A Framework for the Parallel Processing of Datalog Queries

Sumit Ganguly
Avi Silberschatz*
Shalom Tsur†

Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

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 non-redundant 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 non-redundant computation, is also studied The methods reported in this paper clearly demonstrate the trade-offs between redundancy and interprocessor-communication for this class of problems

1 Introduction

The efficient bottom-up evaluation of queries in a deductive database, defined by Datalog programs, is presently an active area of research([14, 4]) 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 [19, 18, 6, 8].

The problem of characterizing Datalog programs that belong to the NC complexity class has been theoretically investigated by Kanellakis, Van Gelder, Ullman, and others [15, 1, 11] A program is in NC, if it can be evaluated in polylogarithmic time given a polynomial number of processors This, however is not very useful for the type of database processing that we are concerned with for the following two reasons

- 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 non shared-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 Datalog queries

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

1 The strategies presented in [19, 18, 6] 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 methods allows for evaluations over partitioned base relations in many cases 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

*This material is based in part upon work supported by NSF Grant IRI-8805215

†Microelectronics and Computer Technology Corporation, Austin, Texas 78759

3 Our method of mapping the Datalog programs to processors results in non-redundant computations in the sense that the same firing 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 non-redundancy and communication, and is similar in spirit to the results presented in [13]

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 non-redundant parallelization scheme for linear sirups 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 for linear sirups and shows that a trade-off exists between non-redundancy and communication In Section 7 we present a general non-redundant scheme applicable to all Datalog programs We conclude in Section 8 and suggest extensions to this work

2 Preliminaries and Notation

A *Datalog* program is a finite set of rules 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 $Q \leftarrow Q_1, \ldots, Q_k$ 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 A p-atom is an atom having p as the predicate symbol

A substitution θ 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 (constant or variable) distinct from v_i and the variables $v_1, \ldots, v_n$ are distinct θ is called a *ground* substitution if the t_i are all constants

A *Datalog* program is a finite set of rules whose predicate symbols 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

$$\begin{aligned} anc(X,Y) &- par(X,Y) \\ anc(X,Y) &- par(X,Z), anc(Z,Y) \end{aligned}$$

The relation *par* above is an extensional relation, where $par(X,Y)$ means that X is a parent of Y The relation *anc* above is a derived relation, where $anc(X,Y)$ 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 and 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 [2] 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 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]

In sections 3 through 6, we restrict our attention to *linear sirups* which are Datalog programs with one linear recursive rule r and one non-recursive (exit) rule e Each such program may be canonically represented as

$$\begin{aligned} e \quad t(\bar{Z}) &- s(\bar{Z}) \\ r \quad t(\bar{X}) &- t(\bar{Y}), b_1, \ldots, b_k \end{aligned}$$

where

- t is the output (or derived) predicate symbol
- s is a base relation
- $\bar{Z}$ is the sequence of variables appearing in the head of the exit rule
- $\bar{X}$ is the sequence of variables appearing in the head of the recursive rule
- $\bar{Y}$ 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 *semi-naive evaluation* [4, 14]

3 No redundancy

The basic step in the semi-naive 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 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 now formally describe our parallelization scheme

Let L be a linear sirup with a recursive rule r and an exit rule e Let $v(r)$ be any *sequence* of variables, all of which appear in the recursive rule r This sequence is referred to as the *discriminating sequence* for the recursive rule Similarly, let $v(e)$ be any sequence of variables, all of which appear in the exit rule e This sequence is called the *discriminating sequence* of variables for the exit rule Finally, let P be a finite set of processors, (e g , $\{1, 2, \quad, n\}$) on which the program is to be executed, and let h and h' be two functions defined as follows

h set of ground instances of $v(r) \rightarrow P$

h' set of ground instances of $v(e) \rightarrow P$

These two functions are referred to as the *discriminating* functions

Given a linear sirup L, 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 L Let Q_i denote the program to be executed at processor i It consists of the following five execution steps

1 **Initialization.** A new predicate t^i_{out} is defined whose interpretation is the set of all the t-tuples generated at processor i

$$t^i_{out}(\bar{Z}) \quad - s(\bar{Z}), h'(v(e)) = i$$

2 **Processing.** A new predicate t^i_{in} is defined whose interpretation is the set of all t-tuples that are input to processor i at some point in the execution b^i_m is the fragment of the base relation b_m that is accessed at processor i Its computation is given later in this section

$$t^i_{out}(\bar{X}) \quad - t^i_{in}(\bar{Y}), b^i_1, \quad, b^i_k, h(v(r)) = i$$

3 **Sending.** For every i and $j \in P$, the predicate symbol t_{ij} represents the set of tuples transmitted from processor i to processor j For every $j \in P$, we introduce the following rule in Q_i

$$t_{ij}(\bar{Y}) \quad - t^i_{out}(\bar{Y}), h(v(r)) = j$$

4 **Receiving.** For every i and $j \in P$, the predicate t_{ji} represents the set of all the tuples transmitted from processor j to processor i $\bar{W}$ is a sequence of new distinct variables not appearing in the original program For every $j \in P$, we introduce the following rule in Q_i

$$t^i_{in}(\bar{W}) \quad - t_{ji}(\bar{W})$$

5 **Final pooling.** This rule states that all the tuples generated by all the processors are stored in a single relation t as the answer to the query $\bar{W}$ is the same sequence of variables as in the receiving step

$$t(\bar{W}) \quad - t^i_{out}(\bar{W})$$

The program Q_i consists of several recursively defined predicates, namely, t, t^i_{out}, t_{ji} and t^i_{in} The theorem below asserts the relationship between the predicate t defined in the given program L and one of the predicates of Q_is, namely, t

Theorem 1: Let $Q = \cup_{i \in P} Q_i$ as defined above obtained by rewriting a given linear sirup L with recursive predicate t Then for every input of base relations, the interpretation of the predicate t in the least model of Q is identical to the interpretation of t in the least model of L □

We first describe the abstract architecture on which the parallel program is executed Given a set P of processors, we assume that a processor i in P may communicate with every other processor j in 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 this data without error within some finite time This abstraction is easily implementable by either shared memory or message passing

The parallel execution proceeds with each processor evaluating the Datalog program Q_i using a semi-naive evaluation The relations t^i_{out} and t^i_{in} are local to processor i, for each $i \in P$ The predicates t_{ij}, for $i, j \in P$, represent the channel ij in the abstract architecture described above Hence, addition of tuples to the predicate t_{ij}, during the semi-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 **Initialization.** The following rule is evaluated according to a semi-naive evaluation

$$t^i_{out}(Z) \quad -s(Z), h'(v(e)) = i.$$

2 **Processing.** The following rule is evaluated using a semi-naive evaluation scheme

$$t^i_{out}(X) \quad -t^i_{in}(Y), b^i_1, \quad ., b^i_k, h(v(r)) = i.$$

If the variables appearing in $v(r)$ do not appear in b_k, then $b^i_k = b_k$ Otherwise, b^i_k is defined as follows

$$b^i_k \quad - b_k, h(v(r)) = i$$

3 **Sending.** Once tuples are generated at some iteration by processor i, they must be sent to different processors The following rule

$$t_{ij}(Y) \quad -t^i_{out}(Y), h(v(r)) = j$$

sends only those subset 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

4 **Receiving.** In the receive 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

5 **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

6 **Parallel Termination.** The parallel algorithm terminates when every processor in P is idle and all channels are empty This may be detected by standard algorithms of Distributed Computing as given in [5, 7]

So far, we have described the parallel execution of the parallel program, and proved its correctness However, in order to be effectively parallel, we must restrict the choice of the discriminating sequences

The rule in the processing step of Q_i would be evaluated as the following relational algebra expression, $\prod(\sigma_{h(v(r))=i}(t^i_{in} \bowtie b^i_1 \quad \bowtie b^i_k))$ 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 In that case each processor computes the entire join expression, thus repeating the computation done by a sequential processor and defeating the purpose of parallelism 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

The following definition precisely defines the notion of non-redundancy

Definition 1: A parallelization scheme is called *semi-naive non-redundant*, 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 semi-naive evaluation of the same program on the same data □

Theorem 2: The parallelization scheme described above is semi-naive non-redundant □

4 Examples

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

$$\begin{array}{lll} anc(X,Y) & - & par(X,Y) \\ anc(X,Y) & - & par(X,Z), anc(Z,Y) \end{array}$$

The relation *par* above is a base relation, where, $par(X,Y)$ means that X is the parent of Y

We assume that there are N processors, numbered from 1 through N Thus $P = \{1, 2, \quad ., N\}$ 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 *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 *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

4.1 Example 1

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

- Initialization

$$anc^i_{out}(X,Y) \quad - par(X,Y), h(Y) = i$$

- Processing
  $anc^i_{out}(X,Y) \quad - par^i(X,Z), anc(Z,Y), h(Y) = i$
- Sending For every j, $1 \leq j \leq N$,
  $anc_{i,j}(Z,Y) \quad - anc^i_{out}(Z,Y), h(Y) = j$
- Receiving For every j, $1 \leq j \leq N$,
  $anc^i_{in}(W_1,W_2) \quad - anc_{j,i}(W_1,W_2)$
- Final Pooling
  $anc(W_1,W_2) \quad - anc^i_{out}(W_1,W_2)$

Since $v(r) = \langle Y \rangle$, 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, $anc_{i,j}(Z,Y) \quad - anc^i_{out}(Z,Y), h(Y) = j$) does not yield any tuple That is, $anc_{i,j} = \phi$, 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

4.2 Example 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 $i \neq j$, $par^i \cap par^j = \phi$, and $\cup_{i=1}^{n} par^i = par$

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

$h(a,b) = i$ if and only if (a,b) is a tuple in par^i

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

- Initialization
  $anc^i_{out}(X,Y) \quad \cdot - par^i(X,Y)$
- Processing
  $anc^i_{out}(X,Y) \quad - par^i(X,Z), anc^i_{in}(Z,Y)$
- Sending For every j, $1 \leq j \leq N$,
  $anc_{i,j}(Z,Y) \quad - anc^i_{out}(Z,Y), h(X,Z) = j$
- Receiving For every j,$1 \leq j \leq N$,
  $anc^i_{in}(W_1,W_2) \quad - anc_{j,i}(W_1,W_2)$
- Final Pooling
  $anc(W_1,W_2) \quad - anc^i_{out}(W_1,W_2)$

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, $anc_{i,j}(Z,Y) \quad - anc^i_{out}(Z,Y), h(X,Z) = j$ Equivalently, this may be rewritten as follows

$anc_{i,j} = \{(a,b) | (a,b) \in anc^i_{out} \wedge \exists c (c,a) \in par^i\}$

Thus, $anc_{i,j} \subseteq anc^i_{out}$ 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

4.3 Example 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
  $anc^i_{out}(X,Y) \quad - par(X,Y), h(X) = i$
- Processing
  $anc^i_{out}(X,Y) \quad - par(X,Z), anc^i_{in}(Z,Y), h(Z) = i$
- Sending For every j, $1 \leq j \leq N$,
  $anc_{i,j}(Z,Y) \quad - anc^i_{out}(Z,Y), h(Z) = j$
- Receiving For every j, $1 \leq j \leq N$,
  $anc^i_{in}(W_1,W_2) \quad - anc_{j,i}(W_1,W_2)$
- Final Pooling
  $anc(W_1,W_2) \quad - anc^i_{out}(W_1,W_2)$

We note the following properties of Q_i

1 Let (a,b) be a tuple in anc^i_{out} Then, according to the sending rule, a 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 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 $par(X,Z)$ such that $h(Z) = i$. Hence the accesses to the par relation by different processors do not overlap, and thus there is no contention during the recursive processing

The extent of communication is less here, as compared to Example 2 However, all possible horizontal fragmentations of par is allowed in Example 2, but not all fragmentations are allowed here In Example 1, the relation par was replicated/shared among all the processors, whereas, in this case, each of the processors accesses a disjoint fragment of the par relation However, the algorithm here involves communication, whereas in Example 1, there is no communication between the processors Thus, this example essentially depicts a tradeoff between fragmentation and communication

5 Network Connectivity

In Section 3, we presented a general strategy for the parallel execution of linear Datalog sirups on a set of processors The abstract architecture assumed that every processor could communicate with every other processor In this section, we 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 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 2: Consider a linear recursive rule with the head $t(X_1, X_2, \quad, X_m)$ and the recursive atom in the body $t(Y_1, Y_2, \quad, Y_m)$ A *dataflow graph* for this rule is a directed graph $G = (V, E)$ where

- $V \subseteq \{1, 2, \quad, m\}$ and $i \in V$ if $\exists j \in \{1, 2, \quad, m\}$ such that $Y_i = X_j$
- An edge $i \rightarrow j$ exists in the graph if $Y_i = X_j$ □

Example 4: Consider the following recursive rule

$$p(U, V, W) \quad -p(V, W, Z), q(U, Z)$$

The dataflow graph for this recursive rule is presented in Figure 1

$$1 \rightarrow 2 \rightarrow 3$$

Figure 1

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 3: Consider a set of processes P 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 P, does not require any communication □

Example 5: Consider the ancestor example presented in the earlier section The dataflow graph for it is presented in Figure 2 Hence, as shown in Section 4, there is no requirement for communication between the processors when the discriminating variable is Z □

Figure 2

Unfortunately, it is not always the case that a dataflow graph contains a cycle, as shown in Example 4 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 define the notion of a network graph

Definition 3: Given a set of processors P, we define a *network graph* over P as a directed graph $N = (V, E)$ where $V = P$ and E is any subset of $P \times P$ □

A directed edge $i \rightarrow j$ in N means 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 6: Consider the following program

$$\begin{array}{lll} p(X, Y) & - & p(Y, Z), r(X, Z) \\ p(X, Y) & - & q(X, Y) \end{array}$$

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

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 $P = \{(00), (01), (10), (11)\}$ Below, we consider some of the rules of the rewritten program executed at processor (00)

- Initialization
  $p_{out}^{(00)}(X, Y) \ \cdot - \ q(X, Y), h(X, Y) = (00)$
- Processing
  $p_{out}^{(00)}(X, Y) \ : - \ p_{in}^{(00)}(Y, Z), r(X, Z), h(Y, Z) = (00)$
- Sending
  $p_{(00)(ij)}(Y, Z) \ . - \ p_{out}^{00}(Y, Z), h(Y, Z) = (ij)$

We see from the processing and the initialization rules that if $(a, b) \in p^{(00)}$, then $g(b) = 0$ Consider the rule that represents the operation of sending tuples from processor (00) to processor (01) Then, if $(a, b) \in p_{(00)(01)}$, then by the sending rule, $(a, b) \in p_{out}^{(00)}$, 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 $(a, b) \in p_{out}^{(00)}$, 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 other processor, yields the network graph shown in Figure 3 □

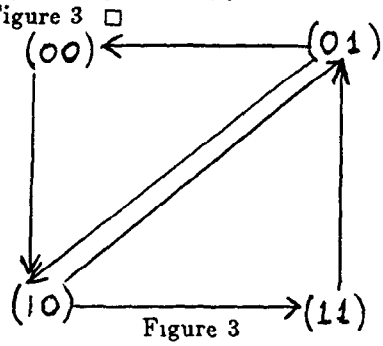

Figure 3

Given a linear sirup L, a sequence of discriminating variables, and discriminating functions satisfying some conditions, there is an algorithm to generate the minimal network graph N to evaluate L 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 L on this database, results in communication along that edge This algorithm and its proof of correctness is described in [9] Here we show an example to illustrate our ideas It further shows that if the discriminating 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 7: Consider the Datalog program

$$\begin{aligned} p(U,V,W) &- s(U,V,W) \\ p(U,V,W) &- p(V,W,Z), q(U,Z) \end{aligned}$$

The dataflow graph for this program as explained in Example 4 is $1 \rightarrow 2 \rightarrow 3$ Let $v(r)$ be $\langle V,W,Z\rangle$, and $v(e)$ be $\langle U,V,W\rangle$ Let g be an arbitrary function from the constants of the database to the set $\{0,1\}$ Define the discriminating functions h and h' to be the following linear function

$$h(a_1,a_2,a_3) = h'(a_1,a_2,a_3) = g(a_1) - g(a_2) + g(a_3)$$

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

$$h(a_1,a_2,a_3) = b_1 - b_2 + b_3 = j \quad (1)$$

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,

$$h'(a_1,a_2,a_3) = h(a_1,a_2,a_3) = b_1 - b_2 + b_3 = i \quad (2)$$

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 to produce $p(a_1,a_2,a_3)$ Let $g(a_4) = b_4$ Hence,

$$b_2 - b_3 + b_4 = i \quad (3)$$

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

$$\begin{aligned} x_1 - x_2 + x_3 &= v \quad (4) \\ x_2 - x_3 + x_4 &= u \quad (5) \end{aligned}$$

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 $(x_1, x_2, x_3, x_4, u, v)$ 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 the Figure 4

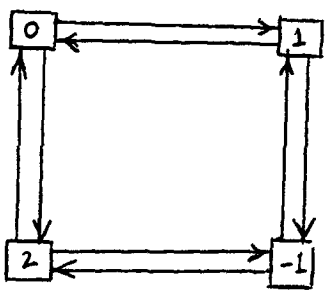

Figure 4

6 Trade-Off

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]

Let L 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^i is defined whose interpretation is the set of all t-tuples that are processed at processor i at some point in the execution

$$t^i(\bar{Z}) \quad -s(\bar{Z}), h'(v(e)) = i$$

2 Recursive Processing

$$t^i(\bar{X}) \quad -t^i(\bar{Y}), b_1, \quad , b_k$$

3 Final Pooling.

$$t(\bar{W}) \quad -t^i(\bar{X})$$

This program scheme and its proof of correctness was 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 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 non-redundancy and communication This general scheme is presented below

The definitions of $v(e)$, $\mathcal{P}$ and h' are the same as in Section 3 We require that every variable in $v(r)$ also appear in $\bar{Y}$ Also, for every processor i in $\mathcal{P}$, we define a discriminating function h_i as follows

h_i set of ground instances of $v(r) \rightarrow \mathcal{P}$

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^i_{in}, t^i_{out} etc are the same as in Section 3 Therefore, we do not repeat the explanations here

1 Initialization.

$$t^i_{out}(\bar{Z}) \quad -s(\bar{Z}), h'(v(e)) = i$$

2 Processing.

$$t^i_{out}(\bar{X}) \quad -t^i_{in}(\bar{Y}), b_1, \quad , b_k$$

3 Sending. For every $j \in \mathcal{P}$, we introduce the following rule in R_i

$$t_{ij}(\bar{Y}) \quad -t^i_{out}(\bar{Y}), h_i(v(r)) = j$$

4 Receiving. For every $j \in \mathcal{P}$, we introduce the following rule in R_i

$$t^i_{in}(\bar{W}) \quad -t_{ji}(\bar{W})$$

5 Final Pooling.

$$t(\bar{W}) \quad -t^i_{out}(\bar{W})$$

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 R_i and the program Q_i defined in Section 3 is that the discriminating functions h_i used by the processors may be different from one another In 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_is 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 The rewritten program R_i consists of several recursively derived predicates, $t^i_{out}, t^i_{in}, t_{ij}$ and t The theorem below asserts the relationship between the predicate t in the R_is and the predicate t given in the original linear sirup L

Theorem 4: Let $R = \cup_{i \in \mathcal{P}} R_i$, where the R_is are obtained from a given linear sirup L as defined above 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 L □

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

1 Let $h_i(a_1, a_2, \quad , a_m) = i$ for every tuple $(a_1, \quad , a_m)$ 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

$$t^i(\bar{Z}) \quad - \quad s(\bar{Z}), h'(v(e)) = i$$

$$t^i_{out}(\bar{X}) \quad - \quad t^i_{in}(\bar{Y}), b_1, \quad , b_k, h(v(r)) = i$$

$$t_{ij}(\bar{Y}) \quad - \quad t^i_{out}(\bar{Y}), h(v(r)) = j$$

$$t^i_{in}(\bar{W}) \quad - \quad t_{ji}(\bar{W})$$

$$t(\bar{W}) \quad - \quad t^i_{out}(\bar{W})$$

Recall that for this section, we have restricted that all variables in $v(r)$ must also appear in $\bar{Y}$ Hence $t^i_{in}(\bar{Y}) \Rightarrow (h(v(r)) = i)$ Therefore, this program is identical to the program Q_i presented in section 3 Thus if each processor uses the same discriminating function for the recursive rule, then the parallel computation is non-redundant

Different choices of discriminating functions h_i and h' may result in different execution schemes In general, the parallel execution is not completely non-redundant, and may require communication We see that in non-redundant execution (i e, when $h_i = h$ for all i), 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 the same processor where they were generated, then non-redundancy may be lost, at the advantage of less communication In general, more communication would lead to lesser redundancy, and vice-versa The above parallelization scheme vividly demonstrates the trade-off between communication and redundancy

By varying the extent of communication in a substitution based parallel execution, we get executions which are points along a spectrum whose extremes are characterized by non-redundancy and no communication

7 A General Scheme

In this section we extend the parallelization scheme presented in Section 3 to include all Datalog programs (that is, non-linear programs, and programs with more than one recursive rule)

Let M be a Datalog program whose rules are numbered from 1 to n, in some order For each rule r_i in M, we choose a discriminating sequence $v(r_i)$ and a discriminating function h_i The discriminating function is defined as follows

h_i set of ground instances of $v(r_i) \rightarrow P$

The meaning of the predicate symbols $t^i_{in}, t_{ij}, t^i_{out}$ etc , and of P are the same as in Section 3 If A is any atom with predicate symbol t, then A^i_{in} is the atom with the predicate symbol t^i_{in} and the same arguments Thus, for example, if A denotes the atom $sg(U, V)$, then A^i_{in} denotes the atom $sg^i_{in}(U, V)$ Likewise, we define A^i_{out} and A_{ij}

The program to be executed at processor i consists of the following four steps, and is denoted by T_i

1 **Processing.** Let $A \quad -B, \quad , C$ be a rule r in M, with discriminating sequence $v(r)$ and discriminating function h Then, include the following rule in T_i

$$A^i_{out} \quad -B^i_{in}, \quad , C^i_{in}, h(v(r)) = i$$

2 **Sending.** Let r be a rule in M, with discriminating sequence $v(r)$ and discriminating function h For every recursive atom C appearing in r and every $j \in P$, include the following rule in T_i

$$C_{ij} \quad -C^i_{out}, h(v(r)) = j$$

3 **Receiving.** Let W be a sequence of all distinct variables not appearing in the original program For every recursive predicate t appearing in the program M and every $j \in P$, introduce the following rule in T_i, where S denotes $t(W)$

$$S^i_{in} \quad -S_{ji}$$

4 **Final Pooling.** Let S be as defined in the receiving step above For every recursive predicate t, include the following rule in T_i

$$S \quad -S^i_{out}$$

Example 8 : Let M be the following non-linear program to compute the ancestor relation anc of a given parent relation par

$$\begin{array}{lll} r_1 & anc(X,Y) & - \quad par(X,Y) \\ r_2 & anc(X,Y) & - \quad anc(X,Z), anc(Z,Y) \end{array}$$

Suppose $v(r_1) = \langle Y \rangle$, and $v(r_2) = \langle Z \rangle$, and $h_1 = h_2 = h$, where h is some arbitrary discriminating function The four execution steps of the program T_i are

1 **Processing.**

$anc^i_{out}(X,Y) \quad -par(X,Y), h(Y) = i$
$anc^i_{out}(X,Y) \; . \; -anc^i_{in}(X,Z), anc^i_{in}(Z,Y), h(Z) = i$

2 **Sending.**

$anc_{ij}(X,Z) \quad -anc^i_{out}(X,Z), h(Z) = j$
$anc_{ij}(Z,Y) \quad -anc^i_{out}(Z,Y), h(Z) = j$

3 **Receiving.**

$anc^i_{in}(W_1,W_2) \quad -anc_{ji}(W_1,W_2)$

4 **Final Pooling.**

$anc(W_1,W_2) \quad -anc^i_{out}(W_1,W_2)$ □

The following theorem asserts the correctness of the transformation

Theorem 5 : Let $T = \cup_{i \in P} T_i$ For every input of the base relations, the interpretation of every derived predicate symbol in the least model of M is identical to the interpretation of the same predicate symbol in the least model of T □

The base relations are distributed among the processors in the following manner Suppose r is a rule with discriminating sequence $v(r)$ and D is a base atom appearing in r If the variables appearing in $v(r)$ do not appear in D, then D is shared/replicated among the processors Otherwise, the fragment of D accessed by processor i is denoted by D^i_{in} and is defined by

$$D^i_{in} \quad - D, h(v(r)) = i$$

As argued in Section 3, all variables appearing in a discriminating sequence of a rule r must also appear in atleast one atom in the body of r The parallelization scheme presented above is non-redundant in the precise sense described below

Definition 4: Let M be a Datalog program, and I be the input database to M Then, we say that a substitution θ is a *successful ground substitution* for a rule $r \quad A \quad -B, \quad , C$ in M, if

1 θ instantiates all the variables occurring in rule r by constants and doesn't instantiate any other variable

2 Each of the atoms in the set $\{A\theta, B\theta, \ , C\theta\}$ is either a fact in the database I or is a fact in the output (i e , the least model)

Theorem 6 . Let M be a given Datalog program Let T be the Datalog program obtained by rewriting M using any choice of discriminating sequences and discriminating functions Then, given any input database I, the number of distinct successful ground substitutions of rules in M is atleast equal to the number of distinct successful ground substitutions of the *processing rules* in T □

8 Conclusions

In this paper we have built on previously reported work and extended the results Our results include the results by Wolfson, Silberschatz and Cohen [19, 18, 6] 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 tradeoff between non-redundancy 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 results presented in this paper and others, that we intend to discuss in future work, form the beginning of a theory for bottom-up, parallel evaluation that is controlled by discriminating functions based upon hashing

References

[1] Afrati F and Papadimitrou C H "Parallel Complexity of Simple Chain Queries", In *Proceedings of the 6th ACM Symposium on Principles of Database Systems,* 1987

[2] Apt K R *Introduction to Logic Programming* Technical Report TR-87-35, Department of Computer Sciences, The University of Texas at Austin, 1988

[3] Bancilhon F "Naive Evaluation of Recursively Defined Relations", MCC Technical Report Number DB-004-85

[4] Bancilhon F and Ramakrishnan R "An Amateur's Introduction to Recursive Query Processing Strategies", In *Proceedings of the 1986 ACM SIGMOD International Conference on the Management of Data*

[5] Chandy K M and J Misra "An Example of Stepwise Refinement of Distributed Programs Quiesence Detection", *ACM TOPLAS,* July 1986

[6] Cohen S and O Wolfson " Why A Single Parallelization Strategy is not enough in Knowledge Bases", In *Proceedings of the 8th ACM Symposium on Principles of Database Systems,* March 1989

[7] Dijkstra E W and C S Scholten "Termination Detection for Diffusing Computations", *Information Processing Letters,* August 1980

[8] Dong G "On Distributed Processibility of Datalog Queries by Decomposing Databases ", In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*

[9] Ganguly S , Silberschatz A and S Tsur " Deriving Networks for the Parallel Evaluation of Datalog Queries", Technical Report, *University of Texas at Austin,* in preparation

[10] Houtsma M A W et al " A Logic Query Language and its Algebraic Optimization for a Multiprocessor Database Machine", Technical Report INF-88-52, *University of Twenete,* December 1988

[11] Kanellakis P "Parallel Complexity of Logic Programs", In *Foundations of Logic Programming and Deductive Databases,* Morgan-Kauffmann 1988

[12] Lloyd J W *Foundations of Logic Programming* Springer-Verlag, Second edition, 1987

[13] Papadimitrou C H and J Ullman "A Communication-Time Trade-Off", *SIAM Journal of Computing,* Vol 16, No 14, 1987

[14] Ullman J *Principles of Database and Knowledge Base Systems* Computer Science Press, 1989

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

[16] Valduriez P and S Khoshafian "Parallel evaluation of the transitive closure of a database relation", In *International Journal of Parallel Programming,* March 1989

[17] van Emden, M H and R A Kowalski "The Semantics of Predicate Logic as a Programming Language", *Journal of the ACM,* October 1976

[18] Wolfson O "Sharing the load of Logic Program Evaluation", In *Proceedings of the 1988 International Symposium on Databases in Parallel and Distributed Systems,* December 1988

[19] Wolfson O and A Silberschatz "Distributed processing of logic programs", In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data,* June 1988