

# A NEW PARADIGM FOR PARALLEL AND DISTRIBUTED RULE-PROCESSING

Ouri Wolfson<sup>1</sup>  
Department of Computer Science  
Columbia University  
New York, NY 10027  
wolfson@cs.columbia.edu

and

Aya Ozeri  
Department of Electrical Engineering  
The Technion - IIT  
Haifa, Israel

## ABSTRACT

This paper is concerned with the parallel evaluation of datalog rule programs, mainly by processors that are interconnected by a communication network. We introduce a paradigm, called data-reduction, for the parallel evaluation of a general datalog program. Several parallelization strategies discussed previously in [CW, GST, W, WS] are special cases of this paradigm. The paradigm parallelizes the evaluation by partitioning among the processors the instantiations of the rules. After presenting the paradigm, we discuss the following issues, that we see fundamental for parallelization strategies derived from the paradigm: properties of the strategies that enable a reduction in the communication overhead, decomposability, load balancing, and application to programs with negation. We prove that decomposability, a concept introduced previously in [WS, CW], is undecidable.

## 1 Introduction

A knowledge base is a relational database augmented with a rule program, e.g. datalog (see [MW]). In this paper we continue the study of parallelization in knowledge bases, begun in [WS, W, CW]. The emphasis in these works was on parallelization

without communication overhead, namely pure parallelization. This type of parallelization is restricted in its applicability, only some classes of programs can be purely parallelized. To overcome this limitation, in [CW] we proposed a strategy that does not incur an overhead, but can be applied to every single-rule program without constants. We show that all the strategies discussed in our previous works are special cases of the data-reduction paradigm, that we introduce in this paper. It stipulates that parallelization is obtained by having each processor evaluate the original program, but with less data. Generally, this is also what parallel processors do when adding two vectors in parallel. In this paper we demonstrate that this idea is applicable to rule processing, and analyze it in this context. In extension to the strategies discussed in our previous works, data-reduction is applicable to datalog programs with multiple rules, constants, and negation.

A data-reduction strategy is obtained as follows. Every single-processor datalog evaluation method can be regarded as a sequence of rule-instantiations, in each rule-instantiation, the variables in the rule are replaced by constants from the input. The purpose of data-reduction is to partition the instantiations among multiple processors, such that if each processor uses a single processor method to evaluate the original program, then it processes less data. The works on the NC-complexity of programs (e.g. [AP, CK, K, UV]) also partition the instantiations, but they assign one instantiation to a processor, assuming a polynomial (in the database size) number of processors. The works identify the programs for which the evaluation can complete in polylogarithmic time. If the number of processors is constant (as we assume), then the NC-type of evaluation algorithms can be adapted, by assigning the work of multiple processors to a single processor. However, it turns out that, which multiple processors are assigned to a single one, i.e. how the pieces of work are grouped, is very important as far as overhead (particularly if the

---

<sup>1</sup> This research was supported in part 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

processors do not have shared memory) and evaluation cost are concerned. The present paper studies the partitioning of work among the processors.

Since this paper is devoted to data-reduction issues, in the next paragraphs we explain the paradigm in detail, and point out its relationship to other relevant work. The database community observed that given massive amounts of data, a declarative program, such as datalog, should be evaluated in a set-oriented, rather than tuple-oriented (a la Concurrent Prolog [DL, Sh]) fashion. The set-oriented, or relational, evaluation of a program  $P$  amounts to iteratively computing a relational algebra expression for each rule of  $P$ , until a fix-point is reached ([U]).

**Example 1** Consider the transitive closure program  $P_1$

$$\begin{aligned} S(x,y) &= S(x,z), A(z,y) \\ S(x,y) &= A(x,y) \end{aligned}$$

The naive evaluation of  $P_1$  initializes the relation  $S$  to  $A$  and then computes the relational algebra expression

$$(1) S(x,y) = S(x,y) \cup \pi_{x,y}(S(x,z) \text{ join } A(z,y))$$

until no new tuples are added to  $S$   $\square$

**Example 2** The program  $P_2$

$$\begin{aligned} S(y) &= S(z), A(z,y) \\ S(y) &= A(a,y) \end{aligned}$$

that finds all the nodes of a graph reachable from the node  $a$  (a constant). The semi-naive evaluation of  $P_2$  initializes  $\Delta S$  and  $S$  to the tuples of  $A$  that have the constant  $a$  in their first position, and then it evaluates the following expressions

$$\begin{aligned} (2) \Delta S(y) &= \pi_y(\Delta S(y) \text{ join } A(z,y)) - S(y) \\ S(y) &= S(y) \cup \Delta S(y) \end{aligned}$$

iteratively, until  $\Delta S = \emptyset$   $\square$

A way of partitioning the rule-instantiations among the processors is the following. Assume that there are  $k$  processors, all of which have access to the extensional database (it is either replicated, or resides in some common memory, or can be received from a processor that "owns" it). It is possible to partition the computation of relational algebra operations among the processors. For this purpose, one can use some technique from the existing literature on parallel evaluation of relational operations (e.g. [BDDW]). However, we postulate that the work partitioning can better be performed by appending some predicate  $h = j$ , to the body of each rule, where  $h$  is some hash function, and  $j$  is the identification of some processor. The function  $h$  gets as arguments a subset of the variables in the relational expression, and it maps each instantiation of the variables into a unique processor of the set of processors  $\{p_0, \dots, p_{k-1}\}$ . The next paragraph motivates this postulate.

Consider the relational algebra expression (1) of Example 1. Assume that the optimal way of joining  $S$  and  $A$  on a single processor is by a nested loop, where  $S$  is the outer relation, and  $A$  is the inner one, each block of  $S$ , in sequence, is joined with the appropriate blocks of  $A$ . This is a likely situation, considering that  $A$  will probably have an index on the column  $z$ , whereas  $S$ , constructed dynamically, will probably not. In order to divide the

work between two processors we can use a hash function (e.g.  $x \bmod 2$ ) that maps each  $x$ -value into either  $p_0$  or  $p_1$ . Assume that the hash function maps half the  $x$ -values in  $S$  to  $p_0$ , and the other half to  $p_1$ , and the distribution of  $S$ -facts among the  $x$ -values is uniform. Then the computation time is clearly split in half.

Suppose now that a single processor evaluation method is parallelized by parallelizing each join operation as explained above (and possibly other relational algebra operations as well). This has two drawbacks. First, the  $k$  processors must be synchronized at each join, they all complete the computation of the join in one iteration, exchange the newly generated tuples, and then begin the next iteration. Second, as shown in our previous works (e.g. [WS]), many tuples are transmitted unnecessarily among the processors. However, the evaluation load can be partitioned without the negative side effects, by appending the hash functions to the rules from which the relational expressions are derived, rather than to the expressions themselves. Then each processor evaluates the modified version of the program, obtained in this fashion. The hash function appended to each rule depends on the evaluation method (semi-naive, Henschen-Naqvi, or another (see [BR])), and on the access plan for computing the relational algebra expression for the rule. It is selected with the purpose of best dividing the processing load among the processors. The question of how to achieve this purpose algorithmically is outside the scope of this paper. However, we use the semi-naive evaluation method for demonstrating our ideas.

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]). However, the issues discussed in this paper have not been addressed previously.

In this paper, we first introduce the paradigm for datalog programs without negation, and we discuss how it is specialized to a particular parallel algorithm. Then we discuss some desirable properties of strategies. These are single-source and single-destination, which enable a lower communication overhead. For a given strategy, we present a sufficient condition for each one of these properties. Actually, the decomposability concept discussed [CW] is a combination of the single-source and single-destination properties. Specifically, the decomposable programs are the ones for which there exist strategies that have both properties. Therefore we ask whether these programs can be characterized algorithmically. Unfortunately, we prove that it is undecidable to determine whether a program is decomposable. We also point out that this result cannot be straight-forwardly obtained from a Rice style theorem in [GMSV]. Then we address the problem of load balancing. Particularly, we discuss changing the data-reduction strategy while the parallel evaluation is in progress. It turns out that this change of strategy can be performed more efficiently for a linear program. Finally, we discuss how to extend the paradigm to programs with stratified negation. The data-reduction paradigm among the processors. However, the same paradigm requires synchronization when applied to programs with negation. It indicates that there is a relationship, that we feel is fundamental in parallel computation, between monotonicity and synchronization. We also show that the single source and destination properties, when present, enable the elimination of the need for synchronization.

The rest of the paper is organized as follows. In section 2 we introduce the terminology used throughout the paper, and in section 3 we present the paradigm. In section 4 we discuss the variables of the paradigm that have to be fixed in order to obtain a parallel evaluation algorithm. In section 5 we discuss one important variable of the paradigm, that determines the overhead, namely the transmission set of tuples, between processors. In section 6 we discuss the single-source and single-destination properties, and in section 7 we prove the undecidability result. In section 8 we address the problem of balancing the load by strategy change. In section 9 we discuss the application of the paradigm to datalog programs with stratified negation, in section 10 we conclude, and in section 11 we discuss future work.

## 2 Preliminaries

In this section we define the basic terminology. A *literal* is a predicate symbol followed by a list of arguments. An *atom* is a literal with a constant or a variable in each argument position. A *constant* is any natural number. (The results in this paper are applicable to character strings as well, since their binary representation is a natural number.) The other arguments of an atom are the *variables*. 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. A *rule* consists of an atom designated as the *head*, and a conjunction of one or more atoms, designated as the *body*. We assume that a rule is range restricted, i.e., every variable in the head of a rule also appears in the body of the rule. 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.

For a rule, *r*, an arithmetic predicate (see [BR]) of the form  $h(x_1, \dots, x_q)$ , where  $x_1, \dots, x_q$  are distinct variables, each of which appears in *r*, is called a *restricting* predicate. For example, for a rule that has variables  $x_1$  and  $x_7$ , the predicate  $(x_1 + x_7) \bmod 5 > 2$  is a restricting predicate. A *restricted version*, *r'*, of a rule *r*, is obtained by appending to the body of *r* a restricting predicate. A restricted version of a program, *P*, is a collection of rules that is obtained by replacing each rule of *P* by a restricted version of it. We assume that only restricted versions of programs have arithmetic predicates, programs do not.

The *input* *I* to a program *P* is a finite set of *R*-facts, where *R* is some extensional predicate symbol. Let *Q* be some intentional predicate in *P*. Given some input *I*, we define the *Q-query*, or the *output for Q*, and denote it  $O(P, Q, I)$ , it is the set of *Q*-facts that have a derivation tree in *P* given *I*. A *derivation tree* for a fact, *a*, is a finite tree with the nodes labeled by facts, *a* is the root, the leaves are facts in *I*, and for each internal node, *b*, with children  $b_1, \dots, b_k$ , there is an instantiation of a rule of *P* that has *b* as the head and  $b_1, \dots, b_k$  as the body, if *r* is a restricted version, then the instantiation must satisfy the restricting predicate. The *output* of *P* is the union of the outputs for all the intentional predicates. The set of input and output facts is called the *database* of the program *P*.

A predicate *Q* in a program *P* *directly derives* a predicate *R* if it occurs in the body of a rule whose head is a *R*-atom. *Q* is *recursive* if  $(Q, Q)$  is in the nonreflexive transitive closure of the

"directly derives" relation. Predicate *Q* *derives* predicate *R* if  $(Q, R)$  is in the reflexive transitive closure of the "directly derives" relation (particularly, every predicate derives itself). A program is recursive if it has a recursive predicate.

## 3 The Data-Reduction Paradigm

We first describe the paradigm assuming that the database resides in a memory common to all the processors. Then we consider the case in which there is no common memory.

Let *P* be a program with *m* rules, that we denote  $\{r_1, \dots, r_m\}$ . Let  $\{p_0, \dots, p_{k-1}\}$  be a set of  $k > 1$  processors. For each rule *r<sub>i</sub>*, we designate *k* restricting predicates,  $h_{ij}(x_1, \dots, x_{q_i})$ , for  $0 \leq j \leq k-1$ . The arguments  $x_1, \dots, x_{q_i}$  are the same for all the *k* predicates, and, by definition, all the arguments are variables of *r<sub>i</sub>*. We require that for each instantiation of the variables  $x_1, \dots, x_{q_i}$ , the predicate  $h_{ij}$  is true for exactly one *j*. Denote by  $r_{ij}$  the restricted version of the rule *r<sub>i</sub>* having the restricting predicate  $h_{ij}(x_1, \dots, x_{q_i})$  appended to its body. Denote by  $P_j$  the restricted version of *P* consisting of the set of rules  $\{r_{ij} \mid 1 \leq i \leq m\}$ . The set  $\{P_0, \dots, P_{k-1}\}$  is called a *data-reduction parallelization strategy*, or, for short, a *parallelization strategy* for *P*. For example, the set of restricted versions

$$\begin{aligned} S(x, y) - S(x, z), A(z, y), (z + y) \bmod k = j \\ S(x, y) - A(x, y), (y + 1) \bmod k = j \end{aligned}$$

for  $j = 0, \dots, k-1$ , constitutes a parallelization strategy for the program of Example 1.

The set of processors  $\{p_0, \dots, p_{k-1}\}$  cooperate in evaluating *P* in parallel as follows. The processors start with a global database, residing in common memory, consisting of the input. Processor *p<sub>i</sub>* performs the instantiations of the rules in the restricted version  $P_i$  (i.e. instantiations that satisfy  $P_i$ 's restricting predicates). If the head of the instantiated rule is not in the database, but each one of the facts in the body is there, then the fact in the head is added to the global database. The instantiations of *p<sub>i</sub>* can be performed by using any single-processor evaluation method on  $P_i$ , however, the method has to be adjusted, to account for additions to the database made by other processors, not just *p<sub>i</sub>*. The parallel algorithm ends when none of the processors can perform an instantiation of a rule in its restricted version, such that a new fact is added to the database. Actually, the number of processors can be smaller than the number of restricted versions, in which case more than one restricted version is assigned to a processor. This way the class of instantiation-partitions can be extended. For the sake of simplicity, the discussion is restricted to the one-restricted-version-per-processor case.

Now assume that there is no global database, but a local one for each processor. Assume further that the input is either replicated, or transmitted at the outset to all the processors. The message-passing, or shared-nothing variation of the data-reduction paradigm is as follows. Each processor, *p<sub>i</sub>*, starts with the local database consisting of the input to the program, and performs the instantiations of  $P_i$ , as before. Processor *p<sub>i</sub>* transmits to each other processor, *p<sub>j</sub>*, the set of tuples that *p<sub>i</sub>* computes. Actually, this set, denoted  $T_{ij}$ , may be less than the whole set of tuples computed by *p<sub>i</sub>*. This issue is addressed in section 5. The processor *p<sub>i</sub>* also receives from each other processor the set of tuples the latter computed. This way common memory is simulated. The communication among the processors is totally asynchronous during the

computation, and the only synchronization requirement is reflected in the termination condition, specified below. In other words, correctness of the paradigm is independent of the time (relative to the computation of each processor) at which messages containing tuples are sent and received by the processors. The algorithm performed by processor  $p_i$  is some variation of the procedure below. The procedure is executed iteratively, until the termination condition is satisfied.

#### DATA-REDUCTION

- 1 Add to the local database new tuples obtained by instantiations of rules of restricted version  $P_i$ ,
- 2 Transmit to some, or all, of the other processors the new tuples computed
- 3 Add to the local database new tuples obtained by instantiations of rules of restricted version  $P_i$ ,
- 4 Receive from some, or all, of the other processors new tuples and add them to the local database

The termination condition of the message-passing paradigm is the following: no processor can generate any new tuples (i.e. tuples that do not exist in the global database for the common memory architecture, or in the local database for the message passing architecture), by instantiating rules of its restricted version, also, there are no "in transit" tuples, i.e. tuples that have been sent but not received. We shall say more about the distributed termination protocol in the next section. Denote by  $S^i$  the relation for intentional predicate  $S$  existing at  $p_i$ , when the termination condition is satisfied. The output of the program for each intentional predicate,  $S$ , is

$$S \text{ is an intentional predicate } \cup_{i=1}^k S^i$$

#### 4 Specializing the Paradigm to an Algorithm

Let  $P$  be a program. In order to obtain a parallel algorithm on  $k$  processors from the data-reduction paradigm, the following four parameters have to be fixed: the restricting predicates that determine the strategy, the sets of tuples  $T_{ij}$  transmitted among the processors (discussed in section 5), the evaluation algorithm of each processor (including how it communicates with other processors), and the distributed termination protocol. In this section we discuss the last two parameters, starting with the evaluation algorithm.

In this paper we consider algorithms based on the semi-naive evaluation (see [Ban, Bay]) of each restricted version of a strategy. In [CW] we discuss an evaluation algorithm for a single-rule program,  $P$ , without constants. Communication among the processors is by message passing. Extended to an arbitrary datalog program, the algorithm PSNE executed by some processor,  $p_j$ , is given in Fig. 1. We use Ullman's notation for the semi-naive evaluation algorithm ([U])  $EVAL-INC(R_1, \dots, R_k, S_1, \dots, S_m, \Delta Q_1, \dots, \Delta Q_m)$  is a function that computes the tuples of  $S$ , that can be obtained by the instantiations of rules, given extensional relations  $R_1, \dots, R_k$ , intentional relations  $S_1, \dots, S_m$ , and differential relations

---

PSNE for processor  $p_j$

- 1 for  $i = 1$  to  $m$  do begin,
- 2  $\Delta S_i = EVAL(S_i, R_1, \dots, R_k, \emptyset, \dots, \emptyset)$ ,  
/\* evaluation of restricted version  $P_j$  \*/
- 3  $S_i = \Delta S_i$ ,
- 4 end,
- 5 send from the  $\Delta S_i$ 's the subset  $T_{jn}$ , to each processor  $p_n$ ,
- 6 repeat
- 7 for  $i = 1$  to  $m$  do,
- 8  $\Delta Q_i = \Delta S_i$ , /\* save old  $\Delta S$ 's \*/
- 9 for  $i = 1$  to  $m$  do begin,
- 10  $\Delta S_i =$   
 $EVAL-INC(R_1, \dots, R_k, S_1, \dots, S_m, \Delta Q_1, \dots, \Delta Q_m)$ ,  
/\* evaluation of restricted version  $P_j$  \*/
- 11  $\Delta S_i = \Delta S_i - S_i$ ,  
/\* remove "new" tuples that actually appeared before \*/
- 12 end,
- 13 send from the  $\Delta S_i$ 's the subset  $T_{jn}$ , to each processor  $p_n$ ,
- 14 for  $i = 1$  to  $m$  do,
- 15  $\Delta S_i \leftarrow \Delta S_i \cup \{S_i - \text{facts received the other}$   
*processors during the last iteration}\},*
- 16  $S_i = S_i \cup \Delta S_i$ ,
- 17 end,
- 18 until  $\Delta S_i = \emptyset$  for all  $i$ 's,
- 19 Wait until termination detection, or until some tuples are received from other processors,
- 20 if termination detection, then output the  $P_j$ 's and quit
- 21 if tuples received, then add them to the  $S_i$ 's, initialize the  $\Delta S_i$ 's to them, and go to r

Figure 1 The parallel semi-naive evaluation algorithm, PSNE, consists of multiple procedures as above, one for each processor

---

$\Delta Q_1, \dots, \Delta Q_m$ . Similarly for the function  $EVAL$ . Steps 5, 13, 15, and 19-21, constitute the modification to the well-known serial semi-naive evaluation algorithm. We shall denote by PSNE this parallel version of semi-naive evaluation. The algorithm PSNE can be seen as the following specialization of the data-reduction paradigm. In step 1 of paradigm, one iteration of semi-naive evaluation is performed for the restricted version  $P_i$ , in step 2, a subset of the newly computed tuples in step 1, i.e. of the differentials ( $\Delta$ 's) for all the intentional predicates, are transmitted to all the other processors (which subset, will be discussed in section 5), in step 3 no evaluation takes place, and in step 4, all the tuples received from other processors during the last iteration are added to the database, and to the differentials. If at this point the

differentials are empty, then processor  $p_i$  waits until termination is detected, or some tuples are received

Another variation of the paradigm is that in step 1, semi-naive evaluation is performed until a (temporary) fix-point is reached. Then data-reduction is continued as above. The algorithm of Fig 1 is modified as follows to reflect this variation. Step 5 is removed, step 13, referring to  $S_i$  rather than  $\Delta S_i$ , is moved to between steps 18 and 19, and step 15 is removed.

Still another variation of the paradigm is to execute step 4 of the paradigm, namely incorporation of tuples received from the other processors, only when a temporary fix-point is reached. The algorithm of Fig 1 is modified to reflect this variation, by removing Step 15.

The above algorithms do not assume any synchronous operation of the network, or that messages, or tuples, are received in the order in which they are sent.

Another parameter to fix in order to turn the data-reduction paradigm into a parallel algorithm is the distributed termination algorithm. However, for this purpose, one has only to select an algorithm from the many published in existing literature ([CM, F, M1, M2]). There, the distributed termination problem is defined as follows. Let  $p_0, \dots, p_{k-1}$  be a finite set of processors, communicating by messages. A processor is either idle or active. Only active processors may send messages, a process may change from active to idle at any time, and a process may change from idle to active only upon receipt of a message. The algorithms provided in the literature superimpose a termination detection algorithm on the computation. In our terminology, a processor is idle if it reaches a temporary fix-point, otherwise it is active. A processor reaches a temporary fix-point if by instantiating rules of its restricted version of the program, new tuples, i.e. tuples that do not exist in the local database, cannot be generated.

## 5 Transmission Sets

The message-passing version of the data-reduction paradigm transmits between processors more tuples than necessary. In simulating common memory, there is no point in transmitting to some processor tuples that will certainly be eliminated by its restricting predicates. To illustrate this, consider the following **Example 2** (continued from the introduction). Denote by  $h$  some hash function,  $h: z \rightarrow \{0, \dots, k-1\}$ . Suppose that there are  $k$  processors, and each  $p_i$  evaluates the program

$$P_i: \begin{aligned} S(y) - S(z), A(z, y), h(z) = i \\ S(y) - A(a, y), h(y) = i \end{aligned}$$

We shall make three observations about this example. First, assume that the relation  $A$  has an index on the second attribute,  $S$  does not have an index, and the optimal way of joining  $S$  and  $A$  is by a nested loop, where  $S$  is the outer relation, and  $A$  is the inner one. Then partitioning the work by the above strategy will probably result in an optimal speedup, i.e.  $k$ . Second, in a wide-area distributed environment, assume that the relation  $A(z, y)$  is horizontally partitioned on the first column, for example processor  $i$  stores the tuples for which  $h(z) = i$ . Then, the requirement that each processor has the whole input at the outset can be relaxed. Third, which demonstrates the topic of this section, in order to ensure that each output tuple is computed by at least one

processor,  $p_i$  has to transmit to  $p_j$  only the tuples  $S(y)$  that it computes, and for which  $h(y) = j$ .  $\square$

Formally, given an input  $I$  to a program  $P$ , we define the set of tuples  $T_{ij}$ , that processor  $p_i$  sends to  $p_j$ . Let  $S$  be an intentional predicate of the program  $P$ , and  $H$  a parallelization strategy of it. The set of  $S$ -facts transmitted from  $p_i$  to  $p_j$ , denoted  $ST_{ij}$ , consists of the intersection of two other sets, denoted  $SR_i$  and  $SC_i$ . First we define the set  $SR_i$ . An  $S$ -fact,  $f$ , is in  $SR_i$  if and only if (*Condition K1*) there is some rule of the program  $P$ , say  $r_g$ , such that  $f$  is not in  $r_g$ , but there is an instantiation of it that satisfies the predicate  $h_{g_j}$ , and  $f$  appears in the body of the instantiated rule. In other words, a tuple  $f$  is in  $SR_i$ , if there is an instantiation for which  $p_j$  is in charge, that uses  $f$ . Determining whether a given fact is in  $SR_i$  can be done in constant time, under the following assumptions

- (1) The size of the program is constant (this assumption is also made in other works, e.g. [UV])
- (2) For any restricting predicate  $h_j(x_1, \dots, x_n)$ , it can be determined in constant time, for any instantiation of any subset of the  $x_i$ 's, whether or not the rest of them can be instantiated by constants, such that the predicate is *true*.

Next we define the set  $SC_i$ . In contrast to the set  $SR_i$ , the set  $SC_i$  does depend on the input. Intuitively, it is the set of facts computed by processor  $p_i$ . Formally, a *productive* instantiation of a rule at processor  $p_i$  is an instantiation for which, when performed by  $p_i$ , the head is not in the database at  $p_i$ , but all the facts in the body are there. A fact is *computed* by  $p_i$  if it is in the head of a productive instantiation. Note that the same fact may be computed by more than one processor. Furthermore, it may be computed, and later received from another processor. Let  $SC_i$  be the set of  $S$ -facts computed by  $p_i$ . Then  $ST_{ij}$ , the *S-transmission set from  $i$  to  $j$* , is  $SR_i \cap SC_i$ .

We define  $T_{ij} = \bigcup_{S \text{ is an intentional predicate}} ST_{ij}$ .

The set  $T_{ij}$  is called the *transmission set* from  $p_i$  to  $p_j$ .

Observe that the definition of  $T_{ij}$  requires that each processor,  $p_i$ , knows the whole strategy, not only its own restricted version. Furthermore, note that the  $T_{ij}$ 's are not necessarily disjoint. For example, if in the body of some rule of  $P$  appears the atom  $S(y)$ , and if the variable  $y$  is not an argument of a restricting predicate, then any processor that computes a fact,  $S(a)$ , must transmit it to all the other processors. Moreover, it is possible that  $S(a)$  is computed by more than one processor.

(Optimization O1) An algorithm based on the data-reduction paradigm may perform the following optimization, to send less than the whole set  $T_{ij}$ . It may eliminate a fact,  $f$ , from  $T_{ij}$ , if it was received at  $p_i$ , before the latter transmitted  $f$  to  $p_j$ . In other words, it is possible that  $p_i$  has computed  $f$ , included it in  $T_{ij}$ , but has not performed the actual transmission (a possible reason is that it waited to fill up a buffer). If at this point  $f$  is being received at  $p_i$ , then  $f$  can be eliminated from  $T_{ij}$ . The reason this optimization does not violate correctness is that the processor that sent  $f$  to  $p_j$  must have also sent it to  $p_i$ .

## 6 Unique Source and Destination Properties

Let  $P$  be a program, and  $\{p_0, \dots, p_{k-1}\}$  a set of processors, for some parallelization strategies for  $P$ , each possible tuple of an

intentional relation,  $S$ , is transmitted to a unique-processor. This is a desirable situation, since it reduces communication among the processors. Formally, the parallelization strategy  $H$  has the *unique destination* property with respect to the intentional predicate  $S$ , if each  $S$ -fact belongs to a unique  $SR_j$ . This means that each  $S$ -fact,  $f$ , is transmitted to only one processor, by any processor that computes  $f$ . For example, the strategy

$$\begin{aligned} S(x,y) - UP_1(x,z), S(z,w), DOWN_1(w,y), (z+w) \bmod k = j \\ S(x,y) - UP_2(x,z), S(z,w), DOWN_2(w,y), (z+w) \bmod k = j \\ S(x,y) - FLAT(x,y), x \bmod k = j \end{aligned}$$

has the unique destination property with respect to  $S$ . For instance, assuming that there are three processors,  $\{p_0, p_1, p_2\}$ , the tuple  $S(5,3)$  is only transmitted to  $p_2$ . Now consider the strategy identical with the one above, except that the restricting predicates of the second rule are  $x \bmod k = j$ . This strategy does not have the unique destination property.

When does a parallelization strategy have the unique destination property? This question is important because it should be taken into consideration in selecting one, from several candidate parallelization strategies by which to evaluate  $P$ .

**Theorem 1** Let  $P$  be a program, and let  $H = \{P_0, \dots, P_{k-1}\}$  be some parallelization strategy of  $P$ . The strategy  $H$  has the unique destination property with respect to intentional predicate  $S$ , if there is a set of argument positions  $t_1, \dots, t_v$  of the predicate  $S$ , such that

- (1) if  $S_0$  is an  $S$ -atom in the body of some rule,  $r_i$ , of  $P$ , then the variables denoted  $x_1, \dots, x_q$ , i.e., the arguments of the restricting predicates  $h_{ij}$ , appear in positions  $t_1, \dots, t_v$  of  $S_0$ , respectively (and consequently  $v = q$ .)
- (2) if  $r_i$  and  $r_j$  are two rules of  $P$  that have an  $S$ -atom in the body, then for every sequence of constants,  $a_1, \dots, a_v$ , and for every  $m$ ,  $h_{im}(a_1, \dots, a_v)$  is true if and only if  $h_{jm}(a_1, \dots, a_v)$  is true.  $\square$

Another important property of a strategy is the unique-source property. It ensures that any  $S$ -fact,  $f$ , is transmitted from (rather than to) a unique processor. Again, this property reduces communication. Formally the parallelization strategy  $H$  has the *unique-source* property with respect to the intentional predicate  $S$ , if each  $S$ -fact,  $f$ , can belong to a unique  $CT_j$ . In other words, if  $f$  is in the output of the program  $P$ , then it is computed by the processor  $p_j$ , and only by this processor. For example, the strategy

$$\begin{aligned} S(x,y) - UP_1(x,z), S(z,w), DOWN_1(w,y), x \bmod k = j \\ S(x,y) - UP_2(x,z), S(x,w), DOWN_2(w,y), x \bmod k = j \\ S(x,y) - FLAT(x,y), x \bmod k = j \end{aligned}$$

has the unique source property with respect to  $S$ . Consider the strategy identical with the one above, except that the restricting predicates of the second rule, are  $z \bmod k = j$ . This strategy does not have the unique source property. The next theorem, giving a sufficient condition for a strategy to have the unique-source property, is identical to Theorem 1, except that it refers to  $S$ -atoms in the head, rather than body, of rules.

**Theorem 2:** Let  $P$  be a program, and let  $H = \{P_0, \dots, P_{k-1}\}$  be some parallelization strategy of  $P$ . The strategy  $H$  has the unique source property with respect to intentional predicate  $S$ , if there is a set of argument positions  $t_1, \dots, t_v$  of the predicate  $S$ , such that

- (1) if  $S_0$  is an  $S$ -atom in the head of some rule,  $r_i$ , of  $P$ , then the variables denoted  $x_1, \dots, x_q$ , i.e., the arguments of the restricting predicates  $h_{ij}$ , appear in positions  $t_1, \dots, t_v$  of  $S_0$ , respectively (and consequently  $v = q$ .)
- (2) if  $r_i$  and  $r_j$  are two rules of  $P$  that have an  $S$ -atom in the head, then for every sequence of constants,  $a_1, \dots, a_v$ , and for every  $m$ ,  $h_{im}(a_1, \dots, a_v)$  is true if and only if  $h_{jm}(a_1, \dots, a_v)$  is true.  $\square$

Assume that a strategy has both, the unique source and destination properties with respect to some intentional predicate,  $S$ , and furthermore the source and destination coincide, i.e. are the same processor, for each  $S$ -fact. Then each  $S$ -fact is produced during the evaluation by a unique processor, and no  $S$ -fact has to be transmitted among the processors.

## 7. Decomposable Programs

For some programs there exists a strategy that has a coinciding source and destination property for every intentional predicate of the program. Such programs are called decomposable. The processors cooperating in the evaluation of a decomposable program do not have to transmit any tuples, and the output produced by each processor is disjoint from the output of each other processor. The advantage of communication-freedom is obvious, and output-disjointness implies that two processors do not duplicate the effort of producing the same fact, this is appealing since it means that work-partitioning is abstracted, independently of implementation details, such as the inner and outer relations of a nested loop join. For example, the following parallelization strategy for computing the transitive closure (a decomposable program) has the communication-freedom and output-disjointness advantages.

$$\begin{aligned} S(x,y) - S(x,z), A(z,y), x \bmod k = j \\ S(x,y) - A(x,y), x \bmod k = j \end{aligned}$$

The data-reduction paradigm is a syntactic concept. However, as we shall show, decomposability is a semantic property, and in this section we study it in this way. Specifically, we ask the following question: Can we algorithmically identify the programs for which there is a parallel evaluation method (whether or not a specialization of the data-reduction paradigm) that satisfies the above conditions, namely, work disjointness, and communication freedom. In [CW] we have taken this semantic approach, defined the decomposability property and provided necessary and sufficient conditions for decomposability of a single-rule program. These conditions can be checked algorithmically. In this section we first extend the decomposability definition to arbitrary datalog programs, this is necessary since in [CW] the definition was restricted to single-rule-programs. Then we ask whether there exists an algorithm that determines whether or not an arbitrary program is decomposable, and answer negatively.

We start with some preliminaries, that pave the way to the decomposability definition. A program is *Q-minimal* if every predicate in the program derives  $Q$ . We shall assume without loss

of generality that when evaluating a  $Q$ -query, the program is  $Q$ -minimal, otherwise rules can be omitted from the program, for answering the  $Q$ -query. Let  $P$  be a  $Q$ -minimal program, for some intentional predicate  $Q$ . The *output domain* of  $P$ , denoted  $O$ , is the set of all  $R$ -facts, for all intentional predicates,  $R$ . In other words, the output domain is the infinite set  $\{R(\vec{a}) \mid R \text{ is an intentional predicate, and } \vec{a} \text{ is a sequence of constants}\}$ . A set of two or more sets,  $M_1, \dots, M_k, \dots$ , is a  $Q$ -partition of the output domain of  $P$ , if the following requirements are satisfied

- 1 the  $M_i$ 's are pairwise disjoint, and
- 2 each  $M_i$  contains at least one  $Q$ -atom (otherwise the member is useless for the evaluation of a  $Q$ -query), and
- 3 each  $Q$ -fact in the output domain belongs to some  $M_i$ .

Let  $D$  be a  $Q$ -partition of the output domain for the program  $P$ , and let  $M_i$  be a member of  $D$ . A  $Q$ -fact,  $g$ , in  $M_i$  is *proper*, if for every input  $I$  such that  $g$  is in the output  $O(P, Q, I)$ , the fact  $g$  has a derivation tree in which all the intentional-facts are in  $M_i$ . In other words, assume that each processor assumes responsibility for producing the  $Q$ -facts belonging to one or more members of  $D$ . Then, for deriving a proper  $Q$ -fact, a processor does not need to receive facts derived by other processors, regardless of the input. The program  $P$  is  $Q$ -decomposable if it has a  $Q$ -partition for which every  $Q$ -fact in the output domain is proper. Then, the set  $D$  is called an *eligible  $Q$ -partition* of  $P$ .

For example, consider the program  $P1$  below

$$\begin{aligned} Q(x, y, z) - Q(x, y, w), A(w, z) \\ Q(x, y, z) - R(y, x, z) \\ R(x, y, z) - R(x, y, w), B(w, z) \\ R(x, y, z) - C(x, y, z) \end{aligned}$$

The program is  $Q$ -decomposable. One example of an eligible  $Q$ -partition is the following.  $M_1$  consists of the  $Q$ - and  $R$ -facts in which the sum of the constants in the first two positions is odd, and  $M_2$  consists of the ones in which the sum is even. Actually, the program  $P1$  has an infinite  $Q$ -partition.  $M_1$  consists of the facts in which the sum is 1,  $M_2$  consists of the facts in which the sum is 2, etc. When the program has a single intentional predicate, it is easy to see that the decomposability definition above reduces to the definition in [CW].

Decomposable programs are also interesting for sequential processing. Once a fixpoint is reached within a member of the partition, all the facts of the member can be removed from the intentional relations, reducing their sizes for further processing. For example, consider the program  $P1$  above. If at some iteration of semi-naive evaluation, the differential does not contain any intentional facts in which the sum of the first two positions is 3, (but prior iterations it did), then all such facts can be removed from the intentional relations, reducing their size for further iterations.

We prove that for an arbitrary datalog program,  $P$ , and a predicate  $Q$  of  $P$ , the problem of determining  $Q$ -decomposability of  $P$  is recursively unsolvable. First, let us point out that the result cannot be obtained trivially from [GMSV, Theorem 8]. That result is a Rice style theorem, that implies that many interesting properties of datalog programs are undecidable. Specifically, theorem 8 in [GMSV] states that any semantic property that contains boundedness, and is strongly nontrivial is undecidable. A property  $\pi$

contains boundedness if every bounded<sup>1</sup> program has the property  $\pi$ . However, decomposability does not contain boundedness. In fact, there are nonrecursive programs that are not decomposable. For proof we will show that the following nonrecursive program,  $P2$ , is not decomposable.

$$\begin{aligned} Q(x, y) - E(x, w), R(w, v), F(v, y) \\ R(x, y) - G(x, y) \end{aligned}$$

Assume, by way of contradiction, that  $P2$  is  $Q$ -decomposable, and consider two members,  $M_i$  and  $M_j$ , of an eligible  $Q$ -partition. Observe that, since every member of the  $Q$ -partition contains a  $Q$ -fact, every member must also contain an  $R$ -fact. Let  $Q(a, b)$  be in  $M_i$ , and  $R(c, d)$  be in  $M_j$ . Then  $Q(a, b)$  is not proper, since for the input  $\{E(a, c), G(c, d), F(d, b)\}$  the fact  $Q(a, b)$  has a single derivation tree, and  $R(c, d)$ , a fact in this tree, is not in  $M_i$ .

In [WS] we have shown that every nonrecursive single-rule program (a program with one intentional predicate and two nonrecursive rules), is decomposable. Actually, a program with an arbitrary number of rules is decomposable, provided that it has a single intentional predicate. However, the program  $P2$  above has two

**Theorem 3:** The problem of determining whether a given program is  $Q$ -decomposable, is recursively unsolvable.

*Proof idea:* The theorem is proven by a reduction from the problem of determining equivalence of two datalog programs, shown undecidable in [S]. Given two programs,  $P_1$  and  $P_2$ , we construct a third,  $P$ , that has a new predicate,  $Q$ , such that  $P$  is  $Q$ -decomposable, if and only if  $P_1$  and  $P_2$  are equivalent. []

The negative result in this section is "cushioned" by a sufficient condition for decomposability, discussed in [WS]. There we defined a syntactic condition, called *pivoting*, that is sufficient for a program to be decomposable.

## 8 Load Balancing

In the exposition so far, we assumed a fixed set of restricting predicates, determining a priori the restricted version executed by each processor. Clearly, even the best functions will fail to evenly balance the load for some inputs. Then load balancing has to occur. We shall not discuss the problem of determining when to balance the load, but only how to do so. The way we propose is for some processor,  $p_i$ , to change the parallelization strategy used, in order to balance the load. Presumably,  $p_i$  is a processor that is idle for more than some prespecified amount of time. Or,  $p_i$  knows that there are idle processors, although  $p_i$  itself is not idle. How should the strategy be changed? We suggest the following protocol.

There is a processor, e.g.  $p_0$ , designated as the "leader", at the outset. When some processor decides to change the parallelization strategy, it selects the set of restricting predicates of the new strategy (possibly the next set in a list of candidate strategies), and sends this set to the leader, requesting a change. The purpose of this step is for the leader to be able to select a single

<sup>1</sup> A program is bounded if, for each query predicate, it produces the same output as a nonrecursive program, given the same input.

"successful" processor if multiple processors are simultaneously attempting a strategy-change, each with a different set of restricting predicates. Before changing a strategy,  $X$ , the leader verifies two things. First, that all the processors have received  $X$ , and second, that at least one processor has generated new tuples using  $X$ . The purpose of the first verification is to ensure that when the algorithm ends, all the processors use the same strategy, this in turn ensures completeness. The purpose of the second verification is to prevent an infinite loop of strategy-changes, without making any progress in the computation of the output. Only after the two verifications complete positively, the leader sends the new strategy to each processor.

When a processor,  $p_j$ , receives a new restricted version from the leader, it transmits from its local database, to each other processor,  $p_m$ , the subset that satisfies condition K1 (see definition in section 5), according to the new strategy. Then,  $p_j$  simply proceeds with its computation using the new restricted version. The PSNE algorithm of Fig. 1 is adapted to change the strategy dynamically, by adding the following step between steps 6 and 7.

6.1) if a new strategy is requested, then send to each processor,  $p_m$ , from each one of the intentional relations  $S$ , the subset that is also in  $SR_k$  (defined according to the new strategy), then change the restricting predicates according to the new strategy.

In the full paper we demonstrate the strategy-change procedure, and prove that it is correct, namely that no output is lost. Assume now that the program being evaluated in parallel is *linear*, namely a program with at most one intentional predicate in the body of each rule. Then we can apply the following optimization of the load balancing scheme. At step 6.1 of the PSNE algorithm,  $p_j$  should transmit to each other processor,  $p_n$ , only a subset of the facts it transmits in the general case. For each intentional predicate  $S$ , it is the subset of the last differential,  $\Delta S$ , (instead of all the  $S$ -facts in the current database) that satisfies condition K1 according to the new strategy. Note that this reduction in the size of  $T_n$  has two positive effects. First, it reduces the number of tuples transmitted among processors. Second, it reduces the amount of work performed by the receiving processor,  $p_n$ , since the size of both, the differential  $\Delta S$ , and the relation  $S$ , shrinks. In the full paper we prove the correctness of the optimization for linear programs, and we demonstrate that it is incorrect if the program is not linear.

## 9 Extension to Datalog with Negation

In this section, we discuss the application of parallel algorithms based on data-reduction, to datalog programs for which the rules are defined as before, except that some of the atoms in the body of a rule may be negated. We shall assume safe negation, namely that each variable in a negated atom also appears in a non-negated atom in the body of the same rule. Furthermore, we shall assume that the programs are stratified (see [ABW]). This means that there is no path in the dependency graph<sup>1</sup> from  $R$  to  $Q$ , if there is a rule whose head is an  $R$ -atom, and a negated  $Q$ -atom appears in its body (namely  $\sim Q$  defines  $R$ ). Such a program has a

<sup>1</sup> A graph that has the predicate symbols as the nodes, and an arc  $S \rightarrow T$  for each pair  $S, T$  such that there is a rule whose head is a  $T$ -atom, and an  $S$ -atom appears in its body.

stratification, i.e. a nonnegative numbering of the predicate symbols, such that if  $S$  is defined by  $\sim T$ , then  $T$  has a lower number than  $S$ , and if  $S$  is defined by  $T$ , then  $T$  has a lower or equal number than  $S$ . The output of such a program is defined as the set of tuples obtained by evaluating the strata one by one, in increasing order, using the complement of a relation  $S$  as the set of facts in the database, for the atom  $\sim S$  appearing in the body of some rule. A data-reduction algorithm of the type discussed in the previous sections, can be used for the evaluation of each stratum.

Therefore, a parallelization strategy for a program with  $t$  strata consists of  $t$  parallelization strategies each one evaluated by  $k$  processors. Suppose that intentional predicate  $S$  is at stratum  $b$ . At the completion of the evaluation of stratum  $b$ , each processor,  $p_i$ , transmits to all the other processors, the  $S$ -facts that are in  $p_i$ 's database, assuming that the atom  $S$  appears (possibly negated) in higher strata. Actually,  $p_i$  does not have to wait until the completion of stratum evaluation, but can transmit the  $S$ -facts as they are evaluated by  $p_i$ . Furthermore, only a tuple,  $f$ , that satisfies the following condition should be transmitted to  $p_j$ .

*Condition (KIN)* There is some rule,  $r_g$ , whose head is at stratum  $b$  or higher, such that  $f$  is not in  $r_g$ , but there is an instantiation that satisfies the predicate  $h_{r_g}$ , and  $f$  appears, possibly negated, in the body of the instantiated rule.

Therefore, the transmission sets are defined in terms of the currently evaluated stratum, as well as higher ones.

Now suppose that intentional predicate  $S$  appears negated at stratum  $s$ , and the stratum of  $S$  is  $u$ ,  $u < s$ . Then a processor,  $p_i$ , cannot start the evaluation of stratum  $s$  before all the processors have completed the evaluation of stratum  $u$ , otherwise, facts it computes may be "invalidated" by  $S$ -facts received later. In other words, there are inputs, and relative computation speeds (and communication delays), for which invalidation of tuples may occur. Therefore, in general, the processors have to be synchronized at each stratum. Synchronization means that each processor has to wait until all the processors have completed their evaluation, and there are no tuples "in transit", before proceeding to the next stratum.

However, this is not always necessary. For example, consider the following strategy for parallelization of the program that computes in  $S$  the transitive closure of  $A$ , and in  $T$  the tuples of the transitive closure of  $B$ , that are not in  $S$ .

$$\begin{aligned} T(x,y) - T(x,z), B(z,y), \sim S(x,y), x \bmod k = j \\ T(x,y) - B(x,y), \sim S(x,y), x \bmod k = j \\ S(x,y) - S(x,z), A(z,y), x \bmod k = j \\ S(x,y) - A(x,y), x \bmod k = j \end{aligned}$$

for  $j = 0, \dots, k-1$ . In this case there is no tuple that has to be transmitted among the processors, and in particular the processors do not have to be synchronized at the beginning of each stratum evaluation. A way of looking at this, is that the only  $S$ -facts that can "invalidate"  $T$ -facts computed by some processor,  $p_i$ , are  $S$ -facts that are also computed by  $p_i$ .

In general, it is possible that for a parallelization strategy, the processors have to be synchronized at the beginning of the evaluation of some, but not all, of the strata of a program. Such strata are called *synchronous*, in contrast to others, that are *asynchronous*. (Actually, it is possible that for a parallelization

strategy, a stratum is asynchronous for some processors, but not for others. However, for the sake of simplicity, we omit this subtlety from the present discussion.) For example, if to the strategy above we add the rules

$$U(x,y) - C(x,y), \sim T(z,y), x \bmod k = j$$

for  $j=0, \dots, k-1$ , then the third stratum is synchronous

A sufficient condition for a stratum to be asynchronous is the following. Let  $P$  be a program, and let  $H$  be a parallelization strategy for the evaluation of  $P$ . Let  $s$  be a stratum, and denote by  $S_1, \dots, S_m$  the intentional predicates that appear negated at stratum  $s$ . Denote by  $G$  the set that consists of  $S_1, \dots, S_m$ , and the intentional predicates that derive any of the  $S_i$ 's. Denote by  $t$  be the highest stratum below  $s$ , that is synchronous, or, if there is none, then  $t=0$ . Disregard any rules of the strategy that define predicates at a stratum higher than  $s$ , and examine the following. If each intentional predicate that is in  $G$ , and is at a stratum between  $t$  and  $s-1$ , has a coinciding unique source and destination property, then  $s$  is asynchronous.

## 10 Conclusion

In this paper we introduced the data-reduction paradigm for evaluating datalog programs in parallel. It consists of the evaluation of a parallelization strategy, i.e. a partition of the rule-instantiations, such that each processor performs the instantiations in a partition member, and adds the newly generated tuples to a common database. The common database may be simulated by message passing.

We proposed a protocol for dynamic changing of strategies derived from the paradigm. This is required for load balancing. For semi-naive evaluation of a linear program, load balancing can be performed more efficiently, since it is necessary to redistribute only the differentials, rather than the whole output produced so far.

We also discussed the extension of the results to datalog programs with stratified negation. The asynchronous mode of parallel computation is not guaranteed when the paradigm is extended to this type of programs. Some strata may be synchronous, i.e. require synchronization of the processors, before the evaluation begins. Others may be asynchronous. It turns out that the synchrony of a stratum is related to two other important properties of parallelization strategies, namely unique source and destination. They enable a lower communication overhead for programs with and without negation, and we provided a sufficient condition for each property. Programs for which there is a parallelization strategy that has both properties are called decomposable, and we have shown that it is undecidable to determine whether or not a program is decomposable.

## 11 Future work

We intend to continue the exploration of the data-reduction paradigm, and will concentrate in the immediate future on distributed environments. The main deviation from our model is that in such an environment it may not be practical to assume that all processors have access to the whole input. However, as we have pointed out in example 2 at the beginning of section 5, this only means that an additional consideration, i.e. the fact that the whole database may not be accessible locally, contributes to the selection of

the restricting predicates

Specifically, we intend to apply the data-reduction paradigm to rule-processing in databases for network management. Net-mate, a project currently under development at Columbia University (see [SDSWY]), aims to develop a software environment for management of very large (hundreds of thousands of interconnected computers) communication networks. A fault in such a network is a failure or an overload condition, and an important goal in network management is to automatically detect and recover from this condition. Rule based programming can be employed to attain this goal, but two factors combine to complicate this approach. First is that detection of the fault may require the analysis of very large amounts of statistical and configuration data, and second is that this data is usually distributed. One solution is to transmit the data, and analyze it in a central location. However, this would place an unacceptable communication load on the network, and an unacceptable computation load on the single processor. Another solution is to run a rule based program at multiple processors in the network, with each analyzing the data produced locally. However, in this approach, the global view that is often required for proper fault detection, is lost. The right solution seems to require one rule program that has access to the data in the whole network. For the rule programmer, this will hide the complexity introduced by distribution, and enable conceptualization of the fault detection problem as being centralized. However, for performance reasons, the program should be processed at many processors in the network, while minimizing communication overhead. Data-reduction satisfies these requirements perfectly. It speeds up the evaluation of a rule-based program by using multiple processors (the nodes in the network), each working on a different subset of the database (the data stored locally at the node), while minimizing the required communication among the processors.

Data-reduction should also prove helpful in the distributed processing of triggers. For example, assume that the network-configuration database is partitioned among many processors in the network, and consider the following trigger: "if the delay on 20% of the communication lines exceeds 5 seconds, then execute a certain alarm". Continuously collecting the tuples representing the lines that satisfy the condition, would place an unacceptable communication and computation load. Processing of the trigger under the data-reduction paradigm will hopefully consist of local trigger-evaluation (counting the number of culprits stored in the processor), with minimal communication among the processors (transmission of the count rather than the tuples).

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

## 12 References

- [ABW] K R Apt, H Blair, A Walker "Towards a Theory of Declarative Knowledge," unpublished memorandum, IBM Yorktown Heights, NY
- [AJ] R Agrawal and H V Jagadish, "Multiprocessor Transitive Closure algorithms", Proc International Symposium on Databases in Distributed and Parallel Systems, Austin TX, Dec 1989

- [AP] F Afrati and C H Papadimitriou "The Parallel Complexity of Simple Chain Queries", Proc 6th ACM Symp on PODS, pp 210-213, 1987
- [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
- [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
- [CK] S S Cosmadakis and P C Kanellakis "Parallel Evaluation of Recursive Rule Queries", Proc 5th ACM Symp on PODS, pp 280-293, 1986
- [CM] K M Chandy and J Misra "On Proofs of Distributed Algorithms with Application to the problem of Termination Detection", Manuscript, Dept of CS, University of Texas
- [CW] S Cohen and O Wolfson, "Why a Single Parallelization Strategy is not Enough in Knowledge Bases," Proc 8th ACM Symp on PODS, pp 200-216, 1989 Also, invited and submitted for publication in a special issue of the Journal of Computer and Systems Sciences
- [D] G Dong, "On Distributed Processibility of Logic Programs by Decomposing Databases", Proc ACM-SIGMOD conf, 1989
- [DL] D DeGroot and G Lindstrom eds "Logic Programming - Functions Relations and Equations", Prentice Hall, 1986
- [F] N Francez, "Distributed Termination", ACM Transactions on Programming Languages and Systems, 2(1), pp 42-55, 1980
- [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
- [GMSV] H Gaifman, H Mairson, Y Sagiv, M Y Vardi, "Undecidable Optimization Problems for Database Logic Programs," IBM Technical Report RJ 5583 (56702) 4/3/87
- [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
- [K] P C Kanellakis "Logic Programming and Parallel Complexity", Springer-Verlag Lecture Notes in CS Series, no 243, pp 1-30, 1986
- [MW] D Maier and D S Warren "Computing with Logic Introduction to Logic Programming", Benjamin-Cummings Publishing Co, 1987
- [M1] F Mattern, "Algorithms for Distributed Termination Detection", Distributed Computing, pp 161-175, 1987
- [M2] F Mattern, "Global Quiescence Detection Based on Credit Distribution and Recovery", to appear, Information Processing Letters
- [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
- [S] O Shmueli "Decidability and Expressiveness Aspects of Logic Queries," Proc 6th ACM Symp on PODS, pp 237-249, 1987
- [Sh] E Y Shapiro "Concurrent Prolog, Collected Papers", MIT Press, 1987
- [SDSWY] S Sengupta, A Dupuy, J Schwartz, O Wolfson, Y Yemini, "The Net-mate Model for Network Management", in IEEE Network Operations and Management Symposium, Feb 1990
- [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
- [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
- [WS] O Wolfson and A Silberschatz, "Distributed Processing of Logic Programs," Proc of the ACM-SIGMOD Conf, pp 329-336, 1988
- [W] O Wolfson, "Sharing the Load of Logic Program Evaluation", Proc of the Intl Symp on Databases in Parallel and Distributed Systems, Austin, TX, Dec 1988