THE DATA REDUCTION PARADIGM FOR PARALLELIZATION IN KNOWLEDGE BASES

by

Ouri Wolfson

Technical Report #571

June 1989
THE DATA REDUCTION PARADIGM FOR PARALLELIZATION IN KNOWLEDGE BASES

Preliminary Version

Ouri Wolfson
Computer Science Dept.
The Technion - IIT
Haifa, Israel

ABSTRACT

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 the instantiations of the rules of the program among the processors. 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 reduction in communication overhead, load balancing, and application to programs with negation.
1. Introduction

A knowledge base is a relational database augmented with a declarative 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 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 incur an overhead, but can be applied to every single-rule program without constants. All the strategies discussed in our previous works are based on data-reduction - each processor evaluates the original program, but with less data.

In this paper we introduce the data-reduction paradigm, and show that the strategies discussed in the previous works are specializations of this paradigm. Every datalog evaluation is a process of rule-instantiations, and the paradigm partitions the instantiations among the processors. The purpose is to do so, such that each processor evaluates the original program with less data. The works on the NC-complexity of programs (e.g. [AP, CK, 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. We, on the other hand, assume a constant number of processors, and divide the instantiations to achieve work-load partitioning, and low overhead.

The data-reduction paradigm, in contrast to the strategies discussed in our previous works, is applicable to datalog programs with multiple rules, constants, and negation. Efficiency of the paradigm is based on the observation made by the database community, 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 [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 $P_1$:

$$S(x,y) := S(x,z), A(z,y)$$
$$S(x,y) := A(x,y)$$

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

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

until no new tuples are added to $S$. \hfill \Box$$

Technion - Computer Science Department - Technical Report CS0571 - 1989
Example 2: The program $P_2$:

\[
S(y):= S(z), A(z, y) \\
S(y):= A(a, y)
\]

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:

\[
\begin{align*}
\Delta S(y) & := \pi_y (\Delta S(y) \ join A(z, y)) - S(y) \\
S(y) & := S(y) \cup \Delta S(y)
\end{align*}
\]

iteratively, until $\Delta S = \emptyset$. □

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. [BBDW]). However, we postulate that the work partitioning can better be performed by appending some predicate $h \cdot j$ 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 $\{p_0, \ldots, p_{k-1}\}$. 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. $z \ mod \ 2$) 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.

We first introduce the paradigm for datalog programs without negation, and we discuss how it is specialized
to a particular parallel algorithm. 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 course can be
performed more efficiently for a linear program. Then we discuss some desirable properties of strategies. These are
single-source and single-destination, and they enable lower communication overhead. Actually, strong-
decomposability discussed [CW] is a combination of these two properties. For a given strategy, we present a
sufficient condition for each one of the properties. Finally, we discuss how to extend the paradigm to programs with
stratified negation. The data-reduction paradigm for datalog without negation, does not require synchronization
among the processors. However, the same paradigm requires synchronization when applied to programs with nega-
tion. It points out that there is a relationship, that we feel is fundamental in parallel computation, between monoto-
nicity 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
address the problem of balancing the load by strategy change, and in section 7 we discuss the single-source and
single-destination properties. In section 8 we discuss the application of the paradigm to datalog programs with
stratified negation. In section 9 we conclude.
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. An R-atom is an atom having R as the predicate symbol. A rule consists of an atom Q, designated as the head, and a conjunction of one or more atoms, denoted Q₁,...,Qₖ, designated as the body. Such a rule is denoted Q :- Q₁,...,Qₖ, which should be read "Q if Q₁ and Q₂, and ..., and Qₖ".

If an atom has a constant in each argument position, then it is a fact. A datalog program (see [MW]), or a program for short, is a finite set of rules whose predicate symbols are divided into two disjoint subsets: the extensional predicates, and the intentional predicates. The extensional predicates are distinguished by the fact that they do not appear in any head of a rule. The rules of a program are range restricted, i.e., every variable in the head of a rule also appears in the body of the rule.

For a rule, r, an arithmetic predicate (see [BR]) of the form h(x₁,...,x₇), where x₁,...,x₇ are distinct variables, each of which appears in r, is called a restricting predicate. For example, for a rule that has variables x₁ and x₇, the predicate \((x₁ + x₇) \mod 5 > 2\) 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.

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, where S is an intentional-predicate atom. 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 b₁,...,bₖ, there is an instantiation of a rule r which has b as the head and b₁,...,bₖ as the body; if r is a restricted version, then the instantiation must satisfy the restricting predicate. The set of input and output facts is called the database of the program P.

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 \( \{r_1, \ldots, r_m\} \). Let \( \{p_0, \ldots, p_{k-1}\} \) be a set of \( k \) processors. For each rule \( r_i \), we designate \( k \) restricting predicates, \( h_{ij}(x_1, \ldots, x_q) \), for \( 0 \leq j \leq k-1 \). The arguments \( x_1, \ldots, x_q \) are the same for all the \( k \) predicates, and, by definition, all the arguments are variables of \( r_i \). We require that for each instantiation of the variables \( x_1, \ldots, x_q \), the predicate \( h_{ij} \) is true for exactly one \( j \). Denote by \( r_{ij} \) the restricted version of the rule \( r_i \) having the restricting predicate \( h_{ij}(x_1, \ldots, x_q) \) appended to its body. Denote by \( P_{ij} \) the restricted version of \( P \) consisting of the set of rules \( \{r_{ij} \mid 1 \leq i \leq m\} \). The set \( \{P_{0j}, \ldots, P_{k-1j}\} \) is called a data-reduction parallelisation strategy, or, for short, a parallelisation strategy for \( P \). For example, the set of restricted versions:

\[
S(x,y) :- S(x,z), A(z,y), (z+y) \mod k = j \\
S(x,y) :- A(x,y), (y+1) \mod k = j
\]

for \( j = 0, \ldots, k-1 \), constitutes a parallelisation strategy for the program of Example 1.

The set of processors \( \{p_0, \ldots, p_{k-1}\} \) cooperate in evaluating \( P \) in parallel as follows. The processors start with a global database, residing in common memory, consisting of the input. Processor \( p_i \) performs the instantiations of the rules in the restricted version \( P_{ij} \) (i.e., instantiations that satisfy \( P_{ij} \)'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 \( p_i \) can be performed by using any single-processor evaluation method on \( P_{ij} \); however, the method has to be adjusted, to account for additions to the database made by other processors, not just \( p_i \). 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 the 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, \( p_i \), starts with the local database consisting of the input to the program, and performs the instantiations of \( P_{ij} \) as before. Processor \( p_i \) transmits to each other processor, \( p_j \), the set of tuples that \( p_i \) computes. Actually, this set, denoted \( T_{ij} \), may be less than the whole set of tuples computed by \( p_i \). This issue is addressed in section 5. The processor \( p_i \) 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.

**VERTICAL-PARTITIONING:**

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 \( P_j \).

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:

\[
\bigcup_{S \text{ is an intentional predicate}} \bigcup_{i=1}^k S^i
\]

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]). $EVAL-INCR(S_1,R_1,...,R_k,S_1,...,S_m,\Delta Q_1,...,\Delta Q_m)$ is a function that computes the tuples of $S_i$ that can be obtained by the instantiations of rules, given extensional relations $R_1,...,R_k$, intentional relations $S_1,...,S_m$, and differential relations $\Delta Q_1,...,\Delta Q_m$. 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 $P_j$; in step 2, a subset of the newly computed tuples in step 1, i.e. of the differentials ($\Delta$'s)

---

PSNE for processor $P_j$:

1. for $i:=1$ to $m$ do begin;
2. $\Delta S_i := EVAL(S_i,R_1,...,R_k,\emptyset,\ldots,\emptyset)$; /* evaluation of restricted version $P_j$ */
3. $S_i := \Delta S_i$;
4. end;
5. send from the $\Delta S_i$'s the subset $T_m$, to each processor $P_n$;
6. r 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. $\Delta S_i := EVAL-INCR(S_i,R_1,...,R_k,S_1,...,S_m,\Delta Q_1,...,\Delta Q_m)$; /* 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 $T_m$, to each processor $P_n$;
14. for $i:=1$ to $m$ do;
15. $\Delta S_i \leftarrow \Delta S_i \cup \{S_i-facts received the other processors during the last iteration\}$;
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 $P_i$'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.
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 \( p_0, \ldots, p_{2-1} \) 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 above 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 relational algebra expression (2) of Example 2. Denote by \( h \) is some hash function \( h: z \to [0,1] \). Suppose that there are two processors, \( p_0 \) and \( p_1 \), where \( p_0 \)
evaluates the program:

\[
P_{20}: \quad S(y) : - S(z), A(z,y), h(z) = 0 \\
S(y) : - A(a,y), h(y) = 0
\]

and \( p_1 \) evaluates the program:

\[
P_{21}: \quad S(y) : - S(z), A(z,y), h(z) = 1 \\
S(y) : - A(a,y), h(y) = 1
\]

Then, in order to ensure that each output tuple is computed 'by at least one processor, \( p_0 \) has to transmit to \( p_1 \) only the tuples \( S(y) \) that it computes, and for which \( h(y) = 1 \); and vice-versa, \( p_1 \) transmits only the tuples for which \( h(y) = 0 \).

**Remark:** In [CW] we proposed similar work partitioning, but there we did not consider partitioning with a hash function such as \( h \) above, that takes as parameter a nondistinguished variable. The reason is that we analyzed a special case of instantiation-partitioning, namely output partitioning. Now assume that we partition the evaluation of the recursive rule as in [CW], based on the distinguished variable, \( y \). First, note that this will necessitate transmission of all newly generated tuples at each iteration. Second, 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 evaluation work on the variable \( y \), will probably not result in much savings compared to a single processor (since the same number of \( S \)-blocks will have to be scanned), whereas partitioning the work on the variable \( z \) will probably cut the evaluation load in half, for each processor.

Formally, given an input \( I \) to a program \( P \), we define the set of tuples \( T_y \), 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 \( SR_j \) and \( SC_i \). First we define the set \( SR_j \). An \( S \)-fact, \( f \), is in \( SR_j \) if and only if:

(Condition K 1) there is some rule of the program \( P \), say \( r_z \), such that \( f \) is not in \( r_z \), but there is an instantiation of it that satisfies the predicate \( h_{ij} \), and \( f \) appears in the body of the instantiated rule.

In other words, a tuple \( f \) is in \( SR_j \), if there is an instantiation for which \( p_j \) is in charge, that uses \( f \). Determining whether a given fact is in \( SR_j \) 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 \( h_{ij}(x_1, \ldots, x_m) \), 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 $SR_j$, 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 $SR_j \cap SC_i$. 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 $p_i$ must have also sent it to $p_j$.

6. 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 "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 $K_1$ (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.

The next example demonstrates the strategy-change procedure.

Example 3: Consider the canonical strongly linear program ($csl$):

$$r_1: S(x,y) :- UP(x,u), S(u,v), DOWN(v,y)$$

$$r_2: S(u,v) :- FLAT(u,v)$$

Assume that the initial strategy consists of $csl_0$ and $csl_1$, where $csl_0$ is $csl$ with the predicate even ($u$) appended both $r_1$ and $r_2$; $csl_1$ is $csl$ with the predicate odd ($u$) appended to $r_1$ and $r_2$. Processor $p_i$ evaluates $csl_i$ ($i = 0,1$) using semi-naive evaluation.

Consider the following input (the tuples of the input relations $UP$, $DOWN$, $FLAT$ are represented by corresponding arcs; for example $(48,50)$ is a tuple of $UP$):
In this case, the whole computation burden falls on $p_0$. Consider the scenario in which $p_1$ requests a change of strategy after $p_0$ has performed three iterations of semi-naive evaluation. Then, the $S$ relation at $p_0$ consists of the set of tuples

$$ \{(50,200),(48,248),(46,246),(44,244), (100,300),(98,398),(96,396),(94,394)\} $$

The $S$ relation at $p_1$ is empty. Suppose that in the new strategy requested, $csl'_0$ consists of $csl$ with the predicate $v < 300$ appended to both rules, and $csf'_1$ consists of $csl$ with the predicate $v \geq 300$ appended to both rules. Then $p_0$ sends to $p_1$ the following subset of tuples of the relation $S$

$$ \{(100,300),(98,398),(96,396),(94,394)\} $$

and continues the semi-naive evaluation with $\Delta S = \{(44,244)\}$. Processor $p_1$ starts the evaluation with $S = \Delta S = \{\text{the set of received tuples}\}$. []

Why does this protocol ensure that no output is lost? A moment of reflection reveals the following. First, the protocol by which the leader changes strategies ensures that when the computation ends, each processor has the same "current", or last parallelization strategy, $H$. Now suppose that some fact, $f$, is not in any local database of a processor when the parallel evaluation algorithm completes. Assume without loss of generality, that $f$ can be obtained by some productive instantiation, $p$, of some rule $r_q$ of the program $P$, given a database that is the union of all the local databases. Since $H$ is a parallelization strategy, then the instantiation $p$ must satisfy the restricting predicate of rule $r_q$, for some restricted version $P_s$ of $H$. Then, it is not possible for the processor $p_s$, evaluating $P_s$, to
to be idle, if, as the distributed termination protocol ensures, there are no messages "in transit", and all processors are idle. The reason is, that at this point the facts in the body of \( r_q \) instantiated by \( p \), must exist in the local database of \( p \).

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. For the rest of this section we shall discuss an optimization of the load balancing scheme, that is appropriate for the PSNE algorithm. As explained, each processor, \( p_j \), changes strategy at the beginning of a semi-naive evaluation iteration, specifically at step 6.1. However, if the program evaluated is linear, then at that point \( p_j \) should transmit to each other processor, \( p_k \), 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. In Example 3, this means that \( p_o \) sends to \( p_1 \) only the tuple \((94,394)\), instead of the set \([ (100,300),(98,398),(96,396),(94,394) ]\).

Note that this reduction in the size of \( T_{j\alpha} \) 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, \( p_k \), since the size of both, the differential \( \Delta S \), and the relation \( S \), shrinks. Note also that, in general, the optimization cannot be performed if the program is not linear. The following example demonstrates this.

**Example 4:** Consider the following program, \( P \), that has exactly one intentional predicate, \( S \).

\[
\begin{align*}
S(w,x) & :=- S(w,y), S(z,w), A(x,y,z) \\
S(x,w) & :=- S(w,y), S(z,w), A(x,y,z) \\
S(w,x) & :- B(w,x)
\end{align*}
\]

The initial strategy consists of restricted versions \( P_0 \) and \( P_1 \); \( P_0 \) is obtained from \( P \) by appending the predicate \( even(w) \) to the body of the three rules. Similarly, \( P_1 \), is obtained by appending the predicate \( odd(w) \) to the body of the three rules. Consider the input consisting of:

\[
\begin{align*}
A & = \{(4,2,2),(6,4,4),(8,4,6)\} \\
B & = \{(50,2),(2,50),(60,2),(2,60)\}
\end{align*}
\]

Processor \( p_1 \) does not generate any new tuples during the first iteration, therefore it waits. Processor \( p_0 \), after the first iteration, has \( \Delta S^{(1)} = \{(4,50),(50,4),(60,4),(4,60)\} \). After the second iteration it has \( \Delta S^{(2)} = \{(6,50),(50,6),(60,6),(60,6)\} \). Up to this point \( p_0 \) does not transmit any tuples to \( p_1 \), because \( p_0 \) has not computed, so far, any tuple that belongs to \( T_{01} \) (since all constants in the \( S \)-tuples are even).
Assume that at this point $p_1$ initiates a change of strategy, requesting that $p_0$ appends to the body of all rules the predicate $w \leq 50$, and $p_1$ appends to the body of all rules the predicate $w > 50$. The subset of $\Delta S^{(2)}$ that satisfies condition K1 is $[(6,60),(60,6)]$. When receiving this set from $p_0$, processor $p_1$ initializes $S$ and $\Delta S$ to it, and executes one iteration of semi-naive evaluation. The iteration does not produce any new tuples, and $p_1$ stops. Meanwhile, $p_0$ performs one more iteration in which the tuples $(8,50)$ and $(50,8)$ are produced. No more tuples are transmitted among the processors, and the evaluation terminates, without producing the output tuples $((8,60),(60,8))$. But at strategy-change time, if $p_0$ had transmitted to $p_1$, the subset that satisfies condition K1, of its whole $S$ relation, then tuples $((60,4),(4,60))$ would have also been transmitted, enabling $p_1$ to produce the missing tuples.

Next, we shall argue that the optimization for a linear program $P$, is correct, namely, that no output tuple is lost. Suppose, by way of contradiction, that some intentional fact $f$, is lost, i.e. not produced by any processor. Assume without loss of generality, that there is a derivation tree, $T$, of $f$ in $P$, such that each intentional fact in $T$ is produced by some processor of the set $\{p_0, \ldots, p_{t-1}\}$. Denote by $e$ the intentional fact in $T$ that is immediately below $f$ (there is only one, since the program is linear). $e$ must be in the differential of at least one processor, say $p_i$, that evaluates the restricted version $P_i$ of at least one of the strategies that $p_i$ executes. There are two possibilities. First, assume that after producing $e$, and before starting the next iteration of semi-naive evaluation, processor $p_i$ does not send $e$ to any other processor (sending may occur in steps 13 or 6.1 of the PSNE algorithm). In this case, the instantiation, $\rho$, that produces $f$ from $e$ must satisfy $p_i$'s restricting predicates, and $p_i$ will produce $f$ in the next iteration. Second, assume otherwise, namely that $e$ is sent to another processor, say $p_j$. Processor $p_j$ also, either performs the instantiation $\rho$ in its next iteration, in which case $f$ is produced, or $p_j$ sends $e$ to another processor (it may eliminate $p$ by its restricting predicates, if the strategy has already changed since $e$ was sent by $p_i$, and before it was received by $p_j$). In any case, each processor performs the instantiation $\rho$, or it transmits $e$. Since the algorithm does not end while tuples are in transit, and when it ends all processors must have the same "current" strategy, $f$ must be produced.

7. Unique Source and Destination Properties

Let $P$ be a program, and $\{p_0, \ldots, p_{t-1}\}$ 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 $SR_{f_0}$. This means that each $S$-fact, $f$, is transmitted to only one processor, by any processor that computes $f$. For example, the strategy:

$$
S(x,y) :- \text{UP}_1(x,z), S(z,w), \text{DOWN}_1(w,y), (x+w) \mod k = j
$$

$$
S(x,y) :- \text{UP}_2(x,z), S(z,w), \text{DOWN}_2(w,y), (x+w) \mod k = j
$$

$$
S(x,y) :- \text{FLAT}(x,y), x \mod k = j.
$$

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 $x \mod k = j$. 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 $P$.

**Proposition 1:** Let $P$ be a program, and let $H = \{P_0, ..., P_{2^e}\}$ 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 $t_1, ..., t_e$ 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 $x_1, ..., x_n$, i.e., the arguments of the restricting predicates $h_j$, appear in positions $t_1, ..., t_e$ of $S_0$, respectively (and consequently $v = q_j$).

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, $a_1, ..., a_n$, and for every $m$, $h_{jm}(a_1, ..., a_n)$ is true if and only if $h_j(a_1, ..., a_n)$ is true.

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 $CT_{f_0}$. In other words, if $f$ is in the output of the program $P$, then it is computed by the processor $p_{f_0}$, and only by this processor. For example, the strategy:

$$
S(x,y) :- \text{UP}_1(x,z), S(z,w), \text{DOWN}_1(w,y), x \mod k = j
$$

$$
S(x,y) :- \text{UP}_2(x,z), S(z,w), \text{DOWN}_2(w,y), x \mod k = j
$$

$$
S(x,y) :- \text{FLAT}(x,y), x \mod k = j
$$
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 $z \mod k = j$. This strategy does not have the unique source property.

The next proposition, giving a sufficient condition for a strategy to have the unique-source property, is identical to Proposition 1, except that it refers to $S$-atoms in the head, rather than body, of rules.

**Proposition 2:** Let $P$ be a program, and let $H = \{P_0, ..., P_{k-1}\}$ 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 $t_1, ..., t_v$ 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 $x_1, ..., x_q$, i.e., the arguments of the restricting predicates $h_j$, appear in positions $t_1, ..., t_v$ 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, $a_1, ..., a_v$, and for every $m$, $h_m(a_1, ..., a_v)$ is true if and only if $h_m(a_1, ..., a_v)$ is true.

Assume that a strategy has both, the unique source and destination properties with respect some intentional predicate, $S$, and furthermore the source and destination coincide, i.e. are the same processor, for each $S$-fact. Then no $S$-fact has to be transmitted among the processors. In [CW] we discussed pure decomposition schemes, namely parallel evaluation algorithms that do not incur any communication or synchronization overhead. For example, the following parallelization strategy for computing the transitive closure (a strongly decomposable program) does not incur any overhead, regardless of the evaluation method of each processor.

$$S(x,y) :- S(x,y), A(x,y), x \mod k = j$$
$$S(x,y) :- S(x,y), x \mod k = j.$$ 

Actually, the pure decomposition schemes evaluate strategies with a coinciding unique source and destination properties. Thus, we have shown here that the pure decomposition strategies discussed in [CW] are strategies that have two independent properties, each one important in its own right.

8. 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 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 $Q$ defines $R$). Such a program has a stratification, i.e. a nonnegative numbering of the predicate symbols, such that if $S$ is defined by $T$, 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 $\neg S$ 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, $r_s$, whose head is at stratum $b$ or higher, such that $f$ is not in $r_s$, but there is an instantiation that satisfies the predicate $h_{ij}$, 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

---

1. 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
in $S$:  

\[
T(x,y) : - T(x,z), B(z,y), S(x,y), x \mod k = j
\]

\[
T(x,y) : - B(x,y), S(x,y), x \mod k = j
\]

\[
S(x,y) : - S(x,z), A(z,y), x \mod k = j
\]

\[
S(x,y) : - A(x,y), x \mod k = j
\]

for $j = 0, \ldots, k-1$. 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:

\[
U(x,y) : - C(x,y), T(z,y), x \mod k = j
\]

for $j = 0, \ldots, k-1$; 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 $S_1, \ldots, S_m$ the intentional predicates that appear negated at stratum $s$. Denote by $G$ the set that consists of $S_1, \ldots, S_m$, 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.

9. 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, and we discussed the set of tuples that should be
transmitted for this simulation. Is the set minimal? In some cases it may not be. For example, if a fact, \( f \), produced by a processor, \( P_j \), can be used in an instantiation, \( \rho \), that \( P_i \) is responsible for, then \( P_j \) will send \( f \) to \( P_i \). However, \( \rho \) may not produce any new fact, in which case \( f \) is transmitted unnecessarily. However, in an upcoming paper, we address the problem of determining decomposability of an arbitrary datalog program, i.e. existence of a strategy for which no communication is necessary. We show that the problem is undecidable. Based on this, we conjecture that determining the minimum set of tuples that has to be transmitted for a given strategy, is also undecidable.

Next 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 by redistributing 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.

10. References


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