A FRAMEWORK FOR THE PARALLEL PROCESSING OF DATALOG QUERIES

Sumit Ganguly, Avi Silberschatz, and Shalom Tsur

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-05 March 1989

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 a class of Datalog queries. Each of these methods consists of three steps:

1. A *rewrite step* that renders an equivalent program to the original one, explicitly amenable to parallel execution.
2. An *assignment step* that assigns the rules and data of the rewritten program to processors.
3. An *execution step* that performs the computation, either with or without processor intercommunication.

We introduce the notion of a *discriminating predicate* by which the computation is partitioned among the processors and parallelism is acheived. The methods demonstrate the trade-offs between redundancy (duplication of computation by processors) and interprocessor-communication.

*Also visiting the Microelectronics and Computer Technology Corporation, Austin, Texas
†This material is based in part upon work supported by NSF Grant IRI-8805215
‡Microelectronics and Computer Technology Corporation, Austin, Texas 78759

1 Introduction

This paper is concerned with the parallel evaluation of linear Datalog programs consisting of one non-recursive rule (also called exit rule) and one linear recursive rule. These programs have one derived predicate (also called intensional predicate or output predicate). The canonical Datalog program that we study in this paper is of the form :

$$\begin{array}{lcl} T(\bar{z}\ ) & \leftarrow & R(\bar{z}\ ). \\ T(\bar{x}\ ) & \leftarrow & B(\bar{u}\ ), T(\bar{y}\ ). \end{array}$$

where:

- T is the output predicate symbol.
- R is an input predicate symbol.
- $\bar{x}$ 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.
- $\bar{z}$ is the sequence of variables appearing in the head of the exit rule.
- $B(\bar{u}\ )$ is the conjunction of all the extensional atoms appearing in the body of the recursive rule. $\bar{u}$ is the set of all the variables appearing in the extensional atoms of 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. In terms of $\bar{u}$, $\bar{x}$ and $\bar{y}$ this means that every variable appearing in $\bar{x}$ appears in at least one of $\bar{u}$ or $\bar{y}$.

We present three *parallelization schemes* in this paper, each of which take a Datalog program of the kind stated above, and produces a parallel execution scheme for the Datalog program. In the sequel, we show how these parallelization techniques can be implemented on two processors. However, the parallelization schemes are completely scalable, and may be used to obtain parallel executions on any number of processors. The parallelization techniques proceed in three stages:

1. A *rewrite* stage. During this stage, the original Datalog program is rewritten in an equivalent form that preserves the semantics of the original program. In its rewritten form, the program becomes amenable to a distribution over more than one processor.
2. An *assignment* stage. At this stage, the rules of the rewritten program are assigned to specific processsors.
3. An *execution* stage. During this stage, the program is executed in parallel. Each of the processors uses the semi-naive evaluation method. At the end, the results are collected.

For each of the parallelization schemes that we will present, the rewritten programs are different. We will introduce the notion of a *discriminating predicate* which partitions the computation carried out at both the processors. The rewriting method, the assignment of rules of the rewritten program to the processors and the choice of the discriminating predicate affect the performance of the parallel execution. Some of the factors that characterize the preformance are:

- *non-redundancy :* Intuitively, this is a measure of the amount of computation duplicated by each of the processors.
- *communication:* The additional computation overhead forced by the need to move results form one processor to the other.

The first parallelization scheme presented in this paper acheives *non-redundancy* but requires communication in general (although there are some important special cases like Transitive Closure in which no communication is required.) The second parallelization scheme does *not* require communication. However, this scheme might result in the duplication of computations by the processors. The third scheme combines the features of the above parallelization schemes,and exhibits a trade-off between non-redundancy and communication.

The rest of the paper is organized as follows. Section 2 presents an example of one of the parallelization schemes by tracing its execution on given data. Section 3 deals with notation and preliminaries of the underlying theory of Datalog. Sections 4, 5 and 6 respectively present each of the three parallelization schemes presented in this paper. Section 7 shows how the choice of the discriminating predicate may affect the parallel execution. We derive two algorithms that have already appeared in the literature [4, 3] as special cases of our parallelization schemes, by varying the discriminating predicate. In Section 8, we present our conclusions.

2 An Illustration of the Parallelization Technique

In this section we introduce our method by showing how a given Datalog program can be executed in parallel. Given a logic program in the canonical form specified above, we wish to execute it in parallel using two communicating sequential processes. We have chosen the "same-generation" Datalog program as an example. We will refer to this program by SG.

$$\begin{array}{llll} SG: & sg(x,y) & \leftarrow & p(u,x), sg(u,v), p(v,y). \\ & sg(x,x) & \leftarrow & human(x). \end{array}$$

Intuitively, the atom $sg(a,b)$ is true if a and b have a common ancestor c, and the "generation gap" between a and c and between b and c is the same. Logically, the recursive rule states that x and y are of the same generation if they have parents u and v that belong to the same generation. The relation $p(u,v)$ has the intended interpretation: u is the parent of v. The first step in the parallelization scheme is to rewrite SG as an equivalent Datalog program. We define two Datalog programs, $SG1$ and $SG2$ as follows:

$$\begin{array}{llll} SG1: & sg_{11}(x,x) & \leftarrow & human(x), odd(x). \\ & sg_{11}(x,y) & \leftarrow & p(u,x), sg_{11}(u,v), p(v,y), odd(x). \\ & sg_{12}(x,y) & \leftarrow & p(u,x), sg_{11}(u,v), p(v,y), even(x). \\ & sg_{12}(x,y) & \leftarrow & p(u,x), sg_{12}(x,y), p(v,y), even(x). \\ & sg_{11}(x,y) & \leftarrow & p(u,x), sg_{12}(x,y), p(v,y), odd(x). \end{array}$$

$$\begin{array}{llll} SG2: & sg_{22}(x,x) & \leftarrow & human(x), even(x). \\ & sg_{22}(x,y) & \leftarrow & p(u,x), sg_{22}(u,v), p(v,y), even(x). \\ & sg_{22}(x,y) & \leftarrow & p(u,x), sg_{21}(u,v), p(v,y), even(x). \\ & sg_{21}(x,y) & \leftarrow & p(u,x), sg_{22}(u,v), p(v,y), odd(x). \\ & sg_{21}(x,y) & \leftarrow & p(u,x), sg_{21}(u,v), p(v,y), odd(x). \end{array}$$

Let us denote by $M_{SG}(p, human)(sg)$ the set of results produced in the relation sg by the program SG for the given input relations p and $human$. We will demonstrate in this example that:

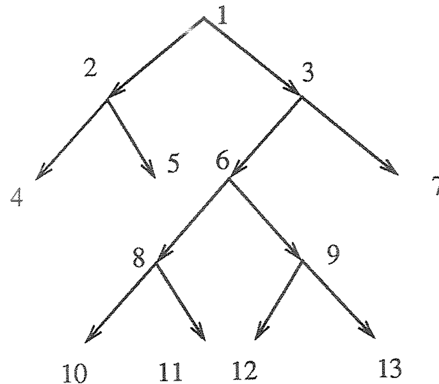
$$M_{SG}(p, human)(sg) = M_{SG1}(p, human)(sg_{11}) \cup M_{SG2}(p, human)(sg_{22})$$

The example will use a particular data set; the equality in general will be shown in section 5.

Let the base relations be as follows:

$$human = \{(1,1), (2,2), \ldots, (13,13)\}$$

The relation p is depicted by the graph below. An arc $i \rightarrow j$ implies $(i, j) \in p$.



We will assign the program $SG1$ to one processor and $SG2$ to the other processor. The following is a trace of the computation.

	Processor 1	Processor 2
initial value:	$\{(1,1),(3,3),\ldots,(13,13)\}$	$\{(2,2),(4,4),\ldots,(12,12)\}$
New Tuples after first iteration :	$\{(2,3),(3,2),(6,7),$ $\{(7,6),(12,13),(13,12)\}$	$\{(4,5),(5,4),(8,9),$ $\{(9,8),(10,11),(11,10)\}$
New tuples after second iteration :	$\{(4,6),(4,7),(5,6),(5,7),$ $\{(6,4),(6,5),(7,4),(7,5)\}$	$\{(10,12),(10,13),(11,12),(11,13),$ $\{(12,10),(12,11),(13,11),(13,10)\}$
After third iteration:	$\{\}$	$\{\}$

Fixpoint was reached for both computations and the results are:

Tuples generated by processor 1	=	21
Tuples generated by processor 2	=	21
Tuples duplicated	=	0

If we had executed the sequential program SG on the same input data, then the number of tuples produced in the output would be 42. Thus, the parallelization scheme adopted in this example cuts the execution time in two with no additional overhead. In general, the results obtained for other programs and different data sets are not always without overhead. This overhead, in particular the trade-off between redundant computation and communication will be explored in the sequel.

3 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. An $S - atom$ is an atom having S as the predicate symbol. A *rule* consists of an atom Q, designated as the head, and a conjunction of one or more atoms, denoted by $Q_1 \dots Q_k$ designated as the body. Such a rule is denoted as $Q \leftarrow Q_1 \dots Q_k$.

A *Datalog* program is a finite set of rules whose predicate symbols are divided into two disjoint subsets: the *base* predicates, (also called sl 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 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. Let $R_1 \dots R_n$ be the base predicate symbols appearing in the program. Then the smallest model satisfying P and containing $R_1 \dots R_n$ is denoted as $M_P(R_1 \dots R_n)$. The interpretation of a specific derived predicate T occurring in P in this smallest model is denoted by $M_P(R_1 \dots R_n)(T)$. The notion of *completion* of a program is frequently used in the paper and is defined next.

Let $p(t_1, \dots, t_n) \leftarrow L_1, \dots, L_n$ be a program clause in a Datalog program P, where $t_1, \dots t_n$ are Datalog terms. We will require a new predicate symbol "=" , not appearing in P, whose intended interpretation is the equality relation. The first step is to transform the given clause into

$$p(x_1, \dots, x_n) \leftarrow (x_1 = t_1) \wedge \dots (x_n = t_n) \wedge L_1 \wedge \dots \wedge L_m.$$

where $x_1, \dots, x_n$ are variables not appearing in the clause. Then if $y_1, \dots, y_d$ are the variables of the original clause, we transform the above formula into

$$p(x_1, \dots, x_n) \leftarrow \exists y_1, \dots, y_d (x_1 = t_1) \wedge \dots \wedge (x_n = t_n) \wedge L_1 \wedge \dots \wedge L_m.$$

Now, suppose that this transformation is performed for each rule with the predicate symbol p in its head. Then we obtain $k, k \geq 1$ transformed formulas of the form:

$$\begin{aligned} p(x_1, \dots, x_n) &\leftarrow E_1. \\ &\vdots \\ p(x_1, \dots, x_n) &\leftarrow E_k. \end{aligned}$$

The union of the formulas above is given by the formula:

$$\forall x_1, \dots, x_n (p(x_1, \dots, x_n) \Leftrightarrow E_1 \vee \dots E_k)$$

and is called the *completed definiton of p.*
For a program P, denote by $IFF(P)$ the union of the completed definitions of each of the predicate symbols appearing in P. The completion of P, denoted by $comp(P)$ is $IFF(P) \cup IFF(=)$. We will use the following results from the theory of logic programming [1, 2].

1. Every model of the completion of a Datalog program is a model of the datalog program.
2. The smallest model of the completion of a Datalog program is the smallest model of the datalog program.

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 non-reflexive transitive closure of the "derives" relstion. 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 [2] and in [1].

4 Parallelization with no Redundancy

In this section we present a parallelization scheme that results in no redundancy. Given the following program:

$$\begin{array}{lcl} L: \; T(\bar{z}\,) & \leftarrow & R(\bar{z}\,). \\ T(\bar{x}\,) & \leftarrow & B(\bar{u}\,), T(\bar{y}\,). \end{array}$$

Let $\bar{v}$ be any subset of all the variables occurring in the body of the recursive rule, including the base predicate symbols and the derived predicate symbol. We now introduce the notion of a discrminating predicate. p, q are called *discriminating predicates* over some domain D if they satisfy the following properties:

1. For any $x \in D$, $p(x) \vee \; q(x) \equiv true$.
2. For any $x \in D$, $p(x) \wedge \; q(x) \equiv false$.

Examples of such predicates are *odd* and *even* , where the domain is the set of natural numbers. In the following, we assume that p, q and s, t are discrminating predicates. The predicate symbols p,q,s and t are new predicate symbols that do not appear in the original program. In the next section we rewrite the program L.

4.1 Rewrite Stage

The following program Q consists of the subprograms $Q1, Q2, C, E1, I$. Let T be an n-ary predicate symbol of L and let $\bar{w}$ be a sequence of n new variables not appearing in the program L. Introduce new n-ary predicate symbols T_{11}, T_{12}, T_{21} and T_{22}. Let $\bar{v}$ be a sequence of some (or possibly all) of the variables appearing in the body of the recursive rule of L. Let $\bar{e}$ be a sequence of some (or possibly all) of the variables appearing in the body of the exit rule of L. The sequences $\bar{x}$, $\bar{y}$, $\bar{z}$ and $\bar{u}$ are identical to the corresponding sequences in L. The parallelism is obtained by partitioning the derivation of tuples. The final output T is the union of the individual relations T_{11} and T_{22} computed in parallel by processors 1 and 2 respectively. The relation T_{12} contains the set of tuples that must be communicated from processor 1 to processor 2, and vice-versa for T_{21}. The subprogram I gives the rules for the inital assignment of values to T_{11} and T_{22} respectively. The initial values of T_{12} and T_{21} are empty relations. $Q1$ and $Q2$ represent the recursive components of the parallel program. $Q1$ evaluates the set of all tuples in T_{11} whose derivation satisfies the discriminating predicate p (and vice-versa for $Q2$). However, it is possible that some results in the final output are generated by applying the derivation of $Q1$ to the reaults of $Q2$. Hence the communication step in the subprogram is necessary in general. The double subscripts i, j used in the predicates $T_{i,j}$ are to be interpreted as follows. The first subscript i denotes the processor number (1 or 2) where this relation is computed. The second subscript j denotes whether the derivation of the tuples for $T_{i,j}$ satisfy the discriminating predicate p (1) or q (2). We emphasize

that the above discussion is for intuitive clarification alone, and from a formal point of view is not necessary.

$$\begin{array}{lrcl} Q1: & T_{11}(\bar{x}) & \leftarrow & B(\bar{u}), T_{11}(\bar{y}), p(\bar{v}). \\ & T_{12}(\bar{x}) & \leftarrow & B(\bar{u}), T_{11}(\bar{y}), q(\bar{v}). \end{array}$$

$$\begin{array}{lrcl} Q2: & T_{22}(\bar{x}) & \leftarrow & B(\bar{u}), T_{22}(\bar{y}), q(\bar{v}). \\ & T_{21}(\bar{x}) & \leftarrow & B(\bar{u}), T_{22}(\bar{y}), p(\bar{v}). \end{array}$$

$$\begin{array}{lrcl} C: & T_{11}(\bar{w}) & \leftarrow & T_{21}(\bar{w}). \\ & T_{22}(\bar{w}) & \leftarrow & T_{12}(\bar{w}). \end{array}$$

$$\begin{array}{lrcl} I: & T_{11}(\bar{z}) & \leftarrow & R(\bar{z}), s(\bar{e}). \\ & T_{22}(\bar{z}) & \leftarrow & R(\bar{z}), t(\bar{e}). \end{array}$$

$$\begin{array}{lrcl} E1: & T(\bar{w}) & \leftarrow & T_{11}(\bar{w}). \\ & T(\bar{w}) & \leftarrow & T_{22}(\bar{w}). \end{array}$$

The program Q lends itself to parallelism in a very natural way. However, we first need to establish the equivalence of the logic programs L and Q while preserving the semantics of Datalog. The claim is that given input relations B ad R, the interpretation of the relation T occurring in the smallest model of L is identical to the interpretation of the relation T in the smallest model for Q.

Lemma 1: *comp*$(Q) \Rightarrow$ *comp*(L).

Proof: Let $\bar{l}$ be the set of variables appearing in the body of the recursive rule of L, but not in the head.
Let eq_1 denote the following formula:

$$eq_1: \ (w_1 = x_1) \wedge \ldots \wedge (w_n = x_n)$$

where x_i is the term appearing in the ith argument position in the head of the recursive rule. Let eq_2 denote the following formula :

$$eq_2: \ (w_1 = z_1) \wedge \ldots \wedge (w_n = z_n)$$

where z_i is the term appearing in the ith argument position of the head of the exit rule. Then,

$$IFF(L) \equiv \forall \bar{w} \ (T(\bar{w}) \Leftrightarrow \exists \bar{l} \ (B(\bar{u}) \wedge T(\bar{y}) \wedge eq_1) \vee (R(\bar{z}) \wedge eq_2))$$

$$\begin{array}{rcll} IFF(Q) & \equiv & \forall \bar{w} \ (T(\bar{w}) & \Leftrightarrow \ T_{11}(\bar{w}) \vee T_{22}(\bar{w}))\wedge \\ & & \forall \bar{w} \ (T_{11}(\bar{w}) & \Leftrightarrow \ \exists \bar{l} \ (B(\bar{u}) \wedge T_{11}(\bar{y}) \wedge p(\bar{v}) \wedge eq_1)\vee \\ & & & \quad T_{21}(\bar{w}) \vee (R(\bar{z}) \wedge s(\bar{e}) \wedge eq_2))\wedge \\ & & \forall \bar{w} \ (T_{12}(\bar{w}) & \Leftrightarrow \ \exists \bar{l} \ (B(\bar{u}) \wedge T_{11}(\bar{y}) \wedge q(\bar{v}) \wedge eq_1))\wedge \\ & & \forall \bar{w} \ (T_{21}(\bar{w}) & \Leftrightarrow \ \exists \bar{l} \ (B(\bar{u}) \wedge T_{22}(\bar{y}) \wedge p(\bar{v}) \wedge eq_1))\wedge \\ & & \forall \bar{w} \ (T_{22}(\bar{w}) & \Leftrightarrow \ \exists \bar{l} \ (B(\bar{u}) \wedge T_{22}(\bar{y}) \wedge q(\bar{v}) \wedge eq_1)\vee \\ & & & \quad T_{12}(\bar{w}) \vee (R(\bar{z}) \wedge t(\bar{e}) \wedge eq_2)) \end{array}$$

$$
\begin{array}{lllll}
IFF(Q) & \Rightarrow & \forall \bar{w}\ (T(\bar{w}\ ) & \Leftrightarrow & T_{11}(\bar{w}\ ) \vee\ T_{22}(\bar{w}\ )) \\
& \Rightarrow & \forall \bar{w}\ (T(\bar{w}\ ) & \Leftrightarrow & \exists \bar{l}\ (B(\bar{u}\ ) \wedge\ T_{11}(\bar{y}\ ) \wedge\ p(\bar{v}\ ) \wedge\ eq_1) \vee \\
& & & & T_{21}(\bar{w}\ ) \vee\ (R(\bar{z}\ ) \wedge\ s(\bar{e}\ ) \wedge\ eq_2) \vee \\
& & & & \exists \bar{l}\ (B(\bar{u}\ ) \wedge\ T_{22}(\bar{y}\ ) \wedge\ q(\bar{v}\ ) \wedge\ eq_1) \vee \\
& & & & T_{22}(\bar{w}\ ) \vee\ (R(\bar{z}\ ) \wedge\ t(\bar{e}\ ) \wedge\ eq_2)) \\
& \Rightarrow & \forall \bar{w}\ (T(\bar{w}\ ) & \Leftrightarrow & \exists \bar{l}\ (B(\bar{u}\ ) \wedge\ T_{11}(\bar{y}\ ) \wedge\ p(\bar{v}\ ) \wedge\ eq_1) \vee \\
& & & & \exists \bar{l}\ (B(\bar{u}\ ) \wedge\ T_{22}(\bar{y}\ ) \wedge\ p(\bar{v}\ ) \wedge\ eq_1) \vee \\
& & & & \exists \bar{l}\ (B(\bar{u}\ ) \wedge T_{11}(\bar{y}\ ) \wedge\ q(\bar{v}\ ) \wedge\ eq_1) \vee \\
& & & & \exists \bar{l}\ (B(\bar{u}\ ) \wedge\ T_{22}(\bar{y}\ ) \wedge\ q(\bar{v}\ ) \wedge\ eq_1) \vee \\
& & & & (R(\bar{z}\ ) \wedge\ eq_2)) \\
& \Rightarrow & \forall \bar{w}\ (T(\bar{w}\ ) & \Leftrightarrow & \exists \bar{l}\ (B(\bar{u}\ \wedge\ T_{11}(\bar{y}\ ) \wedge\ eq_1) \vee \\
& & & & \exists \bar{l}\ (B(\bar{u}\ ) \wedge\ T_{22}(\bar{y}\ ) \wedge\ eq_1) \vee\ (R(\bar{z}\ ) \wedge\ eq_2)) \\
& \Rightarrow & \forall \bar{w}\ (T(\bar{w}\ ) & \Leftrightarrow & \exists \bar{l}\ (B(\bar{u}\ ) \wedge\ (T_{11}(\bar{y}\ ) \vee\ T_{22}(\bar{y}\ )) \wedge\ eq_1) \vee\ (R(\bar{z}\ ) \wedge\ eq_2)) \\
& \Rightarrow & \forall \bar{w}\ (T(\bar{w}\ ) & \Leftrightarrow & \exists \bar{l}\ (B(\bar{u}\ ) \wedge\ T(\bar{y}\ ) \wedge\ eq_1) \vee\ (R(\bar{z}\ ) \wedge\ eq_2)) \\
& \Rightarrow & IFF(L) & &
\end{array}
$$

Let $EQ(L)$ denote the equality theory for the program L, and $EQ(P)$ denote the equality theory for Q. Since every variable and constant appearing in L also appears in Q, we can claim that $EQ(Q) \Rightarrow\ EQ(L)$. Hence,

$$
\begin{array}{lll}
comp(Q) & \Leftrightarrow & IFF(Q) \wedge\ EQ(Q) \\
& \Rightarrow & IFF(L) \wedge\ EQ(L) \\
& \Leftrightarrow & comp(L) \Box
\end{array}
$$

Lemma 2: For any input relations B and R, $M_Q(B,R) \supseteq M_L(B,R)$.

Proof: Since L and Q are positive definite programs (as all Datalog programs are), $M_Q(B,R)$ is a model of *comp(Q)*. From lemma 1, it is a model of *comp(L)*. $M_L(B,R)$ is the smallest model for *comp(L)* and hence $M_Q(B,R) \supseteq M_L(B,R)$. $\Box$

Lemma 3 Let T be an n-ary predicate symbol, and $\bar{x_0}$ be a sequence of n constants. Then, if $T(\bar{x_0}\ )$ or $T_{11}(\bar{x_0}\ )$ or $T_{12}(\bar{x_0}\ )$ or $T_{21}(\bar{x_0}\ )$ or $T_{22}(\bar{x_0}\ )$ has an SLD-refutation of length n from Q, then $T(\bar{x_0}\ )$ has an SLD-refutation from L.

Proof: For the purposes of this proof we will asume that the symbol $\Box$ denotes the empty clause and signifies the end of a successful SLD-refutation. The proof is by induction on the length of the SLD-refutation of $T(\bar{x_0}\ )$ or $T_{11}(\bar{x_0}\ )$ or $T_{12}(\bar{x_0}\ )$ or $T_{21}(\bar{x_0}\ )$ or $T_{22}(\bar{x_0}\ )$ from Q.

$n = 1:$ $T(\bar{x_0}\ )T_{21}(\bar{x_0}\ )$ and $T_{12}(\bar{x_0}\ )$ cannot have SLD-refutation of length 1. Thus we have two cases: either $T_{11}(\bar{x_0}\ )$ has an SLD-refutation of length 1, or $T_{22}(\bar{x_0}\ )$ has an SLD-refutation of length 1. Suppose $T_{11}(\bar{x_0}\ )$ has an SLD-refutation of length 1. Then $T_{11}(\bar{x_0}\ ) \xrightarrow{I} R(\bar{x_0}\ )s(\bar{x_0}\ ) \longrightarrow \Box$
Hence $T(\bar{x_0}\ ) \xrightarrow{Q} R(\bar{x_0}\ ) \longrightarrow \Box$
The same argument holds for $T_{22}(\bar{x_0}\ )$.

$n > 1:$ Consider $n = k + 1$.

Case 1: $T(\bar{x}_0)$ has an SLD-refutation of length $n = k+1$ from Q. The clause used in the first step of the refutation must be from $E1$. Suppose the first clause of $E1$ is used. Then
$T(\bar{x}_0) \xrightarrow{E1} T_{11}(\bar{x}_0) \xrightarrow{n-1steps} \ldots \longrightarrow \Box$
By the induction hypothesis we know that $T(\bar{x}_0)$ may be refuted from L.
The proof is similar if the second clause of $E2$ is used in the first step of the SLD-refutation.

Case 2: Suppose $T_{11}(\bar{x}_0)$ has an SLD-refutation of length $n = k+1$ from Q. There are two possibilities.

1. The first step in the SLD-refutation may use a clause from $Q1$.
   $T_{11}(\bar{x}_0) \xrightarrow{Q1} B(\bar{u}_0)T_{11}(\bar{y}_0)p(\bar{v}_0) \xrightarrow{n-1steps} \ldots \longrightarrow \Box$
   Hence $T_{11}(\bar{y}_0)$ may be refuted from Q using $n-1$ steps. By the induction hypothesis, $T(\bar{y}_0)$ may be refuted from L. Hence $T(\bar{x}_0)$ may be refuted from L as follows:
   $T(\bar{x}_0) \longrightarrow B(\bar{u}_0)T(\bar{y}_0) \longrightarrow \ldots \longrightarrow \Box$

2. The first step in the SLD-refutation may use a clause from C.
   $T_{11}(\bar{x}_0) \xrightarrow{C} T_{21}(\bar{x}_0) \xrightarrow{n-1steps} \ldots \longrightarrow \Box$
   $T_{21}(\bar{x}_0)$ has an SLD-refutation of $n-1$ steps from Q. Hence $T(\bar{x}_0)$ has an SLD-refutation from L.

Case 3: The first step in the SLD-refutation must use a clause from $Q2$.

$T21(\bar{x}_0) \xrightarrow{Q2} B(\bar{u}_0)T_{22}(\bar{y}_0)p(\bar{v}_0) \xrightarrow{nsteps} \ldots \longrightarrow \Box$
Hence $T_{22}(\bar{x}_0)$ has an SLD-refutation of length $n-1$ from Q. By the induction hypothesis, $T(\bar{y}_0)$ has an SLD-refutation from L. hence $T(\bar{y}_0)$ may be refuted from L as follows.
$T(\bar{x}_0) \longrightarrow B(\bar{u}_0)T(\bar{y}_0) \longrightarrow \ldots \longrightarrow \Box$

The other cases for $T_{22}(\bar{x}_0)$ and $T_{21}(\bar{x}_0)$ are symmetric. $\Box$

Lemma 4: $M_L(B,R)(T) \supseteq M_Q(B,R)(T)$.

Proof:

$$\begin{array}{lcl} M_Q(B,R)(T) & = & \{T(\bar{x}_0) \mid T(\bar{x}_0) \text{ has an SLD-refutation from } Q\} \\ \text{by lemma 3} & \subseteq & \{T(\bar{x}_0) \mid T(\bar{x}_0) \text{has an SLD-refutation from } L\} \\ & = & M_L(B,R)(T) \end{array}$$

Theorem 4: $M_L(B,R)(T) = M_Q(B,R)(T)$.

Proof: From lemma 2 we have :

$M_L(B,R)(T) \subseteq M_Q(B,R)(T)$

From lemma 4 we have :

$$M_L(B,R)(T) \supseteq M_Q(B,R)(T)$$

Hence :

$$M_L(B,R)(T) = M_Q(B,R)(T)\Box.$$

4.2 Assignment and Execution Stages

The equivalence of the Datalog programs L and Q have been proven in the previous section. In this section we will discuss the assignment of the subprograms of Q to two processors and their execution. However, this schema can be generalised to any number of processors. One possibility might be to assign $Q1$ to processor 1,and assign $Q2$ to processor 2; once this is done, the computation proceeds as follows. The parallel termination of this algorithm will not be discussed here.

Processor 1:
evaluate the rule
$T_{11}(\bar{z}\ ) \leftarrow R(\bar{z}\ ), s(\bar{e}\ ).$
repeat
  evaluate the following rule to a fixpoint
  $T_{11}(\bar{x}\ ) \leftarrow B(\bar{u}\ ), T_{11}(\bar{y}\ ), p(\bar{v}\ ).$
  evaluate the following rule once
  $T_{12}(\bar{x}\ ) \leftarrow B(\bar{u}\ ), T_{12}(\bar{y}\ ), p(\bar{v}\ ).$
  send T_{12} to processor 2
  $T_{12}(\bar{w}\ ) \leftarrow T_{22}(\bar{w}\ ).$
  receive T_{21} from processor 2
  $T_{11}(\bar{w}\ ) \leftarrow T_{21}(\bar{w}\ ).$
until termination

Processor 2:
evaluate the rule
$T_{22}(\bar{z}\ ) \leftarrow R(\bar{z}\ ), t(\bar{e}\ ).$
repeat
  evaluate the following rule to a fixpoint
  $T_{22}(\bar{x}\ ) \leftarrow B(\bar{u}\ ), T_{11}(\bar{y}\ ), p(\bar{v}\ ).$
  evaluate the following rule once
  $T_{12}(\bar{x}\ ) \leftarrow B(\bar{u}\ ), T_{12}(\bar{y}\ ), p(\bar{v}\ ).$
  send T_{21} to processor 1
  $T_{11}(\bar{w}\ ) \leftarrow T_{21}(\bar{w}\ )$
  receive T_{12} from processor 1
  $T_{22}(\bar{w}\ ) \leftarrow T_{12}(\bar{w}\ )$
until termination

4.3 Discussion

The algorithm presented above is non-redundant in the following sense. Given input relations B and R we define :

1. $count(\bar{x_0}\ )$ = the number of times tuple $T_{11}(\bar{x_0}\ )$ or $T_{22}(\bar{x_0}\ )$ was generated by the parallel evaluation algorithm.

2. $count-parallel(T)$ = the sum of the $count(\bar{x_0}\ )$ for every tuple $T_{11}(\bar{x_0}\ )$ or $T_{22}(\bar{x_0}\ )$ generated by the parallel algorithm. Thus,

$$count - parallel(T) = \sum\{count(\bar{x_0}\ ) \mid T(\bar{x_0}\ ) \text{ is a tuple in the output}\}$$

3. $count - seminaive(T)$ = the sum of the number of times a tuple $T(\bar{x_0}\ )$ is produced in the semi-naive evaluation of the original logic program L.

We claim that $count\!-\!seminaive(T) \geq count\!-\!parallel(T)$. The proof of this claim is quite lengthy. For brevity we will give only an intuitive justification.

If $T(\bar{x}_0)$ is generated in the evaluation of the logic program $Q1$, then there exists $\bar{w}_0$ such that $B(\bar{u}_0) \wedge T(\bar{y}_0) \wedge p(\bar{v}_0)$ is true. Hence $B(\bar{u}_0) \wedge T(\bar{y}_0) \wedge q(\bar{v}_0)$ is false(because p and q are discriminating predicates) and hence the same derivation of the output tuple $T(\bar{x}_0)$ cannot be replicated in the evaluation of $Q2$. Hence, processors 1 and 2 do not duplicate any work.

5 Parallelization without Communication

In this section we present another parallelization scheme for the logic program L that does not require communication between processors.

5.1 Rewrite Stage

Rewrite the logic program L as the union of the subprograms $P1, P2, I$ and E. The assumptions about the predicate symbols $T, T_{11}, T_{12}, T_{21}, T_{22}$ and the varible sequences $\bar{e}$, $\bar{v}$ and $\bar{w}$ are the same as for program Q described earlier. Notice that in this parallelization scheme, the C-component that appeared in program Q is omitted.

$$
\begin{aligned}
P1:\quad T_{11}(\bar{x}) &\leftarrow B(\bar{u}), T_{11}(\bar{y}), p(\bar{v}).\\
T_{12}(\bar{x}) &\leftarrow B(\bar{u}), T_{11}(\bar{y}), q(\bar{v}).\\
T_{12}(\bar{x}) &\leftarrow B(\bar{u}), T_{12}(\bar{y}), q(\bar{v}).\\
T_{11}(\bar{x}) &\leftarrow B(\bar{u}), T_{12}(\bar{y}), p(\bar{v}).
\end{aligned}
$$

Notice that the first two rules of P1 are the same as Q1.

$$
\begin{aligned}
P2:\quad T_{21}(\bar{x}) &\leftarrow B(\bar{u}), T_{21}(\bar{y}), p(\bar{v}).\\
T_{22}(\bar{x}) &\leftarrow B(\bar{u}), T_{21}(\bar{y}), q(\bar{v}).\\
T_{22}(\bar{x}) &\leftarrow B(\bar{u}), T_{22}(\bar{y}), q(\bar{v}).\\
T_{21}(\bar{x}) &\leftarrow B(\bar{u}), T_{22}(\bar{y}), p(\bar{v}).
\end{aligned}
$$

Notice that the last two rules of P2 are the same as Q2.

$$
\begin{aligned}
E:\quad T(\bar{w}) &\leftarrow T_{11}(\bar{w}).\\
T(\bar{w}) &\leftarrow T_{12}(\bar{w}).\\
T(\bar{w}) &\leftarrow T_{21}(\bar{w}).\\
T(\bar{x}) &\leftarrow T_{22}(\bar{x}).
\end{aligned}
$$

$$
\begin{aligned}
I:\quad T_{11}(\bar{z}) &\leftarrow R(\bar{z}), s(\bar{e}).\\
T_{22}(\bar{z}) &\leftarrow R(\bar{z}), t(\bar{e}).
\end{aligned}
$$

We now demonstrate the equivalence of the logic programs L and $P, P = P1 \cup P2 \cup I \cup E$.

Lemma 5: $comp(P) \Rightarrow comp(L)$

Proof: Let $\bar{l}$ be the set of variables appearing in the body of the recursive rule of L, but not in the head.
Let eq_1 denote the following formula:

$$eq_1 : \ (w_1 = x_1) \wedge \ldots \wedge (w_n = x_n)$$

where x_i is the term appearing in the ith argument position in the head of the recursive rule. Let eq_2 denote the following formula :

$$eq_2 : \ (w_1 = z_1) \wedge \ldots \wedge (w_n = z_n)$$

where z_i is the term appearing in the ith argument position of the head of the exit rule. Then,

$$IFF(L) \equiv \forall \bar{w} \, (T(\bar{w}) \Leftrightarrow \exists \bar{l} \, (B(\bar{u}) \wedge eq_1) \vee (R(\bar{z}) \wedge eq_2))$$

$$
\begin{array}{lllll}
IFF(P) & \Leftrightarrow & \forall \bar{w} \, (T(\bar{w}) & \Leftrightarrow & T_{11}(\bar{w}) \vee T_{12}(\bar{w}) \vee T_{21}(\bar{w}) \vee T_{22}(\bar{w})) \\
 & & \forall \bar{w} \, (T_{11}(\bar{w}) & \Leftrightarrow & \exists \bar{l} \, (B(\bar{u}) \wedge T_{11}(\bar{y}) \wedge p(\bar{v}) \wedge eq_1) \vee \\
 & \wedge & & & \exists \bar{l} \, (B(\bar{u}) \wedge T_{12}(\bar{y}) \wedge p(\bar{v}) \wedge eq_1) \vee \\
 & & & & R(\bar{z}) \wedge s(\bar{e}) \wedge eq_2)) \\
 & \wedge & \forall \bar{w} \, (T_{12}(\bar{w}) & \Leftrightarrow & \exists \bar{l} \, (B(\bar{u}) \wedge T_{11}(\bar{y}) \wedge q(\bar{v}) \wedge eq_1) \vee \\
 & & & & \exists \bar{l} \, (B(\bar{u}) \wedge T_{12}(\bar{y}) \wedge q(\bar{v}) \wedge eq_1)) \\
 & \wedge & \forall \bar{w} \, (T_{21}(\bar{w}) & \Leftrightarrow & \exists \bar{l} \, (B(\bar{u}) \wedge T_{22}(\bar{y}) \wedge p(\bar{v}) \wedge eq_1) \vee \\
 & & & & \exists \bar{l} \, (B(\bar{u}) \wedge T_{21}(\bar{y}) \wedge p(\bar{v}) \wedge eq_1)) \\
 & & \forall \bar{w} \, (T_{22}(\bar{w}) & \Leftrightarrow & \exists \bar{l} \, (B(\bar{u}) \wedge T_{22}(\bar{y}) \wedge p(\bar{v}) \wedge eq_1) \vee \\
 & \wedge & & & \exists \bar{l} \, (B(\bar{u}) \wedge T_{21}(\bar{y}) \wedge p(\bar{v}) \wedge eq_1) \vee \\
 & & & & R(\bar{z}) \wedge s(\bar{e}) \wedge eq_2))
\end{array}
$$

After simplification as in Lemma 1, we can prove that $IFF(P) \Rightarrow IFF(L)$. Let $EQ(P)$ and $EQ(L)$ denote the equality theories for P and L respectively. Since every constant, variable and predicate symbol appearing in L also appear in P, we can claim that $EQ(P) \Rightarrow EQ(L)$. Thus:

$$
\begin{array}{lll}
comp(P) & \equiv & IFF(P) \cup EQ(P) \\
 & \Rightarrow & IFF(L) \cup EQ(L) \\
 & \equiv & comp(L) \square
\end{array}
$$

Theorem 6: $M_L(B,R)(T) = M_P(B,R)(T)$.

Proof: By a sequence of lemmas as in lemmas 2 through 5 ,the above result is obtained. □

5.2 Assignment and Execution Stage

One way to get a parallel implementation of P is to assign the various rules of P to different processors. Notice that the programs $P1$ and $P2$ only share the base relations but not the derived relations. Hence $P1$ and $P2$ can be independently assigned to processors without any need for communication. The computation proceeds operationally as follows.

Processor 1:
evaluate the following datalog program to a fixpoint

$$
\begin{array}{lll}
T_{11}(\bar{x}) & \leftarrow & R(\bar{x}), s(\bar{x}). \\
T_{11}(\bar{x}) & \leftarrow & B(\bar{u}) T_{11}(\bar{y}) p(\bar{v}). \\
T_{12}(\bar{x}) & \leftarrow & B(\bar{u}) T_{11}(\bar{y}) q(\bar{v}). \\
T_{12}(\bar{x}) & \leftarrow & B(\bar{u}) T_{12}(\bar{y}) q(\bar{v}). \\
T_{11}(\bar{x}) & \leftarrow & B(\bar{u}) T_{12}(\bar{y}) p(\bar{v}).
\end{array}
$$

Processor 2:
evaluate the following datalog program to a fixpoint

$$
\begin{array}{lll}
T_{22}(\bar{x}) & \leftarrow & R(\bar{x}), t(\bar{x}). \\
T_{21}(\bar{x}) & \leftarrow & B(\bar{u}), T_{21}(\bar{y}), p(\bar{v}). \\
T_{22}(\bar{x}) & \leftarrow & B(\bar{u}), T_{21}(\bar{y}), q(\bar{v}). \\
T_{22}(\bar{x}) & \leftarrow & B(\bar{u}), T_{22}(\bar{y}), q(\bar{v}). \\
T_{21}(\bar{x}) & \leftarrow & B(\bar{u}), T_{22}(\bar{y}), p(\bar{v}).
\end{array}
$$

5.3 Discussion

In the above parallel implementation, each processor works with a subset of the input. Hence, it is expected that each processor would evaluate its output faster than in the case where a single processor does the evaluation over the entire input relations. Because there is no communication we can claim that this parallel program performs no worse than a sequential semi-naive evaluation. To argue that the parallel program actually outperforms the sequential evaluation, we must further establish that each processor computes only a part of the output and not the entire output. While intuitively this would almost always be true, one could construct pathological cases where one of the processors would compute the entire output, and hence there would be no gain over the sequential case. Consider shared memory architechtures where all inut relations are shared. In that case there is no overhead of replication of base relations at the processor sites. We can therefore argue that the above scheme has enough merits to be considered "better" over a sequential semi-naive evaluation scheme. However, since some computation may be replicated by the processors redundant computation may result. Contrast this with the algorithm of section 4, which had no redundancy but required communication. If communication cost is free, the first algorithm is certainly better than the present one. However, if communication is expensive, it might be better to use the above algorithm. These are the two extremes, non-redundancy but with communication vs no communication accompanied by redundancy. In the next section we present an algorithm whch demonstrates a trade-off betwen the two extremes.

6 Tradeoff between Communication and Redundancy

Let us now compare the properties of the parallelization schemes presented in sections 4 and 5. The strategy presented in section 4, had the property of no redundancy, but required communication in general. The strategy presented in section 5 did not require communication, but in general might result in some redundant computation. In this section, we present a strategy which combines these strategies. Consider the logic program $S = P1 \cup P2 \cup C \cup I \cup E$; where $P1, P2, C, I$ and E are the same as the respective subprograms used in programs Q and P. We claim that this program is equivalent to the program L.

Lemma 7: $M_S(B, R) \supseteq M_P(B, R)$

Proof : $S = P1 \cup P2 \cup C \cup I \cup E$
and $P = P1 \cup P2 \cup I \cup E$
Hence, $S \Rightarrow P$. Therefore $M_S(B, R) \supseteq M_P(B, R)$. □

Theorem 8 $M_L(B, R)(T) = M_P(B, R)(T) = M_S(B, R)(T)$

Proof: By an argument exactly similar to lemmas 3 through 5, we establish the above result. □

Since equivalence of S with L has been proved, any correct implementation of S would be a correct implementation of L. Let us assign processor 1 to evaluate $P1$, processor 2 to evaluate $P2$, and let C represent a synchronous communication between the processors. Suppose that in a parallel implementation of S, the subprogram C is never executed. In that case the parallel implementation reduces to that of P, whose equivalence with L we have already showed. Thus, executing the communication step is not critical to the correctness of a parallel implementation

of$S$. We noted earlier that the first two rules of $P1$ are the same as $Q1$, and the last two rules of $P2$ are the same as $Q2$. Hnce one way to compute the fixpoint of $P1$ and $P2$ is the following:

computing fixpoint of $P1$	computing fixpoint of $P2$
repeat	repeat
1. compute subprogram $Q1$ to a fixpoint	1. compute subprogram $Q2$ to a fixpoint
2. evaluate the last two rules of $P1$ once	2. evaluate the first two rules of $P2$ once
until fixpoint is reached.	until fixpoint is reached.

Let us now suppose that the communication step is executed once after each iteration of the above computation for $P1$ and $P2$. Such a communication scheme results in an execution that is identical to the one for Q presented in section 4. As we already showed there, the scheme resulted in no redundancy. Thus both the previous parallelization schemes can be derived as different execution schemes for the above rewriting S, one resulting in no redundancy and the other not requiring communication. In general, more frequent executions of the communication step would result in lesser redundancy and vice-versa. The above parallelization scheme vividly illustrates the trade-off between communication and redundancy.

7 The Role of the Discriminating Predicate

In this section we explore the influence of the discriminating predicate on the parallelization schemes presented. We apply the discriminating predicate to variables respectively appearing in the input and derived relations. The parallelization scheme of section 5, when applied to a Transitive Closure and a discriminating predicate applied to the derived relation, is essentially the schema presented by Wolfson and Silberschatz [4]. The schema of section 4 with the discriminating predicate applied to the input relation is the same as that presented by Valduriez and Khoshafian [3].

7.1 Discriminating Predicate Applied to the Derived Relation

Consider the folowing Transitive Closure program.

$$
\begin{array}{lrcl}
TC: & T(x,y) & \leftarrow & R(x,y). \\
 & T(x,y) & \leftarrow & T(x,z), R(z,y).
\end{array}
$$

Suppose we choose the discriminating predicate to be on the first attribute of the derived relation T (i.e. attribute x). Let p and q be the discriminating predicates *odd* and *even* respectively. Also let s and t be *odd* and *even* respectively. Consider the parallelization scheme described in section 5. The rewrite stage of this scheme when applied to TC above, with p, q, s and t as interpreted above, generates a new Datalog program WS, as follows:

$$
\begin{array}{lrcl}
WS1: & T_{11}(x,y) & \leftarrow & R(x,y), odd(x). \\
 & T_{11}(x,y) & \leftarrow & T_{11}(x,z), R(z,y), odd(x). \\
 & T_{11}(x,y) & \leftarrow & T_{12}(x,z), R(z,y), odd(x). \\
 & T_{12}(x,y) & \leftarrow & T_{11}(x,z), R(z,y), even(x). \\
 & T_{12}(x,y) & \leftarrow & T_{12}(x,z), R(z,y),, even(x).
\end{array}
$$

$$
\begin{array}{llll}
WS2: & T_{22}(x,y) & \leftarrow & R(x,y), even(x). \\
 & T_{22}(x,y) & \leftarrow & T_{22}(x,z), R(z,y), even(x). \\
 & T_{22}(x,y) & \leftarrow & T_{21}(x,z), R(z,y), even(x). \\
 & T_{21}(x,y) & \leftarrow & T_{22}(x,z), R(z,y), odd(x). \\
 & T_{21}(x,y) & \leftarrow & T_{21}(x,z), R(z,y), odd(x).
\end{array}
$$

After the assign stage, processor 1 is assigned the logic program $WS1$ and procesor 2 is assigned the logic program $WS2$. Let us take a closer look at the execution of $WS1$. It follows from the rules of $WS1$ that during the execution phase if $(x,y) \in T_{11}$ then x is odd. Therefore the third rule of $WS1$ above may be eliminated since it would never contribute anything to the result. Similarily, if $(x,z) \in T_{12}$ then x is even. Therefore, the fourth rule of $WS1$ may also be eliminated. Thus $WS1$ may be rewritten equivalently as:

$$
\begin{array}{lll}
T_{11}(x,y) & \leftarrow & R(x,y), odd(x). \\
T_{11}(x,y) & \leftarrow & T_{11}(x,z), R(z,y), odd(x). \\
T_{12}(x,y) & \leftarrow & T_{12}(x,z), R(z,y), even(x).
\end{array}
$$

Notice that the predicate definition of T_{12} is purely self-recursive, and it is initialised to ϕ. Hence T_{12} contributes nothing to the result. Therefore the last rule above may be dropped from $WS1$. After this final transformation,$WS1$ reduces to:

$$
\begin{array}{llll}
WS1': & T_{11}(x,y) & \leftarrow & R(x,y), odd(x). \\
 & T_{11}(x,y) & \leftarrow & T_{11}(x,z), R(z,y), odd(x).
\end{array}
$$

Carrying out similar transformations for $WS2$, we get:

$$
\begin{array}{llll}
WS2': & T_{22}(x,y) & \leftarrow & R(x,y), even(x). \\
 & T_{22}(x,y) & \leftarrow & T_{22}(x,z), R(z,y), even(x).
\end{array}
$$

The parallel program $WS' = WS1' \cup WS2'$. The important property about WS is that

$$
\begin{array}{lll}
T_{11}(x,y) & \Rightarrow & odd(x). \\
T_{22}(x,y) & \Rightarrow & even(x).
\end{array}
$$

Processor 1 computes the logic program $WS1'$ which contains only one recursive predicate T_{11}. Processor 2 computes the logic program $WS2'$ which contains only one recursive predicate T_{22}. Because of the property noted above we can claim that there is no tuple (x,y) in the output which is computed by both the processors. Hence the parallel computation is non-redundant.

Interpreted as a computation over the graph of the relation R, $WS1'$is the logic program which computes all tuples (x,y) such that there is a path in the graph from x to y and x is odd. Similarily $WS2'$ is the logic program which computes all tuples (x,y) such that x is even and there is a path from x to y. An important thing to note is that the entire relation R is accessed by both the processors and hence must be either shared or replicated as the case may be. The reasoning presented above is applicable to any *pivotal* programs presented in [4].

7.2 Discriminating Predicate Applied to the Variables of the Input.

Suppose that the base relation R is partitioned (horizontally) into $R1$ and $R2$ such that processor 1 has access to $R1$ and processor 2 has access to $R2$. Define the discriminating predicate to be

$$\begin{array}{rcl} p(x,y) & = & (x,y) \in R1 \\ q(x,y) & = & (x,y) \in R2 \end{array}$$

By applying the rewriting procedure of Section 4, and using the above discriminating predicate, we get the following Datalog program $VK = VK1 \cup VK2$.

$$\begin{array}{llcl} VK1: & T_{11}(x,y) & \leftarrow & R(x,y), p(x,y). \\ & T_{11}(x,y) & \leftarrow & T_{11}(x,z), R(z,y), p(z,y). \\ & T_{21}(x,y) & \leftarrow & T_{22}(x,z), R(z,y), p(z,y). \\ & T_{11}(x,y) & \leftarrow & T_{21}(x,y). \end{array}$$

$$\begin{array}{llcl} VK2: & T_{22}(x,y) & \leftarrow & R(x,y), q(x,y). \\ & T_{22}(x,y) & \leftarrow & T_{22}(x,z), R(z,y), q(z,y). \\ & T_{12}(x,y) & \leftarrow & T_{11}(x,z), R(z,y), q(z,y). \\ & T_{22}(x,y) & \leftarrow & T_{12}(x,y). \end{array}$$

Assign program $VK1$ to processor 1 and program $VK2$ to processor 2. Notice that the evaluation of program $VK1$ requires the relation $\sigma_{p(u,v)}(R) = R1$ only. Thus the program $VK1$ can be evaluated by processor 1 because the partial input $R1$ is accessible by it. Smilarily the program $VK2$ can be evaluated by processor 2 because the partial input $R2 = \sigma_{q(u,v)}(R)$ is accessible by processor 2. However, communication is necessary between the processors because relations T_{11} and T_{22} appear in both $VK1$ and $VK2$. Thus the evaluation of $VK1$ cannot be done independent of the program $VK2$. Intuitively, the Datalog program $VK1$ computes the set of all pairs of vertices (x, y) such that there is a path from x to y, and the last edge in this path belongs to the relation $R1$. Similarily, the Datalog program $VK2$ computes the set of all pairs of vertices (x, y) such that there is a path from x to y in the graph of R, and the last edge in this path comes from the relation $R2$. Thus, if $R1$ and $R2$ is an exhaustive partition of all edges of the graph of relation R, it is clear that the transitive closure is correctly computed. This also explains why it is not possible to compute the sets computed by $VK1$ and $VK2$ independent of each other, for example suppose there is precisely one path from x to y, with the edges in this path exactly alternating between $R1$ and $R2$. The algorithm obtained as above is essentially the same as the one presented in [3].

7.3 Discussion

The two algorithms above illustrate the impact made by the choice of the discriminating predicate and its arguments upon the operational nature of the parallel iimplementation. It also shows how one might be led to choose a discriminating predicate and design an algorithm based upon constraints such as no replication of data or complete non-redundancy and no communication. The first algorithm does not require communication, is non-redundant but requires that the input relation R be shared. The second algorithm requires communication, is non-redundant but does not require that the input relation be shared.

8 Conclusions

We have presented several complementary methods for the parallel, bottom-up evaluation of linear Datalog programs. Each of these methods consists of three steps:

1. A *rewrite step* that renders an equivalent program to the original one, explicitly amenable to parallel execution.

2. An *assignment step* that assigns the rules and data of the rewritten program to processors.

3. An *execution step* that performs the computation, either with or without processor intercommunication.

We introduced the notion of a *discriminating predicate* by which the computation is partitioned among the processors and parallelism is acheived. The methods demonstrate the trade-offs between redundancy (duplication of computation by processors) and interprocessor-communication.

We would like to extend the applicability of the above strategies to more general classes of Datalog queries. The choice of discriminating predicates influences the performance of the parallel execution, and is a very important parameter to the parallelization process. In order for this method to be useful to a compiler, it would be important to know how these predicates should be chosen, depending upon the underlying architechture, availability of data and other such factors. The problem of parallel evaluation of recursive queries has also been addressed in [5, 6, 8, 7].

Acknowledgements

We would like to thank Carlo Zaniolo for many fruitful discussions. We would also like to thank Ouri Wolfson for discussions during his visit to Austin.

References

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

[2] Lloyd J.W. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

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

[4] 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.

[5] 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.

[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] Dong G. On Distributed Processibility of Datalog Queries by Decomposing databases. *Personal Communication.*

[8] Houtsma M.A.W. et al. A Logic Query Language and its Algebraic Optimization for a Multiprocessor Database Machine, *University of Twenete*, December 1988.