This paper presents several complementary methods for the parallel, bottom-up evaluation of Datalog queries. We introduce the notion of a discriminating predicate, based on hash functions, that partitions the computation between the processors in order to achieve parallelism. A parallelization scheme with the property of nonredundant computation (no duplication of computation by processors) is then studied in detail. The mapping of Datalog programs onto a network of processors, such that the result is a nonredundant computation, is also studied.

1. INTRODUCTION

The efficient bottom-up evaluation of queries in a deductive database, defined by Datalog programs, is presently an active area of research [4, 14]. The bulk of the work has centered around optimization techniques for the sequential evaluation of such programs. Recently, the idea of using parallel evaluation as a means for improving performance has been suggested by Wolfson, Silberschatz, and others [6, 8, 18, 19].

The parallel evaluation of a single logic program is based on the use of multiple cooperating processes that work concurrently. The problem has been theoretically investigated by Kanellakis, Van Gelder, Ullman, and others [1, 11, 15], by characterizing logic programs that belong to the NC complexity class. A program is in NC, if it can be evaluated in polylogarithmic time given a polynomial number of processors. These nice properties, however, are not very useful for the type of database processing that we are concerned with for the following two reasons:

*This material is based in part upon work supported by NSF Grant IRI-8805215 and IRI-9003341.
*Microelectronics and Computer Technology Corporation, Austin, Texas 78759.
Address correspondence to S. Ganguly, A. Silberschatz, or S. Tsur, Department of Computer Sciences, The University of Texas, Austin, Texas 78712.
Accepted April 1991.
A polynomial number of processors in the size of the database may not be realistic given the current technology, since the size of real database systems may be in the order of hundreds of megabytes.

Algorithms in the NC class are assumed to communicate extensively and hence, their theory is of little utility in nonshared-memory architectures.

In this paper, we assume an environment with a constant (though unbounded) number of processors that communicate either through message passing or through shared memory. We present several methods for the parallel, bottom-up evaluation of a restricted class of Datalog queries.

Our paper extends and generalizes the original results of Wolfson et al. [6, 18, 19]. In particular, our scheme differs from the published ones in the following respects:

1. The strategies presented in [6, 18, 19] do not allow for partitioned base relations, i.e., all of the participating processors are assumed to share the same base data. The parallelization scheme presented in this paper is based on a more general paradigm, and allows us to derive new parallel algorithms. Thus, our methods allow for evaluations over partitioned base relations. For instance, the parallel computation of the transitive closure by Valduriez and Khoshafian [16], is a particular case of our method, as we show in Section 4.

2. The strategy presented by Dong [8] is based on decomposing databases such that they do not share the set of constants appearing in each. The practical limitations of this approach are the following. First, arbitrary fragmentations of the database may actually share constants. Second, the scheme has limited scalability. Hence, we need a more widely applicable theory.

3. Our method of mapping the Datalog programs to processors results in nonredundant computations in the sense that the same firing of the rules is never used by two distinct processors.

4. By restricting our attention to linear sirups, we show that often, limited forms of communication among the processors are sufficient. For the class of linear sirups, we develop a technique for deriving a minimal communicating network in the sense that links exist in this network only for those pairs of processors that need to communicate during the computation. This derivation can be performed at compile time and can be used to adapt the parallel execution onto an existing parallel architecture.

5. We show that the scheme for parallelizing linear programs without communication, as presented in [18], is a special case of a general scheme described in Section 6. Our scheme explicitly demonstrates the trade-off between nonredundancy and communication and is similar in spirit to the results presented in [13]. However, the noncommunicating schemes are not always superior to the communicating ones since they may lead to the duplication of work on some of the participating processors. This duplication can be avoided by resorting to communication.

The remainder of the paper is organized as follows. In Section 2 we present the preliminaries and the notation we use throughout the paper. In Section 3 we introduce a general parallelization scheme without redundancy through the use of discriminating variables and hash functions. In Section 4, we demonstrate the generality of our scheme by deriving some previously known examples and also a
new example. In Section 5 we discuss the relationship between the discriminating variables and the resulting minimal communication network. Section 6 generalizes our results and discusses cases in which communication may be required. We conclude in Section 7 and suggest extensions to this work.

2. PRELIMINARIES AND NOTATION

A term in Datalog is a constant or a variable. An atom is a predicate symbol with a constant or a variable in each of its arguments. A ground atom is an atom with a constant in each of its arguments. We assume that the constants are natural numbers. Given an atom \( A \), \( \text{var}(A) \) denotes the set of variables appearing in \( A \). A \( p \)-atom is an atom having \( p \) as the predicate symbol. A \( p \)-tuple is a ground \( p \)-atom. A rule consists of an atom \( Q \), designated as the head, and a conjunction of one or more atoms, denoted by \( Q_1, \ldots, Q_k \) designated as the body. Such a rule is denoted as \( Q: - Q_1, \ldots, Q_k \). A substitution \( \theta \) is a finite set of the form \( \{v_1/t_1, \ldots, v_n/t_n\} \), where each \( v_i \) is a variable, each \( t_i \) is a term distinct from \( v_i \) and the variables \( v_1, \ldots, v_n \) are distinct. \( \theta \) is called a ground substitution if the \( t_i \) are all ground terms. An expression is either a term, a sequence of terms, a literal or a conjunction or disjunction of literals. Let \( \theta = \{v_1/t_1, \ldots, v_n/t_n\} \) be a substitution and \( E \) be an expression. Then \( E\theta \), the instance of \( E \) by \( \theta \), is the expression obtained from \( E \) by simultaneously replacing each occurrence of the variable \( v_i \) in \( E \) by the term \( t_i \), \( i = (1, \ldots, n) \). If \( E\theta \) is ground, then \( E\theta \) is called a ground instance of \( E \).

A Datalog program is a finite set of rules whose predicate symbols (a name with a constant or a variable in its argument positions) are divided into two disjoint subsets: the base predicates, (also called extensional predicates) and the derived predicates, (also called intensional predicates). The base predicates may not appear in the head of any rule in a Datalog program. An example of a Datalog program is the following:

\[
\text{anc}(X,Y): - \text{par}(X,Y).
\]

\[
\text{anc}(X,Y): - \text{par}(X,Z), \text{anc}(Z,Y).
\]

The relation \( \text{par} \) above is an extensional relation, where \( \text{par}(X,Y) \) means that \( X \) is the parent of \( Y \). The relation \( \text{anc} \) above is a derived relation, where \( \text{anc}(X,Y) \) means that \( X \) is an ancestor of \( Y \). The first rule states that if \( X \) is a parent of \( Y \), then \( X \) is an ancestor of \( Y \). The second rule recursively states that, if \( X \) is a parent of \( Z \), and \( Z \) is an ancestor of \( Y \), then \( X \) is an ancestor of \( Y \). An input to a program \( P \) is a relation for each base predicate. An output of \( P \) is a relation for each derived predicate of \( P \). The declarative semantics for the output is the smallest model satisfying \( P \) that contains the input relations. A predicate \( Q \) in a program derives a predicate \( R \) if it occurs in the body of a rule whose head is an \( R \)-atom. \( Q \) is recursive if \( (Q, Q) \) is in the nonreflexive transitive closure of the "derives" relation. A program \( P \) is recursive if it has a recursive predicate. A rule is recursive if the predicate in its head transitively derives some predicate in its body. The theory of logic programming is comprehensively treated in [12] and in [2].
A few sections in this paper restrict their attention to linear sirups which are Datalog programs with one linear recursive rule \( r \) and one nonrecursive (exit) rule \( e \). Each such program may be canonically represented as:

\[
e: t(\overline{Z}) : - s(\overline{Z}).
\]

\[
r: t(\overline{X}) : - t(\overline{Y}), b_1, \ldots, b_k.
\]

where:

- \( t \) is the output (or derived) predicate symbol.
- \( s \) is a base relation.
- \( \overline{Z} \) is the sequence of variables appearing in the head of the exit rule.
- \( \overline{X} \) is the sequence of variables appearing in the head of the recursive rule.
- \( \overline{Y} \) is the sequence of variables which appear as arguments to the unique occurrence of \( t \) in the recursive rule.
- \( b_1, b_2, \ldots, b_k \) are the atoms with base predicates appearing in the body of the recursive rule.
- In order to ensure the safety property (i.e., finite set of answers), we assume that every variable appearing in the head of the recursive rule also appears in its body.

There are several known techniques for the bottom-up evaluation of Datalog programs, [4, 14]. In this paper, we assume that the bottom-up evaluation of Datalog programs is done using differential evaluation [4, 14].

3. PARALLELIZATION WITH NO REDUNDANCY

A very simple paradigm of parallelization is to take a sequential program and to divide the work between the processors. The parallelization scheme is said to be nonredundant, if in some sense, the processors do not duplicate computation among themselves. We present a parallelization scheme based on such a paradigm. The basic step in the bottom-up evaluation of Datalog programs [3] consists of substituting the variables in a rule by constants in the database such that each ground atom in the body of the rule is true in the extensional database or in the (partially computed) intensional database. Each iteration of the differential evaluation uses a set of possible ground substitutions applicable to a rule. We divide the workload between the processors by partitioning the set of possible ground substitutions used by the semi-naive evaluation. This is done by using discriminating functions based on hashing. Thus each processor uses only a subset of the set of possible ground substitutions, and two distinct processors do not use the same ground substitution. We term our scheme the substitution partitioned scheme. With this intuitive explanation, it may now seem obvious that the parallelization scheme is in some sense nonredundant, because a processor does not duplicate the work (i.e., ground substitutions) performed by other processors. We now formally describe our parallelization scheme.

Let \( P \) be a Datalog program whose rules are numbered from 1 to \( n \). Let \( u(r_i) \) be any nonrepetitive sequence of variables appearing in the body of the rule \( r_i \). This sequence is referred to as the discriminating sequence of variables for \( r_i \). Let
be a finite set of processors, on which the program is to be executed. For every $i$, $1 \leq i \leq n$, we define discriminating functions $h_i$ as follows:

$$h_i : \text{set of ground instances of } u(r_i) \to \mathcal{P}.$$ 

Given a Datalog program $P$, we derive a set of Datalog programs to be executed at the various processors. The parallel execution of this derived set of Datalog programs is equivalent (i.e., produces the same answer for every input) to the sequential execution of $P$.

Let $Q_i$ denote the program to be executed at processor $i$. For every recursive atom $t$, we introduce new predicate symbols $t_{in}^i$, $t_{out}^i$, and $t_{ij}^i$. The predicate symbol $t_{out}^i$ denotes the set of all $t$-tuples generated at processor $i$. The predicate symbol $t_{in}^i$ denotes the set of all $t$-tuples that are input to processor $i$. The predicate symbol $t_{ij}^i$ denotes the set of all $t$-tuples that are generated at processor $i$ and are communicated to processor $j$. If $A$ is an atom with predicate symbol $t$, then $A_{in}^i$ is the atom with the predicate symbol $t_{in}^i$ and the same arguments. Likewise, we define $A_{out}^i$ and $A_{ij}$. The program $Q_i$ consists of the following four steps:

1. **Processing.** Let $A : - B, \ldots, C$ be a rule in $P$, with discriminating sequence $v$ and discriminating function $h$. Then, include the following rule in $Q_i$.

   $$A_{out}^i : - B_{in}^i, \ldots, C_{in}^i, h(v) = i.$$ 

2. **Sending.** Let $r$ be a rule in $P$ with discriminating sequence $v$ and discriminating function $h$. Then, for every recursive atom $C$ appearing in the body of $r$ and every $j \in \mathcal{P}$, include the following rule in $Q_i$.

   $$C_{ij} : - C_{out}^j, h(v) = j.$$ 

3. **Receiving.** Let $D$ denote the atom $t(W)$, where $W$ is a sequence of all distinct variables not appearing in the original program. For every recursive predicate $t$ appearing in the program $P$ and every $j \in \mathcal{P}$, introduce the following rule in $Q_i$.

   $$D_{in}^j : - D_{ji}.$$ 

4. **Final Pooling.** Let $D$ denote the atom $t(W)$, where $W$ is a sequence of all distinct variables not appearing in the original program. For every recursive predicate $t$, include the following rule.

   $$D : - D_{out}^i.$$ 

**Theorem 3.1.** Let $Q = \bigcup_{i \in \mathcal{P}} Q_i$. Then for every input of base relations, the interpretation of a recursive predicate $t$ in the least model of $Q$ is identical to the interpretation of the recursive predicate $t$ in the least model of $P$.

**Proof.** See Appendix A. \qed

We now explain the implementation of some of the steps of the rewritten program. In order to do so, we first abstract the parallel architecture on which the parallel execution of the program is described. The abstraction is done in such a manner that it is easily implementable by either shared-memory or message-passing architectures.
Given a set $\mathcal{P}$ of processors, we assume that a processor $i$ in $\mathcal{P}$ may communicate with every other processor $j$ in $\mathcal{P}$. (This is an idealization and will be relaxed in the later sections.) We assume that communication is done by a channel numbered $ij$, denoting that the sending processor is $i$ and the receiving processor is $j$. We require that if a processor $i$ puts some data in channel $ij$, then processor $j$ (and no other processor except $j$) receives these data without error within some finite time.

We now describe the parallel execution on the above abstract architecture. The parallel execution proceeds with each processor evaluating the Datalog program $Q_i$ using the differential evaluation scheme. For each $i \in \mathcal{P}$, the relations $t_{i\text{out}}^i$ and $t_{i\text{in}}^i$ are local to processor $i$. The predicates $t_{ij}$, for $i, j \in \mathcal{P}$, represent the channel $ij$ in the abstract architecture described above. Hence, addition of tuples to the predicate $t_{ij}$ during the differential-naive evaluation, should be interpreted as processor $i$ sending the tuples to processor $j$, along channel $ij$. Similarly, assignment of tuples from the predicate $t_{ij}$ onto another predicate should be interpreted as processor $j$ receiving the tuples sent by processor $i$, along channel $ij$. The general structure of the parallel execution is:

```
evaluate initialization rule.
repeat
  evaluate processing rule.
  evaluate sending rules.
  evaluate receiving rules.
until "termination,"
```

where "termination" is the condition that all processors are idle and all channels are empty.

We now describe the implementation of each of the rules and the condition for parallel termination in some detail.

(1) **Processing.** The rule $A_{i\text{out}}^i :- B_{i\text{in}}^i, \ldots, C_{i\text{in}}^i$ is processed according to a differential evaluation scheme.

(2) **Sending.** Once tuples are generated at some iteration by processor $i$, they must be sent to different processors. The rule $C_{ij}:- C_{i\text{out}}^i, h(v) = j$ sends only those subsets of tuples generated at processor $i$ which might successfully fire the processing rule of processor $j$. Duplicate tuples generated by the same processor may be detected by a difference operation and need not be sent repeatedly. This helps to reduce the extent of communication at the expense of an extra difference operation.

(3) **Receiving.** In the processing step, the rule is evaluated using a differential algorithm. Hence in the receiving step, duplicate tuples received must be eliminated. This is done by a difference operation. Thus, after executing the processing step and the sending step, each processor collects the tuples received from all other processors, selects the set of new tuples received and uses them to fire the processing step in the next iteration. Note that the receives are asynchronous, that is, processor $i$ does not wait for data from processor $j$ if on a particular iteration, it does not receive any data from processor $j$. This is a very important property of the parallel executions resulting from our schemes.
(4) **Final Pooling.** The tuples generated by all the processors are pooled together in a common relation which, depending upon the requirements of the query and the underlying architecture, might require communication from all processors to a single processor.

(5) **Parallel Termination.** The parallel algorithm terminates when every processor in \( P \) is idle and all channels are empty. This may be detected by a distributed termination detection algorithm (e.g., [5, 7]).

So far, we have described the parallel execution of the parallel program and proved its correctness. However, the correctness of the scheme does not necessarily imply that the parallel executions would be faster than a sequential differential execution. The following discussion shows that we must restrict the choice of the discriminating sequence of variables such that each processor in the parallel evaluation may evaluate its processing step without having to replicate the effort of a sequential evaluation.

The rule in the processing step would be evaluated as the following relational algebra expression:

\[
\prod (\sigma_{h(v(r)) = 1}(B_{in} \bowtie \ldots \bowtie C_{in})).
\]

The details may be found in [14]. Consider the evaluation resulting from a choice of \( v(r) \). If the variables appearing in \( v(r) \) do not appear in any of the atoms in the body, then the selection cannot be pushed into the joins. As a result, each processor in the parallel execution computes the joins, then applies the selection and then projects on the relevant attributes. A processor engaged in sequential evaluation of the above program would compute the joins followed by the projection. Thus, each processor taking part in the parallel execution has necessarily repeated the computation done by a sequential processor.

On the other hand, if all the variables appearing in \( v(r) \) also appear in one of the atoms in the body, then the selection can be pushed within the join (as a selection on one of the join operands). Hence, if \( h \) is a discriminating function with a reasonable skew, then it can be expected that each processor has less work to do than the processor executing the entire sequential program. This implies that we should impose some restriction on the choice of the discriminating sequence \( v(r) \). Thus, for the remainder of the paper, we assume that all the variables appearing in a discriminating sequence for the recursive rule must also appear in at least one atom in the body of the recursive rule.

In order to prove that our parallelization scheme does not result in duplication of computation by processors, we must first define the following.

**Definition 3.1.** A parallelization scheme is called *semi-naive nonredundant* if for any program within the scheme, the total number of times a tuple is generated by all the processors is no more than the number of times the same tuple is generated by a sequential differential evaluation of the same program on the same data.

**Theorem 3.2.** The parallelization scheme described above is nonredundant.

**Proof.** See Appendix B. □

In the rewritten program \( Q_i \), the initialization and the processing rules contribute to the generation of tuples. The rules for sending and receiving are
essentially part of the machinery for dividing the work between the processors. These rules never generate new tuples. We make this distinction in the proof of the above theorem, where we consider the total number of times a tuple is generated by all the processors by the application of the initialization and the processing steps alone.

4. EXAMPLES

In this section we demonstrate the generality of our parallelization technique by applying it to the following Datalog program:

\[
\begin{align*}
\text{anc}(X, Y) & : - \text{par}(X, Y). \\
\text{anc}(X, Y) & : - \text{par}(X, Z), \text{anc}(Z, Y).
\end{align*}
\]

The relation \textit{par} above is a base relation, where \textit{par}(X, Y) means that \(X\) is the parent of \(Y\).

We assume that there are \(N\) processors, numbered from 1 through \(N\). Thus \(\mathcal{P} = \{1, 2, \ldots, N\}\). We present three parallel algorithms derived from our scheme by using different choices of discriminating sequence of variables. The first algorithm derived is the one presented by Wolfson and Silberschatz in [19]. This algorithm does not require any communication between the processors but requires that the base relation \textit{par} be shared among the processors. The second algorithm derived is presented by Valduriez and Khoshafian in [16]. This algorithm works on any arbitrary fragmentation of the relation \textit{par}, although in general, it requires communication. The third algorithm is a new one that was developed using our parallelization scheme. This algorithm lies between the other two algorithms in the sense that it requires less communication than the second one, but only allows for some possible fragmentations, whereas it requires more communication than the first one but does not require that the base relation be shared.

\textbf{Example 4.1.} Let \(u(r) = u(e) = \langle Y \rangle\) and let \(h' = h\) be an arbitrary discriminating function on the domain of \(Y\) with range = \(\{1, 2, \ldots, N\}\). The rewritten program for processor \(i\), denoted \(Q_i\) earlier, is defined as follows:

- **Initialization:**
  \[
  \text{anc}_{out}^i(X, Y) : - \text{par}(X, Y), h(Y) = i.
  \]

- **Processing:**
  \[
  \text{anc}_{out}^i(X, Y) : - \text{par}^i(X, Z), \text{anc}(Z, Y), h(Y) = i.
  \]

- **Sending:** For every \(j, 1 \leq j \leq N\),
  \[
  \text{anc}_{ij}(Z, Y) : - \text{anc}_{out}^i(Z, Y), h(Y) = j.
  \]

- **Receiving:** For every \(j, 1 \leq j \leq N\),
  \[
  \text{anc}_{in}(W_1, W_2) : - \text{anc}_{ji}(W_1, W_2).
  \]

- **Final Pooling:**
  \[
  \text{anc}(W_1, W_2) : - \text{anc}_{out}(W_1, W_2).
  \]
Since \( u(r) = \langle Y \rangle \), and \( Y \) does not appear in \( par(X, Z) \), it follows that \( par^i = par \). In other words, the base relation \( par \) must be either shared or replicated by the processors.

The first two rules are the only rules that derive tuples in \( anc^i_{out} \). Therefore, if \((a, b) \in anc^i_{out} \), then \( h(b) = i \). Hence, if \( i \neq j \), then evaluating the sending rule from processor \( i \) to processor \( j \) (namely, \( anc_{ij}(Z, Y) : - anc^i_{out}(Z, Y), h(Y) = j \)) does not yield any tuple. That is, \( anc_{ij} = \emptyset \), whenever \( i \neq j \). Thus, by the above choice of the discriminating sequence of variables, no communication is incurred during the recursive computation. Some communication is incurred, however, during the final pooling of the output to a common destination.

**Example 4.2.** Suppose that the base relation \( par \) is horizontally partitioned among the processors. Let the partition in processor \( i \) be denoted by \( par^i \). Thus, for \( i \neq j \), \( par^i \cap par^j = \emptyset \), and \( \bigcup_{i=1}^{n} par^i = par \).

Let \( u(r) = \langle X, Z \rangle \) and \( v(e) = \langle X, Y \rangle \). Let \( h' = h \) be defined as follows:

\[ h(a, b) = i \text{ if and only if } (a, b) \text{ is a tuple in } par^i. \]

Hence, \((par(X, Y) \land (h(X, Y) = i)) \equiv par^i(X, Y) \). The rewritten program \( Q_i \) executed by processor \( i \) is defined as:

- **Initialization:**
  \[ anc^i_{out}(X, Y) : - par^i(X, Y). \]

- **Processing:**
  \[ anc^i_{out}(X, Y) : - par^i(X, Z), anc^i_T(Z, Y). \]

- **Sending:** For every \( j, 1 \leq j \leq N \),
  \[ anc_{ij}(Z, Y) : - anc^i_{out}(Z, Y), h(X, Z) = j. \]

- **Receiving:** For every \( j, 1 \leq j \leq N \),
  \[ anc^i_T(W_1, W_2) : - anc_{ji}(W_1, W_2). \]

- **Final Pooling:**
  \[ anc(W_1, W_2) : - anc^i_{out}(W_1, W_2). \]

Thus the execution of \( Q_i \) needs access to only a given fragment \( par^i \) of the \( par \) relation, as intended.

Consider the rule that represents the sending operation from processor \( i \) to processor \( j \), namely, \( anc_{ij}(Z, Y) : - anc^i_{out}(Z, Y), h(X, Z) = j \). Equivalently, this may be rewritten as follows:

\[ anc_{ij} = \{(a, b) \mid (a, b) \in anc^i_{out} \land \exists c(c, a) \in par^j \}. \]

Thus, \( anc_{ij} \subseteq anc^i_{out} \). Since the relation \( par^j \) is not available at processor \( i \), the second conjunct of the above expression cannot be verified at processor \( i \). Hence, all tuples in \( anc^i_{out} \) are communicated to processor \( j \). Note that, in this case, the extra communication does not make the parallel execution either incorrect or redundant.

**Example 4.3.** The two examples presented above depict two extremes in the properties of interprocessor communication and sharing/replication of the base
relation $par$. We now present an algorithm that lies between these two extremes. Let $v(e) = \langle X \rangle$, $v(r) = \langle Z \rangle$ and let $h' = h$ be any discriminating function on the domain of $X$ and $Z$. The rewritten program $Q_i$ executed by processor $i$ is:

- **Initialization:**
  \[ \text{anc}_{out}^i(X,Y) : - \text{par}(X,Y), h(X) = i. \]

- **Processing:**
  \[ \text{anc}_{out}^i(X,Y) : - \text{par}(X,Z), \text{anc}_{in}^i(Z,Y), h(Z) = i. \]

- **Sending:** For every $j$, $1 \leq j \leq N$,
  \[ \text{anc}_{ij}(Z,Y) : - \text{anc}_{out}^i(Z,Y), h(Z) = j. \]

- **Receiving:** For every $j$, $1 \leq j \leq N$,
  \[ \text{anc}_{in}^i(W_1,W_2) : - \text{anc}_{ij}(W_1,W_2). \]

- **Final Pooling:**
  \[ \text{anc}(W_1,W_2) : - \text{anc}_{out}^i(W_1,W_2). \]

We note the following properties of $Q_i$.

1. Let $(a,b)$ be a tuple in $\text{anc}_{out}^i$. Then, according to the sending rule, tuple $(a,b)$ is sent to processor $j$ only if $h(a) = i$. Thus every tuple is sent to, and processed by, a unique processor. This differs from Example 4.2, where the output of a processor was sent to all the processors.

2. After the firing of the initialization rule, the processing step of $Q_i$ requires access to those tuples of $\text{par}(X,Z)$ such that $h(Z) = i$. Hence the accesses to the $\text{par}$ relation by different processors do not overlap, and thus there is no contention during the recursive processing. However, it is not true that every possible fragmentation of $\text{par}$ is permitted in this example, because only the second attribute of the $\text{par}$ relation has been arbitrarily fragmented.

The extent of communication is less here, as compared to Example 4.2. However, all possible horizontal fragmentations of $\text{par}$ is allowed in Example 4.2, but not all fragmentations are allowed here. In Example 4.1, the relation $\text{par}$ was replicated/shared among all the processors, whereas, in this case, each of the processors accesses a disjoint fragment of the $\text{par}$ relation. However, the algorithm here involves communication, whereas in Example 4.1, there is no communication between the processors. Thus, this example essentially depicts a trade-off between fragmentation and communication.

5. NETWORK CONNECTIVITY

In Section 3, we presented a general strategy for the parallel execution of Datalog programs on a set of processors. The abstract architecture assumed that every processor could communicate with every other processor. In this section, we restrict our attention to linear Datalog sirups and study how the rules of a program and the choice of the discriminating variables affect the interconnections necessary between the processors. We show that a given discriminating sequence and a given
discriminating function may yield a parallel execution, where some of the communication channels are never utilized. This property is data-independent, in the sense that for every input of base relations to the linear sirup, the parallel execution never utilizes those channels. This implies that it may not be necessary for a processor to communicate with every other processor.

In this section, we formalize these ideas. We present an algorithm, which takes as input a linear sirup, a discriminating sequence of variables, and discriminating functions (in a restricted form). The output of the algorithm consists of pairs of processors that might communicate with each other during the parallel execution. Moreover, if the discriminating functions are chosen to be linear functions (subject to some restrictions), then one can derive the optimal topological structure of the network of processors (defined later) by solving a system of linear equations.

Definition 5.1. Consider a linear recursive rule with the head \( t(X_1, X_2, \ldots, X_m) \) and the recursive atom in the body \( t(Y_1, Y_2, \ldots, Y_m) \). A dataflow graph for this rule is a directed graph \( G = (V, E) \), where:

- \( V \subseteq \{1, 2, \ldots, m\} \) and \( i \in V \) if \( \exists j \in \{1, 2, \ldots, m\} \) such that \( Y_j = X_i \).
- An edge \( i \rightarrow j \) exists in the graph if \( Y_i = X_j \).

Example 5.1. Consider the following recursive rule:

\[ p(U, V, W) : \neg p(V, W, Z), q(U, Z) . \]

The dataflow graph for this recursive rule is:

\[ 1 \rightarrow 2 \rightarrow 3. \]

The edge \( 1 \rightarrow 2 \) is in the graph because the variable \( V \) appears in the first attribute position in the predicate \( p \) in the body and also appears in the second attribute position in the head. Similarly, the edge \( 2 \rightarrow 3 \) is in the graph because the variable \( W \) appears in the second attribute position in the predicate \( p \) in the body and also appears in the third attribute position of the head.

The following theorem states a property of dataflow graphs. It is similar to the theorem presented about pivotal programs in [19].

Theorem 5.1. Given \( N \) processors and a linear sirup with a corresponding dataflow graph \( G \). If \( G \) contains a cycle, then there exists a choice of discriminating sequence of variables and functions such that the parallel execution of the linear sirup on the given \( N \) processes does not require any communication.

Proof. See Appendix C.

Example 5.2. Consider the ancestor example presented in the earlier section. Figure 1 shows the dataflow graph for it. Hence, as shown in Section 4, there is no requirement for communication between the processors when the discriminating variable is \( Z \).
Unfortunately, it is not always the case that a dataflow graph contains a cycle, as shown in Example 5.1. In such cases, the dataflow graph still provides us with an insight into the choice of the discriminating variables so that the interconnections between the processors can be reduced. To formalize this, we need to define the notion of a network graph.

**Definition 5.2.** Given a set of processors $\mathcal{P}$, we define a network graph over $\mathcal{P}$ as a directed graph $N = (V, E)$ where:

- $V = \mathcal{P}$.
- $E$ is any subset of $\mathcal{P} \times \mathcal{P}$.

If there is a directed edge $i \rightarrow j$ in $E$ then this implies that in the parallel execution of a program, data communication from processor $i$ to processor $j$ is permissible. The absence of a directed edge from $i$ to $j$ indicates that processor $i$ may not communicate with processor $j$, either directly or indirectly. Hence routing of information from $i$ to $j$ via other intermediary processors is not permitted during the parallel execution.

**Example 5.3.** Consider the following recursive rule:

$$p(X, Y) : \neg p(Y, Z), r(X, Z).$$
$$p(X, Y) : \neg q(X, Y).$$

Let $g$ be any arbitrary function on the domain of variables $X, Y$ and $Z$ with range $\{0, 1\}$. Let $\nu(e) = \langle X, Y \rangle$, and $\nu(r) = \langle Y, Z \rangle$.

Let $h'(a, b) = h(a, b) = (g(a), g(b))$. Thus, there are four possible values that $h$ can take, $(00), (01), (10)$ and $(11)$. Accordingly, let $\mathcal{P} = \{(00), (01), (10), (11)\}$. Below, we consider some of the rules of the rewritten program executed at processor $(00)$:

- **Initialization:**
  $$p_{\text{out}}^{(00)}(X, Y) : \neg q(X, Y), h(X, Y) = (00).$$

- **Processing:**
  $$p_{\text{out}}^{(00)}(X, Y) : \neg p_{\text{in}}^{(00)}(Y, Z), r(X, Z), h(Y, Z) = (00).$$

- **Sending:**
  $$p_{\text{out}}^{(00)}(Y, Z) : -p_{\text{in}}^{(00)}(Y, Z), h(Y, Z) = (ij).$$

We see from the processing and the initialization rules that if $(a, b) \in p_{\text{out}}^{(00)}$, then $g(b) = 0$. Consider the rule that represents the operation of sending tuples from processor $(00)$ to processor $(01)$. Then, if $(a, b) \in p_{\text{out}}^{(00)}$, then by the sending rule, $(a, b) \in p_{\text{out}}^{(00)}$, and $h(a, b) = (01)$. If $h(a, b) = (01)$, then $g(b)$ must be 1. Thus we conclude that for any input of the base relations and any choice of the function $g$, there is no communication from processor $(00)$ to processor $(01)$. By the same argument, there is no communication from processor $(00)$ to processor $(11)$.

On the other hand, if $(a, b) \in p_{\text{out}}^{(00)}$, then $g(a)$ could be 1, and there is the possibility of communication from processor $(00)$ to processor $(10)$. Carrying out this analysis for every processor yields the network graph shown in Figure 2.

Given a linear sirup $P$, a sequence of discriminating variables, and discriminating functions satisfying some conditions, there is an algorithm to generate the
minimal network graph $N$ to evaluate $P$. The network is minimal in the sense that, for every communication edge in the network, there exists an input database, such that the parallel execution of $P$ on this database, results in communication along that edge. This algorithm and its proof of correctness are described in [10]. Here we show some examples to illustrate our ideas.

Consider the variables that appear in the recursive atom in the body in those attribute positions which are the vertices on the corresponding dataflow graph. In general, using these variables (or some of these variables) as the discriminating sequence helps to reduce the connectivity of the network graph. Thus, the dataflow graph gives an insight into the choice of the discriminating variables to minimize connectivity and communication. The following example illustrates the idea. It further shows that if the combination functions are chosen to be linear functions, then the network graph can be derived by solving a system of linear equations subject to some constraints.

Example 5.4. Consider the Datalog program:

\[
\begin{align*}
p(U, V, W) & : - s(U, V, W). \\
\end{align*}
\]

The dataflow graph for this program as explained in Example 5.1 is:

\[
1 \rightarrow 2 \rightarrow 3.
\]

Let $v(r)$ be $<V, W, Z>$, and $v(e)$ be $<U, V, W>$. Let $g$ be an arbitrary function from the constants of the database to the set $\{0, 1\}$. Define the discriminating functions $h$ and $h'$ as follows:

\[
h(a_1, a_2, a_3) = h'(a_1, a_2, a_3) = g(a_1) - g(a_2) + g(a_3).
\]

Hence the range of $h$ is $\{0, 1 - 1, 2\}$ and thus, $\mathcal{P} = \{0, 1 - 1, 2\}$.

If processor $i$ communicates with processor $j$, then, there must be a tuple $p(a_1, a_2, a_3)$ that is produced by processor $i$ and used as input by processor $j$. Let $g(a_1) = b_1$, $g(a_2) = b_2$ and $g(a_3) = b_3$. If $p(a_1, a_2, a_3)$ is used as input at processor $j$ then,

\[
h(a_1, a_2, a_3) = b_1 - b_2 + b_3 = j.
\]

(1)
If \( p(a_1, a_2, a_3) \) is produced by processor \( i \), it could be produced by firing either the recursive rule or the exit rule. If the exit rule is used then,

\[
h'(a_1, a_2, a_3) = h(a_1, a_2, a_3) = b_1 - b_2 + b_3 = i. \tag{2}
\]

The only solutions of equations (1) and (2) above are when \( i = j \). This means that processor \( i \) communicates with processor \( j \) only when \( i = j \). Hence, this solution is trivial. Suppose that the tuple \( p(a_1, a_2, a_3) \) is produced at processor \( i \) by firing the recursive rule. Then there must be a tuple \( p(a_2, a_3, a_4) \) for some \( a_4 \) which enables the successful firing of the processing step at processor \( i \), and is used as input. Let \( g(a_4) = b_4 \). Hence,

\[
b_2 - b_3 + b_4 = i. \tag{3}
\]

Equations (1) and (3) are subject to the constraint that \( b_1, b_2, b_3, b_4 \in \{0, 1\} \).

Since we are interested in finding all pairs of processors \( i \) and \( j \) such that there is communication from \( i \) to \( j \), we solve the set of equations (1) and (3) for all values of \( b_1, b_2, b_3, b_4 \in \{0, 1\} \) and \( i, j \in \{0, -1, 1, 2\} \). Equivalently, we solve the following system of equations:

\[
x_1 - x_2 + x_3 = v \tag{4}
\]

\[
x_2 - x_3 + x_4 = u \tag{5}
\]

subject to the constraints that \( x_1, x_2, x_3, x_4 \in \{0, 1\} \).

A solution to the above system of equations is a vector of the form \((x_1, x_2, x_3, y, u, v)\). Since we are interested in the last two components alone, we introduce an edge from processor \( u \) to processor \( v \) in the network graph whenever \( u \) and \( v \) appear as the last two components of some solution vector. The network graph thus obtained is shown in Figure 3.

6. TRADE-OFF BETWEEN REDUNDANCY AND COMMUNICATION

In this section we present a scheme that exhibits a trade-off between redundancy and communication. We start our discussion by presenting a parallelization scheme that requires no communication. This scheme was first presented in [18].

![Network graph](image-url)
Let $P$ be a linear sirup and let $v(e)$, $P$ and $h'$ be as defined in Section 3. The program to be executed at processor $i$ consists of the following three execution steps:

1. **Initialization.** A new predicate $t'$ is defined whose interpretation is the set of all $t$-tuples that are processed at processor $i$ at some point in the execution:
   
   \[ t'(\bar{Z}): = s(\bar{Z}), h'(v(e)) = i. \]

2. **Recursive Processing:**
   
   \[ t'(\bar{X}) := t'(\bar{Y}), b_1, \ldots, b_k. \]

3. **Final Pooling:**
   
   \[ t(\bar{W}) := t'(\bar{W}). \]

This program scheme and its proof of correctness were first presented in [18]. Here we list some of the properties of this scheme.

1. No communication is necessary during the recursive computation.
2. The computation may be semi-naive redundant. That is, the same tuple may be generated in the parallel execution more times than in the sequential semi-naive evaluation. Hence computation may be duplicated at the processors.
3. In general, base relations need to be either shared or replicated.

The scheme presented above is a special case of a more general parallelization scheme which exhibits a trade-off between nonredundancy and communication. This general scheme is presented below.

The definitions of $v(e)$, $P$ and $h'$ are the same as in Section 3. However, we require that every variable in $v(r)$ also appears as an argument to the recursive atom in the body. In other words we require that every variable in $v(r)$ also appears in $\bar{Y}$, following the canonical representation of a linear sirup given in Section 2. Also, for every processor $i$ in $P$, we define a discriminating function $h_i$ as follows:

\[ h_i: \text{set of ground instances of } v(r) \rightarrow P. \]

As in Section 3, we derive a set of Datalog programs to be executed at the various processors, and whose parallel execution is equivalent to the sequential execution of the given Datalog sirup. Let $R_i$ denote the program to be executed at processor $i$. It consists of the following five execution steps. The meaning of the predicate symbols $t_{in}^i, t_{out}^i$ etc. are the same as in Section 3. Therefore, we do not repeat the explanations here.

1. **Initialization:**
   
   \[ t_{out}^i(\bar{Z}) := s(\bar{Z}), h'(v(e)) = i. \]

2. **Processing:**
   
   \[ t_{out}^i(\bar{X}) := t_{in}^i(\bar{Y}), b_1, \ldots, b_k. \]

3. **Sending:** For every $j \in P$, we introduce the following rule in $R_i$.
   
   \[ t_{ij}(\bar{Y}) := t_{out}^i(\bar{Y}), h_i(v(r)) = j. \]
(4) **Receiving:** For every \( j \in \mathcal{P} \), we introduce the following rule in \( \mathit{R}_i \).

\[
t'_{ji}(\overline{W}) : \neg t_{ji}(\overline{W}).
\]

(5) **Final Pooling:**

\[
t(\overline{W}) : \neg t'_{\text{out}}(\overline{W}).
\]

The parallel execution of the above program on the abstract architecture proceeds in exactly the same manner as described in Section 3. Note that the major distinction between the program \( \mathit{R}_i \) and the program \( \mathit{Q}_i \) defined in Section 3 is that the discriminating functions \( h_i \) used by the processors may be different from one another. In \( \mathit{Q}_i \), this was not allowed.

In operational terms, this rewriting allows a processor to transmit any arbitrary fragment of the computed result to the other processors and retain the remaining for self-processing. The decision as to whether to communicate tuples is a local decision, since the various \( h_i \)’s may be distinct. However, such flexibility may result in redundant computation with the advantage of less communication. The correctness of the transformation is asserted in the next theorem.

**Theorem 6.1.** Let \( R = \bigcup_{i \in \mathcal{P}} \mathit{R}_i \). Then for every input of base relations, the interpretation of \( t \) in the least model of \( R \) is identical to the interpretation of \( t \) in the least model of \( P \).

**Proof.** See Appendix D.

Having established the correctness of the transformation, let us now examine some of the properties of this scheme.

(1) Let \( h_i(a_1, a_2, \ldots, a_m) = i \) for every tuple \((a_1, a_2, \ldots, a_m)\). If \( i, j \in \mathcal{P} \), and \( i \neq j \), the set of tuples transmitted from processor \( i \) to processor \( j \) is empty. Hence for this specific choice of the discriminating functions, the parallel execution does not require any communication, and proceeds exactly like the one presented in the beginning of the section.

(2) Suppose that \( h_i = h \), for every \( i \in \mathcal{P} \). The rewritten program for processor \( i \) now looks as follows

\[
t'_{\text{out}}(\overline{Z}) : s(\overline{Z}), h'(v(e)) = i.
\]

\[
t'_{\text{out}}(\overline{X}) : -t_{\text{in}}(\overline{Y}), b_1, \ldots, b_k.
\]

\[
t_{ij}(\overline{Y}) : -t'_{\text{out}}(\overline{Y}), h(v(r)) = j.
\]

\[
t'_{\text{in}}(\overline{W}) : -t_{ji}(\overline{W}).
\]

\[
t(\overline{W}) : -t'_{\text{out}}(\overline{W}).
\]

Recall that for this section, we have restricted that all variables in \( v(r) \) must also appear in \( Y \). Hence \( t'_{\text{in}}(Y) \Rightarrow h(v(r)) = i \). Thus, the processing step of \( \mathit{Q}_i \) may be simplified to:

\[
t'_{\text{out}}(\overline{X}) : -t'_{\text{in}}(\overline{Y}), b_1, \ldots, b_k.
\]

Hence the above program is equivalent to \( \mathit{Q} \) presented in Section 3. Thus if each processor uses the same discriminating function for the recursive rule, then the parallel computation is nonredundant.
The above rewriting scheme and its associated parallel execution subsume the two schemes that were presented earlier, the ones with no communication and no redundancy. However, many different execution schemes may result by making different choices of the discriminating functions \( h_i \) and \( h' \). These were inadmissible in the two schemes presented earlier. By making arbitrary choices of the discriminating functions, one can alter the properties of the parallel execution.

In general, the parallel execution is not completely nonredundant, and may require communication. We see that in nonredundant execution, every tuple is processed by a unique processor, and hence if this tuple is produced by different processors, then some communication is incurred to transfer all of them to the same destination. However, if uniqueness of processing sites is not maintained for every tuple but instead, some tuples are processed at the same processor where they were generated, then nonredundancy may be lost with a possible decrease in communication. The following example illustrates this trade-off between nonredundancy and communication for a specific linear sirup.

**Example 6.1.** Consider the following linear sirup for computing the “same-generation” relation.

\[
\text{sg}(X, X) : - \text{ vertex}(X).
\]

\[
\text{sg}(X, Y) : - \text{ up}(U, X), \text{ sg}(U, V), \text{ dn}(V, Y).
\]

Suppose that we are given \( N \) processors and that we wish to parallelize the bottom-up evaluation of this program using the general scheme presented above. Let the discriminating variable sequence for the recursive rule be \( \langle U \rangle \). We assume that the tuples of the form \( \text{sg}(X, X) \) are an insignificant fraction of the set of all \( \text{sg} \) tuples and hence do not partition the evaluation of the exit rule. Note that this is an approximation of the general scheme whose only purpose is to make the analysis easier. Thus the program executed at processor \( i \) is as follows:

\[
\text{sg}_{\text{out}}^i(X, X) : - \text{ vertex}(X).
\]

\[
\text{sg}_{\text{in}}^i(X, Y) : - \text{ up}(U, X), \text{ sg}_{\text{in}}^i(U, V), \text{ dn}(V, Y).
\]

\[
\text{sg}_{\text{out}}^i(U, V) : - \text{ sg}_{\text{out}}^i(U, V), h_i(U) = j, \quad \text{for } 1 \leq k \leq N.
\]

\[
\text{sg}_{\text{in}}^i(W_1, W_2) : - \text{ sg}_{\text{in}}^i(W_1, W_2), \quad \text{for } 1 \leq k \leq N.
\]

\[
\text{sg}(W_1, W_2) : - \text{ sg}_{\text{out}}^i(W_1, W_2).
\]

Let us denote the probability of an event \( A \) by \( Pr[A] \). For \( 1 \leq i \leq N \) and for \( 0 \leq s \leq 1 \), let \( Pr[h_i(a) = i] = s \) independent of \( i \) and the data value \( a \). Clearly, \( Pr[V \sim i \setminus h_i(a) = i] = 1 \). Hence, it follows that \( N \times s \geq 1 \) or that \( s \geq 1/N \).

If the \( i \)th processor uses the hash function \( h_i \) to process its input, it is reasonable to assume that the average processing time spent by processor \( i \) is proportional to \( s \times (\text{processing time for the sequential evaluation}) \). Let \( R(s) \) denote the average processing time spent by processor \( i \) as a function of \( s \). Let \( C(s) \) denote the average number of tuples sent by processor \( i \) as a function of \( s \).

Let \( S \) denote the event that a particular tuple is communicated from a given processor. Let \( P \) denote the event that a particular tuple is produced at a given processor. The set of tuples produced by all the processors are equal to the set of tuples received by all the processors. Hence the probability of \( P \) is equal to the probability that a particular tuple is processed by the same processor. This
probability may be likewise assumed to be equal to \( s \). Thus \( Pr[P] = s \) and \( R(s) = s \times C_0 \) where \( C_0 \) is the cost of sequential evaluation.

Now \( Pr[S|P] = 1 - s \). Hence, \( P[S] = Pr[S|P] \times Pr[P] = (1 - s) \times s \). Thus, \( C(s) = s \times (1 - s) \times M \), where \( M \) is the size of the output. We note the following:

1. For \( s \geq 1/2 \), the communication cost decreases whereas the processing cost increases, confirming the intuition that there is a trade-off between redundant computation and communication.

2. For \( 1/N < s < (N - 1)/N \), we have \( R(s) < R(1/N) \) and \( C(s) < C(1/N) \). Thus, we can conclude that for almost all reasonable metrics of performance (e.g., response time, \( R + C \), etc.) the value of the metric for \( s \) lying within the interval will be greater than when \( s = 1/N \). Hence, the interesting values of \( s \), where optimizer effort should be focused, is \( s = 1/N \) and \( s > (N - 1)/N \). The interval \( (1/N, (N - 1)/N) \) is most likely to be uninteresting for practical cost metrics. We illustrate this analysis by assuming that the cost metric is:

\[
F(s) = R(s) + C(s) \times C_1 = s \times (1 - s) \times M \times C_1 + s \times C_0,
\]

where \( C_1 \) is the average cost of transmitting a tuple. Since \( F(s) \) is a second-degree polynomial whose highest coefficient is negative, we can conclude that \( F \) has a unique maximum. Furthermore, since both \( R \) and \( C \) are increasing functions of \( s \) in \([0, 1/2]\), the maximum value of \( F \) occurs at a value of \( s > 1/2 \). By setting \( dF/ds = 0 \), we obtain that the maximum value of \( F \) occurs at \( s_{\text{max}} = 1/2 + C_0/(2MC_1) \). This leads us to the following case analysis.

(a) \( C_0 < MC_1 \). In this case, \( 1/2 < s_{\text{max}} < 1 \). Since \( F \) is a second-degree polynomial, after it attains its maximum, it can only decrease. Hence, the smallest value of \( F \) in the interval \([1/N, 1]\) occurs either at value \( 1/N \) or at 1.

(b) \( C_0 \geq MC_1 \). In this case, \( s_{\text{max}} \geq 1 \). Hence, the smallest value of \( F \) in the interval \([1/N, 1]\) occurs at \( s = 1/N \).

We have reached the above conclusion by assuming that the exit rule evaluation was not partitioned. The analysis in the general case would be more complex and we leave it as a problem to be solved in the future.

7. BOUNDED REDUNDANCY

In Section 3, we presented a parallelization strategy that resulted in nonredundant parallel executions. From a performance point of view, however, nonredundancy could at times result in low processor utilization. Hence, it would be beneficial to develop a theory that will allow us to understand the trade-offs between nonredundancy and communication. A lower degree of communication would mean that the processors do not require significant amounts of data from other processors and hence are more loosely coupled. Thus, we might expect parallel executions with low communication to have a higher processor utilization. Clearly, high processor utilization is not the only determinant of a reasonable performance metric (e.g., response time). Trivial parallel executions, which simulate the sequential execution on each of the processors, have a 100% processor utilization. The problem with
this, however, is that there is no bound on the duplication of work done by the processors. This motivates the definition of parallel executions with bounded redundancy given below.

Definition 7.1. A parallel execution of a Datalog program is said to have a redundancy factor of \( p \) if the maximum number of times an identical instantiation may be repeated by different processors is bounded by \( p \).

If the redundancy factor of some parallel execution has redundancy factor \( p \), then we are assured that the sum total of the computational cost, which is proportional to the set of instantiations used, is no more than \( p \) times the computational cost of a sequential evaluation.

We now show an application of the parallelization strategy presented in Section 6 to obtain parallel executions which are bounded in terms of redundancy. Suppose that the number of processors in the set \( \mathcal{P} \) is \( N \). Let us choose a set of \( k \) hash functions, \( f_1, f_2, \ldots, f_k \). Define the hash function used by processor \( i \in \mathcal{P} \) to be as follows:

\[
h_i(x) = \begin{cases} f_i(x) & \text{if } \left( \prod_{j=1}^{k} \left( f_j(x) - i \right) \right) \neq 0 \\ i & \text{otherwise.} \end{cases}
\]

Clearly, a tuple may be processed in at most \( k \) sites, hence the redundancy factor of this scheme \( k \). Notice, that if \( k = 1 \), then there is only one hash function and the scheme reduces to the nonredundant scheme presented in Section 3. As noted above, the bounded redundancy is useful only if it is accompanied by the promise of less communication. Let us see if this is the case for the above scheme.

Let \( f_1, \ldots, f_k \) be a set of independent random hash functions such that the probability that a particular tuple hashes to a processor \( i \) is identical to \( s = 1/N \). Thus, \( \Pr[f_i(y) = j] = s \), independent of \( i \) and \( j \), where \( y \) is an arbitrary tuple. Hence,

\[
\Pr[h_i(y) \neq i] = \Pr[(f_1(y) \neq i) \land (f_2(y) \neq i) \land \cdots \land f_k(y) \neq i] = (1 - s)^k.
\]

Let \( C \) denote the total communication among all processors and let \( |t_i| \) denote the size of the recursive predicate \( t_i \) produced at processor \( i \). Then,

\[
C = \sum_{i=1}^{N} |t_i| \times \Pr[h_i(x) \neq i].
\]

By symmetry, \( C = N |t_1| \Pr[h_1(y) \neq i] \).

If the redundancy factor is \( p \), then \( N \times |t_i| \leq p \times |t_i| \). Hence, \( C \leq p \times N \times |t_i| / \Pr[h_i(y) \neq i] = p \times (1 - s)^k \). If \( k \ll N \), then \( (1 - s)^k \approx e^{-sk} \).

Therefore, we obtain the following relation between redundancy factor and communication.

\[
C \leq p \times e^{-sk} \quad \text{if } 1 \leq k \ll N.
\]

8. CONCLUSIONS

In this paper we have developed a theory for bottom-up, parallel evaluation of Datalog programs that is controlled by discriminating functions based upon hashing. Our results include the previous results presented by Wolfson, Silberschatz, Cohen [6, 18, 19] and Valduriez [16] as special cases.
We have observed that, for the class of programs considered, there is a spectrum of equivalent parallel executions and that a trade-off between nonredundancy and communication exists for these. Consequently, the particular scheme used in a compiler may be dependent on the underlying characteristics of the architecture; e.g., computation cost as opposed to communication cost. Our results in Section 5 further show how the rewriting method at compile time can be adapted to the architecture of the system.

The results in this paper are qualitative and obviously, are no substitute for detailed performance studies that would consider such issues as load balancing, processor utilization, etc. We intend to investigate these systematically in the future. The converse of the problem studied in Section 5 is whether the bottom-up execution of a given linear Datalog sirup can be parallelized on an arbitrary network of processors without any redundancy. We discuss the solution to this problem in [10].

APPENDIX A

In this appendix, we present the proof of Theorem 3.1, which states the equivalence between the rewritten program $Q$ and the given Datalog program $P$.

Let $P$ be a program defined with respect to a database $D$, and let $I$ be a set of intensional facts, that is, facts concerning intensional predicates only. We define, as in [17], the immediate consequence operator $T_p$ as follows:

$$T_p(I) = \{ A | A: - A_1, \ldots , A_k \}$$

is a ground instance of a clause in $P$ and every

$A_i$ is either a fact in $I$ or a fact in the database $D$.

We define the operator $T_p \uparrow n$ for $n \geq 1$, as follows:

$$T_p \uparrow 1 = T_p(\emptyset).$$

$$T_p \uparrow (n + 1) = T_p(T_p \uparrow n) \cup T_p \uparrow n.$$

$$T_p \uparrow \omega = \bigcup_{n \geq 1} T_p \uparrow n.$$

The least model for a Datalog program $P$ is denoted by $M_p$. The database is implicit in the notation. The following theorem is from [17], stating the equivalence of the operational and the declarative semantics.

Theorem A.1. $T_p \uparrow \omega = M_p$.

PROOF. Given in [17].

The extension of a predicate $q$ in any general interpretation $I$ is denoted $I(q)$. Thus the extension for a predicate $t$ appearing in the least model for $P$ is denoted $M_p(t)$, and in $T_p \uparrow n$ is denoted $T_p \uparrow n(t)$.

We will refer to a general Datalog program by $P$. The set of all the rewritten rules as presented in Section 3 will be represented as $Q$. Hence $Q = \bigcup_{i \in \mathcal{P}} Q_i$. We prove Theorem 3.1 by first establishing the following two lemmas.

Lemma A.1. For every recursive predicate symbol $t$, $M_p(t) \subseteq M_Q(t)$.

PROOF. Let $G$ be a ground recursive atom. We will first prove by induction that $\forall n \geq 1, \{ G \in T_p \uparrow n \Rightarrow \exists l \in \mathcal{P} G_{out} \in M_Q \}$. 

n = 1. Let $G \in T_p \uparrow 1$. Then there exists a rule $A: - B, \ldots, C$ and a ground substitution $\theta$ such that $A\theta = G$, and $B\theta, \ldots, C\theta$ are facts in the database. Let $h(\theta) = i$. Each of $B_i^\mu \theta, \ldots, C_i^\mu \theta$ are facts in the database. Hence $\theta$ is a ground substitution of the rule corresponding to the above rule in $Q_i$. Hence, $A_{out}^i \theta = G_{out}^i \in M_Q$.

$n = k + 1$. Let $G \in T_p \uparrow (k + 1)$. Then there exists a rule $A: - B, \ldots, C$ and a ground substitution $\theta$ such that

1. $A\theta = G$.
2. $(B\theta, \ldots, C\theta) \subseteq T_p \uparrow n \cup$ database.

Suppose $h(\theta) = i$. If $E$ is a base atom appearing in the body of the rule, then $E_i^\mu \theta \in M_Q$. So, let $E$ be a recursive atom appearing in the body of the rule. Hence $E\theta \in T_p \uparrow n$. By the induction hypothesis, $E_{out}^i \theta \in M_Q$ for some $j \in \mathcal{R}$. Using the sending rule of $Q_j$, $E_j^\mu \theta \in M_Q$, and by the receiving rule of $Q_i$, $E_j^\mu \theta \in M_Q$. Hence $A_{out}^i \theta = G_{out}^i \in M_Q$.

We have thus shown above that $\forall n \geq 1, \{G \in T_p \uparrow n \Rightarrow \exists i \in \mathcal{R}, G_{out}^i \in M_Q\}$. From the final pooling rule of $Q_i$, we see that if $G_{out}^i \in M_Q$ then $G \in M_Q$. □

**Lemma A.2.** Let $p$ be any derived predicate symbol of $Q$. Then for any recursive predicate $i$ in $P$, $M_Q(t) \subseteq M_P(t)$.

**Proof.** Any derived predicate symbol $p$ of $Q$ is of exactly one of the following types:

1. $p = t_i^i$, for some recursive predicate $t$ of $P$, and for some $i \in \mathcal{R}$.
2. $p = t_{out}^i$, for some recursive predicate $t$ of $P$, and for some $i \in \mathcal{R}$.
3. $p = t_{ij}$, for some recursive predicate $t$ of $P$, and for some $i$ and $j \in \mathcal{R}$.
4. $p = t$, for some recursive predicate $t$ of $P$.

We will prove the following (stronger) form of the lemma. For any recursive predicate $i$ of $P$, $M_Q(t_{in}^i) \subseteq M_P(t)$, $M_Q(t_{out}^i) \subseteq M_P(t)$, $M_Q(t_{ij}) \subseteq M_P(t)$ and $M_Q(t) \subseteq M_P(t)$. The proof is by induction on the height of the derivation.

$n = 1$. From the rules of $Q_i$, we see that $T_Q \uparrow 1(t_{in}^i) = \emptyset$, $T_Q \uparrow 1(t_{ij}) = \emptyset$ and $T_Q \uparrow 1(t_j^i) = \emptyset$, for every $i$ and $j \in \mathcal{R}$. Suppose that $T_Q \uparrow 1(t_{out}^i) \neq \emptyset$. Suppose $A_{out}^i \theta$ is a tuple in $T_Q \uparrow 1(t_{out}^i)$. Then there exists a ground substitution $\theta$ for some rule $A_{out}^i := B_{in}^i, \ldots, C_{in}^i, h(v) = i$ in $Q_i$. Thus $\theta$ is a successful substitution of the following rule in $P$: $A: - B, \ldots, C$. Hence, $A \in M_P$.

$n = k + 1$. Again let us consider the four possibilities separately.

$p = t_{in}^i$. Let $s$ be a tuple in $T_Q \uparrow (k + 1)(t_{in}^i)$. Then there exists a rule $D_{in}^i: - D_{in}^i$ in $Q_i$ and a ground substitution $\theta$ such that $D_{in}^i \theta = t_{in}^i(s)$. Hence $s \in T_Q \uparrow (k)(t_{in}^i).$ By the induction hypothesis, $s \in M_P(t)$.

$p = t_{ij}$. Let $s$ be a tuple in $T_Q \uparrow (k + 1)(t_{ij})$. Then there exists a rule $C_{ij}^i: - C_{ij}^i, h(v) = j$ and a ground substitution $\theta$ such that $C_{ij}^i \theta = t_{ij}^i(s)$ and $s \in T_Q \uparrow (k)(t_{out}^i)$. By the induction hypothesis, $s \in M_P(t)$.

$p = t$. Let $s \in T_Q \uparrow (k + 1)(t)$. By a similar argument as above, $s \in T_Q \uparrow (k)(t_{out}^i)$, for some $i \in \mathcal{R}$. By the induction hypothesis, $s \in M_P(t)$. 

Theorem 3.1 follows directly from Lemmas A.1 and A.2.

APPENDIX B

In this appendix we present a proof of Theorem 3.2 which asserts the nonredundancy of the parallelization scheme.

Let \( r \) be a rule \( A : - B, \ldots, C \) in \( P \). Then, \( \text{succ} - \text{substn}(r, P) \) denotes the following set:

\[
\{ \theta | \theta \text{ is a ground substitution for } r \text{ and } A \theta, B \theta, \ldots, C \theta \text{ are true facts in } M_Q \text{ or in the database} \}\.
\]

For any Datalog program \( P \), we define \( \text{succ} - \text{substn}(P) \) as follows:

\[
\text{succ} - \text{substn}(P) = \sum_{r \in P} \text{succ} - \text{substn}(r, P).
\]

Each program \( Q_i \) has a restricted version of \( r \) which we denote by \( r_i \). The statement of Theorem 3.2 may be mathematically rewritten as follows.

**Lemma B.1.** \( |\text{succ} - \text{substn}(P)| \geq \sum_{i \in \mathcal{P}, r \in P} |\text{succ} - \text{substn}(r_i, Q_i)| \).

**Proof.** Given any rule \( r \in L \), we observe the following:

1. If \( i \neq j \), then \( \text{succ} - \text{substn}(r_i, Q_i) \cap \text{succ} - \text{substn}(r_j, Q_j) = \emptyset \).
2. For any \( i \in \mathcal{P} \), \( \text{succ} - \text{substn}(r_i, Q_i) \subseteq \text{succ} - \text{substn}(r, P) \).

From (2), \( \bigcup_{i \in \mathcal{P}} \text{succ} - \text{substn}(r_i, Q_i) \subseteq \text{succ} - \text{substn}(r, P) \).
Hence, \( |\bigcup_{i \in \mathcal{P}} \text{succ} - \text{substn}(r_i, Q_i)| \leq |\text{succ} - \text{substn}(r, P)| \).
By (1), \( \sum_{i \in \mathcal{P}} |\text{succ} - \text{substn}(r_i, Q_i)| \leq |\text{succ} - \text{substn}(r, P)| \).
Since the above is true for any \( r \in L \), hence

\[
\sum_{i \in \mathcal{P}, r \in P} |\text{succ} - \text{substn}(r_i, Q_i)| \leq \sum_{r \in P} |\text{succ} - \text{substn}(r, P)| = |\text{succ} - \text{substn}(P)|.
\]

Theorem 3.2 asserts that the total number of times tuples are generated by the parallel scheme is no more than that generated by a sequential differential evaluation. It is a property of differential evaluation that the total number of times tuples are generated is identical to the number of successful substitution of rules in the program. Theorem 3.2 now follows from this above observation and Lemma B.1.

APPENDIX C

In this appendix we prove Theorem 5.1 which states that in the case that a given linear Datalog sirup has a cyclic dataflow graph, then one can choose the discrimi-
nating functions and the sequences in a manner such that no communication is required among the processors.

Suppose the dataflow graph has a cycle \( a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_k \rightarrow a_1 \), where the \( a_i \)'s are the argument positions of the recursive predicate \( t \) appearing in the body of the recursive rule of \( P \). Let us define the discriminating sequences \( \nu(r) \) and \( \nu(e) \) as follows:

\[
\nu(r) = \langle Y_{a_1}, \ldots, Y_{a_k} \rangle, \text{ where } Y_{a_i} \text{ is the variable appearing in argument position } a_i \text{ of } t(\bar{Y}) \text{ in the body of the recursive rule of } P.
\]

\[
\nu(e) = \langle Z_{a_1}, \ldots, Z_{a_k} \rangle, \text{ where } Z_{a_i} \text{ is the variable appearing in argument position } a_i \text{ of } t(\bar{Z}) \text{ in the head of the exit rule of } P.
\]

Suppose that we are given \( N \) processors numbered from 1 through \( N \). Let \( g \) be any function from the constants of the database and the program to the set \( \{1, 2, \ldots, N\} \). Define the discriminating functions \( h \) and \( h' \) as follows:

\[
h'(x_1, \ldots, x_k) = h(x_1, \ldots, x_k) = (g(x_1) + \cdots + g(x_k)) \mod N.
\]

**Lemma C.1.** Suppose a given linear sirup has a cycle in its dataflow graph \( a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_k \rightarrow a_1 \). Let \( h, h', \nu(r) \) and \( \nu(e) \) be defined as above. If a tuple \( (c_1, \ldots, c_n) \in M_Q(t_{\text{out}}) \), then \( h(c_1, \ldots, c_n) = i \).

**Proof.** Suppose \( (c_1, \ldots, c_n) \in M_Q(t_{\text{out}}) \). Then the above tuple could be derived either by the initialization step or the processing step of \( Q_i \). If the tuple was derived using the initialization step, then, \( h'(c_1, \ldots, c_n) = i \). If the tuple was derived by the rule in the processing step, then there exists a substitution \( \theta \) and a tuple \( (d_1, \ldots, d_n) \in M_Q(t_{\text{in}}) \) such that \( h(d_1, \ldots, d_n) = i \) and \( \bar{X} = (c_1, \ldots, c_n) \). By definition of the dataflow graph, an arc \( a_i \rightarrow a_j \) means that \( Y_{a_i} = X_{a_j} \). Hence, \( d_1 = c_1, d_2 = c_2, \ldots, d_{a_k-1} = c_{a_k} \) and \( d_{a_k} = c_{a_k} \). Hence,

\[
h(c_1, \ldots, c_n) = (g(c_1) + g(c_2) + \cdots + g(c_n)) \mod N
\]

\[
= (g(d_1) + g(d_2) + \cdots + g(d_{a_k})) \mod N
\]

\[
= h(d_1, \ldots, d_{a_k})
\]

\[
i.
\]

**Theorem C.1.** Let \( P \) be a linear Datalog sirup with a cycle in its dataflow graph. Let \( h, h', \nu(r) \) and \( \nu(e) \) be defined as above. Then, if \( i \neq j \), then \( M_Q(t_{ij}) = \emptyset \).

**Proof.** Consider the rule in \( Q_i \) that sends tuples from processor \( i \) to processor \( j \), namely,

\[
t_{ij}(\bar{Y}) := t_{\text{out}}^j(\bar{Y}), h(\nu(r)) = j.
\]

Suppose \( M_Q(t_{ij}) \neq \emptyset \). So, let \( (c_1, \ldots, c_n) \in M_Q(t_{ij}) \). Then, there exists a successful ground substitution \( \theta \) of the above rule such that \( \bar{Y} = (c_1, \ldots, c_n) \) and \( h(\nu(r)\theta) = j \). Also, \( h(\nu(r)\theta) = h(Y_{a_1} \theta, \ldots, Y_{a_k} \theta) = h(c_{a_1}, \ldots, c_{a_k}) = i \), by Lemma C.1. Hence \( i = j \).
APPENDIX D

In this section we present the proof of Theorem 6.1, which states the equivalence between the rewritten program \( R \) and the given linear Datalog sirup \( P \). The notation followed in this section is treated in Appendix A.

**Lemma D.1.** For every \( n \geq 1 \), \( T_p \uparrow n(t) \subseteq M_R(t) \).

**Proof.** From the receiving rule of \( R_i \), we see that \( M_R(t_{out}^i) \subseteq M_R(t) \). Hence \( \bigcup_{i \in \mathcal{R}} M_R(t_{out}^i) \subseteq M_R(t) \). Thus, in order to prove the lemma, it would be sufficient to show that for every \( n \geq 1 \), \( T_p \uparrow n(t) \subseteq \bigcup_{i \in \mathcal{R}} M_R(t_{out}^i) \). The proof is by induction on the height of the derivation \( n \).

\( n = 1 \). Let \( A \in T_p \uparrow 1(t) \). Then there exists a ground substitution \( \theta \) of \( P \) such that \( \theta \theta = A \). Let \( h'(\nu(e)\theta) = i \). Then \( \theta \) is a ground substitution for the initialization rule of \( R_i \). Hence, \( \theta \theta \in M_R(t_{out}^i) \) and \( \theta \theta \in \bigcup_{i \in \mathcal{R}} M_R(t_{out}^i) \).

\( n = k + 1 \). Let \( A \in T_p \uparrow (k + 1)(t) \). Then there exists a substitution \( \theta \) of the recursive rule of \( P \), such that \( \theta \theta = A \). Also, \( \theta \theta \in M_R(t_{out}^i) \) and by the induction hypothesis, \( \theta \theta \in M_R(t_{in}^j) \). Therefore, \( A = \theta \theta \in M_R(t_{out}^i) \subseteq \bigcup_{i \in \mathcal{R}} M_R(t_{out}^i) \).

**Lemma D.2.** \( M_R(t) \supseteq M_P(t) \).

**Proof.** \( M_P(t) = \bigcup_{n \geq 1} T_p \uparrow n(t) \subseteq M_R(t) \), by Lemma D.1.

**Lemma D.3.** Let \( p \) be any derived predicate symbol of \( R \) (i.e., \( p \) is either \( t \) or any of the \( t_{in}^i \) or any of the \( t_{out}^i \) or \( t_{ij} \)). Then for any \( n \geq 1 \), \( T_R \uparrow n(p) \subseteq M_R(t) \).

**Proof.** The proof is by induction on the height of derivation \( n \).

\( n = 1 \). From the rules in \( R_i \), we see that \( T_R \uparrow 1(t_{in}^i) = \emptyset \), \( T_R \uparrow 1(t_{ij}) = \emptyset \) and \( T_R \uparrow 1(t) = \emptyset \), for every \( i \) and \( j \in \mathcal{R} \). Thus, if \( p = t_{in}^i \) or \( t_{ij} \) or \( t \), \( T_R \uparrow 1(p) = \emptyset \subseteq M_R(t) \).

Now suppose that \( p = t_{out}^i \), the only other derived predicate in \( R \). Let \( A \in T_R \uparrow 1(t_{out}^i) \). Then there exists a ground substitution \( \theta \) for the initialization rule of \( R_i \) such that \( A = \theta \theta \). Then \( \theta \theta \in M_R(t_{out}^i) \). Hence, \( A = \theta \theta \in M_R(t_{out}^i) \).

\( n = k + 1 \). There are four possibilities for \( p \), namely \( p = t_{in}^i \), \( p = t_{our} \), \( p = t_{ij} \) or \( p = t \). We consider each of the cases separately below.

\( p = t_{in}^i \). We must show that \( T_R \uparrow (k + 1)(t_{in}^i) \subseteq M_R(t) \). Let \( A \in T_R \uparrow (k + 1)(t_{in}^i) \). Then there exists a substitution \( \theta \) for the receiving rule in \( R_i \) such that \( \theta \theta = A_{ij} \), and \( \theta \theta \in M_R(t_{in}^j) \), for some \( j \). Then, by the induction hypothesis, \( \theta \theta \in M_R(t_{in}^j) \) and hence, \( A = \theta \theta \in M_R(t_{in}^j) \).

\( p = t_{out}^i \). Let \( A \in T_R \uparrow (k + 1)(t_{out}^i) \). Then there exists a substitution \( \theta \) for the processing rule in \( R_i \) such that \( \theta \theta = A \). Also, \( \theta \theta \in M_R(t_{out}^i) \). By the induction hypothesis, \( \theta \theta \in M_R(t_{out}^i) \). Hence, \( A = \theta \theta \in M_R(t_{out}^i) \).
Let $A \in T_R \uparrow (k + 1)(t_{ij})$. Then there exists a ground substitution $\theta$ for the sending rule in $R_i$ such that $A = \overline{Y}\theta$, and $\overline{Y}\theta \in T_R \uparrow k(t_{out}')$. By the induction hypothesis, $\overline{Y}\theta \in M_p(t_{out}')$.

Let $A \in T_R \uparrow (k + 1)(t_i)$. Then there exists an $i \in \mathcal{I}$ and a substitution $\theta$ for the final pooling rule of $R_i$ such that $a = \overline{W}\theta$, and $\overline{W}\theta \in T_R \uparrow k(t_{out}')$. By the induction hypothesis, $A = \overline{W}\theta \in M_p(t_{out}')$.

Lemma D.4. $M_R(t) \subseteq M_p(t)$.

**Proof.** $M_R(t) = \bigcup_{n \geq 1} T_R \uparrow n(t) \subseteq M_p(t)$, by Lemma D.3. □

Theorem 6.1 follows from Lemmas D.2 and D.4.

**REFERENCES**
