Mapping Datalog Program Execution to Networks of Processors

Sumit Ganguly, Abraham Silberschatz, Member, IEEE, and Shalom Tsur

Abstract — The problem of mapping the parallel bottom-up execution of Datalog programs to an interconnected network of processors is studied. The parallelization is achieved by using hash functions that partition the set of instantiations for the rules. We first examine this problem in an environment where the number of processors and the interconnection topology is known, and communication between program segments residing at non-adjacent processors is not permitted. An algorithm is presented that decides whether a given Datalog program can be mapped onto such an architecture. We then relax the constraint on the architecture by allowing program segments residing at non-adjacent processors to communicate. A theory of approximate mappings is developed, and an algorithm to obtain the closest approximate mapping of a given Datalog program onto a given architecture is presented.

I. INTRODUCTION

The efficient bottom-up evaluation of queries in a deductive database, defined by Datalog programs, has been an active area of research [15], [3], [7]. 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 a number of different authors [16], [17], [5]. The general paradigm of parallelization on which the above work is based is to take a sequential evaluation and divide the work among the processors by using hash functions.

In the context of Datalog, the idea of using hash functions to partition the work between the processors was first proposed for a subset of Datalog programs in [16], [17]. The above idea was then extended into a parametric framework for the parallel evaluation of general Datalog programs in [5], [18]. In the context of general rule based systems, work along similar lines may be found in [9], [10].

All previous work on this topic assumed that the architecture on which the parallel program is to be executed allows every processor to communicate directly with every other processor through message passing or through shared memory. No work has been done in addressing the problem of mapping the evaluation of Datalog programs when the underlying architecture does not satisfy the above property. In other words, the parallel algorithms developed thus far are not applicable, in general, to a system where the underlying architecture has a specific topology which makes communication between non-adjacent processors either impossible or highly expensive. We note, however, that there has been some work in the area of characterizing classes of Datalog programs which can be evaluated in parallel without any communication, [16], [17], [8].

In this paper we develop a theory for mapping the bottom-up execution of constant-free Datalog programs onto architectures that satisfy the following two properties:

1) The number of processors and the interconnection topology between the processors is known. For sake of generality, we assume that the interconnection topology can be represented as a directed graph between the processors.

2) A processor can communicate with only its immediate neighbors with respect to the interconnection topology. Thus, processors that are not connected by a direct interconnection link are not allowed to communicate with each other.

Our work is based upon [5], where we presented a parallelization scheme for the parallel execution of Datalog programs, which has the following three desirable properties:

1) It is parametric, in the sense that the parallelization strategy is controlled by some input parameters. Thus, for a given program, we can choose different values of the parameters to obtain different parallel algorithms to evaluate the same Datalog program.

2) It is non-redundant, in the sense that in the resulting parallel execution, different processors do not duplicate the computation of each other.

3) Each processor works on the same Datalog program, but on a smaller input data set.

The parallelization scheme takes as input a sequential Datalog program \( R \) and a set \( P = \{P_1, \ldots, P_n\} \) of processors available for execution. The output is a set \( Q = \{Q_1, \ldots, Q_n\} \) consisting of \( n \) sequential Datalog programs. Each \( Q_i \) is to be run on processor \( P_i \). The parallel execution of the \( Q_i \)'s is equivalent to the sequential execution of \( R \).

We have assumed in [5] that all the processors in the set \( P \) can communicate directly with each other. In this paper we assume that the interconnection topology is more restricted. Given this restriction, we closely examine its ramifications on our previous approach. Our main results can be summarized as follows: we show that the choice of every parameter in the parallelization strategy presented in [5] requires a certain minimal connectivity among the subprograms. Furthermore,
we design an algorithm, which when given a choice of the parameters, predicts the minimal interconnections required among the subprograms to make the parallel algorithm work correctly. Moreover, we develop another algorithm for deciding whether or not, for a given program and a processor network, the program can be transformed into a collection of Datalog programs subject to the above restrictions. If the answer is yes, then the algorithm outputs the transformed program as well. A key observation that makes the design of such an algorithm possible is the symmetry among the required interconnections among the subprograms. We make use of the symmetry property to develop a simple notation to express the interconnections between subprograms for all values (infinite) of the parameters. This notation naturally yields a decidable solution to the mapping problem, which is formulated in terms of graph homomorphism.

Given that not every Datalog program $R$ can be transformed into a collection of programs $Q$ subject to the above constraints on the architecture, we then relax the constraints and allow non-adjacent processors to communicate via intermediate processors. We develop a theory for approximate mappings and design an algorithm for finding the closest approximate mapping of a given Datalog program on the architecture.

The remainder of the paper is organized as follows. In Section II, we review the basic concepts from Logic Programming and also present our parallelization strategy [5]. In Section III, we show how parallelization parameters determine network topologies and give an algorithm to compute them. In Section IV we develop the theory of mapping Datalog programs onto networks of processors and in Section V we develop its approximate counterpart. We present our conclusions in Section VI.

II. PRELIMINARIES

In this section we review the parallelization scheme as presented in [5]. In order to do so, we first present the basic notation and theory of Logic Programming that we will be using in this paper.

A. Basic Notation

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

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

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

The relation $\text{par}$ above is an extensional relation, where $\text{par}(X, Y)$ means that $X$ is the parent of $Y$. The relation $\text{anc}$ above is a derived relation, where $\text{anc}(X, Y)$ means that $X$ is an ancestor of $Y$. The first rule states that if $X$ is a parent of $Y$, then $X$ is an ancestor of $Y$. The second rule recursively states that, if $X$ is a parent of $Z$, and $Z$ is an ancestor of $Y$, then $X$ is an ancestor of $Y$.

An input to a program $R$ is a relation for each base predicate. An output of $R$ is a relation for each derived predicate of $R$. The declarative semantics for the output is the smallest model satisfying $R$ that contains the input relations. Let $p$ be a predicate defined in a program $R$ and let $D$ be the input database. Then $Mp(R, D)$ denotes the extension of $p$ in the least model of $R$ when evaluated over the database $D$. A predicate $Q$ in a program $R$ is recursive if it occurs in the body of a rule whose head is an $R$-atom. $Q$ is recursive if the predicate in its head transitively derives some predicate in its body. The theory of logic programming is comprehensively treated in [1], [6].

B. Overview of Parallelization Strategy

The parallelization strategy used in this paper is centered around the semi-naive evaluation scheme of Datalog programs [2]. The basic step in this scheme 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.

Let $R$ be a Datalog program whose rules are numbered from 1 to $k$, in some order. This program is to be executed on a set of processors $P$. For each rule $r_i \in R$ we choose a non-repetitive sequence of variables, called discriminating variables, all of which appear in the body of $r_i$. The discriminating
variables of rule \( r \), are denoted by \( \nu(r) \). Also, for each \( r \), we choose a \textit{discriminating function} \( h \), defined as follows:

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

Given a Datalog program \( R \), a set of Datalog programs to be executed at various processors is derived. The parallel execution of this rewritten set of programs is equivalent (that is, produces the same answer for every input) to the sequential execution of \( R \). Let \( Q \), denote the program to be executed at processor \( P \). We will introduce new predicate symbols whose intended meanings are explained below.

1) For every derived predicate symbol \( t \in R \) and for every \( P \in P \) we introduce a new predicate symbol \( t^\prime \). The intended meaning of this predicate is the set of all the \( t \) tuples that are generated at processor \( P \).

2) For every derived predicate symbol \( t \in R \) and for every \( P \in P \) we introduce a new predicate symbol \( t^\prime \). The intended meaning of this predicate is the set of all the \( t \) tuples that are input to processor \( P \) at some point in the execution.

3) For every derived predicate symbol \( t \in R \) and for every \( P \) and \( P \in P \) we introduce a new predicate symbol \( t^\prime \). This predicate represents the set of the \( t \)-tuples transmitted from processor \( P \), to processor \( P \).

If \( A \) is any atom with predicate symbol \( t \), then \( A^\prime \) is the atom with the predicate symbol \( t^\prime \) and the same arguments. Thus, for example, if \( A \) denotes the atom \( sg(U, V) \), then \( A^\prime \) denotes the atom \( sg^\prime(U, V) \). Likewise, we define \( A^\prime \) and \( A^\prime \).

The program to be executed at processor \( P \), denoted by \( Q \), consists of the following four steps:

1) \textbf{Processing.} Let \( A : B, \ldots, C \) be a rule \( r \) in the given Datalog program \( R \), with discriminating sequence \( \nu \) and discriminating function \( h \). Then, include the following rule in \( Q \):

\[
A^\prime = B^\prime, \ldots, C^\prime, h_\nu = t.
\]

2) \textbf{Sending.} Let \( r \) be a rule in \( R \), with discriminating sequence \( \nu \) and discriminating function \( h \). For every recursive atom \( C \) appearing in \( r \) and every \( P \) in \( P \), include the following rule in \( Q \):

\[
C = -C^\prime, h_\nu = j.
\]

3) \textbf{Receiving.} Let \( \bar{W} \) be a sequence of all distinct variables not appearing in the original program. For every recursive predicate \( t \) appearing in the program \( R \) and every \( P \) in \( P \), introduce the following rule in \( Q \), where \( S \) denotes \( t(\bar{W}) \). (Hence, \( S^\prime \) denotes \( t^\prime(\bar{W}) \)).

\[
S^\prime = -S.
\]

4) \textbf{Final Pooling.} Let \( S \) be as defined in the receiving step above. For every recursive predicate \( t \), include the following rule in \( Q \).

\[
S : -S^\prime.
\]

\[\]

C. Uniform Discriminating Functions

The basic question that we examine in this paper is whether, for a given a Datalog program and a processor network, it is possible to choose the discriminating variables and the discriminating functions in such a manner that the resulting parallel execution can be mapped onto the network. Ideally, we would like to consider this question independently of the input database. Unfortunately, it is possible in this case to have a situation where the set of discriminating functions chosen performs skewedly on many input databases resulting in poor performance. For example, an answer to the above basic question for a particular Datalog program could be \( h(x) = x \mod 5 \). The performance of an algorithm using such a discriminating function is very susceptible to skews in the data. Such a case might occur if, for example, the database contained many more multiples of five (giving \( h(x) = 0 \)) than all the other possible values put together. In general, schemes that use predetermined hash functions cannot perform as well when compared to a scheme that chooses a hash function based on the distribution of the data. Hence, from a performance point of view, it is best to leave at least one degree of freedom to the runtime system.

We approach this problem by restricting the class of discriminating functions that are considered. Intuitively, we let the discriminating functions be constructed in two stages: 1) The \textit{combination functions} and 2) the \textit{basic discriminating function}, with the combination functions instantiated at \textit{compile time} and the basic discriminating function instantiated at \textit{run time}. Thus, the runtime system is left with one degree of freedom, namely the choice of the basic discriminating function to control performance. This class of discriminating functions, called \textit{uniform}, is defined below.

\textbf{Definition 1:} Let \( \mathcal{H} = \{ h_1, \ldots, h_n \} \) be a set of discriminating functions where the arity of \( h_i \) is \( a_i \) for \( 1 \leq i \leq n \). We say that \( \mathcal{H} \) is a uniform class of discriminating functions if the following two conditions are satisfied:

1) There is a fixed natural number \( n \) and a unary discriminating function \( h : \text{constants of the database } \rightarrow \{ 1, \ldots, n \} \).

This unary discriminating function is called the \textit{basic discriminating function}.

2) For every \( h_i \in \mathcal{H} \), there exists a function \( f_i \) of arity \( a_i \) such that:

\[
h(x_1, \ldots, x_{a_i}) = f_i(h(x_1), \ldots, h(x_{a_i})).
\]

For every permissible value of the \( x_i \)'s, these \( f_i \)'s are called the \textit{combination functions}. Without loss of generality we will assume that the constants in the database
are natural numbers. In such a case the $x_i$'s may be thought of as arbitrary natural numbers.

We are now in a position to precisely state the key question that we examine in this paper, which is: given a Datalog program and a processor network, is it possible to choose the discriminating variables and uniform discriminating functions in such a manner that the resulting parallel execution can be mapped onto the network, for every choice of the input database and every choice of the basic discriminating function?

We use the following notation throughout the remainder of the paper. Let $R$ be a given Datalog program with $k$ rules numbered from 1 to $k$. The discriminating sequence for rule $r$ is denoted by $\nu_r$, and the vector of discriminating variables, one for each rule, is denoted by $DV$. Thus, $DV$ is the discriminating sequence of the $i$th rule in $R$. The basic discriminating function whose range is the set $\{1, \ldots, n\}$ is denoted by $h$. We select a combination function for every rule $r$, which is denoted by $f_r$. The vector of combination functions, one for each rule is denoted by $F$. The rewriting strategy reviewed in Section II A assigns a Datalog program $Q_i$ to processor $P_i$. The union of these $Q_i$'s is denoted by $Q$, and in order to make its dependence on $h$, $F$, $DV$, and $n$ explicit, we denote it by $Q(h, F, DV, n)$.

D. Datalog Subclass

The subclass of Datalog programs to which the results in this paper are applicable to is characterized by the following two properties:

1) Programs are constant-free.
2) Predicates appearing in the head of a rule do not have duplicate variables (that is, $p(X, X)$ may not appear in the head of a rule).

Property 2 above does not restrict the class of Datalog programs that we can deal with. In fact, all Datalog programs can be rectified, as shown in [15]. Property 2 above is a special case of rectified programs. In addition, without loss of generality, it may be assumed that variables in different rules have different names. Hence, our results are applicable to the class of constant-free Datalog programs.

III. NETWORK TOPOLOGY DETERMINATION

In this section we show that the choice of the parallelization parameters leads to parallel executions whose communication patterns are very well defined and computable. This implies that in a given processor network, it may not be necessary for every processor to communicate with every other processor, as long as the communication patterns are preserved.

Given a set of processors $P$, we define a processor network 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 $P_i$ to processor $P_j$ is permissible. The absence of a directed edge from $P_i$ to $P_j$ indicates that processor $P_i$ may not communicate with processor $P_j$, either directly or indirectly. Hence routing of information from $P_i$ to $P_j$ via other intermediate processors is not permitted during the parallel execution.

We now formally define the communication pattern that arises due to a choice of the parallelization parameters.

**Definition 2:** Let $R$ be a Datalog program and let $F$, $DV$, and $n$ be a specific choice of the combination function vector, the discriminating sequence vector, and the range of the basic discriminating function, respectively. The network graph $H(N, F, DV, n)$ has the vertex set $V = \{v \mid v \in range(f)\}$ for some combination function $f$ in the vector $F$. There is an edge from one vertex $i$ to another vertex $j$ in $V$, if, for some derived predicate symbol $t$ in $R$, $M(t)(i)(Q(h, F, DV, n)) \neq \phi$ for some choice of the basic discriminating function $h$ and the input database $D$.

**Definition 2** does not allow us to construct the network graph for a given $F$, $DV$, and $n$, since the choices of the input databases could be infinite.

We evaluate $Q$ on a specially constructed database with the property that if the predicate $t$, is empty during the evaluation of $Q(id, F, DV, n)$, then it is empty after the evaluation of the program on any database. This database is called the complete hash database and is formally defined in Definition 3 below.

**Definition 3:** Let $n$ be a natural number $\geq 1$. The complete hash database for a given program $R$ denoted by $CHD(n)$ assigns to each $k$-ary base predicate $r$ the following relation:

$$r = \{r(a_1, \ldots, a_k) \mid 1 \leq a_i \leq n, 1 \leq i \leq k\}$$

**Theorem 1:** Let $t$ be an arbitrary derived predicate symbol of $Q(id, F, DV, n)$ and let $id$ denote the identity function (that is, $id(x) = x$). Then,

$$M(t)(Q(id, F, DV, n), CHD(n)) = \phi \iff \forall h, DB(M(t)(Q(h, F, DV, n), DB) = \phi$$

**Proof:** See Appendix A.

**begin**

Evaluate $Q(id, F, DV, n)$ on the data base $CHD(n)$.

for every derived predicate symbol $t$ do

if $t \neq \phi$ then introduce an arc from processor $P_i$ to processor $P_j$

endfor

**end.**

Fig. 1. Algorithm 1, for computing a network graph.

**Theorem 1** can be used as the basis for an algorithm, shown in Fig. 1, that computes the network graph. For a given $F$, $DV$, and $n$, suppose that we evaluate $Q$ using the identity function as the basic discriminating function and the complete hash database as input. If in this evaluation, a predicate $t$ is found to...
be empty, then for every choice of \( h \) and input database, \( t \)
would be empty (and vice versa).

**Example 1:** Consider the following Datalog program:

1. \( r_1: \text{sg}(X, Y) \leftarrow \text{flat}(X, Y) \)
2. \( r_2: \text{sg}(X, Y) \leftarrow \text{up}(X, U), \text{sg}(U, Y) \)
3. \( r_3: \text{sg}(X, Y) \leftarrow \text{up}(X, V), \text{flat}(V, Y) \)

Let \( n = 2 \), (that is, the range of the basic discriminating function
is \( \{0, 1\} \), \( DV = \{(X, Y), (U, V), (X, V)\}\) and
\( F = (id, id, id) \). Thus, the set of processors \( P = \{P_{oo}, P_{oa}, P_{oo}, P_{oa}\} \). If Algorithm 1 is used for this case, we get a network graph which is shown in Fig. 2.

![Network graph](image1)

**Fig. 2. Network graph**

Consider the following Datalog program:

\[
\begin{align*}
\text{Example 1:} & \quad \text{sg}(X, Y) \leftarrow \text{flat}(X, Y), \\
& \text{sg}(X, Y) \leftarrow \text{up}(X, U), \text{sg}(U, Y), \\
& \text{sg}(X, Y) \leftarrow \text{up}(X, V), \text{flat}(V, Y).
\end{align*}
\]

Now suppose that we change \( DV \) without changing \( n \) and \( F \).
Let \( DV = \{(X, Y), (X, U, V), (X, V, Y)\} \). Notice that in the for loop in Algorithm 1, we derive a network for every predicate symbol in \( R \) and then superpose these networks. Thus, Algorithm 1 can be further conceptually decomposed to obtain network graphs corresponding to each derived predicate, such that the final network graph is the superposition of each of these networks. If we ignore the communication resulting from the computation of the exit rule, we obtain the following network graphs for the predicate \( \text{sg} \) and \( s \) which are shown in Fig. 3.

We are now in a position to introduce the notion of mapping a Datalog program to a processor network.

**Definition 4:** We say that a Datalog program \( R \) can be
mapped onto a processor network \( N = (V, E) \) if for some \( F, DV \) and \( n, N = (V, E) \) is a homomorphic image of \( NG(F, DV, n) \).

Since the vertices of a network graph consist of restricted
copies of the program (also referred to as sub-programs), a mapping of a network graph onto a processor network implies
that in the parallel evaluation of the program, each processor
executes an entire sub-program. The edge preserving property
implicit in the notion of homomorphism implies that for every
choice of the basic discriminating function and the input data-
base, there will be no necessity for non-adjacent processors to
communicate.

**IV. ALGORITHM FOR MAPPING**

In this section, we develop an algorithm for deciding
whether a given Datalog program \( R \) can be mapped onto a
given processor network \( N = (V, E) \). That is, can the parallelization
parameters \( F, DV \), and \( n \) be chosen in such a manner
that there is a homomorphism from \( NG \) onto \( N \). The basic idea
behind the algorithm is that for a given choice of \( F \) and \( DV \), the set of network graphs \( \{NG(F, DV, n) \mid n \geq 1 \} \) can be represented concisely. We first discuss this concise notation called
transformations. We then discuss how network graphs can be
represented as transformations. Finally, we present the mapping
algorithm.

**A. Transformations**

**Definition 5:** An \( m \) to \( n \) transformation \( \tau \), where \( m \) and \( n \) are
natural numbers, is denoted by a sequence \( [X_1, X_2, \ldots, X_n] \)
where each \( X_i \in \{1, \ldots, m\} \) or \( X_i \) is the special symbol \( T \).

The meaning of the transformation is given below.

**Definition 6:** An \( m \) to \( n \) transformation \( \tau \), where \( m \) and \( n \) are
natural numbers, maps a sequence of length \( m \) over a finite set
\( D \) to sets of sequences of length \( n \) over \( D \). Let
\( \tau = [X_1, X_2, \ldots, X_n] \), and let \( \langle s_1, s_2, \ldots, s_m \rangle \) and \( \langle t_1, t_2, \ldots, t_n \rangle \)
denote sequences of lengths \( m \) and \( n \), respectively, over \( D \).
Then, \( \langle t_1, t_2, \ldots, t_n \rangle \) if one of the following conditions hold:

1) \( X_i \in \{1, \ldots, m\} \) and \( t_i = s_i \); 
2) \( X_i = T \) and \( t_i \in D \).

A transformation concisely represents a family of graphs
defined as follows.

**Definition 7:** A transformation \( \tau \) defines an infinite family
of graphs \( \bar{G}_\tau = \{G_1, (1), G_2, (2), \ldots\} \) as follows:

- **Vertex Set of** \( \bar{G}_\tau(k) = \{1, \ldots, k\}^m \cup \{1, \ldots, k\}^n \). 
- **There is an edge from** \( s \in \{1, \ldots, k\}^m \) to \( t \in \{1, \ldots, k\}^n \) if \( t \in \tau(s) \).

**Example 2:** Consider the 3 to 3 transformation
\( \tau = [3, 1, T] \). Let \( D = \{1, 2, 3\} \).
Then, \( \pi((1, 2, 3)) = \{(3, 1, 1), (3, 1, 2), (3, 1, 3)\} \). The graph \( G_1(2) \) is shown in Fig. 4. Let \( \tau_1 = [1, 3, \top] \) and \( \tau_2 = [\top, 1, 2] \). Then, the network graphs in Fig. 3 may be described as \( G_{\tau_1}(2) \) and \( G_{\tau_2}(2) \), respectively.

![Network graph for \( \tau = [1, 1, \top] \).](image)

We now illustrate how network graphs can be represented as transformation graphs through an example.

**Example 3:** Consider the following Datalog program consisting of two rules labeled \( s \) and \( r \) respectively.

\[
\begin{align*}
\text{s:} & \quad \pi(X, Y) := \rho(X, Y), \\
\text{r:} & \quad \pi(X, Y) := \rho(X, Z), \rho(Z, Y).
\end{align*}
\]

Let \( \pi = (X, Y) \), \( \rho = (X, Y) \) and let the combination functions be the identity functions. Suppose a tuple \( \pi(a, b) \) is generated from rule \( s \) by a processor. Clearly, then \( \rho(a, b) = \pi(a, b) \). This tuple is sent to all processors processing rule \( r \) which can use it. A processor can use a tuple if it can match any one of the \( \pi \)-atoms in the body. Let us derive the condition when the first recursive atom, namely \( \pi(X, Z) \) can use a tuple generated at processor \( (c, d) \). A processor \( (c, d) \) uses all those substitutions where \( \rho(c) = c \) and \( \rho(d) = d \). Hence, if a tuple \( \pi(a, b) \) can match against \( \rho(X, Z) \), then \( \rho(X) = c = \pi(a) \). No condition is imposed on \( Z \). This can be simply expressed as the transformation \([1, \top]\). Similarly, matching the second recursive atom gives the transformation \([\top, 2]\).

The pair of transformations \([1, \top]\) and \([\top, 2]\) represent the network graph family, \( \{NG(F, DV, n) \mid n \geq 2\} \), where \( F \) is chosen as identity functions and \( DV \) is chosen as described above. For the same \( DV \), if \( F \) is chosen to be any other function, then the resulting network graph is a homomorphic image of the network graph family, when \( F \) is the identity function. □

Communication between the processors occurs because one processor may generate a tuple (that is, at the head) that the other processor needs (that is, in the body). Thus, there is a possibility of communication between processors resulting from every pair of predicates with the same symbol, one appearing in the head of a rule and the other in the body of a rule (possibly the same). The entire communication network is simply the union of such pair-wise communication. Example 3 points to the basic property that each pair-wise communication can be represented as a transformation.

We now show how to represent network graphs using a set of transformations. For brevity, given an atom \( C \), the term occupying the \( j \)th argument position of \( C \) is denoted by \( C[I] \), and the \( l \)th member of a sequence \( s \) is denoted by \( s[l] \).

**Definition 8:** For every predicate \( A \) that appears in the head of some rule \( r \in R \), and every predicate \( B \) that appears in the body of some rule \( s \in R \) such that \( A \) and \( B \) have the same predicate symbol, we construct a \( [v] \) to \( [v] \) transformation \( \tau_{A,B} \) as follows:

\[
\tau_{A,B}[k] = \begin{cases} 
1, & \text{if for some } l, \\
\pi[l] = A[I] \text{ and } \\
\theta[l] = v[k] \text{ or } \top, & \text{otherwise}
\end{cases}
\]

If \( A \) and \( B \) are predicates that do not share the same predicate symbol, then \( \tau_{A,B} = \phi \).

For a given Datalog program \( R \) and discriminating sequence vector \( DV \), the set of transformations \( \{\tau_{A,B}\} \) is denoted by \( G \). A family of graphs, one for each natural number \( n \geq 1 \), denoted by \( \{G(n)\} \) is defined by \( G \) as follows:

- For every rule \( r \) in the program \( R \) with discriminating sequence \( [v] \), the set of vertices is \( V = \{1, \ldots, n\}^{[v]} \) is associated. Thus, every vertex in the graph is uniquely associated with a rule.
- For rules \( r, s \in R \), there is an edge from a vertex \( u \in V_r \) to a vertex \( v \in V_s \) if \( u \in \tau_{A,B} \), where \( A \) is the head of \( r \) and \( B \) appears in the body of \( s \).

Let us study the relationship of \( G(n) \) to a network graph \( NG(F, DV, n) \). In the network graph, the vertex set is \( V = \{v \mid v \in f(V_r)\} \). Hence, for every choice of the discriminating function vector, there is an onto mapping from \( V \) to the vertex set of the network graph which is defined by \( f \). The following theorem shows that this mapping is a homomorphism.

**Theorem 2:** Let \( G = \{G(1), G(2), \ldots\} \) be a family of graphs corresponding to a program \( R \) and discriminating vector \( DV \). Let \( F \) be a discriminating function vector. Then, for all \( n \geq 1 \), \( G(n) \) can be homomorphically mapped onto \( NG(F, DV, n) \) under the vertex mapping induced by \( F \).

**Proof:** See Appendix B. □
The problem of mapping a Datalog program onto a network of processors is to find \( F, DV \), and \( n \) such that \( NG(F, DV, n) \) can be homomorphically mapped onto the network of processors. Since, graph-homomorphism is transitive, it follows from Theorem 2, that the mapping problem is solved, if we can find a homomorphic mapping from \( g(n) \) onto the network of processors, for some \( n \). Additionally, the homomorphic mapping should be an onto map from the set of vertices associated with each rule (denoted by \( V \)) to the set of processors.

Since, the choices of \( n \) are infinite, the observation above does not yield a decidable procedure. The following theorem makes this possible by allowing us to check for only one value of \( n \).

**Theorem 3:** Let \( N = (V, E) \) be a processor network and \( g \) represent a family of graphs defined by a set of transformations. \( N \) is a homomorphic image of some member of \( g \) that preserves onto-ness of each \( V \), if and only if it is a homomorphic image of \( g(p) \) that preserves onto-ness of each \( V \), where \( p = |V| \times c \) and \( c \) is a constant that depends on \( g \) alone.

**Proof:** See Appendix C. □

**Input:** A constant-free and arity complete Datalog program \( R \).

**Output:** A decision whether \( R \) can be mapped onto \( N \).

If the answer is affirmative, then the parameters are output.

```plaintext
begin
   possible := false;
   DV := Init-discr-seq(nil);
   While not possible and DV ≠ nil do
      T := Construct-transformation;
      if Homomorph(N, T) then
         possible := true;
         print("Mapping Possible");
      endif
      DV := next-discr-sequence(DV);
   endwhile
   if not possible then
      print("Mapping not possible");
   endif
end
```

**Fig. 5 Algorithm 2.**

We now present an algorithm for deciding whether a given constant-free and arity complete Datalog program can be mapped onto a given processor network. In case of an affirmative answer, the algorithm also outputs the specific choice of the combination function vector \( F \), the discriminating sequence vector \( DV \), and the range of the basic hash function. This algorithm, referred to in the sequel as Algorithm 2, is depicted in Fig. 5, and uses the following four sub-routines:

- Init-discr-sequence\((R)\) and its companion next-discr-sequence\((R, DV)\), where \( R \) and \( DV \) are the given program and the discriminating variable, respectively, are subroutines that enumerate the set of all possible vectors of discriminating sequences, one for each rule in \( R \). We initialize the vector \( DV \) by invoking Init-discr-sequence\((R)\) and then repeatedly invoke next-discr-sequence\((R, DV)\) to get a new vector \( DV \) until either a mapping is found or all possible choices of the vector have been exhausted.
- The procedure Construct-transformation constructs the set of transformations \( T \) that represent the family of network graph as discussed earlier in this section.
- The procedure Homomorphic\((N, T)\) determines if \( N \) is a homomorphic image of one of the members of \( T \).

The correctness of Algorithm 2 follows directly from Theorems 2 and 3. The fact that the algorithm always terminates follows from the following arguments. Each of the four subroutines used in the algorithm terminates. Furthermore, the number of iterations of the while loop is bounded by the number of possible choices of the discriminating variable vector, which is finite. Hence the algorithm terminates on all inputs.

V. APPROXIMATE MAPPINGS

As discussed in the previous sections, there exist Datalog programs that cannot be mapped onto a processor network subject to the restriction that non-adjacent processors do not communicate. However, if non-adjacent processors are allowed to exchange data via intermediate vertices, then it is intuitively clear that any program can be executed on such a network by using any mapping of the subprograms to the processor nodes. Thus, in the presence of communication via routing, we need to devise a metric to measure the "goodness" of the mapping, or how closely the mapping approximates a processor network. For our purpose, a mapping will be considered to be good if communicating pairs of subprograms are mapped onto processor nodes that are close by, if not adjacent.

We first quantify the intuition of approximate mappings.

**Definition 9:** A Datalog program \( R \) is said to be approximately \( m \)-mappable onto a processor network \( N = (V, E) \), if for some choice of \( F, DV \), and \( n \), there is a 1-1 onto function \( f \) from the set of vertices of the network graph \( NG(F, DV, n) \) to \( V \) such that whenever there is an edge between two vertices \( u \) and \( v \) in \( NG \), the shortest path between \( f(u) \) and \( f(v) \) is at most \( m \). □

The reader may have noticed that the notion of mapping as defined in the previous section corresponds to an approximately I-mapping. It is intuitively clear that a Datalog program can always be approximately mapped onto a connected network. That is, it can be shown that there is an \( m \geq 0 \) such that \( R \) is approximately \( m \)-mappable onto \( N \). Thus, an interesting problem would be to find the smallest \( m \) such that \( R \) is approximately \( m \)-mappable onto \( N \) and the values of the parameters \( F, DV \), and \( n \) that makes such a mapping possible. A smaller \( m \) would imply less hops for intermediate routing and a larger \( m \) would imply large hops for intermediate routing.

Similar to notion of approximate \( m \)-mappings, we define the notion of approximate \( m \)-homomorphisms. We say that a
graph \( G(U_1, E_1) \) can be approximately \( m \)-homomorphically mapped onto \( H(U_2, E_2) \) if there exists an onto function \( f \) from \( U_1 \) to \( U_2 \) such that for every \((u, v) \in E_1\), the shortest distance between \( f(u) \) and \( f(v) \) in \( H \) is at most \( m \).

**Theorem 4:** Let \( N = (V, E) \) be a processor network and \( \mathcal{G} \) represent a family of graphs defined by a set of transformations. Then \( N \) is an approximate \( m \)-homomorphic image of some member of \( \mathcal{G} \) if and only if it is an \( m \)-homomorphic image of \( G_p \) where \( p = |V| \times c \), where \( c \) is a constant that depends on \( \mathcal{G} \) alone.

**Proof:** See Appendix D.

Since approximate homomorphisms are now needed in place of homomorphisms, the only change to Algorithm 2 required is that the procedure for checking Homomorphism should be replaced by a procedure for approximate-m-homomorphism.

Theorem 4 provides us with a computable procedure for finding the smallest \( m \) such that a given Datalog program is approximately \( m \)-mappable onto \( N = (V, E) \). The algorithm is presented in Fig. 6.

**APPENDIX A**

In this appendix we present the proof of Theorem 1. To so do, we must first present some notation necessary to proceed further.

Let \( A \) denote the atom \( p(t_1, \ldots, t_n) \). Then \( h(A) \) denotes the atom \( p(h(t_1), \ldots, h(t_n)) \). (Recall that \( h \) denotes the basic discriminating function.)

**Lemma A.1:** Let \( A \) be any atom and \( q \) a ground substitution. Then \( h(Aq) = A(hq) \) if \( A \) has no constants in its argument positions.

**Proof:** Let \( A = \pi(x_1, \ldots, x_n) \) and \( \theta = \{x_1/c_1, \ldots, x_n/c_n\} \). Then, \( h(A\theta) = h(\pi(x_1, \theta, \ldots, x_n, \theta)) = h(\pi(c_1, \ldots, c_n)) = h(\pi(h(c_1), \ldots, h(c_n))) = h(\pi(h(x_1), \ldots, h(x_n))) = A(h\theta). \)

**Lemma A.2:** Let \( R \) be a given constant free Datalog program and let \( D \in M(Q(h, F, DV, n), DB) \). Then \( h(D) \in M(Q(id, F, DV, n), h(DB)) \) where \( id \) is the identity function.

**Proof:** We first note that if \( R \) is constant free, then so is the transformed program. The proof is by induction on the height of derivation.

- **Base Case:** Suppose \( D \in \mathcal{T}(Q(h, F, DV, n), DB) \). Then there exists a processing rule \( r \) of the form
\[ A : = B, \ldots, C, \text{send}(h(X_1), \ldots, h(X_n)) = m \] in \( Q(h, F, DV, n) \) and a substitution \( \theta \) such that \( B \theta \ldots, C \theta \) are true in the database and \( f_i(h(X_1, \theta), \ldots, h(X_n, \theta)) = m \). The corresponding rule in \( Q(id, F, DV, n) \) is: \( A : = B, \ldots, C, \text{send}(h(X_1), \ldots, X_n) = m \).

By definition of \( h(DB) \), if \( B \theta \) is true in \( DB \) then \( h(B \theta) = h(\theta \theta) \) is true in \( h(DB) \). Also, \( f_i(h(X_1, \theta), \ldots, h(X_n, \theta)) = m \). Thus, \( h(\theta) \) is a valid substitution for this rule in \( Q(id, F, DV, n) \). Thus, \( A(\theta) = h(A(\theta)) = h(DB) \in T(Q(id, F, DV, n), DB) \)

**Induction Case:** The proof technique for the induction case is identical to that used for the base case. Let \( D \in T(Q(h, F, DV, n), DB) \). Then \( D \in T(Q(h, F, DV, n), DB) \).

1) By instantiating a sending rule.

Then there exists a processing rule \( r \) of the form \( A : = B, \ldots, C, \text{send}(h(X_1), \ldots, h(X_n)) = m \) in \( Q(h, F, DV, n) \) and a substitution \( \theta \) such that \( B \theta \ldots, C \theta \) are true in the database or in \( T(Q(id, F, DV, n), DB) \). Also, \( f_i(h(X_1, \theta), \ldots, h(X_n, \theta)) = m \). By the induction hypothesis, each of \( B(\theta) \ldots, C(\theta) \) is true either in \( h(DB) \) or in \( T(Q(id, F, DV, n), DB) \). As in Lemma 1, we can show that \( f_i(h(X_1), \ldots, X_n)(h(\theta) = m) \). Hence \( h(\theta) \) is a valid substitution for this rule in \( Q(id, F, DV, n) \). Thus, \( A(\theta) = h(A(\theta)) = h(DB) \in T(Q(id, F, DV, n), DB) \).

2) By instantiating the sending rule. Then there exists a sending rule of the form \( C_{r,\theta, \xi} = -C^{r, \theta}_{\text{send}}, h(v_i) = z \) if and only if \( \phi \).

**Theorem 1:** Let \( t \) be an arbitrary derived predicate symbol of \( Q(n, h, DV, F) \). Then

\[
M_t(Q(id, F, DV, n), \text{CHD}(n)) \neq \phi \iff \exists h, DB \ (M_t(Q(h, F, DV, n), DB) \neq \phi)
\]

**Proof:** Let \( A \) be a ground \( t \)-atom and let \( A \in M(Q(id, F, DV, n), DB) \). Then \( A \in M(Q(id, F, DV, n), h(DB)) \) by Lemma 2. Further \( h(DB) \in \text{CHD}(n) \). By monotonicity of the least fixpoint operator \( M \), \( A \in M(Q(id, F, DV, n), \text{CHD}(n)) \). Conversely, let \( M_t(Q(id, F, DV, n), \text{CHD}(n)) \neq \phi \). So let \( h = \text{id} \) and \( DB = \text{CHD}(n) \).

**APPENDIX B**

In this section, we present a proof of Theorem 2. In order to do so, we need to establish the following lemmas:

**Lemma B.1:** Let \( A \) be the head of rule \( r \) with discriminating sequence \( v_i \) and \( B \) be a predicate in the body of rule \( s \) with discriminating sequence \( v_j \). Then \( z \in \tau_{x, y} (x) \) if and only if there is a successful substitution of the conjunction \( A \land B \) such that \( v_i = x, v_j = z \).

**Proof:** Let \( 1 \leq k \leq l \), let \( A = \left( \bigcup_{k} \right) \) and \( B = \left( \bigcap_{k} \right) \). Suppose that for a choice of \( k \), there exists \( \ell \) such that \( U \{ k \} = v_\ell \) and \( V \{ k \} = v_\ell \). Then, for every successful substitution \( \theta \) of the conjunction \( A \land B \), \( v_i = x, v_j = z \). This corresponds to \( \tau_{k, \ell} \). Hence proved.

**Lemma B.2:** Let \( A \) be the head of a rule \( r \) and let \( D \) be a predicate with the same predicate symbol in the body of rule \( s \). If \( w \in \tau_{x, y} (u) \), then either \( f(u) = f(w) \) or there exists an edge between \( f(u) \) and \( f(w) \) in \( NG(F, DV, n) \), for every \( n \geq 1 \).

**Proof:** Suppose that \( w \in \tau_{x, y} (u) \). By Lemma B.1, there exists a successful substitution \( \theta \) of the conjunct \( v_i = u, v_j = w, A = D \).

Consider the following rules that define the communication from rule \( r \) to \( s \) due to the predicate pair \( A, D \).

\[
A_{n, x} = -A_{n, x}^*, v_\ell = u.
\]

\[
A_{n, y} = -A_{n, y}^*, v_\ell = w.
\]

\[
A_{n, y}^* = -D_{n, v}
\]

Clearly, \( \theta \) can be extended to a successful substitution \( \theta' \) of the rule above, since we have assumed that different rules never use the same variables. Since \( \theta \) is a successful substitution of rules 2 and 3, hence \( \theta' \) is also too. Hence \( \theta' \) is the \( \delta \) that we set out to construct. Hence, \( \theta' \) is a successful substitution of the following rules that define the communication between subprograms in the network graph due to the predicate pair \( A, D \), where \( i \) denotes \( f(u) \) and \( j \) denotes \( f(w) \).

\[
A_{n, x} = -B_{n, x}^*, \ldots, C_{n, v}^*, f(v_i) = i.
\]

\[
A_{n, y} = -A_{n, y}^*, f(v_j) = j.
\]

\[
D_{n, v} = -D_{n, v}
\]

Hence there is an edge between \( f(u) \) and \( f(w) \) in \( NG(F, DV, n) \).

**Proof of Theorem 2:** Let \( \tau \) be the set of transformations \( \{ \tau_{x, y} \} \) constructed for predicate pairs \( A, B \) in the Datalog program \( R \). By Lemma B.2, there is a homomorphism from the subgraph of \( G(n) \) defined by \( \tau_{x, y} \) onto \( NG(F, DV, n) \). Since \( G(n) \) is the union of all such subgraphs defined by \( \tau_{x, y} \), \( B \) for various predicate pairs \( A, B \), the result follows.
has \( k \) rules, such that the length of the discriminating sequence of the \( i \) th rule is \( a_i \). Let \( m \) be the largest of the \( a_i \)'s. Let \( p = |I| \times m \times k \). Suppose \( n > p \). The vertex set of \( G(n) \) can be partitioned into \( k \) sets, one for each of the rule given the program. The vertex set for the rule \( i \) is \( V_i = \{ 1, \ldots, n \}^k \). Let the mapping from \( G(n) \) to \( N = (V, E) \) be the set \( \{ f_1, \ldots, f_k \} \), that is, each of the \( f_i \) is an onto map for \( V_i \) onto \( V \).

Let \( D = \{ 0, \ldots, n-1 \} \). The powerset of any set \( S \) is denoted by \( \text{Powerset}(S) \). For each \( 1 \leq i \leq k \), let \( D \) denote the cartesian product of \( D \) with itself \( a_i \) times.

For \( 1 \leq i \leq k \), define two functions \( \alpha_i, \beta_i : D \rightarrow \text{Powerset}(V) \) as follows:

\[
\alpha_i(x) = \{ u \in V \mid \exists \in D, 1 \leq i \leq a \alpha_i(f_i) = u \text{ and } \{|f_i\} = x \} \\
\beta_i(x) = \{ u \in V \mid \forall \in D, 1 \leq i \leq a \alpha_i(f_i) = u \text{ and } \{|f_i\} = x \} 
\]

Let \( T_i = \{ x \mid V \neq \beta_i(x) \neq \phi \} \) and \( S_i = \{ x \mid \beta_i(x) = V \} \).

If \( u \in V \neq \beta_i(x) \) then every tuple in \( f_i^{-1}(u) \) contains \( x \). Hence there can be at most \( a_i \) distinct \( x \)'s such that \( a \in V \neq \beta_i(x) \). Thus, \( |T| \leq a_i \times |V| \) and therefore \( |S| \geq n - a_i \times |V| \). Hence, \( |S| \geq \frac{p - (\max a_i)}{v} \times |V| \). Hence, \( \cap_1 S_i \neq \phi \). Consequently, let \( a \in \cap_1 S_i \). Consider the subgraph \( H(n, a) \) of \( G(n) \) which does not contain the element \( a \) in any of its vertices. We will show that \( H(n, a) \) can be homomorphically mapped onto \( N = (V, E) \).

The vertices of \( H(n, a) \) is a subset of the vertices of \( G(n) \). Hence, a vertex of \( H(n, a) \) is mapped, quite naturally, onto the same vertex in \( N = (V, E) \) as it was mapped before by some \( f_i \). We denote this mapping by \( g_x \), for \( 1 \leq i \leq k \).

The edge preserving property is obvious, since originally, each of the \( f_i \)'s is edge preserving on the larger graph, and no new edges have been added to the graph. Also, \( H(n, a) \neq V \), for \( 1 \leq i \leq k \), and hence \( \alpha_i(a) \subseteq \beta_i(a) \). Hence \( g_x^{-1}(v) \neq \phi \), for every \( v \in V \). Further, \( H(n, a) \) is isomorphic to \( G(n - 1) \) and hence \( G(n - 1) \) can be homomorphically mapped onto \( N = (V, E) \).

If \( n \neq 0 \), the homomorphism is similar to the graph \( G(n) \) can be mapped onto \( N = (V, E) \), then so can \( G(n+1) \). This follows from the fact that \( G(n+1) \) can be first contracted to obtain \( G(n) \) which can then be mapped onto \( N = (V, E) \).

\[\square\]

**APPENDIX D**

In this section we present a proof for Theorem 4.

**Proof of Theorem 4:** The proof is virtually identical to the proof of Theorem 3, that is, \( p = (\max \alpha_i) \times k \times |V| \). Then, following the arguments in the proof of Theorem 2, we construct an element \( a \in D \) and set of functions \( g \). The property that the shortest path between any two sequences over \( D - \{a\} \) be of length no greater than \( m \) continues to be true since no new edges are added to \( G(p - 1) \) and no edges are deleted from the network \( N = (V, E) \). The balance of the argument proceeds as in the proof of Theorem 2. \[\square\]

**ACKNOWLEDGMENTS**

The research of Abraham Silberschatz was supported in part by the Texas Advanced Technology Program under Grant No. ATP-024, the US National Science Foundation under Grant Nos. IRI-9003341 and IRI-9106450, and grants from the IBM and Hewlett-Packard corporations.

**REFERENCES**


Sumit Ganguly received a PhD degree in computer science in 1992 from the University of Texas at Austin. Today, he is an assistant professor in the Department of Computer Sciences at Rutgers University in New Brunswick, N.J. His current interests lie in parallel and scientific databases.

Abraham Silberschatz received a PhD degree in computer science from the State University of New York, Stonybrook, in 1976. Today, he is an endowed professor in the Department of Computer Sciences of the University of Texas at Austin, where he specializes in the area of concurrent processing. His research interests include operating systems, database systems, and distributed systems.

Dr. Silberschatz joined the UT Austin faculty as an associate professor in 1980, earned the rank of full professor in 1984, and received the endowed professorship in 1988. Since January 1993, Silberschatz has been on leave at AT&T Bell Labs, where he is serving as a senior research scientist. He is active in the field of databases, and has served as program committee chair and general conference chair for a number of prestigious symposia in this field.

Shalom Tsur holds a PhD in computer science from the Hebrew University of Jerusalem, and MSc and BSc degrees in electrical engineering from the Technion-Israel Institute of Technology. At present, he is with the Center for High Performance Computing at the University of Texas, Austin.

Dr. Tsur's research interests include the theory, system issues, and applications of deductive, object-oriented (DOOD) database systems. Application areas of particular interest include scientific databases and problems involving a combination of procedural and declarative programming, for example, the financial engineering domain. He is the author or coauthor of 45 papers and one book in his research area.

Prior to joining the CHPC, Dr. Tsur was with MCC, where he was principally involved in the design of the LDL (Logical Data Language) deductive database system and its successor the LDL++ system. Before that, he held technical, faculty, and management positions at the IBM Scientific Center in Haifa, Israel, at AT&T Bell Labs, and at Pennsylvania State University.