WHY A SINGLE PARALLELIZATION STRATEGY IS NOT ENOUGH IN KNOWLEDGE BASES

Simona Cohen, Ouri Wolfson

Columbia University
Dept. of Computer Science

Technical Report CUCS-015-90

WHY A SINGLE PARALLELIZATION STRATEGY IS NOT ENOUGH IN KNOWLEDGE BASES

Simona Rabinovici Cohen
Ouri Wolfson [1]

Department of Computer Science
The Technion - Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

We address the problem of parallelizing the evaluation of logic programs in data intensive applications. 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 several parallelization strategies based on the concept of data-reduction - the original logic-program is evaluated by several processors working in parallel, each using only a subset of the database. The strategies differ on the 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. The relationship to the NC complexity class is demonstrated. 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] Present address: Department of Computer Science, Columbia Univ., New York, NY 10027, USA.

Research was supported in part by the Foundation for Research in Electronics, Computers and Communications, in Israel, by DARPA Research Grant #F-29601-87-C-0074, and by the Center for Advanced Technology at Columbia University under contract NYSSTF-CAT(89)-5.

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. An example of a recursive Datalog program, consisting of two rules, is the following:

$$T(x,y) :- T(x,z),\ A(z,y)$$
$$T(x,y) :- A(x,y)$$

It computes the transitive closure of the relation A. The program is evaluated in a set-oriented fashion by initializing the relation T to A, and then iteratively adding to T the new tuples obtained by joining (or composing) the relations T and A.

At the heart of our present 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 some fundamental parallelization properties. The most powerful property is strong-decomposability. It enables the evaluation to be separated into any number of completely independent tasks, that can be carried out in parallel. Program classes that enable partial independence of the evaluation tasks, are also introduced (section 6).

The other part of the paper is devoted to parallelization strategies, i.e., sets of algorithms cooperatively evaluating a program. We propose several classes of parallelization strategies, and analyze their properties and limitations. The parallelization strategies proposed are based on the data-reduction principle; each processor evaluates the original logic-program, but using only a subset of the database. This principle underlies parallelization in many domains, including computer vision and vector-computing. In this paper we demonstrate its application to logic programming. For example ([WS]), the transitive closure program presented above can be evaluated in parallel, by having one processor start from the even nodes, thus computing the tuples of the relation T in which the first component is even, and the second processor start from the odd nodes. This can be done if one processor evaluates the program having the predicate *even*(x) appended to the body of both rules, and the other evaluates the program with *odd*(x) appended. The performance of such methods is analyzed in this paper.

The strategies analyzed in this paper differ in their evaluation cost, overhead, and classes of programs to which they can be applied. We postulate that the performance of a parallelization strategy depends on these two factors: the total evaluation cost, and the overhead of communication and synchronization among the algorithms of the strategy [2].

[2] As often demonstrated (e.g. [LY, DIY]), communication overhead limits the potential gains in performance by parallelization. For example, the parallel evaluation of the transitive closure, described above, does not need any communication between the two processors.

We first formally define and study pure parallelization, i.e., parallelization without overhead. We focus on pure parallelization strategies of minimal evaluation cost, and show that a program can be parallelized by such a strategy, if and only if it is strongly decomposable. By cost-minimality we mean that the total evaluation cost of all the processors working in parallel, is not higher than the cost of a single processor performing the evaluation single-handedly; and this holds for every input of the logic program. The distribution of the work to processors is static, i.e. independent of the input to the datalog program, and is performed by a hash function. If it distributes the workload evenly, then minimal total evaluation cost translates into optimal speed-up. Our result is obtained using three different evaluation cost measures. One, "the number of successful inferences", was introduced by Bancilhon and Ramakrishnan ([BR]). The others are introduced in this paper.

The results of this paper combined with the results in [W], demonstrate an interesting class structure of programs, with respect to pure parallelization. Most amenable is the class of strongly-decomposable programs. A program in this class can be purely parallelized, with minimal total evaluation-cost. Next is the class of sharable programs. Such a program can be purely parallelized, but the evaluation cost may not be minimal. Finally, the class of nonsharable programs cannot be purely parallelized. The relationship between these classes of programs, and the programs in the NC complexity class is demonstrated. The class of NC programs intersects the class of sharable and strongly decomposable programs, but is neither contained in, nor contains, any of them.

Next we consider strategies 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. We propose an independent parallelization strategy, i.e. a strategy that incurs control- but not data-overhead. It is still restricted in applicability to the strongly decomposable programs, but it has minimal total evaluation cost and it balances the evaluation work-load among the processors. Load-balancing is dynamic: when a processor becomes idle it takes work from the other processors. Finally, we introduce a strategy, called DS3, that can be applied to parallelize all programs, and it incurs minimal data-overhead. DS3 also has minimal evaluation-cost for the linear programs.

The parallelization strategies 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 (such a program is called single rule program, or sirup, in [CK]). 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], [K], [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. If the number of processors is constant, then the NC-type of evaluation algorithms can be adapted by assigning the work of multiple processors to a single processor. However, it turns out that, which multiple processors are assigned to the single one, is very important as far as overhead (particularly if the processors do not have shared memory) and evaluation cost are concerned. The work in this paper can be regarded in some sense, as the study of this issue - how to partition the work among the processors.

An approach, called horizontal partitioning, is taken in the parallelization of production systems ([IS,M,St,TM]). It partitions the rules among the processors, and each processor evaluates its own set of rules, while communicating with the other processors. The data-reduction approach advocated here, is orthogonal to horizontal partitioning (perhaps should also be called vertical partitioning). Data-reduction partitions the data (or some of it) rather than the rules. However, a variant of data reduction, named "copy and constrain", was proposed independently in the production-system literature ([SMM]), and its merit was demonstrated experimentally using OPS5 ([P]). But, the topics of this paper, namely program classification and parallelization strategies, have not been addressed previously.

Another body of relevant research has been performed on parallel and concurrent variations of PROLOG ([DL]). 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 whole sequence of relational operations, performed in all the iterations, rather than each individual operation. In some sense, the work in this paper amounts to studying the parallelization of a sequence of relational operations.

Finally, [W] and [WS] 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. [W] also formally defined pure parallelization in terms of algorithms that evaluate a program in parallel, and studied the class of sharable programs. This paper extends that work in several ways. First, it demonstrates that there is something fundamental about decomposability, independent of the parallelization methods proposed in [W] and [WS]. This fundamental property, partitioning of the output domain, is introduced in section 3 of this paper. In fact, we show that there are two notions, decomposability and strong decomposability. We provide a complete characterization of the single rule programs with respect to both, and show their relationship to parallelization strategies with certain desirable properties. Second, in the present paper we analyze the evaluation cost of parallelization strategies. Third, we propose and analyze strategies with communicating algorithms, that overcome the limitations of pure parallelization.

The rest of this paper is organized as follows. In the next section we provide main definitions used in the rest of the paper. In section 3 we study decomposability, and in section 4 we study pure parallelization, and analyze its cost and its limitations. In section 5 we introduce control-overhead for the purpose of load balancing. In section 6 we discuss the general parallelization strategy, DS3, and related program classes. We conclude and discuss future research in section 7.

2. PRELIMINARIES

In this section we define the basic terminology as well as provide some relevant definitions. The Datalog language has three building blocks: predicate symbols, variables and constants. With each predicate symbol is associated a fixed *arity*. A predicate symbol with arity k followed by a list of k arguments is a *literal*. 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). We shall denote constants by lowercase letters from the beginning of the alphabet, a through n, and variables by lowercase letters from the end of the alphabet, w, x, y, z. Predicate symbols are denoted by uppercase letters.

If an atom has a constant in each argument position, then it is a *fact*. An R*-atom* is an atom having R as the predicate symbol. An atom has a *repeated variable* if it has a variable that appears in more than one argument position. A *rule* consists of an atom, Q, designated as the *head*, and a finite set 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." A variable that appears in the head of the rule is called a *distinguished* variable. An *instantiation* of a rule, or a set of atoms, H, is a function that maps each variable in H to a constant. If H is a set, and f is an instantiation of it, then the *instantiated set*, denoted $H \cdot f$ is the set of facts obtained by replacing the variables in H according to f. If H is a rule of the form head :- body, then the *instantiated rule*, $H \cdot f$, consists of two sets: *head*$\cdot f$, the atom *derived* by f, and *body*$\cdot f$. When no confusion can arise, the instantiated rule, $H \cdot f$, rather than denoting two sets, shall denote only one set: *head*$\cdot f \cup$ *body*$\cdot f$.

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 restrict our discussion to datalog programs with one intentional predicate, denoted S, that is unary or binary.

Furthermore, programs do not have any constants, and each one consists of two rules: an exit rule, denoted $S(x,y)$:-$B(x,y)$ 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 $O(P,I)$, 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 instantiated rule which has b as the head and $b_1, \ldots, b_k$ as the body. For self containment of this paper, we describe in appendix C the most popular, bottom-up, serial algorithm that produces the output of a program, namely semi-naive evaluation ([Ban,Bay]). Given an input I to a program, an instantiation f of the recursive rule is *useless* if (1) the atom *head·f* has a derivation tree of height one (representing the instantiated exit rule), or (2) *head·f*∈ *body·f*. A derivation tree for a fact is *free from useless instantiations* if none of its instantiations is useless. A derivation tree with useless instantiations can always be replaced by a smaller tree; thus this kind of tree is not interesting, and whenever we refer to a derivation tree, we assume that it is free from useless instantiations.

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 (i.e. a program that produces the same output for every input). Sagiv provides a polynomial-time algorithm that minimizes a given sirup ([Sa]). Let A,B be two predicates symbols, and H a set of facts. Then, an *A-to-B substitution of H* is the set of facts obtained by replacing every occurrence of the predicate symbol A in H, by the predicate symbol B. For example, S-to-B substitution of the set $\{S(1,1), A(1,2), S(2,1)\}$, is the set $\{B(1,1), A(1,2), B(2,1)\}$. The following theorem is an immediate consequence of Sagiv's algorithm.

Theorem 2.1 [Sa]: Let P be a program (minimal of course), and f a 1-1 instantiation of the recursive rule. Let I be a S-to-B substitution of *body·f*, and I' an arbitrary nonempty subset of I. Then *head·f* is not in $O(P, I-I')$. □

In other words, the theorem says that if we take a 1-1 instantiation of the recursive rule, eliminate at least one atom of it, and feed the resulting set as an input to P, then the head of the instantiated rule cannot be obtained.

3. CHARACTERIZATION OF (STRONGLY) DECOMPOSABLE PROGRAMS

In this section we study the notion of decomposability. If a program is decomposable, it means that its output domain, i.e., the infinite set of possible output tuples, can be partitioned such that the following condition is satisfied. For each input, each intentional fact, a, has a derivation tree in which all the intentional facts belong to the same partition-member as a. In other words, the evaluation of the decomposable programs can be partitioned a priori into a number of completely independent tasks, each working on a disjoint set of partition-members. As we shall explain in the first subsection, the decomposability 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.

3.1 Definitions and Complete Characterization of Unary Programs

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 $\{ S(a) \mid a$ 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 $O(P,I)$, 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 (such an algorithm, e.g. semi-naive evaluation in appendix C, consists of iteratively evaluation the output, when at each iteration, a join involving the relations S and/or ΔS is performed). 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 $P1$:

$$P1: \qquad S(x,y) :- S(x,z),\ A(z,y)$$
$$S(x,y) :- B(x,y)$$

As we shall see it is decomposable, and assume that it is semi-naively evaluated. If at some iteration the differential ΔS 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.

The next lemma is widely referred to in the proofs of this section concerning unary, as well as binary programs. We define two S-facts to be *neighbors with respect to P*, if there is a 1-1 instantiation f of the recursive rule of P, such that $S(\vec{e}_1)$, $S(\vec{e}_2) \in$ *rule·f*.

Lemma 3.1: If a program P has an eligible partition D, then every two neighbors with respect to P are in the same member of D.

Proof: Assume that there is a 1-1 instantiation, f, of the recursive rule of P such that $S(\vec{e}_1), S(\vec{e}_2) \in$ *rule·f* but $S(\vec{e}_1)$, $S(\vec{e}_2)$ are not in the same member of D. Let I be the input I = S-to-B substitution of *body·f*. Suppose that *head·f* is in member M_i of D. At least one of the facts $S(\vec{e}_1)$, $S(\vec{e}_2)$ is in *body·f* and is not in M_i. By the definition of an eligible partition D, *head·f* has a derivation tree in which all the S-facts are in M_i. If $B(\vec{e}_1)$ is a node in this tree, then $S(\vec{e}_1)$ must also be a node in the tree (remember that predicate symbol B does not appear in the recursive rule). Therefore, *f·head* is in $0(P, I - \{B(\vec{e}_1)\})$. Contradiction to Theorem 2.1. □

Next we characterize the decomposable programs. First the unary programs and then the binary ones.

Theorem 3.1: A unary program is not decomposable.

Proof: Let P be an unary program in which $S(x)$ is the head of the recursive rule. Assume that P has an eligible partition D. We shall exhibit that every two S-facts in the output domain are in the same member of D, and therefore D is not a partition. Let $S(a), S(b)$ be two S-facts in the output domain. Note that there exists an S-atom in the body of the recursive rule, in which the argument is a variable different than x, say y. Such an atom exists since (1) P is recursive, so there is at least one S-atom in the body of the recursive rule, and (2) P is minimal, so $S(x)$ is not in the body of the recursive rule. Let f be a 1-1 instantiation of the recursive rule of P, in which x is instantiated to 'a' and y is instantiated to 'b'. By Lemma 3.1, $S(a), S(b)$ are in the same member of D. □

3.2 Sufficient Conditions for Decomposability

For the rest of this section we only consider binary programs. A set of atoms is *first-fixed* (*second-fixed*) if all the S-atoms in that set have the same variable in the first (second) argument position. A program P is *first-fixed* (*second-fixed*) if the set of atoms in the recursive rule is first-fixed (second-fixed). For example, the transitive closure program $P1$, 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 the following one: $S(x,y) :- S(z,y), S(w,y), A(z,w,x), C(y)$. For each natural number i, denote by M_i the infinite set of facts $\{S(i,k) \mid k \geq 1\}$. Let P be a first-fixed program. Define the infinite set $\{M_i \mid i \geq 1\}$ to be the *natural partition* for the first-fixed program. Similarly a natural-partition is defined for a second-fixed program ($M_i = \{S(k,i) \mid k \geq 1\}$).

Lemma 3.2: A program, P, which is first-fixed, or second-fixed, is decomposable. The natural partition for P is also an eligible partition for P.

Proof: Let P be a first-fixed program, and $S(i,j)$ a fact in member M_i of the natural partition for P. It is easy to see that for every I, such that $S(i,j) \in O(P,I)$, all the derivation trees for $S(i,j)$ contain only S-facts with the constant i as their first argument. These S-facts belong to member M_i. Therefore any fact is proper, and the natural partition is an eligible partition of P. The proof for a second-fixed program is similar. □

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 $S(x,x){:-}S(y,y),\ S(z,z),\ A(x,y,z)$ is repeating. Define the partition $\{M_1,M_2\}$, where $M_1 = \{S(i,j) \mid (i=j)\}$ and $M_2 = \{S(i,j) \mid i \neq j\}$, to be the *degenerate* partition. A fact is a *one-constant* fact if the same constant appears in its two arguments. Otherwise, the fact is a *two-constant* fact. For example, $S(a,a)$ is a one-constant S-fact, while $S(a,b)$ is a two-constant S-fact.

Lemma 3.3: A repeating program P, is decomposable. An eligible partition for P is the degenerate one.
Proof: Note that in the output of P, the two-constant S-facts are derived only by instantiations of the exit rule. Therefore, these facts are proper in any partition of the output domain, particularly, in the degenerate one. The one-constant S-facts have the following property. All the derivation trees of a one-constant S-fact contain only one-constant S-facts. Therefore, these facts are also proper in the degenerate partition. □

Next we define a discriminating program. The definition, in contrast to the others in this paper, is not entirely syntactic. For an input I to a program P, define the *nontrivial output*, denoted $nt(I)$, to be the set of S-facts which are in $O(P,I)$, but not in the B-to-S substitution of I. In other words, $S(a,b)$ can be in $nt(I)$, only if $B(a,b)$ is not in I. Intuitively, the nontrivial output is the output that cannot be obtained only by instantiations of the exit rule, i.e, the facts that do not have derivation trees of height one. Furthermore, define the *two-constant subinput*, denoted I^*, to be the input obtained by eliminating from I all the one-constant B-facts. Define a program to be a *reverse* program if 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.

A program is *discriminating* if the following two conditions are satisfied:

(1) the program is reverse, and

(2) for each input I, $nt(I) = nt(I^*)$.

In appendix A we provide an algorithm for determining whether or not a program P is discriminating. The program with the recursive rule $S(x,y){:-}S(y,x),S(x,z),S(z,y)$ is an example of a discriminating program. The two conditions in the above definition are independent. For example, the reverse program having the recursive rule $S(x,y){:-}S(y,x),S(x,z),A(x,y,z)$ does not satisfy condition (2). To see this, consider the input $I = \{B(2,1),\ B(1,1),\ A(1,2,1)\}$. $S(1,2)$ is in $nt(I)$, but is not in $nt(I^*)$.

On the other hand, there are programs, that satisfy only the second condition. For example, the program with the recursive rule $S(x,x) :- S(x,n),\ S(n,x)$ is not reverse, but satisfies condition (2).

Lemma 3.4: A discriminating program P, is decomposable. An eligible partition for P is the degenerate one.

Proof: In a reverse program, the nontrivial output, $nt(I)$, contains only two-constant S-facts for every I. To see that, note that every instantiation of the recursive rule, which derives a one-constant S-fact, contains that one-constant S-fact in the body of the instantiated rule. Hence, that S-fact could be derived by an instantiation of the exit rule. The rest of this proof is similar to the proof of Lemma 3.3. For every I, the one-constant S-facts can be derived by an instantiation of the exit rule, so they are proper in any partition, particularly in the degenerate one. The two-constant S-facts have a derivation tree in which all the S-facts are two-constant S-facts (by condition 2 in the definition of a discriminating program). Therefore, these facts are also proper in the degenerate partition. □

3.3 Necessary Conditions for Decomposability

Next, we characterize all the decomposable programs. We prove that if a program is not first-fixed, nor second-fixed, nor repeating, nor discriminating, then it is not decomposable. This proof involves a lengthy case analysis. First, we prove two lemmas that introduce two properties of decomposable programs. Then, Lemma 3.7 shows that among the reverse programs, only the discriminating ones are decomposable. Lemma 3.8 proves that among the non-reverse and non-repeating programs, only the first-fixed or second-fixed programs are decomposable. In general, we prove that a program is not decomposable, by showing that all the output domain facts of that program must be in one member of any eligible partition. Define a set of atoms H to be a *variant* of another set of atoms H' if H can be obtained from H' by renaming the variables in H' (different variables are renamed by different variables).

Lemma 3.5: Let P be a decomposable program. If the recursive rule r of P, contains two S-atoms, M and N, such that:

(1) At most one of the atoms has a repeated variable, and

(2) The set $\{N,M\}$ is not first-fixed, nor second-fixed, nor a variant of $\{S(x,y),S(y,x)\}$.

Then in every eligible partition D of P, all the two-constant facts in the output domain are in the same member of D.

Proof: Consider two two-constant facts, $S(\vec{e}_1) = S(i,j)$ and $S(\vec{e}_2) = S(k,l)$. We shall divide the proof into two cases.

<u>case 1:</u> Assume that $i \neq k$, and $i \neq l$, and $j \neq k$, and $j \neq l$. Then there are three subcases to consider.

(1.1) M and N do not have a shared variable, and none of them has a repeated variable. Then there is a 1-1 instantiation f of r such that $N \cdot f = S(i,j)$ and $M \cdot f = S(k,l)$. Using Lemma 3.1 ends this subcase.

(1.2) M and N have at least one shared variable, but do not have any repeated variable. Since $\{M,N\}$ is not a variant of $\{S(x,y),S(y,x)\}$, there is exactly one variable with two occurrences in the set $\{M,N\}$. Additionally, since the set $\{M,N\}$ is not first-fixed, nor second-fixed, these two occurrences are not in the same position. Therefore, $\{M,N\}$ is $\{S(x,y),S(z,x)\}$ (actually a variant of the set). Consider the following two instantiations:

Let f be a 1-1 instantiation of r, in which $f(y) = k$, $f(x) = j$, $f(z) = i$. By Lemma 3.1, $S(j,k)$, $S(i,j)$ are in the same member of D.

Let g be a 1-1 instantiation of r, in which $g(y) = l$, $g(x) = k$, $g(z) = j$. By Lemma 3.1, $S(k,l)$, $S(j,k)$ are in the same member of D.

Therefore, the three facts $S(i,j)$, $S(k,l)$, and $S(j,k)$ are in the same member, particularly the first two.

(1.3) At least one of the atoms M, N has a repeated variable. Assume that $N = S(x,x)$ is that atom. Since $\{M,N\}$ is not first-fixed, nor second-fixed, x does not appear in M. Consequently, we can assume that M is $S(y,z)$. Let o be a new constant. The following two instantiations end this case.

Let f be a 1-1 instantiation of r, in which $f(x) = o$, $f(y) = i$, $f(z) = j$. By Lemma 3.1, $S(o,o)$, $S(i,j)$, are in the same member of D.

Let g be a 1-1 instantiation of r, in which $g(x) = o$, $g(y) = k$, $g(z) = l$. By Lemma 3.1, $S(o,o)$ and $S(k,l)$ are in the same member of D. As in subcase 1.2, $S(i,j)$ and $S(k,l)$ are in the same member.

case 2: Assume that $i=k$, or $i=l$, or $j=k$, or $j=l$. Then, we can find a third two-constant fact, $S(\vec{e}_3)$, with constants that are pairwise-different from both, $S(\vec{e}_1)$ and $S(\vec{e}_2)$. Based on case 1, $S(\vec{e}_1)$ and $S(\vec{e}_3)$ are in the same member of D, and $S(\vec{e}_2)$, $S(\vec{e}_3)$ are in the same member of D. □

Lemma 3.6: Let P be a decomposable program, and assume that the recursive rule of P, has three atoms that are variants of the set $\{S(z,z),\ S(w,z),\ S(z,w)\}$ (Note that no pair of atoms satisfies the condition of Lemma 3.5). Then, in every eligible partition D of P, all the two-constant S-facts are in the same member of D.

Proof: We shall prove that every two two-constant facts, $S(i,j)$ and $S(k,l)$, with pairwise-different constants, are in the same member of D. The proof is obtained by the following five 1-1 instantiations of the recursive rule.

Let f_1 be: $f_1(w) = j$, $f_1(z) = i$. By Lemma 3.1, $S(i,i)$, $S(i,j)$, $S(j,i)$ are in the same member of D. Denote it M_i.

Let f_2 be: $f_2(w) = i$, $f_2(z) = j$. By Lemma 3.1, $S(j,j)$, $S(j,i)$ and $S(i,j)$ are in M_i.

Let f_3 be: $f_3(w) = k$, $f_3(z) = j$. By Lemma 3.1, $S(j,j)$, $S(k,j)$, and $S(j,k)$ are in M_i.

Let f_4 be: $f_4(w) = j$, $f_4(z) = k$. By Lemma 3.1 $S(k,k)$, $S(k,j)$ and $S(j,k)$ are in M_i.

Let f_5 be: $f_5(w) = l$, $f_5(z) = k$. By Lemma 3.1, $S(k,k)$, $S(k,l)$ and $S(l,k)$ are in M_i. Therefore, $S(k,l)$, $S(i,j)$ are in the same member of D.

If the two facts have common constants, the proof is identical to *case* 2 of Lemma 3.5. □

A *linear* program is a program with only one S-atom in the body of the recursive rule. Define a program to be *switching* if it is reverse and linear. A switching program is, in fact, equivalent to a non-recursive program (replace the S predicate symbol in the body of the recursive rule by B). Note that this equivalence does not contradicts our notion of minimality, since we did not delete an atom, but rather replaced one atom by another.

Lemma 3.7: If a reverse program is decomposable, then it is discriminating.

Proof: We first prove the following three claims.

Claim 3.7.1: A reverse program, whose recursive rule has an S-atom with a repeated variable, is not decomposable.

Proof: Let P be a reverse program, and N an atom of the recursive rule, r, that has a repeated variable. Denote the head of the recursive rule by $S(x,y)$. Assume, by way of contradiction, that P is decomposable. Hence, P has an eligible partition $D = \{M_1,...,M_r,...\}$. We show that all the facts in the output domain are in the same member of D. First, we show this for the two-constant facts. There are two cases.

1) The repeated variable in N is not a distinguished variable. In this case, N and the head of r are two atoms that satisfy the conditions of Lemma 3.5. Consequently, all the two-constant S-facts are in the same member of D.

2) The repeated variable in N is a distinguished variable. Let $N = S(x,x)$. Then the recursive rule contains the atoms $S(x,x)$, $S(x,y)$, $S(y,x)$. By Lemma 3.6, all the two-constant S-facts are in the same member of D.

Now, suppose that M_i is the member of D that contains all the two-constant facts. Let $S(\vec{e})$ be a one-constant fact. We select a 1-1 instantiation, f, in which $N \cdot f = S(\vec{e})$ and $head \cdot f$ is a two-constant fact. By Lemma 3.1, $S(\vec{e})$ is in M_i. $\square_{Claim\ 3.7.1}$

Claim 3.7.2: A switching program is discriminating.

Proof: A switching program is a reverse program, and therefore for every input I, the nontrivial output, $nt(I)$, contains only two-constant facts. Additionally, in such a program, every derivation tree of a fact in $nt(I)$ contains only one B-fact, which is also a two-constant fact. Therefore, every fact in $nt(I)$ is in $nt(I^*)$. $\square_{Claim\ 3.7.2}$

Claim 3.7.3: Let D be an eligible partition of a reverse, non-linear, and decomposable program. Then, all the two-constant facts in the output domain are in the same member of D.

Proof: By Claim 3.7.1, the referred program does not contain an S-atom with a repeated variable. Additionally, because of the non-linearity of the program, there is an S-atom in the body of the recursive rule, having at most one distinguished variable. Denote this atom by M. Denote the head of the recursive rule by $S(x,y)$. Next, we show that there are two S-atoms in the rule that satisfy the conditions of Lemma 3.5. Thus, all the two-constant S-facts are in the same member of D. If M does not have a distinguished variable, then $S(x,y)$ and M are the two desired S-atoms. If M has a distinguished variable, say x, then the following holds. If x appears in the first position of M, then M and $S(y,x)$ are the two desired atoms; otherwise M and $S(x,y)$ are the two desired atoms. $\square_{Claim\ 3.7.3}$

Proof of Lemma 3.7: Consider a decomposable reverse program P. Assume, by way of contradiction, that P is not discriminating. We shall show that in any eligible partition D of P, all the facts in the output domain are in the same member of D, contradicting the fact that D is a partition. By Claim 3.7.2, P is not linear; thus by Claim 3.7.3, all the two-constant S-facts are in the same member of D, say M_i.

It remains to show that all the one-constant S-facts are also in M_i. Let $S(j,j)$ be a one-constant fact. P is a reverse program but not a discriminating one. Thus there is an input, I, such that $nt(I^*) \subset nt(I)$. We shall assume without loss of generality, that I has the following property (minimality): $nt(I^*) \subset nt(I)$, but if we eliminate any one-constant B-fact, $B(\vec{e})$, from I, then $nt((I - \{B(\vec{e})\})^*) = nt(I - \{B(\vec{e})\})$. Such an input can be obtained by starting with an input for which the proper containment is satisfied, and eliminating one-constant B-facts, repeatedly, until equality is obtained; then return the last eliminated B-fact. Now suppose, again without loss of generality, that the constant j is not in I (otherwise we can add $j+1$ to all the constants in I). Obviously I has a fact $B(i,i)$. Denote by I_0, the input obtained from I, by replacing each occurrence of the constant i by j. It is easy to see that I_0 satisfies the following properties: (*i*) $B(j,j)$ is in I_0, and (*ii*) $nt(I_0^*) \subset nt(I_0)$; in other words there is a two-constant fact, a, in $nt(I_0)$, that is not in $nt(I_0^*)$. Consequently (remember minimality), I_0 forces $S(j,j)$ and a to be in the same member of D, namely M_i. $\Box_{Lemma\ 3.7}$

Lemma 3.8: If a non-reverse and non-repeating program is decomposable, then the program is first-fixed or second-fixed.

Proof: We first prove the following 2 claims.

> **Claim 3.8.1:** A non-reverse and non-repeating program, P, that has at least two S-atoms with repeated variables in the recursive rule, is not decomposable.
>
> **Proof:** Assume, by way of contradiction, that the program P has an eligible partition D. We show that all the facts in the output domain are in the same member of D. Every two one-constant facts are in the same member of D because they are neighbors with respect to P. Denote this member by M_i. P is not a repeating program, therefore the recursive rule includes an S-atom with two different variables. Denote it N. Now consider a two-constant S-fact, $S(\vec{e})$, and let f be a 1-1 instantiation of the recursive rule in which $N \cdot f = S(\vec{e})$. Since $rule \cdot f$ contains a one-constant S-fact, we obtain, using Lemma 3.1, that $S(\vec{e})$ is in M_i. $\Box_{Claim\ 3.8.1}$
>
> **Claim 3.8.2:** If a non-reverse and non-repeating program, P, is decomposable, and has an eligible partition, D, then, there are two two-constant S-facts that are not in the same member of D.
>
> **Proof:** Assume, by way of contradiction, that all the two-constant S-facts are in one member, M_i, of D. Then, there is a one-constant S-fact that is not in M_i. Denote this fact by $S(i,i)$. We consider two cases.
>
> 1) The recursive rule includes an S-atom with a repeated variable. Assume it is $M = S(z,z)$. Since the program is not repeating, there is an S-atom with two different variables. Denote this atom by N. Let f be a 1-1 instantiation such that $f(z) = i$. By Lemma 3.1, both $M \cdot f$ and $N \cdot f$ are in the same

member of D, M_i. Contradiction to $S(i,i)$ not being in M_i.

2) The recursive rule does not include an S-atom with a repeated variable. Denote the head of the recursive rule by $S(x,y)$, and let f be the following instantiation: $f(x) = f(y) = i$ (which means that f is not 1-1) and for all the other variables f substitutes distinct constants, that are different from i. Let I be the input consisting of the S-to-B substitution of $body \cdot f$. The relation B in I contains only two-constant facts since (i) the recursive rule does not include an atom with a repeated variable, and (ii) the program is non-reverse. Therefore, I forces $S(i,i)$ to be in the same member of the partition as some two-constant S-fact. This member is M_i, contradiction. $\square_{Claim\ 3.8.2}$

Proof of Lemma 3.8: Let P be a decomposable, non-reverse and non-repeating program. By Claim 3.8.1, P contains at most one S-atom with a repeated variable. Assume, by way of contradiction, that P is neither first-fixed, nor second-fixed. We shall show that in any eligible partition D of P, all the two-constant S-facts are in the same member of D, which contradicts Claim 3.8.2.

There are two cases:

(1) The head of the recursive rule has two distinct variables. Since the program is non-reverse, there are two S-atoms in the recursive rule, that satisfy the conditions of Lemma 3.5. From Lemma 3.5 we conclude that all the two-constant S-facts are in the same member of D.

(2) The head of the recursive rule has a repeated variable. In this case, one can find either two S-atoms that satisfy the conditions of Lemma 3.5 or, two S-atoms in the body of the recursive rule that together with the head satisfy the conditions of Lemma 3.6. In either case, all the two-constant S-facts are in the same member of D. $\square_{Lemma\ 3.8}$

Theorem 3.2: A program is decomposable if and only if it is first-fixed, or second-fixed, or repeating, or discriminating.

Proof: (if) from Lemmas 3.2, 3.3, 3.4.

(Only if) from Lemmas 3.7, 3.8. □

3.4 Strong Decomposability

For some decomposable programs, having multiple processors does not provide a real advantage compared to a single processor, particularly if the latter, as explained in subsection 3.1, removes members as it reaches member-fixpoint. For example, consider a repeating program with the degenerate 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 produce any nontrivial output, i.e. output for which the recursive rule has to be instantiated. 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 nontrivial facts. We shall prove in

Theorem 3.3 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. In this subsection we completely characterize the strongly decomposable programs, i.e the program for which there is an eligible partition such that more than one processor does "real" work. We determine that of the decomposable programs, only the first-fixed, second-fixed, and switching are strongly decomposable. Furthermore, a program in each of these classes has a natural partition, i.e., a partition with an infinite number of members, each of which requires real work to produce. Consequently, as we shall show in the next section, an arbitrary number of processors can be effectively utilized for producing the output, given a large enough input.

For a program P, and an input I, a fact $a \in O(P,I)$ is *nontrivial* if it belongs to the nontrivial output. 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 3.4, that if a program is strongly decomposable, then every fact in the output domain is nontrivial for some input. Therefore, the "real" work is distributed among the processors.

In Claim 3.7.2 we proved that a switching program is discriminating. In addition to the degenerate eligible partition, it has the 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 3.3: A program is strongly decomposable if and only if it is first-fixed, or second-fixed, or switching.

Proof: (if) It is easy to see that the natural partition for each program in one of these classes, is a strongly eligible partition, and therefore those programs are strongly decomposable. For example, if the program is first-fixed, then the desired input is obtained in the following way. Let f be a 1-1 instantiation of the recursive rule in which $f(x)=1$. Let g be a 1-1 instantiation of the same rule, in which $g(x)=2$. Then, I is the S-to-B substitution of $(body \cdot f \cup body \cdot g)$.

(only if) All the programs that are not decomposable cannot, of course, be strongly decomposable. Thus, it suffices to show that repeating programs, and non-linear discriminating programs, are not strongly decomposable. We do it so by showing that for every eligible partition, the nontrivial facts must belong to the same partition member.

(1) Repeating programs - The recursive rule contains at least two S-atoms with repeated variables: the head, and at least one atom in the body. Thus, every two one-constant facts are neighbors with respect to the program. Note that only one-constant facts can be nontrivial, and that by Lemma 3.1 these facts are in the same member of any eligible partition.

(2) Non-linear discriminating programs - These programs are reverse programs, so only two-constant S-facts can be nontrivial. Additionally, by Theorem 3.2, the discriminating programs are decomposable, and by Claim 3.7.3, in every eligible partition for the program, all the two-constant S-facts are in one member. □

The next corollary establishes the robustness of the strong-decomposability concept; when a program is strongly decomposable, then it has a partition with an infinite set of members containing nontrivial facts, and furthermore, every fact can be nontrivial.

Corollary 3.4: 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 $R_1,...,R_k$, and for each k facts $a_i \in R_i$, for $i=1,...,k$, 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 3.5: 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, in every strongly eligible partition of P, a and b belong to the same member.

Proof: Consider a first-fixed program P. The facts a and b have the same constant in their first position; thus they are neighbors with respect to P. By Lemma 3.1, a, b are in the same member of any eligible partition of P.

Similar arguments are used to prove the proposition for a second-fixed program, or a switching one. □

The next proposition establishes the relationship between the family of strongly decomposable programs and the family of programs in the NC complexity class (assuming $P \neq NC$).

Proposition 3.6: There are strongly decomposable programs that are also in NC (e.g. the linear transitive closure), there are strongly decomposable programs that are not in NC (e.g. the first-fixed program $S(w,x)$:- $S(w,y)$,$S(w,z)$,$H(x,y,z)$), and there are programs in NC that are not strongly decomposable (e.g. the program $S(x,y)$:- $A(x,z)$,$S(z,w)$,$C(w,y)$).

Proof: The linear transitive closure, and the program $S(x,y)$:- $A(x,z)$, $S(z,w)$, $C(w,y)$ are in NC by results from [AP, UV]. The program $P2$: $S(w,x)$:- $S(w,y)$, $S(w,z)$,$H(x,y,z)$ is P-complete. We prove it by a reduction from the first known P-complete program, path-systems ([C]), which is $S(x)$:- $S(y)$, $S(z)$, $H(x,y,z)$. Given an input, I, to path-systems, we transform it to an input, I', to program $P2$ as follows. The relation H in I' is the same as in I. Let 'a' be some constant. The relation B in I' consists of all the tuples $B(a,i)$ such that $B(i)$ is in I. Then $S(a,i)$ is in $O(P2, I')$ if and only if $S(i)$ is in $O(path-systems, I)$. □

The next comment concerns the extension of the positive results of theorem 3.3. 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.

In sum, in this section we defined two properties of programs: decomposability and strong decomposability. We completely characterized the programs that have each one of these properties, and the result is that only a narrow class of programs possesses them. In the next section, we prove that *only* the strongly decomposable programs can be evaluated by several processors that do not communicate, nor duplicate any work. Thus, the importance of the above characterization is in its "only if" direction, that is, the negative result. Except for the strongly decomposable programs, there is *no* program that can be evaluated with minimal total evaluation cost and without communication. New strategies, that involve (minimal) communication or duplication of work, are needed. Sections 4.4, 5, and 6 discuss such strategies. Furthermore, in [WO] we have extended the decomposability definition to arbitrary datalog programs (not necessarily binary sirups), and we have shown that for such programs decomposability is undecidable. Similarly, the strong decomposability definition can be extended, and the proof of [WO] can be repeated verbatim to show that strong decomposability is also undecidable. Thus, complete characterization can be obtained only for subclasses of programs, such as the binary sirups considered in this paper. Moreover, in [WS] we syntactically define the *pivoting* property for arbitrary datalog programs. The strongly decomposable programs (semantic property) are exactly the pivoting binary sirups. Also, every pivoting arbitrary-datalog-program is strongly decomposable.

4. 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).

4.1 Parallelization Schemes

In this subsection we provide the formal definition of a parallelization scheme, i.e., a set of parallel algorithms that together evaluate a program. Each algorithm in the scheme evaluates the program with less than the whole input; consequently, it is faster, but, on the other hand, does not produce the whole output. Then we distinguish between two types of parallelization schemes: decomposition and sharing. Both of them guarantee that the whole output is obtained as the union of all the facts produced by the algorithms, therefore, if these facts are sent to an output processor, or a common file, completeness of the result is ensured. However, decomposition schemes also guarantee that the processors executing the parallel algorithms do not duplicate one another's work. Finally, we define pure parallelization schemes, i.e. schemes that do not incur any overhead.

Let P be a program, and let I be an input to P. A *partial computation*, denoted $A_i(I)$, is a partially ordered set of facts from $I \cup O(P,I)$. The subscript i in $A_i(I)$ stands for the identity of the processor that produces the partial computation. Each fact $S(\vec{e})$ in $A_i(I)$ is labeled *computed* or *transmitted*. If $S(\vec{e})$ is computed, then it must be preceded in $A_i(I)$ by all the facts of one of its derivation trees. Intuitively, the partial order in $A_i(I)$ represents the time-order in which the output of P is evaluated, and the requirement that $S(\vec{e})$ must be preceded by all the facts in some derivation tree means that i must "know" all these facts before being able to compute $S(\vec{e})$. The set $A_i(I)$ is called a "partial" computation, since not all facts of $O(P,I)$ have to be in $A_i(I)$. A transmitted fact is received from another processor, thus a derivation tree does not necessarily precede it. For example, the semi-naive evaluation by a single processor produces a partial computation consisting of the input facts, followed by all the output facts, in the order in which they are evaluated; all the facts are computed.

An *r-parallelization-scheme*, A, for partial computation of P, is a function which maps each input, I, into r partial computations, $A(I) = \{A_1(I),...,A_r(I)\}$, such that if some fact is transmitted in some $A_i(I)$, then it is computed in some $A_j(I)$ [3]. A is called a *scheme* for short. The set of all partial computations with subscript i constitutes the (output of) *algorithm* A_i of A. We denote by p_i the processor that executes A_i. A scheme, A, is *sharing* if (i) (completeness) for every input I, each fact $a \in O(P,I)$ is in some partial computation of $A(I)$, and (ii) (time-saving input) for at least one input, I', there is no partial computation in $A(I')$ that contains the whole nontrivial output. A scheme, A, is a *decomposition* scheme if (i) it is sharing, and (ii) (disjointness) for every input I, no fact is a computed fact in more than one partial computation of $A(I)$. Intuitively, 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 decomposition scheme do not duplicate one another's work. Existence of a time saving input, simply ensures that the scheme is not trivial, i.e., does not consists of a single-processor evaluation algorithm.

A scheme A is *independent* if for every input I, each partial computation in $A(I)$ does not contain any transmitted facts (i.e all the intentional facts are computed). Independence ensures that facts are not transmitted between algorithms, i.e. there is no data overhead. In this section we discuss only independent schemes. An independent scheme, A, is *data-driven* if for each input I, and for each fact $b \in O(P,I)$, and for 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 $b \in A_i(I \cup Z)$.

[3] Actually, additional requirements have to be satisfied by the set A(I), but we omit them since they are not used in this paper.

(2) (noncontribution) If $b \notin A_i(I)$, 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 $b \notin A_i(I \cup Z)$.

Intuitively, 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. 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 $A_i(I \cup Z)$.

Remark 4.1: It can be shown that an independent decomposition scheme that satisfies the contribution requirement, also satisfies the noncontribution requirement.

Let A be an independent, data-driven, parallelization scheme for the 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.

4.2 Strong Decomposability and Pure Parallelization

In this subsection we prove that the programs having pure parallelization schemes are exactly the strongly decomposable programs. Then we outline a strategy (i.e. a class of schemes), called $DS\,1$, that contains all the pure decomposition schemes.

Lemma 4.1: 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 other input, I', if $b \in O(P,I')$ then $b \in A_j(I')$.

Proof: Assume, by way of contradiction, that $b \notin A_j(I')$. Then, by completeness of the scheme, $b \in A_k(I')$, $j \neq k$. Let I'' be the input $I \cup I'$. By the contribution requirement of a data-driven scheme, $b \in A_k(I'')$ because $b \in A_k(I')$. By the same requirement, $b \in A_j(I'')$ because $b \in A_j(I)$. Contradiction to the disjointness requirement, i.e that every fact is computed in a unique processor. □

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). Given a program P, a *restricted version* P_t of P (see [WS]) is a program obtained from P by appending evaluable predicates to the body of some, or all, of the rules of P. For example the program with the recursive rule $S(x,y)$:- $S(x,z)$, $A(z,y)$, $odd(x)$ is a restricted version of $P\,1$, defined in subsection 3.1. A set of facts I is an input to P_t if and only if it is an input to P. The output of P_t is defined as the output of P, with the following exception: in a derivation tree, every instantiation of a rule, r, must satisfy the evaluable predicate appended to r.

Theorem 4.1: 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 $i = 1,...,r$. Then $\{M_1, \ldots, M_r\}$ is a strongly eligible partition for P.

Proof: (if): Let P be a strongly decomposable program. By Theorem 3.3 P is first-fixed or second-fixed or switching. Assume P is first-fixed. We show a pure decomposition scheme $A = \{A_1, A_2\}$ for P. A_i is the semi-naive evaluation (see appendix C) of the restricted version, P_i. P_1 (P_2) is P with the evaluable predicate $odd(x)$ $(even(x))$ appended to the recursive and exit rules. Now, let us prove that A is a pure decomposition scheme. Given an input I, the partial computation in processor i consists of the input facts (unordered), followed by all the output facts, in the order in which they are evaluated. $A_1(I)$ contains all the facts in $O(P,I)$ with an odd constant in their first position. $A_2(I)$ contains all the other facts in $O(P,I)$ (i.e. the output facts with an even constant in their first position). Therefore, completeness and disjointness are satisfied. A also has a time-saving input, e.g. the input provided in the proof of Theorem 3.3, the (if) part. Consequently, A is a decomposition scheme. Moreover, all the facts in the partial computations are computed, and therefore A is an independent scheme. Additionally, A_1 and A_2 are monotonic; hence the contribution requirement is satisfied. The non-contribution requirement is also satisfied by Remark 4.1. Consequently, A is pure.

Similarly, we can prove that second-fixed and switching programs have decomposition schemes (for switching programs the odd-even($x+y$) evaluable predicates are used).

(only if): P has a pure decomposition scheme $A = \{A_1, \ldots, A_r\}$. Let D be the following partition of the output domain of P. $D = \{M_1, \ldots, M_r\}$ where M_i contains all the facts that their home-algorithm is A_i. We shall show that D is a strongly eligible partition. Let I be an input to P, and $S(\vec{e}_1)$ a fact in $O(P,I)$ with the home-algorithm A_i. By Lemma 4.1, $S(\vec{e}_1) \in A_i(I)$, and by the independence of the scheme, $S(\vec{e}_1)$ is preceded in $A_i(I)$ by a derivation tree; thus all the intentional facts of that tree are in member M_i of D. Consequently, D is an eligible partition. It remains to show that D is a <u>strongly</u> eligible partition, i.e. there is an input for which two nontrivial facts belong to different members of D. It is easy to see that the input that is time-saving in A, is such an input. □

Next we describe a set of pure decomposition schemes, namely a *strategy*.

Strategy DS1:

A strongly decomposable program P, is evaluated by algorithms $\{A_1,...,A_r\}$, for any number of processors, r. Let h be some hash function that maps each natural number (pair of natural numbers for switching programs) into a unique member of the set $A = \{1,...,r\}$ (for example, the modulo r function). We assume that some number is mapped by h into each member of A. 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-fixed, 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 $h(k) = i$. If P is switching, then h must be commutative. ☐

A scheme in the strategy DS1 is obtained by fixing the hash function, the number of 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.

It can be shown that a scheme is a pure decomposition scheme if and only if it is in $DS1$. The (if) part of the proof is similar to the if part in Theorem 4.1. For showing the converse, let A' be a pure decomposition scheme for r processors. Theorem 4.1 and Proposition 3.5 indicate that in A', each algorithm is responsible for some members of the natural partition. We select a hash function h' which maps each natural number to the set $\{1,...,r\}$, such that for every two natural numbers, i and j, the following is satisfied: $h'(i) = h'(j)$ ($h'(i1,i2) = h'(j1,j2)$ for switching programs) if and only if members M_i and M_j ($M_{i1,i2}$ and $M_{j1,j2}$ for switching programs) of the natural partition belong to the same member of D. Then A' is a scheme in DS1, for which the hash function is h'. ☐

4.3 Evaluation Cost of Strategy DS1

In this subsection, we consider schemes in DS1, where the algorithm of each processor is semi-naive evaluation (see appendix C). We establish that for these schemes, for each given input, the total amount of work performed by all processors in evaluating a program is minimal, i.e., not higher than the amount of work in semi-naively evaluating the program by a single processor. Therefore, given a good hash function, the speed-up is maximum. 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. Identical results can be shown for the naive evaluation algorithm ([U]).

Next we define three cost measures to quantify the amount of work performed by an algorithm for evaluating a program P given an input I. These measures will be used in our cost analysis of the strategies. The first, denoted $cost^1$, assumes that the cost of an iteration i of semi-naive evaluation is $c \cdot | S^{i-1} |$, where $| S^{i-1} |$ is the number of tuples in the relation S at the end of the $i-1$ iteration (i.e. at the beginning of the i-th iteration). c is fixed for a given input (the results presented in this section still hold if $cost^1$ is a super-linear function of $|S^i|$). Then, $cost^1$ of evaluating $O(P,I)$, is the total cost of all the iterations performed during the evaluation. Note that the cost of an iteration increases as the evaluation proceeds.

The second cost measure, denoted $cost^2$, is the number of "successful inferences of rules" performed during the evaluation. An inference of a rule is a substitution of facts, one for each atom in the rule body. For example, every join can be regarded as a sequence of such substitutions. If the inference succeeds, namely, equal constants replace the same variable, then it is a *successful inference*. Note that a fact can be derived by several successful inferences; then, the price of deriving that fact is greater than one. The measure was introduced and justified in [BR].

In the third measure, $cost^3$, the cost of a join of two relations is the multiplication of their sizes. Here we assume that the join is computed by a trivial nested loop. As before, $cost^3$ of evaluating $O(P,I)$ is the total cost of all the joins in all the iterations.

In all the three measures, the instantiations of the exit rule are ignored. The reason is, that these instantiations can be done immediately by copying all the relation B to S. Thus, the first iteration of semi-naive evaluation, that we speak of in the proofs is the first iteration of the "repeat loop" (see appendix C), and contains only instantiations of the recursive rule. Anyway, our results are valid even if the price of the instantiations of the exit rule is not zero.

We demonstrate the three measures by the following example. Consider the transitive closure program $P1$ of subsection 3.1, and the input $I=$ $\{B(1,2), B(2,4), B(1,3), B(3,4), B(4,5), A(1,2), A(2,4), A(1,3), A(3,4), A(4,5)\}$. The costs of performing semi-naive evaluation on $P1$ and I are: $cost^1 = c\cdot(5+6+7)$, $cost^2 = 2+1+0$, and $cost^3 = 5\cdot 5 + 1\cdot 5 + 1\cdot 5$.

In the rest of this subsection, let $\{A_1,...,A_r\}$ denote the execution of a scheme in DS1, that evaluates a strongly decomposable program, P, with an arbitrary input, I. Each A_i is the semi-naive evaluation of the restricted version of P, for some hash function denoted h. The semi-naive evaluation of P, on a single-processor, given the same input I, is called A. We denote the S-relation at the end of the i-th iteration of A (i.e. at the beginning of the $i+1$ iteration), by S^i (S^0 is the S-relation at the beginning of the first iteration, i.e. the set of facts obtained by instantiations of the exit rule). Moreover, the set of new facts derived at that iteration is denoted ΔS^i ($\Delta S^i = S^i - S^{i-1}$ except for $\Delta S^0 = S^0$). Similarly, the S-relation at the end of the i-th iteration of A_j is denoted S_j^i, and the set of new facts derived at that iteration is denoted ΔS_j^i. Thus $\Delta S_j^i = S_j^i - S_j^{i-1}$ except for $\Delta S_j^0 = S_j^0$. Another notation is $h_j(\Delta S^i)$: the set of facts in ΔS^i, that are mapped by the hash function to j. Similarly, $h_j(S^i)$ is the set of facts in S^i that h maps to j. For example, if the program is first-fixed, then $h_2(\Delta S^i)$ is the set of facts in ΔS^i such that h maps their first argument to 2.

Lemma 4.2: For every processor j, and for every iteration i, $\Delta S_j^i = h_j(\Delta S^i)$.

Proof: Simple induction on the iteration number. □

Corollary 4.3: For each j, the number of iterations in algorithm A_j is not higher than the number of iterations in algorithm A (because if in the i-th iteration A_j derives a new fact, so does A). □

Corollary 4.4: Every successful inference performed in iteration i of algorithm A_j, is also performed in iteration i of A, and is performed the same number of times (because at the beginning of iteration i, $S_j^{i-1} = h_j(S^{i-1})$ and $\Delta S_j^{-1} = h_j(\Delta S^{i-1})$). □

Theorem 4.2: The following inequalities hold for $A, A_1, ..., A_r$:

(1) $\sum_{i=1}^{r} cost^1(A_i) \le cost^1(A)$ (2) $\sum_{i=1}^{r} cost^2(A_i) \le cost^2(A)$ (3) $\sum_{i=1}^{r} cost^3(A_i) \le cost^3(A)$.

Proof:

(1) By Lemma 4.2, and the disjointness and completeness requirements, we obtain that in every iteration i:

$$(4.1) \qquad \sum_{j=1}^{r} |S_j^i| \leq |S^i|$$

($|X|$ denotes the number of facts in relation X).
If i is greater than the number of iterations in processor j, then $|S_j^i| = 0$.
For $j = 1,...,r$ we denote by m_j (m), the number of iterations that algorithm A_j (A) performs when evaluating I. By Corollary 4.3, $max(m_1,...,m_r) \leq m$.

We obtain: $cost^1(A_j) = c \cdot \sum_{i=1}^{m_j} |S_j^{i-1}|$, and

$$cost^1(A) = c \cdot \sum_{i=1}^{m} |S^{i-1}|.$$

Based on equation 4.1, it is easy to see that:

$$\sum_{j=1}^{r} cost^1(A_j) \leq cost^1(A)$$

(2) By Corollary 4.4, and the fact that every successful inference is performed by a unique algorithm of the scheme, we obtain that for every iteration i:

$$(4.2) \qquad \sum_{j=1}^{r} p_j^i \leq p^i \ .$$

where p_j^i (p^i) is the number of successful inferences performed during iteration i of algorithm A_j (A).
An arithmetic manipulation, similar to the one done for $cost^1$ in (1), completes the proof.

(3) Assume that in the body of the recursive rule, there are k S-atoms. Suppose that the sizes of the extensional relations joined by algorithms A_j, for $j=1,...,r$, at each iteration are $l_{j,1},...,l_{j,d}$. (The sizes obviously do not change from iteration to iteration). Let $l_j = l_{j,1} \cdot ... \cdot l_{j,d}$. Similarly, we denote the size of the extensional relations joined by A, by $l_1,...,l_d$, and $l = l_1 \cdot ... \cdot l_d$. Clearly, $l_j \leq l$ for $j = 1,...,r$. Note that l_j may be strictly smaller than l. For example, in the restricted version $S(x,y) :- S(y,x), A(x,y), even(x+y)$, the evaluable predicate "cuts" the size of the relation A.
By Lemma 4.2, and the disjointness and completeness requirements, we obtain that in every iteration i:

$$(4.3) \qquad \sum_{j=1}^{r} |\Delta S_j^i| \leq |\Delta S^i|$$

If i is greater than the number of iterations in processor j, then $|\Delta S_j^i| = 0$.

Additionally: $cost^3(A_j) = l_j \cdot k \cdot \sum_{i=0}^{m_j-1} (|S_j^i|^{k-1} \cdot |\Delta S_j^i|)$, and

$$cost^3(A) = l \cdot k \cdot \sum_{i=0}^{m-1} (|S^i|^{k-1} \cdot |\Delta S^i|).$$

An arithmetic manipulation, similar to the one done for $cost^1$ in (1), shows that:

$$\sum_{j=1}^{r} cost^3(A_j) \le l \cdot k \cdot \sum_{i=0}^{m-1} \sum_{j=1}^{r} (|S_j^i|^{k-1} \cdot |\Delta S_j^i|).$$

Thus, using inequalities (4.1) and (4.3), it is easy to see that:

$$\sum_{j=1}^{r} cost^3(A_j) \le cost^3(A) \ \square$$

4.4 Pure Parallelization of Other Programs

Theorem 4.2 indicates that if we insist on pure parallelization of programs, we must relax the disjointness requirement, and consequently the minimal total evaluation cost. This approach was taken in [W], where pure sharing (not decomposition) schemes were examined (pure sharing schemes were defined in section 4.1). Programs that have a pure sharing scheme are named ***sharable***. [W] showed that all linear programs are sharable, while there are programs, such as path-systems, blue-blooded-frenchman (see [CK]), and others, all of which belong to a syntactic class called propagating programs, are not sharable. The class of sharable programs is strictly larger than the class of strongly decomposable programs, and is incomparable to the class of decomposable programs (see figure 1).

For evaluating the linear programs in parallel, the following strategy of pure sharing schemes was proposed in [W]. The strategy is denoted SS1 in this paper. It evaluates a linear program P that is not strongly decomposable, 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 [4]; h is some hash function.

Intuitively, a pure sharing scheme does not guarantee a minimal total evaluation cost, since the algorithms of the scheme do not necessarily produce disjoint sets of facts, and therefore, the same fact may be "examined" in the scheme by more than one algorithm. It can be easily shown that there are programs and inputs, for which the algorithms of SS1 do not satisfy the inequalities of Theorem 4.2.

However, it can be shown that when considering $cost^2$, the following is satisfied. For each input, the maximum (among all participating processors) amount of work in semi-naively 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, $S(x,y){:-}S(w,z),A(x,y,w,z)$ does not have such a cover but is sharable.

[4] $h(y)=i$ or $h(x+y)=i$ work as well.

5. DYNAMIC LOAD DISTRIBUTION

Pure parallelization pays for lack of overhead with two limitations. First, it is applicable only to decomposable programs. Second, even for decomposable programs, the evaluation-load cannot be balanced dynamically among the processors; thus, for DS1 we cannot ensure that minimal total-cost translates into time minimality. Consequently, in this section we examine independent-parallelization i.e., parallelization with control-overhead but without data-overhead. We suggest a strategy, DS2, for the parallel evaluation of a strongly decomposable program. Strategy DS2 is an adaptation of DS1 that balances the work-load dynamically, by using control messages. We assume that every two processors can communicate and every transmitted message arrives to its destination (no failures).

Strategy DS2:

By using a restricted version of a program, as in DS1, every one of the r processors assumes responsibility for computing some members of the natural partition (see subsection 3.2) of the program. Each processor performs its evaluation, one member at a time, in increasing order of members. For example if processors 0 and 1 cooperate, and processor 0 is responsible for the even members, then it evaluates M_2 first, then M_4, then M_6, etc. When some processor, i, terminates evaluating all its members, it announces completion to all the other processors. In response, each one of them broadcasts the identification of the partition member it is currently working on. Processor i assumes responsibility for $1/r$ of the unprocessed members of each processor. Consequently, each processor is left with $(r-1)/r$ of the members it had before the announcement of i. To continue the example, if processor 0 terminates the even members, it sends a control message indicating so to processor 1. Processor 1, that is responsible for the odd members, responds with the identification of the member it is working on, say 7. This indicates to processor 0 that responsibility for the odd partitions that succeed 7, is divided; processor 0 takes the members M_9, M_{13}, M_{17}, and processor 1 takes members M_{11}, M_{15}, $M_{19}, \ldots$, 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. □

Note that only control messages, i.e. partition-identifications and termination messages, are sent between processors, by DS2. It is easy to realize that every scheme in DS2 is an independent parallelization scheme (see definitions in subsection 4.1).

For the rest of this section assume that the algorithm executed by each processor is semi-naive evaluation. For example, a processor of DS2 semi-naively evaluates M_2, then it semi-naively evaluates M_4, then M_9, etc. It is easy to show that each scheme in DS2 has minimal evaluation cost, by the three measures introduced in section 4.3, $cost^1$, $cost^2$, and $cost^3$. In other words, an analog to Theorem 4.2 can be shown for schemes in DS2. The proof is based on the observation that the cost of semi-naive evaluation of a sequence of members by a processor in DS2, is not higher than the cost of a processor in DS1 that is assigned responsibility for the same members (the latter evaluates all of them together, and not one by one as the former does).

We argue that schemes in DS2 are optimal within the strategy of *partition-oriented* independent decomposition schemes. Intuitively, this is the strategy of independent decomposition schemes 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 independent decomposition scheme avoids this extra work. Observe that only strongly decomposable programs have a partition oriented independent decomposition schemes.

Schemes of DS2 are optimal, up to one partition-member [5] , 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. Second, all processors are busy until completion.

Finally, note that strategy SS1 can also be extended to strategy SS2, that distributes the load dynamically. A scheme in SS2 evaluates any linear program, P, as follows. "Member" M_i consists of the set of output facts derived from the input, where the relation B is restricted to the set $\{B(i,c) \mid c \text{ is a constant}\}$. Each processor evaluates the members in increasing order. The evaluation of a member consists of the evaluation of the restricted version of P having $x=i$ appended to the exit rule. Work redistribution occurs when a processor completes, as in DS2.

6. A PARALLELIZATION STRATEGY APPLICABLE TO ALL PROGRAMS

In this section we present a general purpose parallelization strategy, DS3. In contrast to the strategies presented thus far, DS3 can be used for the parallelization of every program. It incurs a data-overhead involved in transmitting tuples among the processors, but we show that in some sense the overhead is minimal. We also show that for the linear programs, the total evaluation cost of the processors is minimal.

6.1 The Strategy DS3

As the previous strategies, DS3 is a data-reduction strategy, i.e., each processor evaluates a program P, using less than the whole database. Given a hash function h, processor i is responsible for computing the facts that satisfy $h(x)=i$, where x is the first variable in the head of the recursive rule [6]. In DS3, each processor executes a modified version of semi-naive evaluation (an adaptation of naive evaluation is also possible, and even simpler). To ensure completeness, each processor communicates with the others in the following way. Processor i has a set of predicates, T_{ij} for $j=1$,.., r and $j \neq i$. Each T_{ij} depends on the program being evaluated, and the hash function. Processor i transmits processor j all the facts that i computes, and that satisfy predicate T_{ij}. Next we provide the formal description.

[5] 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.

[6] Obviously, an analog of DS3 exists for $h(y)=i$ where y is the second variable in the head of the rules.

Strategy DS3:

Given a system of r processors, a hash function h that maps the natural numbers into $\{1, .. ,r\}$, and a program P, processor i executes the following procedure:

(1) If P is strongly decomposable - execute DS1 or DS2 (no facts have to be transmitted).

(2) Determine the transmission predicates T_{ij} for $j=1, .. ,r$ and $j \neq i$, according to the flow-chart in figure 6.2 (next subsection).

(3) Let P_i be P with the hash function $h(x)=i$ appended to the exit and recursive rules; x is the first variable in the head of both rules. Compute $O(P,I)$ by semi-naive evaluation.
At the end of each iteration do:

3.1 Denote by ΔS_i the set of new tuples that i computed during its last iteration. For $j=1, .. ,r$ $j \neq i$ transmit processor j all the facts in ΔS_i that satisfy the predicate T_{ij}.

3.2 Add to the relation ΔS_i, the set of all tuples that were received from other processors during i's last iteration (this set may be empty).

3.3 If ΔS_i is empty, then wait until some tuples are received from other processors.

end.

The computation ends when all processors are in step 3.3, and no tuples are "in transit" i.e. have been sent but not received yet. □

Note that the processors perform their computation completely asynchronously. Also, the only assumption that we make about the communication network is that each tuple that is sent, eventually reaches the destination processor (no FIFO arrival of massages is necessary). In appendix B we provide an example of evaluating a program by a scheme in DS3, and discuss its performance.

6.2 The Transmission Predicates

In strategy DS3, processor i sends to processor j the facts that i computes, and that satisfy the transmission predicate T_{ij}. In this subsection we define these predicates, whose purpose is to reduce the number of transmitted facts. Intuitively, a fact does not have to be transmitted to j, if once arrived there, it will either be eliminated by j's hash function, or, it will not contribute to the evaluation performed by processor j. Such facts will not satisfy the predicate T_{ij}. 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, the program is not decomposable, but regardless of the input, a fact $S(c,d)$ such that $j \neq h(d)$, does not have to be sent to j. Such a fact is never instantiated in the body of the recursive rule of j, because j's hash function prevents this.

In an unary program, T_{ij} is $TRUE$ for any i and j. In other words, each processor transmits all its computed facts to all the other processors. Thus, for the rest of this subsection we consider only binary programs.

The predicate T_{ij} depends on the class of binary program, P, being evaluated. We define several classes of programs, each with its own set of transmission predicates. 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. For example, the program with the recursive rule $S(x,y) :- S(x,z), S(y,x)$ is first-consistent. 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 program (a first-fixed program, or a program with an empty r-body). For example the program with the recursive rule $S(x,y) :- S(x,z), S(y,x), S(z,z)$ is a partially-first-consistent program, while the program with the recursive rule $S(x,y) :- S(x,z), S(x,m), S(y,y), A(z,m)$ is partially-first-fixed.

Now we define a partially-discriminating program. Given an instantiation, f, of the recursive rule of P, denote by I_f^+ the following input to P. It consists of {the facts in S-to-B substitution of *body*$\cdot f$} minus {all the one-constant B-facts, except $B(f(x),f(x))$}. P is *partially-discriminating* if *head*$\cdot f \in O(P,I_f^+)$ for any instantiation f of r. In other words, a program is partially-discriminating if for every instantiation, f, there exist a derivation tree of *head*$\cdot f$, as follows. Each leaf of the tree is in *body*$\cdot f$ (the S predicate symbol is replaced by B); also, each B-fact in the tree is either a two-constant fact, or the fact $B(f(x),f(x))$. In a discriminating program, for every instantiation, f, that derives a new fact, there exist a derivation tree in which each B-fact is a two-constant fact. Thus, every discriminating program is also a partially-discriminating program. The algorithm that decides whether or not a program is partially-discriminating is described appendix A.

The definition of T_{ij} is given using the flow-chart in Figure 6.2.

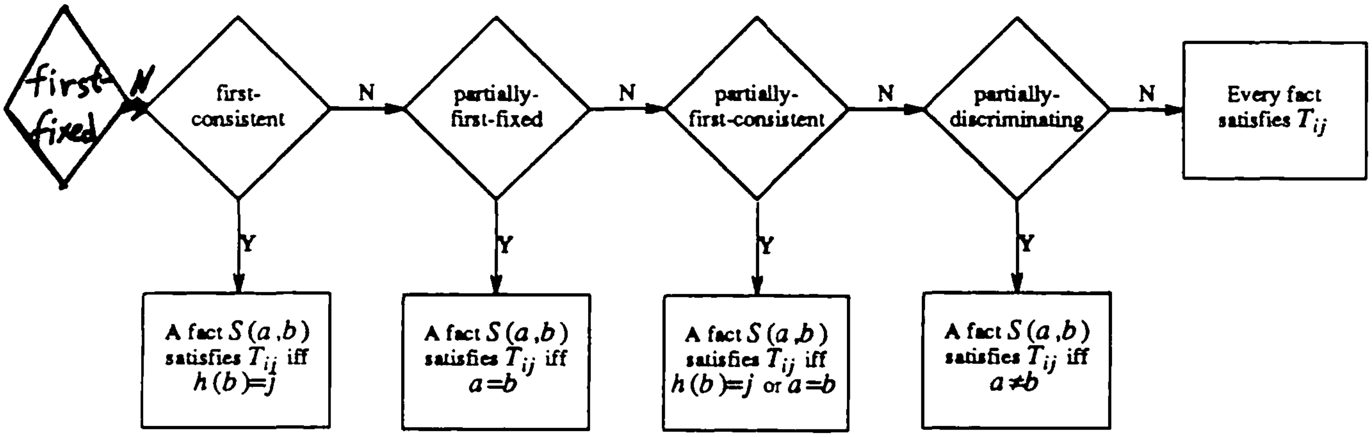
first-fixed
N
first-consistent
N
partially-first-fixed
N
partially-first-consistent
N
partially-discriminating
N
Every fact satisfies T_{ij}
Y
Y
Y
Y
A fact $S(a,b)$ satisfies T_{ij} iff $h(b)=j$
A fact $S(a,b)$ satisfies T_{ij} iff $a=b$
A fact $S(a,b)$ satisfies T_{ij} iff $h(b)=j$ or $a=b$
A fact $S(a,b)$ satisfies T_{ij} iff $a \neq b$

Figure 6.2: Defining the transmission predicates

We end this subsection with the following remark. The decomposable programs, along with the new program classes defined in this section, comprise the set of "coverable" programs. We shall not formally define here this extension of the decomposability concept, but will just mention that, a coverable program has a cover of the output domain, for which every fact is proper. The notion of a cover is weaker than the notion of a partition, in the sense that the members of a cover need not be disjoint, but each one is smaller than the whole output domain.

6.3 Correctness of the Strategy DS3

In this subsection we establish that each scheme in strategy DS3 is actually a decomposition scheme. A moment of thought will reveal that two of the three properties of such a scheme, disjointness and existence of a time-saving input, are trivially satisfied. Completeness is not, particularly since a processor executing DS3 does not send to all the other processors, all the facts that it computes.

Theorem 6.1: Any scheme in DS3 satisfies the completeness requirement.

Proof: Assume, by way of contradiction, that there is a scheme, A, in DS3, an input I, and a fact $S(\vec{e}) \in O(P,I)$ that is not computed in any one of the partial computations $A_i(I)$. Consider a derivation tree of $S(\vec{e})$, with leaves in I. We select an instantiation, f, in that tree that satisfies the following two conditions: (i) f derives a fact, $S(a,b)$, which is not computed in any one of the processors, and (ii) every S-fact in *body*·f is computed by one of the processors. Such an instantiation exists since in the considered derivation tree, the root is not computed in any one of the processors, but every S-fact in the bottom of the tree (derived by the exit rule) is computed by one of the processors. Assume without loss of generality that $h(a) = j$ (i.e. processor j is "responsible" for producing $S(a,b)$). Since the evaluation had terminated, but $S(a,b)$ was not computed, we conclude (remember condition (ii) of f) that there is at least one fact, $S(c,d)$, in *body*·f that is computed in another processor, but not transmitted to j. Suppose that $S(c,d) = M \cdot f$, where M is some atom in the body of the recursive rule, and that $S(c,d)$ was computed by processor i, $i \neq j$. Obviously, P is not first-fixed (otherwise $h(c)=j$). We shall prove that for any other program, $S(c,d)$ must satisfy the predicate T_{ij}, and therefore must have been transmitted to j.

(1) A first-consistent program. In such a program, x appears in all the S-atoms in the body of the recursive rule. We know that $h(c) \neq j$ and that f maps x to a, and that $h(a)=j$. Thus $a=d$, and $S(c,d)$ must satisfy T_{ij}.

(2) A partially-first-fixed program, but not first-consistent. In such a program, every atom in the recursive rule is either an atom with a repeated variable, or an atom with the variable x in the first position. Since $h(c) \neq h(a)$, and consequently $a \neq c$, M is an atom with a repeated variable. Thus $S(c,d)$ is a one-constant fact, and therefore satisfies T_{ij}.

(3) A partially-first-consistent program, but not first-consistent, nor partially-first-fixed. In such a program, every atom in the recursive rule is either an atom with a repeated variable, or it has the variable x in one of its positions. Since $h(c) \neq h(a)$, M is either with a repeated variable, or x is in its second position. Therefore $c=d$, or $d=a \wedge h(d)=j$, and consequently $S(c,d)$ satisfies T_{ij}.

(4) A partially-discriminating program, but not first-consistent, nor partially-first-fixed, nor partially-first-consistent. By definition of a partially-discriminating program, $S(a,b)$ can be derived from *body*·f without using any one-constant facts other than $S(a,a)$. We also know that if $S(a,a)$ is computed, then it must be computed by processor j. Since $S(a,b)$ is not computed, we conclude that $S(c,d)$ is a two-constant fact, and therefore satisfies T_{ij}.

(5) A program which is not of the previous kinds. In such a program, every fact satisfies T_{ij}, particularly $S(c,d)$. □

6.4 The Performance of Strategy DS3

To estimate DS3 performance, we consider two factors: evaluation cost and data-communication overhead. In this subsection we refer to these factors, and we start with the evaluation cost. Intuitively, time-saving occurs in DS3 compared to the single-processor evaluation, 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, the evaluation cost of processor j (smaller S), is reduced. Third, the evaluable predicate $h(x)=i$ cuts the size of every relation having x as an attribute; consequently the joins involving such relations are cheaper. Formally, we establish in the next theorem, that for linear programs the total evaluation-cost of all the processors participating in DS3 is minimal.

Theorem 6.2: Let $\{A_1, \ldots, A_r\}$ be a scheme in DS3 that evaluates a linear program P. Denote by A the algorithm that semi-naively evaluates P on a single processor. For every input I to P: $\sum_{i=1}^{r} cost^2(A_i) \le cost^2(A)$ ($cost^2$ is the measure introduced in subsection 4.3).

Proof: We show that (i) Among the algorithms $A_1, \ldots, A_r$, every successful inference is performed by at most one of the algorithms, and in that algorithm, the successful inference is performed only once. (ii) Every successful inference performed in $\{A_1, \ldots, A_r\}$, is also performed in A.

(i) The scheme $\{A_1, \ldots, A_r\}$ is disjoint, thus every successful inference is performed by at most one of the algorithms. Additionally, each A_i is semi-naive, and the program is linear, thus every successful inference performed by A_i, is performed only once (when the differential relation includes the S-fact in the body of the instantiated rule).

(ii) Consider a successful inference, performed by some algorithm in $\{A_1, \ldots, A_r\}$. The S-fact in the body of the instantiated rule is in the output of A (Theorem 6.1), thus belongs to the differential relation in one of the iterations of A. In that iteration, the inference is performed by A. □

Next, we establish overhead minimality in the following sense. Let A be a decomposition scheme for the partial computation of a program P. For an input I to P, the total number of transmitted facts in $A_1(I), A_2(I), \ldots, A_r(I)$ is the *overhead of A for I* (transmitted facts are defined in subsection 4.1). Given a scheme B in DS3, a decomposition scheme A for the partial computation of the same program, is called *B-alike* if it satisfies the following condition: for every input, A computes the same facts as B at every processor (although A may transmit more or less of them).

For an input I to P, let $S(\vec{e})$ be a computed fact in $A_i(I)$, and a transmitted fact in $A_j(I)$. In other words, *i transmits* $S(\vec{e})$ to j. Then, A is *simple* if for every input, I' to P, the following is satisfied. If $S(\vec{e})$ is a computed fact in $A_i(I')$, then it is a transmitted fact in $A_j(I')$. Intuitively, in a simple scheme, if for some input, a processor, i, sends a computed fact, $S(\vec{e})$, to another processor, j, then for every other input, i will send $S(\vec{e})$ to j. This means that processor i does not incur the additional work of determining when to send a fact and when not to do so. Note that every scheme in DS3 is simple.

To prove the minimal overhead theorem, we first prove the following lemma.

Lemma 6.1: Let P be a program, let $A=\{A_1, \ldots, A_r\}$ be a scheme in DS3 for partial computation of P, and let h be its hash function. Let f be a 1-1 instantiation of the recursive rule of P, such that $S(a,b)$ is the instantiated head, and $S(c,d)$ is in the instantiated body. Assume that $h(a)=j$ and $h(c)=i$. Then, in every simple A-alike scheme, $B=\{B_1, \ldots, B_r\}$, and for every input I, if $S(c,d)$ is computed in $B_i(I)$, then $S(c,d)$ is transmitted in $B_j(I)$.

Proof: Let I_0 be the S-to-B substitution of *body*$\cdot f$. In any scheme for the partial computation of P, for the input I_0, $S(a,b)$ is preceded by all the facts in *body*$\cdot f$, as computed facts or as transmitted ones. Since B is A-alike, $S(a,b)$ is a computed fact in $B_j(I_0)$, but $S(c,d)$ is not; thus $S(c,d)$ is a transmitted fact in $B_j(I_0)$. Since B is simple, this is satisfied for every input. □

The following theorem indicates that DS3 cannot transmit less facts than it actually does.

Theorem 6.3: For every input, a scheme, A, in DS3 has a minimal overhead, among all the simple A-alike schemes.

Proof: Consider an input I to $A=\{A_1, \ldots, A_r\}$, and a fact $S(c,d)$ which is a computed fact in $A_i(I)$, and a transmitted fact in $A_j(I)$, i.e. $T_{ij}(S(c,d)) = TRUE$. We show that in every simple A-alike scheme, $B=\{B_1, \ldots, B_r\}$, with the input I, the fact $S(c,d)$ is transmitted from i to j. We show it by demonstrating a 1-1 instantiation of the recursive rule, in which $S(c,d)$ is in the instantiated body, and another fact $S(a,b)$, such that $h(a)=j$, is the instantiated head. Since $S(c,d)$ is computed by processor i in A, $h(c)=i$, then using Lemma 6.1 the theorem follows.

We break down the analysis by program classes.

(1) A first-consistent program, but not first-fixed. $S(c,d)$ is a transmitted fact in $A_j(I)$ and $h(c)=i$. Thus (see flow-chart in figure 6.2) it is a two-constant fact, and $h(d)=j$. In such a program the head of the recursive rule has the variable x in the first position while there is an atom N in the body with x in the second position. A 1-1 instantiation, f, in which $N\cdot f = S(c,d)$ is the desired one.

(2) A partially-first-fixed program, but neither first-consistent, nor first-fixed. Again, by the flow-chart in figure 6.2, $c=d$. In the body of the recursive rule there is an atom with a repeated variable, $S(z,z)$. A 1-1 instantiation f in which $f(z)=c$, and $f(x)=a$ such that $h(a)=j$, is the desired one.

(3) A partially-first-consistent program, but neither partially-first-fixed, nor first-consistent, nor first-fixed. In this case, $S(c,d)$ is either a one-constant fact, i.e. $c=d$, or a two-constant fact, and $h(d)=j$. If $c=d$, then the desired instantiation exists as argued in case (2). If $c\neq d$, then the desired

instantiation exists as argued in case (1).

(4) A partially-discriminating program, but neither partially-first-consistent nor partially-first-fixed, nor first-consistent, nor first-fixed. In this case, $c \neq d$. Also, there is an S-atom in the recursive rule, N, without a repeated variable and without the variable x (otherwise the program is of previous kinds, or first-fixed). Let a be a constant different than both, c and d, such that $h(a)=j$. Then a 1-1 instantiation, f, in which $N \cdot f = S(c,d)$, and $f(x)=a$ is the desired instantiation.

(5) A program which is neither of the four previous kinds, nor first-fixed. If $c \neq d$, then the desired instantiation exists as argued in case (4). If $c=d$, we do not search for a 1-1 instantiation as before, but use a different approach. Since the program is not partially-discriminating, there is an instantiation, f', such that if I is the S-to-B substitution of $body \cdot f'$, then the following holds. Every derivation tree to $head \cdot f'$ with leaves in I contains a one-constant B-fact other than $B(f'(x),f'(x))$. Consider a subset of I, denoted I', such that $head \cdot f' \in O(P,I')$, and the number of one-constant B-facts in I' is minimal (i.e., if we eliminate a one-constant B-fact from I', then $head \cdot f'$ is not any more in the output). Obviously, I' has a one-constant B-fact other than $B(f'(x),f'(x))$. Assume it is $B(m,m)$, and $m \neq f'(x)$. Let a be a constant satisfying $h(a)=i$, and suppose, without loss of generality, that the constants c and a are not in I' (otherwise we can add $a+c+1$ to all the constants in I'). Furthermore, denote by I'', the input obtained from I', by replacing each occurrence of the constant m by c, and each occurrence of the constant $f'(x)$ by a. The number of one-constant B-facts in I'' is minimal. Thus, for the input I'', $S(c,c)$ has to be transmitted from processor i to processor j in any A- alike scheme, B (otherwise processor j cannot compute $S(a,b)$, contradicting completeness for I''). If B is also simple, then $S(c,c)$ has to be transmitted from processor i to processor j for every input that produces it. □

Note that DS3 can be easily extended to arbitrary datalog programs, provided that an algorithm sends all the new tuples computed at each iteration, to all the other processors. In conclusion, the properties of the strategies discussed in this paper are summarized in the table of Figure 2 (following the references section).

7. 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 single-rule programs, with respect to both notions. Our notion of program-decomposability may be related to algebraic-operator decomposition, discussed in [IW], and to clausal decomposition, discussed in [LM] (although both papers, in contrast to ours, do not require disjointness of the output sets, and not provide a syntactic characterization of programs). In the future, we intend to investigate these possible relationships.

Then we studied data-reduction parallelization and started by examining pure parallelization. We 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 strategy DS3, that can be used for parallelization of every program. Strategy 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 extend the concept of data-reduction parallelization to all Datalog programs, and other rule-based languages, such as OPS5 ([BFKM]). Also, it would be interesting to devise general methods of combining data-reduction parallelization, with single processor optimization techniques. 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:

$MAGIC(xp){:-}\ MAGIC(x),\ PARENT(x,xp)$

$MAGIC(a)$

$SG(x,x){:-}\ H(x)$

$SG(x,y){:-}\ MAGIC(xp),\ PARENT(x,xp),\ PARENT(y,yp),\ SG(xp,yp)$

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

Finally, we intend to study the enhancement of data-reduction with some interesting ideas on parallel processing, that appeared in the literature ([D, GST, HAC, R, RSL, VK]).

Acknowledgement:

We thank the referees, Michael Kifer, Oded Shmueli, and Victor Vianu, for helpful suggestions and comments on this paper. Oded Shmueli also suggested the method of communication among the processors used in DS3.

8. REFERENCES

[AJ] R. Agrawal and H.V. Jagadish, "Multiprocessor Transitive Closure Algorithms ", AT&T Bell Laboratories Manuscript, 1988.

[AP] F. Afrati and C. H. Papadimitriou "The Parallel Complexity of Simple Chain Queries", Proc. 6th ACM Symp. on PODS, pp. 210-213, 1987.

[B] C. Beeri "Data Models and Languages for Databases", Proc. ICDT 1988.

[Ban] F. Bancilhon "Naive Evaluation of Recursively Defined Relations", in On Knowledge Base Management Systems - Integrated Database and AI Systems, Brodie and Mylopoulos, Eds., Springer-Verlag.

[Bay] R. Bayer, "Query Evaluation and Recursion in Deductive Database Systems", unpublished manuscript, 1985.

[BBDW] D. Bitton, H. Boral, D.J. DeWitt and W.K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations", ACM TODS, 8(3), 1983.

[BFKM] L. Brownston, R.Farrell, E. Kant, N. Martin, "Programming Expert Systems in OPS5", Addison Wesley, Reading, Massachusetts, 1985.

[BMSU] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman "Magic Sets and Other Strange Ways to Implement Logic Programs", Proc. 5th ACM Symp. on PODS, pp. 1-15, 1986.

[BR] F. Bancilhon and R. Ramakrishnan "Performance Evaluation of Data Intensive Logic Programs" in Foundations of Deductive Databases and Logic Programming, Ed. J. Minker, Morgan-Kaufman, 1988.

[C] S. Cook, "An Observation on Time Storage Tradeoff", JCSS 9(3), pp. 308-316, 1974.

[CK] S. S. Cosmadakis and P. C. Kanellakis "Parallel Evaluation of Recursive Rule Queries", Proc. 5th ACM Symp. on PODS, pp. 280-293, 1986.

[D] G. Dong, "On Distributed Processibility of Logic Programs by Decomposing Databases", Proc. ACM-SIGMOD conf., 1989.

[DIY] D.M. Dias, B.R. Iyer, P.S. Yu, "On Coupling Many Small Systems for Transaction Processing", Research Report RC11722, IBM T.J. Watson Research Center.

[DL] D. DeGroot and G. Lindstrom eds. " Logic Programming - Functions Relations and Equations", Prentice Hall, 1986.

[GST] S. Ganguly, A. Silberschatz, S. Tsur, "A Framework for the Parallel Processing of Queries", Manuscript, Comp. Sci. Dept., Univ. of Texas at Austin, 1989.

[HAC] M. W. Houtsma, P. M. G. Apers, and S. Ceri, "Parallel Computation of Transitive Closure Queries on Fragmented Databases", University of Twente, TR INF-88-56, Dec. 1988.

[IS] T. Ishida, S.J. Stolfo, "Towards the Parallel Execution of Rules in Production System Programs", Proc. of the 13th annual international symposium on Computer Architecture, pp. 28-37, IEEE/ACM, 1986.

[IW] Y. E. Ioannidis and E. Wong, "Towards an Algebraic Theory of Recursion", University of Wisconsin, CS department, TR #801, Oct. 1988.

[K] P. C. Kanellakis "Logic Programming and Parallel Complexity", Proc. ICDT '86, International Conference on Database Theory, Springer-Verlag Lecture Notes in CS Series, no. 243, pp. 1-30, 1986.

[LM] J. -L. Lassez and M. J. Maher, "Closures and Fairness in the Semantics of Programming Logic", Theoretical Computer Science 29, pp. 167-184, 1984.

[LY] M. S. Lakshmi, P. S. Yu "Effect of Skew on Join Performance in Parallel Architectures", to appear, Proc. of the Int. Symp. on Databases in parallel and Distributed Systems. Austin TX, Dec. 1988.

[M] D.P. Miranker, "Recent Developments in Parallel Production System Algorithms", University of Texas at Austin, manuscript, 1988.

[MNSUV] K. Morris, J. Naughton, Y. Saraiya, J.D. Ullman and A. Van Gelder, "YAWN! (Yet Another Window on NAIL!)" Unpublished Manuscript.

[MW] D. Maier and D. S. Warren "Computing with Logic: Introduction to Logic Programming", Benjamin-Cummings Publishing Co., 1987.

[P] A.J. Pasik, "A Methodology for Programming Production Systems and its Implications on Parallelism", Ph.D. Thesis, Columbia University, 1989.

[R] R. Ramakrishnan, "Parallelism in Logic Programs", Univ. of Wisconsin, Computer Sci. Dept, TR #892, Nov. 89.

[RSL] L. Raschid, T. Sellis, and C. C. Lin, "Exploiting Concurrency in a DBMS Implementation of production Systems, Proc. International Symposium on Databases in Distributed and Parallel Systems, Austin TX, Dec. 1989.

[Sa] Y. Sagiv "Optimizing Datalog Programs," *Proc. 6th ACM Symp. on PODS*, pp. 349-362, 1987.

[Sh] E. Y. Shapiro "Concurrent Prolog, Collected Papers", MIT Press, 1987.

[St] S.J. Stolfo, "Five Parallel Algorithms for Production System Execution on the DADO Machine", Proc. of the National Conference of Artificial Intelligence, 1984.

[SMM] S.J. Stolfo, D.P.Miranker and R.Mills, "A simple processing scheme to extract and load balance implicit parallelism in the concurrent match of production rules", In proc. of the AFIPS symp. on fifth generation computing, AFIPS,1985.

[TM] M.F.M. Tenorio, D.I. Moldovan, "Mapping Production Systems into Multiprocessors", Proc. of the 13th annual international symposium on Computer Architecture, IEEE/ACM, 1986.

[U] J.D. Ullman, "Database and Knowledge-base Systems Volume 1", Computer Science Press, 1988.

[UV] J.D. Ullman and A. Van Gelder, "Parallel Complexity of Logic Programs", TR STAN-CS-85-1089, Stanford University.

[VK] P. Valduriez and S. Khoshafian, "Parallel evaluation of the Transitive Closure of a Database Relation", International Journal of Parallel Programming 17,1, Feb. 1988.

[W] O. Wolfson, "Parallel Bottom-Up Evaluation of Datalog Programs by Load Sharing", TR CUCS-509-89 computer science dept., Columbia Univ., Some of the results appear in Proc. of the Intl. Symp. on Databases in Parallel and Distributed Systems, Austin, TX, Dec. 1988.

[WO] O. Wolfson and A. Ozeri, "A New Paradigm for Parallel and Distributed Rule-Processing", Proceedings of the ACM-SIGMOD 1990, International Conference on Management of Data, Atlantic City, NJ, May 1990. Also, TR CUCS-011-90, Department of Computer Science, Columbia University.

[WS] O. Wolfson and A. Silberschatz, "Distributed Processing of Logic Programs," Proc. of the ACM-SIGMOD Conf., pp. 329-336, 1988.

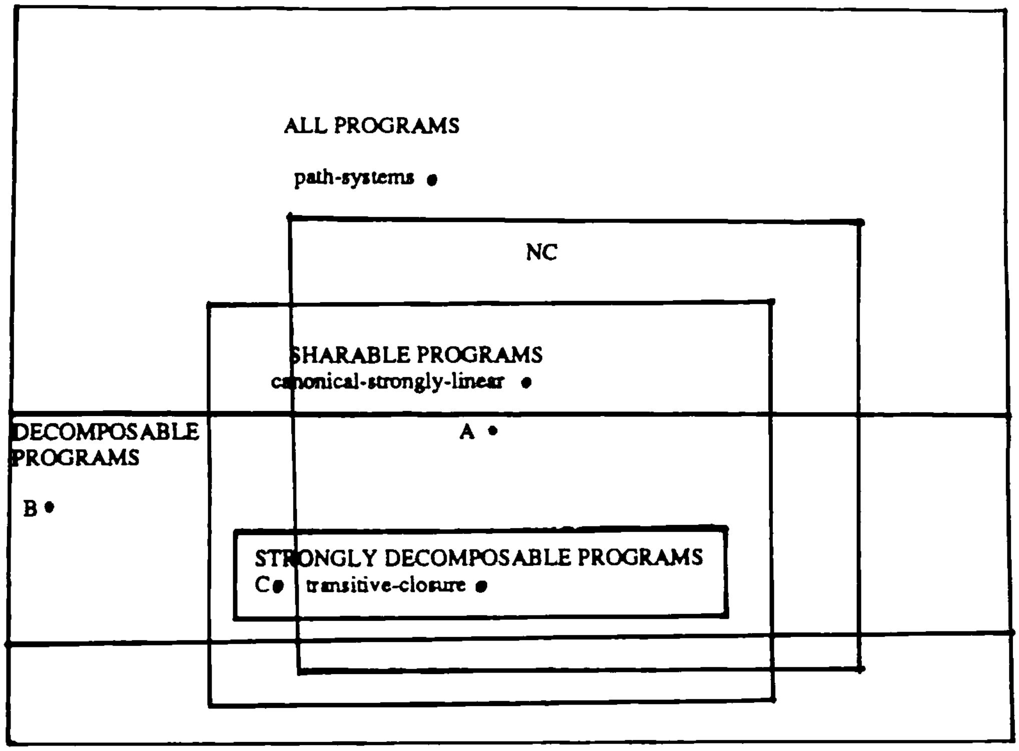

Figure 1: When considering amenability to pure parallelization, the following logic-program class structure is exhibited. The strongly decomposable programs are most amenable to pure parallelization. A representative of this class is the transitive closure program: S(x,y):- S(x,z),A(z,y). Next in the hierarchy, is the class of sharable programs. A representative of this class is the canonical strongly linear program: S(x,y):-UP(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, and the NC complexity class. Point A represents the program: S(x,x):-S(y,y),A(x,y). Point B represents the program: S(x,x):-S(y,y),S(z,z),H(x,y,z). Point C represents the program: S(w,x):-S(w,y),S(w,z),H(x,y,z).

	Applicable to Programs	Overhead	Load Distribution	Total Cost
DS1	strongly decomposable	no overhead	static	minimal
SS1	linear	no overhead	static	not minimal
DS2	strongly decomposable	control overhead	dynamic	minimal
SS2	linear	control overhead	dynamic	not minimal
DS3	all	minimal data overhead	static	minimal for linear programs

Figure 2

APPENDIX A: DISCRIMINATING PROGRAMS

In this appendix we provide two algorithms. The first, A.1, to determine whether or not a program is partially-discriminating, and the other, A.2, to determine whether or not it is discriminating. The two algorithms are quite similar. Both check all the partitions of the variables in the recursive rule, r, of the tested program, P. A *partition* of the variables is a set of pairwise disjoint subsets, such that each variable is in some subset. For each partition, the algorithms consider a corresponding instantiation. An instantiation, f_i, of r, *corresponds to a partition* p_i if for every two variables, the following is satisfied: they are mapped to different constants by f_i, if and only if they are in different subsets of p_i. Note that any instantiation of the recursive rule corresponds to a partition of the variables of that rule. Additionally, instantiations corresponding to the same partition are equivalent, and consequently, only one representative of each equivalence class is considered in our algorithms.

Algorithm A.1 below determines whether or not a program is partially-discriminating. It does so by simply checking the definition, and thus its correctness is trivial.

Algorithm A.1:

```
(1)  Denote by $p_1, \ldots, p_k$ the partitions of the variables in $r$.
     For each partition, $p_i$, do:
       Consider an instantiation $f_i$ of $r$, corresponding to $p_i$. If $head \cdot f_i \notin O(P, I^{+}_{f_i})$ then $P$ is not partially-discriminating. Halt. (The notation $I^{+}_{f_i}$ is introduced in subsection 6.2).
     End.
(2)  $P$ is partially-discriminating.    □
```

The algorithm is exponential in the size of the program, but we assumed, as other works (e.g. [UV]), that the size of the program is a constant. Next we provide the algorithm, A.2, for determining whether or not a program is discriminating.

Algorithm A.2:

```
(1)  If the head of $r$ is an atom with a repeated variable, then $P$ is not discriminating. Halt.
     Otherwise, 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.
(2)  Denote by $p_1, \ldots, p_k$ the partitions of the variables in $r$.
     For each partition, $p_i$, do:
       Consider an instantiation, $f_i$, of $r$, corresponding to $p_i$. Let $I$ be the input obtained by the $S$-to-$B$ substitution of $body \cdot f_i$. If $head \cdot f_i \notin body \cdot f_i$ and $head \cdot f_i \notin nt(I^{*})$, then $P$ is not discriminating. Halt. (The notation $I^{*}$ is explained in subsection 3.2).
     End.
```

(3) P is discriminating. □

In contrast to A.1, correctness of A.2 is not trivial, and the difficulty is due to the difference in definitions of partially-discriminating and discriminating programs. A partially-discriminating program is defined in terms of inputs created by instantiations of the recursive rule. In contrast, the definition of a discriminating program is in terms of an arbitrary input.

Theorem A.3: Algorithm A.2 correctly determines whether or not a program is discriminating.

Proof: If the algorithm halts in step (2), then we found an instantiation f_i, and an input I (which is the S-to-B substitution of $body{\cdot}f_i$) such that (i) $head{\cdot}f_i \in nt(I)$ (because we required that $head{\cdot}f_i \notin body{\cdot}f_i$), and ($ii$) $head{\cdot}f_i \notin nt(I^*)$. Thus for this input, $nt(I) \neq nt(I^*)$.

If the algorithm halts in step (3), then we shall show that the program is discriminating. Assume, by way of contradiction, that it is not discriminating. The program is reverse, thus there is an input INP, for which $nt(INP) \neq nt(INP^*)$. Thus, there is a derivation tree, T, that satisfies the following condition: the root, $S(\vec{e})$, of T is in $nt(INP)$ (i.e. cannot be derived by the exit rule), but is not in $nt(INP^*)$, Without loss of generality, we assume minimality in the following sense. T does not have any subtree whose root is in $nt(INP) - nt(INP^*)$. Furthermore, we shall assume, again without loss of generality, that for every subtree of T, if its root is in $nt(INP^*)$, then all its B-leaves are two-constant facts (by definition of $nt(INP^*)$, the root has such a derivation tree). Since the root of T is not in $nt(INP^*)$, T has a one-constant B-leaf, say $B(i,i)$. Consider its father, $S(i,i)$, and all $S(i,i)$'s brothers. They represent an instantiation, say g. Clearly, $head{\cdot}g$ is not in $nt(INP^*)$, since its derivation tree has $B(i,i)$ as a leaf. By minimality, $head{\cdot}g$ is the root, $S(\vec{e})$. Denote by I_g the S-to-B substitution of $body{\cdot}g$. We shall show that $S(\vec{e}) \notin nt(I_g^*)$. If $S(\vec{e}) \in nt(I_g^*)$, then either every S-fact in $body{\cdot}g$ is not in $nt(INP)$ (i.e. can be derived from the exit rule), or is in $nt(INP^*)$ (by minimality). In both cases, $S(\vec{e}) \in nt(INP^*)$, contradicting the way T was chosen. Consequently, $S(\vec{e}) \notin nt(I_g^*)$. Let f be the instantiation we created at step (2) of algorithm A.2, when we considered the same partition of variables as g performs. Let I_f be the S-to-B substitution of $body{\cdot}f$. Clearly, $head{\cdot}f \in body{\cdot}f$ if and only if $head{\cdot}g \in body{\cdot}g$. Since the tree is free of useless instantiations, $head{\cdot}f \notin body{\cdot}f$. Also $head{\cdot}f \in nt(I_f^*)$ if and only if $head{\cdot}g \in nt(I_g^*)$. Thus, $head{\cdot}f \notin nt(I_f^*)$, and the algorithm must have stopped at step (2). □

APPENDIX B: EXAMPLES

This appendix demonstrates by an example the execution of strategy DS3, using two processors, $P0$ and $P1$. The evaluation method of each processor is semi-naive. We use *the canonical strongly linear (csl)* program:

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

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

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, 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 integer k.

Let the input to csl be the extensional database relations of Figure B.1. 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 $\{(i, 6) \mid i = 1,...,5\}$.

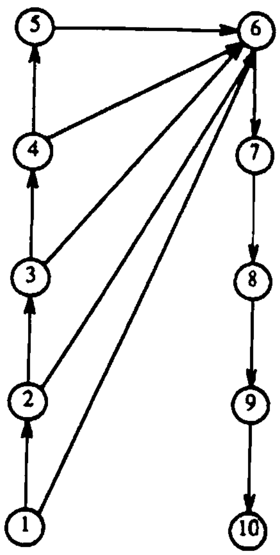

Figure B.1: Sample input to the csl program.

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 in this example, since both processors do approximately the same amount of work at each iteration.

Next we explain the example, and figure B.2 which summarizes it. $P0$ evaluates csl with the predicate $x \bmod 2 = 0$ appended to the exit and recursive rules, therefore, it starts with the differential $\{(2,6),(4,6)\}$. $P1$ appends the predicate $x \bmod 2 = 1$ 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 tuples-exchange the differentials of both processors become the same. Actually after all the following tuples-exchanges, 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 table of figure B.2. A final fix-point is reached after the sixth iteration.

For comparison, consider strategy SS1, in which $P0\,(P1)$ evaluates *csl* with the predicate $x \bmod 2 = 0$ $(x \bmod 2 = 1)$ 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.

For SS1, DS3 and the serial semi-naive evaluation, we summarize the relation sizes at each iteration of each strategy, in the table of Figure B.3. 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 csl 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 csl case. Second, the size-cutting variable x, appears in only one relation.

	Processor 0		Processor 1	
iteration	sizes of UP,ΔS,DOWN	the differentials ΔS	sizes of UP,ΔS,DOWN	the differentials ΔS
1	2,2,4	(2,6)(4,6)	2,3,4	(5,6)(3,6)(1,6)
1.1		∅		∅
2	2,3,4	(1,6)(3,6)(5,6)	2,2,4	(2,6)(4,6)
2.1		(2,7)(4,7)		(1,7)(3,7)
3	2,4,4	(2,7)(4,7)(1,7)(3,7)	2,4,4	(2,7)(4,7)(1,7)(3,7)
3.1		(2,8)		(1,8)(3,8)
4	2,3,4	(2,8)(1,8)(3,8)	2,3,4	(2,8)(1,8)(3,8)
4.1		(2,9)		(1,9)
5	2,2,4	(2,9)(1,9)	2,2,4	(2,9)(1,9)
5.1		∅		(1,10)
6	2,1,4	(1,10)	2,1,4	(1,10)
6.1		∅		∅

Figure B.2: 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.

	single-processor	SS1		DS3	
iteration		processor 0	processor 1	processor 0	processor 1
1	4,5,4	4,2,4	4,3,4	2,2,4	2,3,4
2	4,4,4	4,2,4	4,2,4	2,3,4	2,2,4
3	4,3,4	4,1,4	4,2,4	2,4,4	2,4,4
4	4,2,4	4,1,4	4,1,4	2,3,4	2,3,4
5	4,1,4		4,1,4	2,2,4	2,2,4
6				2,1,4	2,1,4

Figure B.3: Performance comparison. The table entries consist of three numbers, for the size of relations UP, the differential ΔS, and DOWN respectively.

APPENDIX C

In this appendix we provide the semi-naive evaluation algorithm for a single rule program. We use Ullman's terminology ([U]). Denote the atoms in the body of the recursive rule by $T_1, \ldots, T_k$. The function $EVAL\_RULE(T_1, \ldots, T_k)$, is a relation for the predicate S. The relation consists of all the tuples that can be derived by instantiations of the recursive rule, that use facts from relations assigned to the T_i's. It is obtained by joining these relations. $EVAL\_RULE\_INCR(T_1,...,T_k)$, that computes the differential relation obtained from the recursive rule, is defined as $\bigcup_{i=1}^{m} EVAL\_RULE(T_1, \ldots, T_{i-1}\ ,\ \Delta T_i\ ,\ T_{i+1}, \ldots, T_k)$ where the assigned relations are as follows. For $j \neq i$, the assigned relation is the extensional relation, or s, depending on the predicate symbol of T_i (the lowercase letters denote the relations for the corresponding uppercase predicate symbols). For $j = i$, if T_j has an extensional predicate symbol, then the assigned relation is ϕ, otherwise it is Δs (the differential relation computed by the previous iteration).

Semi-Naive Evaluation:

```
    $\Delta s \leftarrow$ the $B$-to-$S$-substitution of the relation $b$.
    $s \leftarrow \Delta s$
    repeat
        $\Delta s \leftarrow EVAL\_RULE\_INCR(T_1, \ldots, T_k)$
        $\Delta s \leftarrow \Delta s - s$        * remove tuples that appeared before *
    until $\Delta s = \phi$
```