., parallelization without overhead. An interesting class-structure of logic programs is demonstrated, when considering amenability to pure-parallelization. Then we propose strategies that do incur an overhead, but are optimal in a sense that will be precisely defined.

This paper makes the initial steps towards a theory of parallel logic-programming.
1. INTRODUCTION

It is accepted by now that declarative languages present numerous advantages over navigational ones, and should constitute the interface to the next-generation databases, such as deductive and object oriented databases ([B]). We feel that parallelization holds the key to acceptable performance of a declarative language. In this paper we continue the study of Datalog parallelization, begun in [WS, W]. Datalog (see [MW]) is a simple logic programming language. At the heart of our study lies the realization that "no single parallelization strategy is appropriate for all logic programs" (to rephrase the analog statement in [MNSUV], that no single evaluation strategy is appropriate). Therefore, a large part of this paper is devoted to the classification of programs according to parallelization properties. The most fundamental property is strong-decomposability. It enables the evaluation to be separated into completely independent tasks, that can be carried out in parallel.

The other part of the paper is devoted to parallelization schemes, i.e., sets of algorithms cooperatively evaluating a program. We analyze the properties and limitations of several classes of such schemes. We postulate that the performance of a parallelization scheme depends on two factors: the evaluation cost, and the overhead of communication and synchronization among the algorithms of the scheme. The parallelization schemes proposed in this paper differ in their evaluation cost, overhead, and programs to which they can be applied. We first formally define and study pure parallelization, i.e., parallelization without overhead. It turns out that there is a hierarchical class structure of programs with respect to pure parallelization. Most amenable is the class of strongly-decomposable programs. They can be purely parallelized, with minimal total evaluation-cost. In other words, the total evaluation cost of all processors, equals the cost of one processor executing the evaluation single-handedly. We propose a parallelization strategy, i.e., class of schemes, that is pure and has minimal cost. Next is the class of sharable programs. They can be purely parallelized, but the evaluation cost may not be minimal. We propose a parallelization strategy for them as well. Finally, the class of non sharable programs cannot be purely parallelized. As we shall explain, pure parallelization is also restricted to static work-load allocation.

Then, next we consider schemes that do incur an overhead. We distinguish between control-overhead and data-overhead. The former consists of control messages being transmitted between processors, and the latter consists of data messages, i.e. relations or part of them, being transmitted. Independent-parallelization, i.e., parallelization with control- but without data-overhead, is still restricted in applicability, but we propose a strategy that uses control messages to dynamically balance the evaluation work-load. Thus, we show that for the strongly-decomposable programs independent parallelization is in some sense optimal. Finally, we introduce a parallelization strategy, called DS3, that incurs minimal data-overhead, can be applied to
parallelize all programs, and we prove that it has minimal evaluation-cost for the linear programs.

The parallelization schemes proposed in this paper are "scalable", i.e., an arbitrary number of processors can be effectively utilized. (In any case, we assume that the number of processors is significantly smaller than the number of tuples in the database). The datalog programs considered are the ones with two rules, and one, unary or binary, intentional predicate. They are syntactically simple, yet as demonstrated, provide a rich test-bed, with subclasses having different parallelization properties. Additionally, the strategy DS3 can be easily generalized to arbitrary datalog programs.

Concerning relevant work, most efforts in the area of parallelization have been devoted to characterization of the logic programs which belong to the NC complexity class (see [UV], [CK], [AP]). If a program is in NC, it means that a query can be evaluated very fast (in polylogarithmic time), given a very large number of processors (polynomial in the number of database tuples). The processors have to communicate extensively, usually through common memory. Unfortunately, this research means very little as far as utilizing a constant number of processors, as we assume here, particularly if they do not share memory (e.g. a hypercube multiprocessor having 1024 nodes).

Another body of relevant research has been performed on parallel and concurrent variations of PROLOG. Much of this research, along with a description of the three leading languages that have emerged (Flat Concurrent Prolog, Parlog, and Guarded Horn Clauses) is summarized in the collection of papers [Sh]. However, there is a fundamental difference between logic program evaluation in knowledge bases, which is performed bottom-up (or forward chaining), and concurrent Prolog, which is evaluated top-down (or backward chaining). As a result of this difference we feel that not much of the research on concurrent Prolog can be utilized in knowledge bases.

Bottom-up-evaluation for logic programs in knowledge bases, usually amounts to iteratively performing several relational algebra operations, and deducing new facts, until a fixed point is reached. There has been work on parallelization of relational algebra operators, particularly the join (e.g. [BBDW]). However, when parallelizing these low level operations in knowledge bases, the processors have to be synchronized at the completion of each iteration, then each processor has to exchange its newly generated facts with the newly generated facts of every other processor, and duplicate elimination has to be centralized at a single processor. Therefore, the communication and required synchronization among the processors is extremely high. Much of this overhead can be avoided by considering the "big picture", i.e., the logic program.

1 As often demonstrated (e.g. [LY, DIY]), communication overhead limits the potential gains in performance by parallelization.
Finally, in [W] and [WS] we proposed methods for pure parallelization, and analyzed their applicability. The methods basically consist of rewriting a program by a set of other programs (each of which works with smaller relations), and evaluating them in parallel. This paper extends that work in several ways. First, it demonstrates that there is something fundamental about decomposability, independent of the method proposed in [W]. In fact, there are two notions, decomposability and strong decomposability. We provide a complete characterization of the programs with respect to both. Second, we formally define parallelization-schemes, overhead, and pure parallelization. Thus, we extend the impossibility results that demonstrate the pure-parallelization class-structure, beyond one method or another. Third, in the present paper we analyze the evaluation A cost of parallelization schemes. Fourth, we propose schemes with communicating algorithms, that overcome the limitations of pure parallelization.

The rest of this paper is organized as follows. In section 2 we study decomposability, and in section 3 we study pure parallelization, and its limitations. In section 4 we propose communicating schemes, and in section 5 we conclude and discuss future research. In appendix A we provide an evaluation-cost analysis of the proposed strategies, and in appendix E we demonstrate them using sample programs and inputs. In appendices B, C, D we provide formal definitions and algorithms for some concepts discussed intuitively in the paper.

2. PARTITIONING OF THE OUTPUT DOMAIN

In this section we study decomposability. If a program is decomposable, it means that its output domain, i.e., the infinite set of possible output tuples, can be partitioned as follows. Regardless of the input, each intentional fact, \( a \), has a derivation tree in which all intentional facts belong to the same partition-member as \( a \). As we shall explain, this notion is important for parallel, as well as sequential processing. We completely characterize the programs that are decomposable, and an interesting phenomenon is exhibited. If a program has a partition in which more than one member is "nontrivial" (i.e. contains facts that cannot be derived from an exit rule alone), then it has a partition with an infinite number of members that are nontrivial. We shall argue that programs that satisfy the above condition are more interesting. We call them strongly decomposable, and completely characterize them as well.

In this paragraph we define the basic terminology. 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 results in this paper are applicable to character strings as well, since their binary representation is a 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, ...,and \( Q^k \)." If an atom has a constant in each argument position,
then it is a fact. A DATALOG program (see [MW]), 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. We discuss datalog programs with one intentional predicate, denoted S, that is unary or binary. Furthermore, a program does not have any constants, and consists of two rules: an exit rule, \( S(x,y) : - B(x,y) \) or \( S(x) : - B(x) \); and a recursive rule, in which the predicate symbol \( B \) does not appear. The recursive rule of a program is range restricted, i.e., every variable in the head of a rule also appears in the body of the rule. The input \( I \) to a program \( P \) is a finite set of \( R \)-facts, where \( R \) is some extensional predicate symbol. The output of \( P \) for the input \( I \), denoted \( O(P,J) \), is a set of \( S \)-facts. A fact, \( a \), is in the output if and only if it has a derivation tree. This is a finite tree with the nodes labeled by facts; \( a \) is the root, the leaves are facts of \( I \), and for each internal node, \( b \), with children \( b_1, \ldots , b_k \), there is an instantiation of a rule which has \( b \) as the head and \( b_1, \ldots , b_k \) as the body. We assume that the recursive rule of a program is minimal, i.e., there is no atom which can be eliminated from the body of the rule to obtain an equivalent program. Equivalence of two programs of the type we discuss can be tested using an algorithm provided in [Sa].

A program is unary (binary) if the intentional predicate \( S \) is unary (binary). For the unary programs we define the output domain, denoted \( O \), to be the set of all \( S \)-facts, namely the infinite set \( \{ S(a) \mid a \text{ is a constant} \} \). Similarly we define the output domain of binary programs. A set of two or more sets, \( M_1, \ldots , M_k,\ldots \) is a partition of the output domain if \( \bigcup_i M_i = O \), and each \( M_i \) is nonempty, and the \( M_i \)'s are pairwise disjoint. Let \( D \) be a partition of the output domain for the program \( P \), and let \( M_i \) be a member of \( D \). The fact \( g \in M_i \) is proper, if: for every input \( I \) such that \( g \) is in the output \( O(P,J) \), the atom \( g \) has a derivation tree in which all the \( S \)-facts are in \( M_i \). A program \( P \) is decomposable if it has a partition \( D \), for which every fact in the output domain is proper. Then, the set \( D \) is called an eligible partition of \( P \).

Decomposable programs are interesting for parallel as well as sequential processing. For parallelism, each processor can assume responsibility for producing the output of the program belonging to some members of an eligible partition. This way, each processor works with a smaller \( S \)-relation during bottom up evaluation. Furthermore, since each output fact is proper, there is no overhead for transmitting intermediate results between processors, and if each member of the partition is assigned to a processor, then the complete output is guaranteed to be produced (see strategy DS1 in the next section). For sequential processing, once a fix-point is reached within a member (of the partition) all the output facts of the member can be removed from the relation \( S \). This in turn reduces the size of \( S \) for further processing. For example, consider the transitive closure program, \( P_1: S(x,y) : - S(x,z), A(z,y) \). As we shall see it is decomposable, and assume that it is semi-naively evaluated. If at some iteration the differential \( \Delta S \) does not contain any more tuples of the form \( (2,k) \) (but in prior iterations it did), then all such tuples...
can be output and removed from $S$. Thus $S$ is reduced for the next iteration.

**Theorem 2.1:** An unary program is not decomposable.

**Proof idea:** For any two facts, $S(a)$ and $S(b)$, there is a one-to-one instantiation of the recursive rule, such that one fact is in the body, and the other is in the head. Consider the input $I$ consisting of the body of the instantiated rule (the $S$-predicate-symbol is replaced by $B$, the predicate symbol in the exit rule). Since the recursive rule is minimal, $I$ forces both facts to be in the same partition-member. []

For the rest of this section we only consider binary programs. A program $P$ is *first-fixed* (*second-fixed*) if all the $S$-atoms of the recursive rule have the same variable in the first (second) argument position. For example, the transitive closure program above, $P_1$, is first-fixed. If the recursive rule is $S(x,y) :- A(x,z), S(z,y)$, then the program is second fixed. Another example of a second-fixed program, this time nonlinear, is one with the rule $S(x,y) :- S(z,y), S(w,y), A(z,w,x), C(y)$. For each natural number $i$, denote by $M_i$ the infinite set $\{S(i,k) | k \geq 1\}$. Let $P$ be a first-fixed program. Define the infinite set $\{M_i | i \geq 1\}$ to be the *natural partition* for the first-fixed program. Similarly a natural-partition is defined for a second-fixed program ($M_i = \{S(k,i) | k \geq 1\}$).

**Lemma 2.2:** A program $P$ which is first-fixed or second-fixed is decomposable. The natural partition for $P$ is an eligible partition for $P$.

**Proof idea:** Consider the transitive closure $S(x,y) :- S(x,z), A(z,y)$. Clearly an atom of the form $S(3,k)$ is never in the derivation tree of an atom $S(2,j)$. []

A program is *repeating* if every $S$-atom in the recursive rule (head and body) has a repeated variable. For example, the program with the recursive rule $S(x,y) :- S(y,y), A(x,y)$ is repeating. Define the partition $\{M_1, M_2\}$, where $M_1 = \{S(i,j) | i = j\}$ and $M_2 = \{S(i,j) | i \neq j\}$, to be the *degenerated partition*.

**Lemma 2.3:** A repeating program $P$, is decomposable. An eligible partition for $P$ is the degenerated one. []

In appendix B we define a *discriminating* program, and provide an algorithm for identifying one. Intuitively, a program is discriminating if for every input $I$, the facts of the output which can only be obtained by instantiating the recursive rule, can be obtained also if we remove from $I$ all the $B$-facts with the same constant in both positions. For example, the program with the recursive rule $S(x,y) :- S(y,x), S(x,z), S(z,y)$ is discriminating since the same-constant $B$-facts of an input do not contribute to the output produced by the recursive rule.

**Lemma 2.4:** A discriminating program $P$, is decomposable. An eligible partition for $P$ is the degenerated one. []

**Theorem 2.5:** A program is decomposable iff it is first-fixed, or second-fixed, or repeating, or discriminating.

**Proof idea:** (if) From lemmas 2.2, 2.3, and 2.4. (only if) Lengthy case analysis. []
For some decomposable programs, having multiple processors does not provide a real advantage compared to a single processor, particularly if the latter removes members as it reaches member-fixpoint. For example, consider a repeating program with the degenerated partition. We can assign responsibility for each one of the two members to a different processor, but the processor that receives the member $M_2 = \{S(i, j) \mid i \neq j\}$ cannot obtain any new fact by instantiating the recursive rule of the repeating program. It does remove from the other processor the burden of handling the members of $M_2$, when generating the members of $M_1$. But a single processor can also remove the members of $M_2$, after the first iteration of (semi-) naive evaluation. Well, maybe a repeating program can have another partition, in which more than one processor can produce new facts by instantiating the recursive rule. We shall prove in Theorem 2.6 that this is not the case, i.e., for every partition of a repeating program, there must be one member which contains all the facts of $M_1$. The same arguments can be made for discriminating programs. For them, the "real" work is carried out by the processor which is assigned responsibility for $M_2$. Therefore, for the purpose of parallelization, we are more interested in the programs with an eligible partition in which the recursive rule has to be "used" for more than one partition-member.

For a program $P$, and an input $I$, a fact $a \in O(P, I)$ is nontrivial if every derivation tree of $a$ has a depth higher than one. In other words, a nontrivial fact cannot be obtained by an instantiation of the exit rule. The program $P$ is strongly decomposable if it is decomposable, and has an eligible partition $D$, such that for some input, more than one partition member contains a nontrivial fact. The partition $D$ is called a strongly eligible partition. Although the definition required "some" nontrivial facts for some input, we shall demonstrate in corollary 2.7, that if a program is strongly decomposable, then every fact in the output domain is nontrivial for some input. Therefore, as we shall see, the work-load can be evenly distributed among processors.

Define a program to be switching if it is linear, and the head of the recursive rule is $S(x, y)$, and the $S$-atom in the body of the recursive rule is $S(y, x)$. Note that a switching program is discriminating, and in addition to the degenerated eligible partition it has an eligible partition $D = \{M_{ij} \mid i \geq 1, j \geq 1\}$ where each $M_{ij}$ is $\{S(i, j), S(j, i)\}$. This partition is called the natural partition of the switching program (different than the natural partition of a first-fixed program).

Theorem 2.6: A program is strongly decomposable if and only if it is first-fixed, or second-fixed, or switching. □

The next corollary demonstrates the robustness of the strong-decomposability concept.

Corollary 2.7: If a program is strongly decomposable, then it has an infinite eligible partition (e.g. the natural partition). Furthermore, for each $k$ members of the partition $R_1, \ldots, R_k$, and for each $k$ facts $a_i \in R_i$, for $i = 1, \ldots, k$, there is an input for which each $a_i$ is nontrivial. []

The next proposition indicates that for the strongly decomposable programs there is no strongly eligible partition which is "finer" than the natural partition.
Proposition 2.8: Let \( P \) be a strongly decomposable program, and let \( a, b \) be two facts of a member, say \( M_i \), of the natural partition of \( P \). Then \( P \) has no strongly eligible partition \( \rho \), such that \( a \) and \( b \) belong to different members of \( \rho \). [1]

The last comment in this section concerns extension of the positive result of theorem 2.6. Specifically, if a program is first-fixed, or second-fixed, or switching, then it is strongly decomposable even if we allow the body of the recursive rule to contain negated extensional-atoms, provided that the variables in these atoms also appear in nonnegated atoms in the body (stratified and safe negation). Furthermore, such programs are strongly decomposable even if the predicate symbol \( B \) is allowed to appear in the body of the recursive rule.

3. PURE PARALLELIZATION

In the previous section we have seen that strongly decomposable programs are amenable to parallelization that does not incur communication or synchronization overhead, namely pure parallelization. It is achieved by replicating the input at multiple processors, and assigning output responsibility for each member of a strongly-eligible-partition to some processor. Two questions immediately arise. First, what is the performance of this parallelization method? Second, what are the limits of pure parallelization, i.e., can other programs be purely parallelized, possibly by another method? In this section we answer these questions, which turn out to be related as follows. There are other programs, although not all of them, that can be purely parallelized. However, the ones that can be purely parallelized while guaranteeing minimal total evaluation-cost, are exactly the strongly decomposable ones. Therefore, we discover a class-structure of programs with respect to pure parallelization. This structure is illustrated in figure 1 (following the references section).

We start by defining parallelization schemes, i.e., sets of parallel algorithms that evaluate a program. Then we define pure schemes, and distinguish between two types of such schemes: decomposition and sharing. The formal definitions are given in appendix C, and here are just the informal ones. Intuitively, a parallelization scheme is a set of algorithms, each of which performs logic-program evaluation, but with less than the whole input; consequently, it is faster, but does not necessarily produce the whole output. A similar situation occurs in all parallel-join algorithms; each processor performs a join, but of less than the whole input relations. More formally, let \( P \) be a program. An \( r \)-parallelization-scheme, \( A \), for partial computation of \( P \), is a function which maps each input, \( I \), into \( r \) partial computations, \( A(I) = \{A_1(I), ... , A_r(I)\} \). (\( A \) is also called an \( r \)-scheme or simply a scheme for short.) A partial computation, \( A_j(I) \), is a sequence of facts. Each fact in the sequence is either computed by processor \( j \), or transmitted to \( j \) from another processor. The sequence \( A_j(I) \) represents the order in which the facts are computed or received at processor \( j \). It is called a "partial" computation, since the whole output \( O(P, J) \) may not be in \( A_j(I) \). The partial computation is produced by algorithm \( A_j \). If for each
input there are no facts transmitted between processors, then \( A \) is an independent scheme. Such a scheme does not incur a data-overhead. If in addition, the output of each algorithm of \( A \) depends solely on the input (and any control-messages received from other processors do not affect this), then \( A \) is data-driven. Such a scheme does not incur control-overhead. If \( A \) is both, independent and data-driven, then it is called a pure parallelization scheme, or, for short, a pure scheme. Such a scheme does not incur any overhead of communication among the algorithms.

Let \( A \) be an \( r \)-scheme. The scheme is sharing, if for each input \( I \), each fact \( a \in O(P,I) \), is computed in some partial computation in \( A(I) \) (i.e., the scheme does not lose output). If \( A \) is also disjoint i.e., each output fact is computed by a unique processor, then \( A \) is called a decomposition \( r \)-scheme. Assuming that a certain amount of work is necessary to produce each output fact, processors executing (the algorithms of) a disjoint scheme do not duplicate one another's work; thus, for some decomposition schemes we can prove (in appendix A) evaluation-cost minimality.

**Lemma 3.1:** Assume that \( A \) is a pure decomposition scheme for a program \( P \), and let \( b \) be a fact of the output domain. If for some input \( I \), and for some algorithm \( A_j \) of \( A \), the fact \( b \in A_j(I) \), then for every input \( I' \), if \( b \in O(P,I') \) then \( b \in A_j(I') \).

*Proof idea:* Otherwise, since \( A \) is data driven, for the input \( I \cup I' \) the fact \( b \) is computed by two algorithms of \( A \); this in turns contradicts disjointness. \( \square \)

Under the assumptions of the previous lemma we say that \( A_j \) is the home algorithm of \( b \) in \( A \). Note that every fact in the output domain has a unique home algorithm (by disjointness and completeness of \( A \)).

**Theorem 3.2:** A program \( P \) has a pure decomposition scheme if and only if it is strongly decomposable. Furthermore, let \( A \) be a pure decomposition \( r \)-scheme and denote by \( M_i \) the set of facts in the output domain, which have \( A_i \) as their home algorithm, for \( i = 1,\ldots,r \). Then \( \{M_1,\ldots,M_r\} \) is a strongly eligible partition for \( P \). (]

Next we outline a parallelization strategy, i.e., a class of schemes, called DS1. We shall show that every pure decomposition scheme is in DS1. The strategy DS1 uses the term restricted-version. Given a program \( P \), a restricted version of \( P \) (see [WS]) is one obtained by appending evaluable predicates to the body of some rules of \( P \).

**Strategy DS1:** It evaluates a strongly decomposable program \( P \) by algorithms \( \{A_1,\ldots,A_r\} \), for any number of processors, \( r \). Let \( h \) be some hash function that maps each natural number into the set \( \{1,\ldots,r\} \). Each algorithm, \( A_i \), evaluates the restricted version of \( P \) with the predicate \( h(x)=i \), or \( h(y)=i \), or \( h(x+y)=i \) (for \( P \) a first-, or second-fixed, or switching program, respectively) appended to the exit and recursive rules. Thus, for a first-fixed program, processor \( i \) is assigned responsibility for the members \( M_k \) of the natural partition, for which \( h(k)=i \). (]}
processors, and each $A_i$ (naive, semi-naive, or some other evaluation method). The scheme obtained is obviously a pure-decomposition-scheme. Observe that uniting several (but not all) members of the natural partition into one member, leaves an eligible partition. In appendix A we establish that for some schemes in DS1, for each given input, the total amount of work of all processors in evaluating a program is minimal, i.e., not higher than the amount of work in naively or semi-naively evaluating the program by a single processor. Intuitively, DS1 saves time for a first-fixed program, because at each iteration of naive or semi-naive evaluation, the predicate $h(x) = i$ cuts (approximately by a factor of $1/r$) the size of every relation, extensional or intentional, having the attribute $x$. Additionally, it can be shown that the number of iterations does not increase.

Theorem 3.2 and Proposition 2.8 indicate that in a pure decomposition scheme, each algorithm is responsible for some members of the natural partition. Let $P$ be a first-fixed program. An algorithm responsible for member $M_i$ of the natural partition, is one which evaluates (by some method) the program $P$ with the predicate $x = i$ appended to the exit rule of $P$. Thus, every pure decomposition scheme is in DS1.

Are there any other programs that have a pure parallelization scheme? The answer is yes, but for them the total evaluation cost may not be minimal for every input. Specifically, consider the programs that have a pure sharing scheme (see definition before lemma 3.1). They are called sharable programs. Intuitively, a pure sharing scheme does not ensure minimal evaluation cost, since the algorithms of the scheme do not do disjoint work. The reason for this is that the same fact is examined during naive or semi-naive evaluation by two or more algorithms. Strategy SS1 below is a class of pure sharing schemes. The next theorem holds for unary, as well as binary programs, and establishes that the class of sharable programs is strictly larger than the class of decomposable ones.

Theorem 3.4: Every linear program is sharable. 

Presently we do not have a complete characterization of the sharable programs, but have syntactically defined a class of programs that are not sharable. For brevity we omit the definition, and just establish that,

Theorem 3.5: There are programs that are not sharable.

The programs path systems $(S(x) :- \neg S(y), S(z), H(x, y, z))$, and blue blooded frenchman $(S(x) :- S(m), S(f), M(x, m), F(x, f))$, discussed previously ([CK, UV]) in the context of the NC complexity class, belong to the nonsharable class of programs. Intuitively the reason for this is the following. If such a program has a pure-sharing-scheme, and for some input each algorithm of the scheme produces less than the whole output, then, because of the independence and data-driven properties, there must be another input for which some output is lost.

---

2 This observation obviously also holds for a second-fixed program, and for a switching program (use $X + Y$, or any other commutative function, instead of $X$).
### 3. STRATEGY SS1

**Strategy SS1:** It evaluates a linear program $P$ that is not strongly decomposable (e.g. $S(x, y) : \neg \text{UP}(x, w), S(w, z), \text{DOWN}(z, y)$), by algorithms $\{A_1, ..., A_r\}$. Each algorithm, $A_i$, evaluates the restricted version of $P$ having the predicate $h(x) = i$ appended to the exit rule only. $h$ is some hash function.

In appendix A we establish that for each input, the maximum (among all participating processors) amount of work in evaluating a program by SS1, is not higher than the amount of work in evaluating the program by a single processor.

In conclusion, the class-structure of programs with respect to pure-parallelization is illustrated in figure 1. Finally, consider the following question. Can the class of sharable programs be characterized in terms of output domain partitioning, as we have done for programs that have a pure decomposition scheme? This is an open problem at this point, but observe that the natural way of doing so does not work. This natural way is in terms of an output domain "cover", i.e. set of fact-sets that are not necessarily disjoint. For example, the program $S(x, y) : \neg S(w, z), A(x, y, w, z)$ does not have such a cover but is sharable.

### 4. OTHER PARALLELIZATION STRATEGIES

Pure parallelization pays for lack of overhead with two limitations. First, it is restricted in its applicability. Second, when applicable, the evaluation-load cannot be balanced dynamically among the processors; thus, for DS1 we cannot translate minimal total-cost into time minimality. Consequently, in this section we consider impure parallelization by schemes. We first examine independent-parallelization i.e., parallelization with control-overhead but without data-overhead. By using control messages we balance the work-load dynamically, and in particular we improve DS1 to minimize execution time. Independent parallelization is still restricted in its applicability, therefore, in subsection 4.2 we provide a parallelization strategy that works for all programs, thus incurs a data overhead, but we prove that the overhead is minimal.

#### 4.1 Independent Strategies

DS2, presented next, is a strategy of independent decomposition schemes (ids's). Each scheme in DS2 works for every strongly decomposable program, and exploits the relaxation of the data-driven requirement, which constrains a pure scheme. It passes control information between the processors to balance the work-load dynamically.

**Strategy DS2:** As in DS1 a strongly decomposable program is evaluated by $r$ algorithms, for any $r$. For simplicity, we just describe the cooperation of two processors, $p_1$ and $p_2$, in evaluating the first-fixed program $P_1: S(x, y) : \neg S(x, z), A(z, y)$. Processor $p_1$ first evaluates the restricted version of the program having the predicate $x=1$ appended to the exit and recursive rules (i.e. natural-partition member $M_1$); then $M_3$ (i.e. all tuples of the transitive closure of the form
(3,k)), then \( M_5 \), etc. Similarly, \( p_2 \) evaluates members \( M_2, M_4, M_8 \) (nothing in \( M_6 \)), etc. Suppose that \( p_2 \) finishes before \( p_1 \). Then work redistribution occurs. \( p_2 \) notifies \( p_1 \) of completion, and \( p_1 \) responds with the index of the partition-member it is working on, say 7. This indicates to \( p_2 \) that responsibility for the odd partitions that succeed 7, is divided; \( p_2 \) takes the indices 9, 13, 17, 21,..., and \( p_1 \) takes 11, 15, 19,..., etc. The work continues with each algorithm notifying its companion upon completion, and the latter responding with the partition number it is working on at that time. 

We claim that schemes in the above strategy are optimal within the strategy of partition-oriented ids’s. Intuitively, this is the strategy of ids’s in which for every input, the output in a member of the natural partition is never "split" between two or more partial computations. Such splitting necessitates extra work to determine that every fact is proper for an input, and a partition-oriented ids avoids this extra work. Observe that only strongly decomposable programs have a partition oriented ids.

Schemes of DS2 are optimal, up to one partition-member\(^4\), for the following two reasons combined. First, the total work-load of all the processors is minimal, i.e. not higher than the work-load of one processor performing the evaluation single-handedly (appendix A). Second, all processors are busy until completion.

Strategy SS2 below extends the class of programs that can be evaluated in parallel without data-overhead, at the expense of relaxing disjointness. It is used for linear programs that are not strongly decomposable.

**Strategy SS2**: Is derived from SS1 as DS2 is derived from DS1. "Member" \( M_i \) consists of the set of output facts derived from the input, where the relation \( B \) is restricted to the set \( \{ B(i,c) \mid c \text{ is some constant } \} \). The processors evaluate the members in increasing order, with work redistribution occurring when a processor completes. []

4.2 General Strategies

The strategy DS3 below can be applied for parallelization of every program. Actually, DS3 is a family of parallelization strategies, one for each class of programs. The classes\(^5\) are defined in appendix D.

**Strategy DS3**: For the strongly decomposable programs this strategy is DS2. Consider any other program, \( P \). Assume that the head of the recursive rule is either \( S(x,y) \) or \( S(x,x) \). Let \( h \) be a hash function. Multiple processors cooperate in evaluating \( P \) as follows. Each processor, \( i \),

\[ h(y) = i \text{ or } h(x+y) = i \]

work as well.

\(^4\) This means that in any other scheme, say DSM, for some input, the last processor to complete may do so before the last processor of DS2 completes. But if so, then the last processor of DS2 trails the last processor of DSM by at most the time it takes to evaluate one partition-member in that input.

\(^5\) The classification in appendix D is related to DS3, as decomposability is related to DS1. In other words, that classification is an extension of the decomposability concept, and can be cast in terms of an output domain "cover", independently of DS3.
evaluates the S-facts in which \( h(x) = i \), by appending the evaluable predicate to the body of the exit and recursive rules\(^6\) (thus cutting the size of every relation that has \( x \) as an attribute). The evaluation is semi-naive or naive. At the completion of each iteration, processor \( i \) does two things. First, it adds to \( S \) (\( \Delta S \) if the evaluation is semi-naive) the set \( G \) of all tuples that were received from other processors during \( i \)'s last iteration. Note that \( G \) may be empty. Second, \( i \) sends to each other processor, \( j \), the subset \( T_{ij} \) of the set of new tuples that \( i \) computed during its last iteration. The subset \( T_{ij} \) may be proper, since some tuples that \( i \) computed may not help \( j \) produce its tuples, regardless of the input. For example, assume that the head of the recursive rule is \( S(x, y) \), and there is a single S-atom in the body, \( S(z, x) \). Then, regardless of the input, a fact \( S(c, d) \) such that \( j \neq h(d) \), cannot help \( j \) produce its S-facts. The formal definition of \( T_{ij} \) depends on the program \( P \), and can be found in appendix D. An important point to observe about DS3 is that the processors perform their computation completely asynchronously\(^7\). □

First, let us establish correctness (completeness is not trivial, since as explained, some S-facts are not transmitted).

**Theorem 4.1:** DS3 is a strategy of decomposition schemes. []

Intuitively, time-saving occurs in DS3 for the following reasons. First, for a given input, each one of the processors computes approximately \( 1/r \) of the output tuples. Second, whenever \( T_{ij} \) is a proper subset of the computed tuples, overhead as well as the evaluation cost of processor \( j \) (smaller \( S \)), are reduced. Third, \( i = x \mod r \) reduces the size of every relation having \( x \) as an attribute. What weighs against these arguments is the potential increase in the number of iterations. In appendix A we determine that for linear programs, the total evaluation-cost of all the processors participating in DS3 is minimal.

Next, we establish overhead minimality in the following sense. Let \( A \) be an \( r \)-scheme for partial computation of a program \( P \), and let \( I \) be an input to \( P \). Then the total number of transmitted facts in \( A_1(I), A_2(I), \ldots, A_r(I) \) is the **overhead of \( A \) for \( I \)**. Let a \( D \)-scheme be a decomposition scheme that for every input, computes the same facts as DS3 at every processor. In addition, we assume that in a \( D \)-scheme the following holds. If processor \( i \) transmits an intentional fact \( f \), that it computes for some input, to processor \( j \), then for every input for which processor \( i \) computes \( f \), it transmits \( f \) to processor \( j \) (otherwise additional work is necessary to determine when to transmit \( f \), and when not to do so).

**Theorem 4.2:** For every input, a scheme in DS3 has a minimal overhead, among all \( D \)-schemes. []

In other words, DS3 cannot transmit less facts than it actually does.

**Note:** DS3 can be easily extended to arbitrary datalog programs, provided that an

---

6. An analog of DS3, and consequently of the program-classification in Appendix D, exists for the second variable in the head of the recursive rule (use the predicate \( I = h(y) \)).

7. For the sake of brevity, we omit description of the distributed termination protocol.
algorithm sends all the new tuples computed at each iteration, to all the other processors. In conclusion, the properties of the strategies devised in this paper are summarized in the table of Figure 2.

5. CONCLUSION AND FUTURE WORK

In this paper we first defined the notions of decomposability and strong decomposability, and provided a complete characterization of all the unary and binary two-rule programs, with respect to both notions. Then we examined pure parallelization, and showed that the programs that can be purely parallelized with minimal total evaluation cost, are exactly the strongly decomposable ones. Strategy DS1 can be used for this purpose. All linear programs can also be purely parallelized (strategy SS1), but not at minimal cost. Although strategy DS1 has minimal total cost, this cost may not be evenly balanced among the processors. Strategy DS2, that is not pure but incurs only control-overhead, overcomes this limitation for the strongly decomposable programs. It is in some sense optimal. Strategy SS2 is an adaptation of SS1 to balance the load, for linear programs. Finally, we proposed a class of strategies, DS3, one of which can be used for parallelization of every program. Strategies of DS3 incur data-overhead, but it is in a sense minimal; also, DS3 has minimal total evaluation cost, for the linear programs.

An obvious future-research direction is to examine applicability of the proposed parallelization strategies to all Datalog programs, and other declarative languages. At this point let us observe that some strategies are applicable in conjunction with the magic sets method (see [BMSU]). For example the same generation program produced by the method in response to a query is:

\[
\begin{align*}
\text{MAGIC (xp)} :&= \text{MAGIC (x)}, \text{PARENT (x, xp)} \\
\text{MAGIC (a)} :&= \\
\text{SG (x, y)} :&= \text{H (x)} \\
\text{SG (x, y)} :&= \text{MAGIC (xp)}, \text{PARENT (x, xp)}, \text{PARENT (y, yp)}, \text{SG (xp, yp)} \\
\end{align*}
\]

Then schemes SS1, SS2, and DS3 can be applied in the evaluation of the program.

Acknowledgement:
We thank Michael Kiffer, Oded Shmueli, and Victor Vianu for helpful discussions and comments about the material in this paper. Oded Shmueli also suggested the method of communication among the processors used in DS3.
6. REFERENCES


Figure 1: A hierarchy of logic program classes. The strongly decomposable programs are most amenable to pure parallelization. A representative of this class is the transitive closure program: $S(x,y) :- S(x,z), A(z,y)$. Next in the hierarchy, is the class of sharable programs. A representative of this class is the canonical strongly linear program: $S(x,y) :- U(x,z), S(z,w), DOWN(w,y)$. Finally, the class of nonsharable programs is not amenable to pure parallelization. A representative of this class is the path-systems program: $S(x) :- S(z), S(y), H(x,z,y)$. For completeness, we also show the class of decomposable programs. Point A represents the program: $A(x,y) :- S(x,x), S(y,y)$. Point B represents the program $S(x,x) :- S(y,y), S(z,z), H(x,y,z)$.

<table>
<thead>
<tr>
<th>Applicable to Programs</th>
<th>Overhead</th>
<th>Load Distribution</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>strongly decomposable</td>
<td>no overhead</td>
<td>static</td>
</tr>
<tr>
<td>SS1</td>
<td>linear</td>
<td>no overhead</td>
<td>static</td>
</tr>
<tr>
<td>DS2</td>
<td>strongly decomposable</td>
<td>control overhead</td>
<td>dynamic</td>
</tr>
<tr>
<td>SS2</td>
<td>linear</td>
<td>control overhead</td>
<td>dynamic</td>
</tr>
<tr>
<td>DS3</td>
<td>all</td>
<td>minimal data overhead</td>
<td>static</td>
</tr>
</tbody>
</table>

Figure 2
APPENDIX A: Cost-analysis of the Proposed Strategies

In this appendix we determine that for some schemes in DS1, DS2, and DS3, the total amount of work performed by the processors executing the scheme is minimal, i.e., not higher than the amount of work done by a single processor, for the same evaluation. Additionally, we show that for some schemes in SS1 and SS2 any participating processor does not do more than the amount of work done by a single processor, for the same evaluation. We shall consider two cost measures to quantify the amount of work performed by an algorithm for evaluating a program \( P \) given an input \( I \). The first, denoted \( \text{cost}^1 \), assumes that the cost of an iteration of naive or semi-naive evaluation is \( c_1 \cdot n \) or \( c_2 \cdot n \), respectively; \( n \) is the number of tuples in the relation \( S \) at the beginning of the iteration. \( c_1 \) and \( c_2 \) are fixed for a given input, but depend monotonically on the size of the input. Then, \( \text{cost}^1 \) of evaluating \( O(P, I) \) is the total cost of all the iterations performed during the evaluation. The second cost measure, denoted \( \text{cost}^2 \), is the number of "successful instantiations of rules" performed during the evaluation. The measure was introduced and justified in [BR].

Theorem A.1: Let \( P \) be a strongly decomposable program, let \( h \) be a hash function, and consider a scheme of DS1 that consists of algorithms \( A_1, \ldots, A_r \); each \( A_i \) is the naive evaluation of the program \( P_i \), which is \( P \) with \( h(x) = i \) appended to the exit and recursive rules. Denote by \( A \) the naive evaluation of \( P \). Then, by each one of the measures \( \text{cost}^1 \) and \( \text{cost}^2 \), for each input, \( \text{cost}^1(A) \geq \text{cost}^1(A_1) + \cdots + \text{cost}^1(A_r) \). Furthermore, the above holds for semi-naive rather than naive evaluation, and also for a scheme in DS2 rather than DS1. []

Let us observe that Theorem A.1 holds even if \( \text{cost}^1 \) is a superlinear function of \( n \) (e.g. \( n \log n \)); actually the inequality in its strict form holds. Another observation is that theorem A.1 is in a sense confirmed empirically in [AJ]. There, scheme DS1 was independently proposed for the transitive closure program, and tested on a random graph with 250 nodes. The speedup reported is linear in the number of processors.

Now we state the result about strategy DS3:

Theorem A.2: Let \( P \) be a linear program, and consider a scheme of DS3 that consists of algorithms \( A_1, \ldots, A_r \); each \( A_i \) performs semi-naive evaluation in step (2). Denote by \( A \) the semi-naive evaluation of \( P \). Then, for each input, \( \text{cost}^2(A) \geq \text{cost}^2(A_1) + \cdots + \text{cost}^2(A_r) \). []

Clearly, disjointness of the schemes in DS1, DS2, and DS3 is essential for proving theorems A.1 and A.2, and their results do not apply to SS1 and SS2. However, about these schemes we can establish:

Theorem A.3: Let \( P \) be a linear program, let \( h \) be a hash function, and let a scheme in SS1 consist of algorithms \( A_1, \ldots, A_r \); each \( A_i \) is the semi-naive evaluation of the program \( P_i \), which is \( P \) with \( h(x) = i \) appended to the exit rule. Denote by \( A \) the semi-naive evaluation of \( P \). Then, for each input, \( \text{cost}^2(A) \geq \max_i [\text{cost}^2(A_i)] \). The above also holds for a scheme in SS2. []
APPENDIX B: Discriminating Programs

In this appendix we define a discriminating program, and provide an algorithm for determining whether or not a program is discriminating. The definition, in contrast to the others in this paper, is not entirely syntactic. Let \( P \) be a program, and denote the recursive rule of \( P \) by \( r \). For an input \( I \) denote by \( I(S) \) the set of \( S \)-facts obtained by substituting in \( I \) the predicate symbol \( S \) for \( B \). In other words, \( I(S) \) is obtained by selecting from \( I \) the \( B \)-facts and replacing the \( B \) predicate symbol by \( S \). For an input \( I \) denote by \( r(I) \) the set of \( S \)-facts which are in \( O(P,I) - I(S) \). For an input \( I \) denote by \( I^* \) the input obtained by eliminating from \( I \) all the \( B \)-facts that have the same constant in both positions, e.g., \( B(a,a) \). A program is discriminating if the following two conditions are satisfied: 1) the head of the recursive rule has distinct variables, and if we denote the head atom of the recursive rule by \( S(x,y) \), then there is an atom \( S(y,x) \) in the body, and 2) for each input \( I \), \( r(I) = r(I^*) \). Next we provide an algorithm for determining whether or not a program \( P \) is discriminating.

Algorithm B.1:

1. If \( r \) has an \( S \)-atom with a repeated variable (in the head or the body), then \( P \) is not discriminating. Halt.
2. Denote the head of \( r \) by \( S(x,y) \). If \( S(y,x) \) is not in the body of \( r \), then \( P \) is not discriminating. Halt.
3. Denote by \( p_1 \cdots p_k \) the partitions of the variables in \( r \). A partition of the variables is a set of pairwise disjoint subsets, such that each variable is in some subset. For each partition \( p_i \) do:
   Consider an instantiation \( f \) of \( r \), in which every two variables are mapped into different constants if and only if they are in different subsets of the partition. Let \( I \) be the set of facts in the body of the instantiated \( r \), except that the \( S \)-predicate symbol is replaced by the \( B \)-predicate symbol. If the fact in the head of the instantiated \( r \) is neither in the instantiated body nor in \( r(I^*) \), then \( P \) is not discriminating. Halt.
4. \( P \) is discriminating.

Theorem B.1: Algorithm B.1 correctly determines whether or not a program is discriminating.

APPENDIX C: Parallelization Schemes

Let \( P \) be a program, and let \( I \) be an input to \( P \). A partial computation, denoted \( c(I) \), is a sequence of facts. The predicate symbols in \( c(I) \) are taken from \( P \), and a fact does not appear
more than once in \( c(I) \). Each intentional fact in the sequence is either \textit{computed} (blue) or \textit{transmitted} (red). The sequence \( c(I) \) satisfies the following two conditions. First, each extensional predicate fact in \( c(I) \) must be in \( I \). Second, every computed fact, \( a \), in \( c(I) \), is in \( O(PJ) \), and is preceded by all facts of some derivation tree of \( a \). A fact of this derivation tree may be either computed or transmitted. The sequence \( c(I) \) represents the order in which the output of \( P \) is evaluated, and is called a "partial" computation, since not all facts of \( O(PJ) \) have to be in \( c(I) \). A transmitted fact is received from another processor, thus a derivation tree does not necessarily precede it. For example, naive evaluation by a single processor produces a partial computation consisting of the input facts (in some order), followed by all the output facts, in the order in which they are evaluated; all the facts are computed.

The partial computation \( c(I) \) \textit{computes less} if there is some fact \( b \in O(PJ) \) that cannot be obtained by instantiating an exit rule, and that is not computed in \( c(I) \). If \( c(I) \) computes all, except some facts which can be obtained by instantiating an exit rule, then it does not do less work than computing the whole set \( O(PJ) \). Let \( C = (c_1(I), \ldots, c_r(I)) \) be a set of partial computations. Input \( I \) is \textit{time-saving} for \( C \) if each sequence in \( C \) computes less. The set \( C \) is \textit{complete} if for each fact, \( a \in O(PJ) \), there is a partial computation \( c_j(I) \) in \( C \), such that \( a \) is in \( c_j(I) \). The set \( C \) is \textit{disjoint} if for every pair of computations in \( C \), \( c_i(I) \) and \( c_j(I) \), for \( j \neq i \), there is no intentional predicate fact which is computed in both, \( c_i(I) \) and \( c_j(I) \).

An \textit{r-parallelization-scheme} for partial computation of \( P, A \), is a function which maps each input, \( I \), into \( r \) partial computations, \( A(I) = \{A_1(I), \ldots, A_r(I)\} \), such that if some fact is transmitted in some \( A_j(I) \), then it is computed in some \( A_j(I) \). \( A \) is called an \textit{r-scheme} for short. The set of all sequences with subscript \( i \) constitutes the (output of) \textit{algorithm} \( A_i \) of \( A \). The algorithm is executed by \textit{processor} \( p_i \). If \( A(I) \) is complete for every \( I \), and is time-saving for some input, then \( A \) is a \textit{sharing} \( r \)-scheme. If in addition \( A(I) \) is disjoint for every input, then \( A \) is a \textit{decomposition} \( r \)-scheme. A complete scheme does not lose output, and, assuming that a certain amount of work is necessary to produce each output fact, processors executing (the algorithms of) a disjoint scheme do not duplicate one another's work (see appendix A). Existence of a time saving input simply ensures that the scheme is not trivial, i.e., does not have a single-processor evaluation algorithm.

A partial computation is \textit{independent} if it does not contain any transmitted facts (i.e. all the intentional facts are computed). Let \( A \) be an \( r \)-scheme. If for every input \( I \), each computation in \( A(I) \) is independent, then \( A \) is an \textit{independent scheme}. Independence ensures that facts are not transmitted between algorithms, i.e. there is no data overhead. In section 3 we discuss only independent partial computations, and the rest of the definitions refer to this property. An independent scheme, \( A \), is \textit{data driven} if for each input \( I \), and for each fact \( b \in O(PJ) \), and for

---

8. Actually, additional requirements have to be satisfied by the set \( A(I) \), but we omit them since they are not used in this paper.
each set of input facts, $Z$, the following two conditions are satisfied for each algorithm $A_i \in A$:

1. (contribution) If $b \in A_i(I)$ and the set of derivation trees of $b$ for the input $I \cup Z$ is a superset (not necessarily proper) of the set of derivation trees of $b$ for the input $I$, then $b \in A_i(I \cup Z)$.

2. (noncontribution) If $b \not\in A_i(I)$, and the set of derivation trees of $b$ for the input $I$ is a superset (not necessarily proper) of the set of derivation trees of $b$ for the input $I \cup Z$, then $b \not\in A_i(I \cup Z)$.

The contribution requirement is simply that if the fact $b$ is in $A_i(I)$, and $Z$ contributes to the derivation of $b$, then its addition to $I$ cannot suppress the production of $b$. Note that if $A_i$ is monotonic, then the contribution requirement is satisfied, but if stratified negation is allowed, $A_i$ is not monotonic but may still satisfy the contribution requirement. The noncontribution requirement is that if the fact $b$ is not in $A_i(I)$, and the set $Z$ does not "contribute" to the derivation of $b$ (i.e., there is no derivation tree which contains a fact in $Z$), then $b$ is also not in $A_i(I \cup Z)$. The fact that a scheme is data driven ensures that the output of each processor depends solely on the input, and not on communication with another processor; in other words, there is no control overhead.

Let $A$ be an independent, data-driven $r$-scheme for partial computation of $P$. $A$ is called a pure parallelization scheme, or, for short, a pure scheme. Such a scheme does not incur the overhead of communication among the algorithms.

The definition of a decomposition scheme required the existence of one time saving input but as a result of Theorem 3.2 we obtain

**Corollary C.1**: If a program has a pure decomposition scheme, then it has a pure decomposition scheme with an infinite set, $R$, of time-saving inputs. Furthermore, any pair of inputs in $R$ is disjoint. []

### APPENDIX D - The Transmission Sets in Strategy DS3

In this appendix we define the transmission sets, $T_{ij}$, that are sent between processors by the algorithms of strategy DS3. The set of tuples $T_{ij}$ has to be transmitted from processor $i$ to processor $j$, when $i$ completes each iteration of naive or semi-naive evaluation. $T_{ij}$ is a subset of the set $R_i$ of tuples that processor $i$ computed in its last iteration. If the program is unary, then $T_{ij}$ is the whole set $R_i$. For binary programs, in many cases less tuples have to be transmitted. Specifically, $T_{ij}$ depends on the type of binary program, denoted $P$, hence we first have to define several types of such programs. Denote the recursive rule of $P$ by $r$ and let the first variable in its head be $x$.

A program is **first-consistent** if every S-atom in the body of $r$ contains the variable $x$. A program is **partially-first-consistent** (partially-first-fixed) if the removal of all the S-atoms with repeated
variables from the body of \( r \), leaves a first-consistent (first-fixed or an empty \( r \)-body) program. Now we define an almost-discriminating program. \( P \) is \textit{almost-discriminating} if for any instantiation \( f \) of \( r \), in which the instantiated head is not in the instantiated body the condition C.1 below holds. Let \( I \) be the set of facts in the body of the instantiated \( r \), except that the \( S \)-predicate symbol is replaced by the \( B \)-predicate symbol. Denote by \( I^+ \) the input obtained from \( I \) by eliminating all the \( B \)-facts with the same constant in both positions, except that if \( B(f(x), f(x)) \) exists in \( I \), it is not removed. Then the condition C.1 is that \( O(P, I^+) \) contains the head of the instantiated rule. The algorithm that decides whether or not a program is almost-discriminating is the following (similar to algorithm B.1 in Appendix B):

1. If there is an \( S \)-atom in the body of \( r \) with a repeated variable other than \( x \), then \( P \) is not almost-discriminating. Halt.
2. Denote by \( p_1 \cdots p_k \) the partitions of the variables in \( r \). A partition of the variables is a set of pairwise disjoint subsets, such that each variable is in some subset. For each partition \( p_i \) do:
   Consider an instantiation \( f \) of \( r \), in which every two variables are mapped into different constants if and only if they are in different subsets of the partition. Let \( I \) be the set of facts in the body of the instantiated \( r \), except that the \( S \)-predicate symbol is replaced by the \( B \)-predicate symbol.
   If the instantiated head is neither in the instantiated body nor in \( O(P, I^+) \) then \( P \) is not almost-discriminating. Halt.
3. \( P \) is almost-discriminating.

The definition of \( T_{ij} \) is given using the flow-chart in Figure D.1. Note that \( T_{ij} \) should be determined only once, at the beginning of each algorithm in DS3.
Figure D.1:  
A is the set of tuples which, when applying the hash function to their second constant, the result is $j$.  
B is the set of tuples with the same constant in both positions.  
C is the set of all tuples that have distinct constants in their positions.  
D is the set of all tuples computed, i.e., $R_i$.  

APPENDIX E  
This appendix demonstrates by examples the execution of the proposed strategies. Performance issues are also addressed. We assume that each strategy uses two processors, $P_0$ and $P_1$. The evaluation method of each processor is semi-naive.  

Strategy DS1: Consider the following strongly decomposable program (the transitive closure):  
$S(x,y):-S(x,z),A(z,y)$.  
$S(x,y):-A(z,y)$.  
Let the input to the program consist of the arcs of the graph in Figure E.1.  

Figure E.1: Sample input to the transitive closure  
Consider strategy DS1, in which $P_0$ semi-naively evaluates a program with the predicate
x mod 2 = 0 appended to the exit and recursive rules, and P1 does so with x mod 2 = 1 appended to them. Both, P0 and P1 perform four iterations. The differentials at the beginning of each iteration of P0 are: { (2,3), (4,1) } { (2,4), (4,2) } { (2,1), (4,3) } { (2,2), (4,4) } , respectively. Similarly, the differentials of P1 are of size two at each iteration. Note that in a single-processor evaluation the number of iterations is also four, but the differentials used in each iteration are of size four each.

**Strategy SS1:** For this strategy we use the canonical strongly linear (csl) program:

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

\[ S(x,y) := \text{FLAT}(x,y) \]

The extensional-database relations UP, FLAT, and DOWN represent a directed graph with three types of arcs. The csl program defines a tuple \((a,b)\) to be in \(S\), iff there is a path from \(a\) to \(b\) having \(k\) UP arcs, one FLAT arc, and \(k\) DOWN arcs, for some integer \(k\).

Let the input to csl be the extensional database relations of Figure E.2. UP is the relation \([(1,2),(2,3),(3,4),(4,5)\]}, DOWN is the relation \([(6,7),(7,8),(8,9),(9,10)\]}, and FLAT is the relation \([(i,6) | i = 1, \ldots, 5]\}.

![Figure E.2: Sample input to the csl program.](image)

In strategy SS1, P0 (P1) evaluates csl with the predicate \(x \mod 2 = 0\) (\(x \mod 2 = 1\)) appended to the exit rule only. The differentials at the beginning of each iteration of P1 (which performs worse than P0) are: \([(1,6),(3,6),(5,6)\]} \([(4,7),(2,7)\]} \([(3,8),(1,8)\]} \([(2,9)\]} \([(1,10)\]}, respectively.

A single processor performs five iterations, each iteration having a differential of greater size. A comparison of performances is summarized in Figure E.4.

**Strategy DS3:** To demonstrate DS3 we use the same program and input as for SS1, i.e., csl with the input depicted in Figure E.2. For simplicity we assume that both processors finish each
iteration of the semi-naive evaluation at the same time, and then messages exchange occurs. This assumption is justified by the fact that in this example both processors do approximately the same amount of work at each iteration.

Next we explain the example, and figure E.3 which summarizes it. P0 evaluates csl with the predicate $x \mod 2 = 0$ appended to the exit and recursive rules, therefore, it starts with the differential $\{(2,6),(4,6)\}$. P1 appends the predicate $x \mod 2 = 1$ and starts with the differential $\{(1,6),(3,6),(5,6)\}$. After the first iteration both processors reach a temporary fix-point, and P0 transmits the set $\{(2,6),(4,6)\}$ while P1 transmits the set $\{(1,6),(3,6),(5,6)\}$. Each processor adds the received set of tuples to its present differential. P0 obtains $\{(1,6),(3,6),(5,6)\}$ and P1 obtains $\{(2,6),(4,6)\}$. After the second iteration P0 obtains the differential $\{(2,7),(4,7)\}$ and P1 obtains $\{(1,7),(3,7)\}$. As result of messages-exchange the differentials of both processors become the same. Actually after all the following messages-exchange, the differentials are the same. Iteration 3 ends with the differential of P0 being $\{(2,8)\}$ and the differential of P1 being $\{(1,8),(3,8)\}$. The rest of the evaluations are in the following table. A final fix-point is reached after the sixth iteration.

<table>
<thead>
<tr>
<th>iteration</th>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sizes of UP,ΔS,DOWN</td>
<td>the differentials ΔS</td>
</tr>
<tr>
<td>1</td>
<td>2,2,4</td>
<td>(2,6)(4,6)</td>
</tr>
<tr>
<td>1.1</td>
<td>∅</td>
<td>(1,6)(3,6)(5,6)</td>
</tr>
<tr>
<td>2</td>
<td>2,3,4</td>
<td>(2,7)(4,7)</td>
</tr>
<tr>
<td>2.1</td>
<td>2,4,4</td>
<td>(2,7)(4,7)(1,7)(3,7)</td>
</tr>
<tr>
<td>3</td>
<td>2,3,4</td>
<td>(2,8)</td>
</tr>
<tr>
<td>3.1</td>
<td>(2,9)</td>
<td>(2,9)(1,9)</td>
</tr>
<tr>
<td>4</td>
<td>2,2,4</td>
<td>(1,10)</td>
</tr>
<tr>
<td>4.1</td>
<td>∅</td>
<td>(1,10)</td>
</tr>
</tbody>
</table>

Figure E.3: csl execution by strategy DS3. In a line marked by iteration $i$ we specify the differentials at the beginning of the $i$-th iteration and after messages-exchange. In a line marked by $i.1$ we specify the differentials at the end of the $i$-th iteration, and before the messages-exchange.

To compare the performance of SS1, DS3 and the sequential strategy, we summarize the relation sizes at each iteration of each strategy, in the table of Figure E.4. The conclusions from this comparison are as follows. For each one of the strategies, SS1 and DS3, the hardest-working processor performs better than the single processor. In SS1, P1 that works harder, has the same
number of iterations as a single processor, and at three of the five iterations the size of the differential is approximately half the size of the single-processor's differentials. In DS3, P0 and P1 perform six iterations (five for the single processor), at each iteration the size of UP is half the size in the single processor case.

Finally, observe that for DS3 the cs1 program is a worst-case example in two respects. First, the program classification that enables less tuples to be transmitted between processors, thus reducing overhead and evaluation-cost, does not help in the cs1 case. Second, the size-cutting variable x, appears in only one relation.

<table>
<thead>
<tr>
<th>iteration</th>
<th>single-processor</th>
<th>SS1</th>
<th>DS3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>processor 0</td>
<td>processor 1</td>
<td>processor 0</td>
</tr>
<tr>
<td>1</td>
<td>4,5,4</td>
<td>4,2,4</td>
<td>4,3,4</td>
</tr>
<tr>
<td>2</td>
<td>4,4,4</td>
<td>4,2,4</td>
<td>4,2,4</td>
</tr>
<tr>
<td>3</td>
<td>4,3,4</td>
<td>4,1,4</td>
<td>4,2,4</td>
</tr>
<tr>
<td>4</td>
<td>4,2,4</td>
<td>4,1,4</td>
<td>4,1,4</td>
</tr>
<tr>
<td>5</td>
<td>4,1,4</td>
<td>4,1,4</td>
<td>4,1,4</td>
</tr>
<tr>
<td>6</td>
<td>2,1,4</td>
<td>2,1,4</td>
<td>2,1,4</td>
</tr>
</tbody>
</table>

Figure E.4: Performance comparison. The table entries consist of three numbers, for the size of relations UP, the differential $\Delta S$, and DOWN respectively.