SHARING THE LOAD OF LOGIC-PROGRAM EVALUATION
Extended Abstract

Ouri Wolfson
Computer Science Department
The Technion - Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

In this paper we propose a method by which multiple processors can share the load of evaluating an intentional-database relation. The relation is defined by (possibly recursive) logic-programs. The load sharing method does not introduce interprocess communication, or synchronization overhead. The method resembles the one introduced in [WS], except that it is less restrictive, and thus applicable to a larger class of logic programs. It includes, for example, all linear single rule programs. We also define a class of programs to which the method cannot be applied.
1. Introduction

The efficient evaluation of intentional database relations, defined by means of recursive logic programs, has recently emerged as a very active area of research ([U], [BR], [K]). Two main methods of improving performance have received most of the attention. One is selection propagation, and the other is parallel evaluation.

Selection propagation reduces the number of relevant input-database tuples, by using constants passed as parameters to the database query processor. This usually necessitates some rewriting of the logic program which defines the intentional relation. The best known rewriting algorithm for this purpose is "magic sets" (see [BMSU]).

Parallel evaluation uses multiple cooperating processors, to reduce the overall evaluation time from start to finish. Most effort in this area has been devoted to characterization of the logic programs which belong to the NC complexity class ([UV], [K], [AP]). If a program is in NC, it means that its intentional relations can be evaluated very fast, given a polynomial (in the number of input-database tuples) number of processors; they have to communicate extensively, usually through common memory. Unfortunately, this research means very little as far as utilizing a constant number of processors, particularly if they do not share common memory (e.g. a hypercube multiprocessor system* having 1024 nodes).

Presently, we are aware of only one method of speeding up logic-program evaluation, given a constant number of processors. It is the method introduced in [WS], which is useful even if the processors are interconnected through an asynchronous network. For intentional relations called decomposable, the [WS] method creates rewritten versions of the original logic program (a la selection propagation), and assigns to each processor a different version. Each processor executes its version on a local copy of the input database, without communicating with the other processors. At the end each tuple of the output relation and each intermediate tuple (i.e., that derives an output tuple) is evaluated by a unique processor. The union of outputs comprises the output of the original program (completeness). Assume that, along the lines suggested in [BR, Section 4], the load of evaluating an intentional relation is measured in terms of the number of new tuples generated in the process (a new tuple maybe an output tuple, or an intermediate tuple). Then the load is partitioned among multiple processors, and since each processor performs less work, we expect a shorter evaluation-time from start to finish.

The problem with the above method of partitioning the load is that it is applicable only to a very restricted class of logic programs. For example, in the class of simple chain programs ([UV], [AP]) it is applicable only to

* see [H]
regular ones. In this paper we address a less restrictive type of load distribution, and thus extend the class of logic programs to which the method is applicable. Specifically, we remove the restriction that each new tuple is computed by a unique processor (but still require completeness). Then, the load may not be partitioned, but it can be shared; each node in a multiprocessor system carries less than the total load. Duplication of effort may occur, but no communication overhead is introduced by load-sharing. If at the end each processor writes its evaluated relation to the same output file, no tuple is lost, but the evaluation completes faster.

For example, consider the following DATALOG (see [MW]) program called in [MPS] the canonical strongly linear:

\[
S(x,y) :\neg UP(x,w), S(w,z), DOWN(z,y)
\]

\[
S(x,y) :\neg FLAT(x,y)
\]

Assume that the extensional-database relations UP, FLAT, and DOWN represent a directed graph with three types of arcs. The above program defines a tuple \((a,b)\) to be in \(S\), if and only if there is a path from \(a\) to \(b\) having \(k\) UP arcs, one FLAT arc, and \(k\) DOWN arcs. The results in [WS] indicate that \(S\) is not decomposable. However, it does have a load sharing scheme. For example, processor \(i\) in the set of processors \(\{1, \ldots, r\}\) can execute the program above, with the predicate \(i = x \mod r\) added to the second rule of the program*. In other words, processor \(i\) computes the tuples \((a,b)\) for which the path goes through a FLAT arc \((c,d)\), with \(i = c \mod r\)**. Since there may be more than one path between \(a\) and \(b\), it is not guaranteed that each tuple is computed by a unique processor. However, it is intuitively clear that for a large random graph, each one in the set of processors \(\{1, \ldots, r\}\) performs less work, thus completes its evaluation before a single processor would have done so.

The purpose of this paper is to formally define what it means for a program to have a load-sharing scheme, and to explore which programs do have such schemes, and which ones do not, i.e., are not amenable to parallel evaluation by the method described. We determine that almost all linear programs (each rule has at most one intentional predicate in the body) have such a scheme. In the class of single rule programs (srups), defined in [CK], the pivoting ones (see [WS]) do have a load sharing scheme; so does a certain subclass of the simple chain programs.

We define a class of srups for which we prove that a load sharing scheme cannot exist. Several famous srups belong to this class (e.g. path systems).

---

* \(i = (x+y) \mod r\) works as well
** This works for character-string labels as well, since the binary representation can be regarded as a natural number
The rest of this paper is organized as follows. In section 2, we provide the preliminaries, and in section 3 we define and discuss what it means for a program to have a load sharing scheme. In section 4 we determine that almost all linear programs have a load sharing scheme, and in section 5 we prove that a whole class of sirups cannot have a load sharing scheme. In section 6 we prove that a subclass of the simple chain programs has a load sharing scheme. In section 7 we discuss future work.

2. Preliminaries

An atom is a predicate symbol with a constant or a variable in each argument position. We assume that the constants are the natural numbers. An \( R \)-atom is an atom having \( R \) as the predicate symbol. A rule consists of an atom, \( Q \), designated as the head, and a conjunction of one or more atoms, denoted \( Q_1, \ldots, Q^k \), designated as the body. Such a rule is denoted \( Q :- Q_1, \ldots, Q^k \), which should be read "\( Q \) if \( Q_1 \) and \( Q_2 \), and, ..., and \( Q^k \)." A rule or an atom is an entity. If an entity has a constant in each argument position, then it is a ground entity. For a predicate \( R \), a finite set of \( R \)-ground-atoms is a relation for \( R \).

A DATALOG program, or a program for short, is a finite set of rules whose predicate symbols are divided into two disjoint subsets: the extensional predicates, and the intentional predicates. The extensional predicates are distinguished by the fact that they do not appear in any head of a rule. An input to \( P \) is a relation for each extensional predicate. An output of \( P \) is a relation for each intentional predicate of \( P \). A substitution applied to an entity, or a sequence of entities, is the replacement of each variable in the entity by a variable or a constant. It is denoted \( x_1/y_1, x_2/y_2, \ldots, x_n/y_n \) indicating that \( x_i \) is replaced by \( y_i \). A substitution is ground if the replacement of each variable is by a constant. A ground substitution applied to a rule is an instantiation of the rule.

A database for \( P \) is a relation for each predicate of \( P \). The output of \( P \) given an input \( I \), is the set of relations for the intentional predicates in the database, obtained by the following procedure, called bottom up evaluation.

BUE1. Start with an initial database consisting of the relations of \( I \).

BUE2. If there is an instantiation of a rule of \( P \) such that all the ground atoms in the body are in the database generated so far, and the one in the head is not, then:

1. add to the database the ground atom in the head of the instantiated rule, and reexecute BUE2.

BUE3. Stop.

This procedure is guaranteed to terminate, and produce a finite output for any given \( P \) and \( I \) ([VEK]). The output is
unique, in the sense that any order in which bottom up evaluation adds the atoms to the database will produce the same output.

For some rule, a variable which appears in the head, is called a distinguished variable. For simplicity we assume that each rule of a program is range restricted, i.e. every distinguished variable also appears in the body of the rule; additionally, we assume that none of the rules of a program has constants.

An evaluable predicate is an arithmetic predicate (see [BR]). Examples of evaluable predicates are sum, greater than, modulo, etc. A rule re is a restricted version of some rule r, if r and re have exactly the same variables, and r can be obtained by omitting zero or more evaluable predicates from the body of re. In other words, re is r with some evaluable predicates added to the body, and the arguments of these evaluable predicates are variables of r. For example, if r is:

$$S(x,y,z) :- S(w,x,y), A(w,z)$$

then one possible re rule is:

$$S(x,y,z) :- S(w,x,y), A(w,z), x-y=5$$

A program $P_i$ is a restricted version of program $P$ if each one of its rules is a restricted version of some rule of $P$. Note that $P_i$ may have more than one restricted version of a rule $r$ of $P$. To continue the above example, if $P$ has the rule $r$, then $P_i$ may have the rule $re$ as well as the rule $re'$:

$$S(x,y,z) :- S(w,x,y), A(w,z), x-y=6$$

Throughout this paper, only restricted versions of a program may have evaluable predicates. The input of a program with evaluable predicates, i.e. a restricted version, is defined as before. The output is also defined as before, except that BUE2 also verifies that the substitution satisfies* the evaluable predicates in the ground rule; only then the atom in the head is added to the database and BUE2 is reexecuted. In other words, in considering instantiations for a restricted version of a rule, bottom up evaluation disregards database atoms which do not satisfy the additional evaluable predicates.

A predicate $Q$ in a program $P$ directly derives a predicate $R$ if it occurs in the body of a rule whose head is a $R$-atom. $Q$ is recursive if $(Q,Q)$ is in the nonreflexive transitive closure of the "directly derives" relation. Predicate $Q$ derives predicate $R$ if $(Q,R)$ is in the reflexive transitive closure of the "directly derives" relation (particularly, every predicate derives itself). A program is recursive if it has a recursive predicate. A rule is recursive if the

* for example, the substitution $x/14,y/8$ satisfies the evaluable predicate $x-y=6$, whereas the substitution $x/13,y/9$ does not.
predicate in its head transitively derives some predicate in its body.

3. Load Sharing Schemes

Let \( P \) be a program, and \( T \) an intentional predicate in \( P \). Denote by \( Q^1, \ldots, Q^r \) the set of intentional predicates which derive \( T \). Intuitively, these are the relations that have to be evaluated to determine the output relation for \( T \). The output of \( P \) for \( T \) given \( I \), denoted \( O(P,T,I) \), is \( \bigcup_{i=1}^{r} Q^i \). In other words, the output contains \( T \)-ground-atoms and ground-atoms of other predicates which derive \( T \). Assume that \( P_1, \ldots, P_r \) are restricted copies of \( P \), for \( r > 1 \). For an intentional predicate \( R \) of \( P \), denote by \( R_i \) the relation output by \( P_i \) for \( R \); the relation output by \( P \) is denoted \( T \). Observe that this is a somewhat unconventional notation, since for \( P_i \) the relation name is different than the predicate name. The set \( D = \{ P_1, \ldots, P_r \} \) is a load sharing scheme for evaluating \( T \) in \( P \), if the following two conditions hold:

1. For each input \( I \) to \( P, P_1, \ldots, P_r \), \( \bigcup_{i} T_i \supseteq T \) (completeness)

2. There is a constant \( c_D > 1 \) such that for every integer \( n \), there is an input \( I \), for which \( |O(P,T,I)| > n \), and \( |O(P,T,I)| / \max_i |O(P_i,T_i,I)| \geq c_D \) (nontriviality).

In order to intuitively explain the above definition, we assume that the whole database, i.e. the set of input base relations, is replicated at each one of \( r \) processors, and each processor has a restricted copy of the program \( P \). Furthermore, we assume that for a given input to \( P \) and to some restricted copy of it, the computation time of a predicate decreases proportionately to the size of the output.

The first requirement in the definition is that no \( T \)-atoms are lost by evaluating the relation for \( T \) in each \( P_i \), rather than the relation for \( T \) in \( P \). Although the requirement is for inclusion in one direction only, the fact that \( \bigcup_{i} T_i \) does not contain any atoms which are not in \( T \) is implied by the fact that each \( P_i \) is a restricted version of \( P \).

Thus, by using multiple processors and taking the union of the \( T_i \)'s, the exact relation for \( T \) is obtained.

The second requirement, stated informally, is that for each one in an infinite set of inputs, the output of each \( P_i \) for \( T \) is at least \( c_D \) times smaller than the output of \( P \) for \( T \) (thus completes faster). We also require that this output-size reduction for each \( P_i \), occurs for arbitrarily large outputs. Although this is still a best case scenario in some sense (which is generally unconventional), for the load sharing schemes that we are discussing in this paper, it
is intuitive that time saving can be achieved for the "average input". The reason is that each load-sharing scheme discussed in this paper is obtained by adding the evaluable predicate \( i = (x_1 \ldots + x_k) \mod r \) to one of the rules. For an input which is distributed evenly across a range of natural numbers, this partitions the output evenly.

In [WS] a decomposable predicate is defined. Decomposability is similar to the existence of a load sharing scheme, with two exceptions. First, decomposability imposes an additional restriction, called lack-of-duplication. It requires that for each input \( I \) to \( P_1 \ldots, P_r \), and for each \( i \neq j \), the relations \( Q_i \) and \( Q_j \) are disjoint for any intensional predicate \( Q \) which derives \( T \) in \( P \). Second, for nontriviality, decomposability requires only that for some arbitrary input each \( T_i \) is nonempty. The reason for the more restrictive definition of load sharing in this respect, is the following. For decomposability there is nothing to lose by partitioning the load, but for load sharing we would not be satisfied if all load is duplicated, except the evaluation of a few tuples.

The positive results obtained in [WS] are applicable here. Before stating them we define one more new concept, the potential speedup. Given a load-sharing scheme \( D = \{P_1, \ldots, P_r\} \) for evaluating a predicate \( T \) in \( P \), the potential speedup of \( D \), denoted \( Ps(D) \), is the maximal number \( M \) for which the following condition is satisfied. For every integer \( n \) and every \( \varepsilon \), there is an input \( I \) for which \( lO(P,T,J)| > n \), and \( |O(P,T,J)|/\max_i |O(P_i,T_i,J)| \geq M - \varepsilon \). Intuitively, the potential speedup is the number to which the ratio \( |O(P,T,J)|/\max_i |O(P_i,T_i,J)| \) can come arbitrarily close, when \( I \) is an arbitrarily large input. When the load to evaluate \( T \) is measured in terms of new ground atoms generated in the evaluation process, the above is the ratio between the load of evaluating \( T \) by \( P \), and the maximum load of a processor, when the sharing scheme is used. The definition is somewhat complicated since there are load-sharing schemes (the ones discussed in section 6) for which the potential speedup cannot be achieved, but to which the ratio can come arbitrarily close. Note that the fact that \( D \) is a load-sharing scheme implies that \( 1 < Ps(D) \leq r \).

A single rule program (see [CK]) is a program which has a single intensional predicate, denoted \( S \) in this paper. It has a nonrecursive rule,

\[
S(x_1,\ldots,x_n) :- B(x_1,\ldots,x_n).
\]

and one other, possibly recursive, rule in which the predicate symbol \( B \) does not appear. In the nonrecursive rule the \( x_i \)'s are distinct variables. Assume that \( R \) is a set of atoms with each atom having a variable in each argument position. The set \( R \) is pivoting if there is a subset \( d \) of argument positions, such that in the positions of \( d \):
1. the same variables appear (possibly in a different order) in all atoms of $R$, and

2. each variable appears the same number of times in all atoms of $R$.

A member of $d$ is called a pivot. Note that a variable which appears in a pivot may or may not appear in a nonpivot position. The recursive rule of a sirup is pivoting if all the occurrences of the recursive predicate in the rule constitute a pivoting set. For example, the rule

$$S(w,x,y,z) :- S(u,y,x,w), S(v,x,y,w), A(u,v,z)$$

is pivoting, with argument positions 2, 3 and 4 of $S$ being the pivots.

Theorem 1: If the recursive rule of a sirup is pivoting, then the sirup has a load-sharing scheme of any size. The potential-speedup equals the size of the scheme. []

4. Linear Programs

In this section we discuss linear programs. A program is linear if the body of each rule contains at most one intentional predicate. A rule of a program $P$ is an exit rule if its body consists of extensional predicates only. An exit rule $r_*$ with extensional predicate symbols $B_1, \ldots, B_k$, is distinct, if there is no other rule of $P$ in which a nonempty subset of the $B_i$'s appears, but another extensional predicate does not. In other words, $r_*$ is not distinct if there is another rule of $P$ in which a subset of the $B_i$'s appears, but no other extensional predicate does. Note that the exit rule of a sirup is distinct. An exit rule $r_*$ in $P$ derives a predicate $R$ if the predicate in $r_*$'s head derives $R$.

An intentional predicate of a linear program is distinct if it is derived by a distinct rule.

Theorem 2: If $T$ is a distinct predicate of a linear program $P$, then there is a load-sharing scheme of any size for evaluating $T$ in $P$. The potential-speedup equals the size of the scheme. []

In [WS] we have shown that a linear sirup without repeated variables in the recursive predicate, is not decomposable unless it is pivoting. Observe that theorem 2 implies that every linear sirup has a load-sharing scheme of any size, indicating (if the reader was not convinced already) that the class of predicates having load-sharing schemes is strictly larger than the class of decomposable predicates.

We conjecture that any predicate (not necessarily distinct) in a linear program has a load-sharing scheme, but do not have a proof of this fact yet. For example, in the program computing the transitive closure of a graph given by the relation $B$
\[ S(x,y) :- S(x,z), B(z, y) \]
\[ S(x,y) :- B(x, y) \]

the predicate \( S \) is not distinct, because the exit rule is not distinct. However, \( S \) obviously has a load-sharing scheme of any size (add \( x \mod r = i \) to the nonrecursive rule).

5. Programs Without a Load Sharing Scheme

In this section we demonstrate that not every sirup has a load sharing scheme. Specifically, we provide a necessary condition for a sirup to have a load sharing scheme (Theorem 3). It turns out that some famous sirups do not satisfy the condition. An example is the first P-complete problem, path-systems ([C]). The sirup for the problem is:

\[ S(x) :- S(y), S(z), H(x,y,z) \]
\[ S(x) :- B(x) \]

Intuitively, the reason that path-systems does not have a load sharing scheme is as follows. If such a scheme exists, and there is an input for which the output of each restricted copy is smaller than the total output (nontriviality), then there must exist a larger input for which completeness is not satisfied.

Another example of a sirup without a load sharing scheme, is a variant of path systems called the blue blooded frenchman ([CK]):

\[ S(x) :- S(y), S(z), H_1(x,y), H_2(x,z) \]
\[ S(x) :- B(x) \]

Some other variations which have not been defined previously, as far as we know, are (nonrecursive rule obvious, thus specification omitted):

\[ S(x,u) :- H_1(x,y,u), H_2(x,u), S(y,u), S(z,w) \]
\[ S(x,u) :- H(x,y,z,u), S(y,u), S(z,w) \]
\[ S(x) :- H_0(x,w), H_1(w,y), H_2(w,x), S(y), S(z) \]
\[ S(x,w,y) :- U_1(x,u), S(t,u,v), FLAT(v,w,z), S(z,t), S(z,u), DOWN(r,s,x) \]

What do the above sirups have in common? This is what the next theorem establishes. Before stating it we need the following definition.
Given a sirup $P$ the extensional-graph of $P$, denoted $G(P)$, is an undirected graph defined as follows. Its nodes are the variables which do not appear in any $S$-atom in the body of the recursive rule. The edge $x \rightarrow y$ is in the graph if and only if there is an extensional-predicate atom, $A$, in the body of the recursive rule such that $x$ and $y$ are variables of $A$.

**Theorem 3:** A sirup $P$ does not have a load sharing scheme if the set of atoms, $ST$, in the body of the recursive rule satisfies all the following conditions.

1. Except for the $S$-atoms, there are no two atoms of $ST$ which have the same predicate symbol.
2. There are at least two $S$-atoms in $ST$, the $S$-atoms in $ST$ have pairwise disjoint variables, and none of them has repeated variables.
3. Denote by $R$ the set of variables which appear in some $S$-atom of $ST$. Each extensional predicate atom has a variable which does not appear in $R$, and each variable in $R$ appears in some extensional predicate atom.
4. The graph $G(P)$ has a distinguished variable in each one of its connected components.

It is obvious that the sirups that have been discussed in this section satisfy the conditions above.

Let us observe that the first two conditions of the previous theorem are not, by themselves, sufficient for nonexistence of a load-sharing-scheme. They are satisfied in the following sirup, $P_0$, although it does have a load sharing scheme.

$$S(x,y) \leftarrow S(x,x), A(z_1, z_2) S(z_2, y)$$

$$S(x,y) \leftarrow B(x,y).$$

It will be shown in the next section that a load-sharing scheme of any size can be obtained for $P_0$.

6. Simple Chain Programs

In this section we establish that each sirup in a subclass of the simple chain programs, has a load sharing scheme. The subclass includes sirups which are not linear, and were shown in [WS] not to be decomposable.

A simple chain program is a recursive sirup in which: (a) all the predicates are binary, (b) the argument positions in the left hand side of the recursive rule have distinct variables, and these variables appear in the first argument position of the first atom in the body, and in the last argument position of the last atom, respectively, (c) all the
argument positions in the body of the recursive rule have distinct variables, except that the first argument position of the second atom has the same variable as the last argument position of the first atom, the first argument position of the third atom has the same variable as the last argument position of the second atom, etc. For example, the following is a simple chain program

\[ S(x, y) :\neg A(x, Z_1), S(z_1, z_2), S(z_2, Z_3), C(z_3, z_4), D(z_4, y) \]

\[ S(x, y) :\neg B(x, y) \]

\((A, B, C, D)\) are extensional predicates.

Two programs are equivalent if they produce the same output (for every predicate), for any given input. A simple chain program is weakly regular if the leftmost (or rightmost) predicate symbol in the body of the recursive rule is \(S\), and by replacing all other \(S\) predicate-symbols in the body of the recursive rule, by \(B\), an equivalent program is obtained. For example, \(P^0\) of the previous section is weakly regular since it is equivalent to the following program (see [UV] for the equivalence proof)

\[ S(x, y) :\neg S(x, z_1), A(z_1, z_2), B(z_2, y) \]

Similarly, any siur, denote it \(P^1\), of the form

\[ S(x, y) :\neg S(x, z_1), S(z_1, z_2), \ldots, S(z_n, y) \]

is weakly regular, since it is equivalent to the program

\[ S(x, y) :\neg S(x, z_1), B(z_1, z_2), \ldots, B(z_n, y) \]

\[ S(x, y) :\neg B(x, y) \]

Although a weakly regular program can be rewritten as a regular program which is decomposable, this may not be desirable for performance reasons.

**Theorem 4:** A weakly regular simple chain program has a load-sharing scheme of any size. The potential-speedup is the size of the scheme. []

**Remark:** The above theorem is proven by adding to the recursive rule in copy \(P_i\), the predicate \(x \mod r = i\). In this respect it is different than previous proofs, where the predicate was added to the nonrecursive rule.

It may not be surprising that weakly regular simple chain programs have a load-sharing schemes, since they are equivalent to linear ones, for which we have proven existence of such schemes. However, note that \(P^0\) and \(P^1\) have been proven in [WS] not to be decomposable, although their equivalent programs have been proven
decomposable.

7. Future Work

An obvious direction for future research is to obtain a complete characterization of programs which have load sharing schemes. In other words, we would like to close the gap between the programs for which we can prove existence of load-sharing schemes, and the ones for which we can prove nonexistence. Another direction, is to quantify the correlation between response time on one hand, and evaluation load as measured in terms of new tuples generated, on the other. For this purpose we shall probably have to conduct extensive experiments, involving randomly generated inputs. Another question is the following. If a load sharing scheme exists, does there always exist one with a potential speedup equal to the size of the scheme? For the programs discussed in this paper we have seen that this is the case. Must it be true? Finally we shall mention that we would like to determine how to distribute the load when communication among the processors participating in the evaluation cannot be avoided. What architectures are preferable?

Acknowledgement

The author thanks Nissim Francez and Oded Shmueli for helpful discussions and comments.

References


