Parallelism in Logic Programs

Raghu Ramakrishnan
Computer Sciences Department
University of Wisconsin-Madison, WI 53706, U.S.A.

Abstract

There is a tension between the objectives of avoiding irrelevant computation and extracting parallelism, in that a computational step used to restrict another must precede the latter. Our thesis, following [BeR87], is that evaluation methods can be viewed as implementing a choice of *sideways information propagation graphs*, or sips, which determines the set of goals and facts that must be evaluated. Two evaluation methods that implement the same sips can then be compared to see which obtains a greater degree of parallelism, and we provide a formal measure of parallelism to make this comparison.

Using this measure, we prove that transforming a program using the Magic Templates algorithm and then evaluating the fixpoint bottom-up provides a "most parallel" implementation for a given choice of sips, without taking resource constraints into account. This result, taken in conjunction with earlier results from [BeR87, Ra88], which show that bottom-up evaluation performs no irrelevant computation and is sound and complete, suggests that a bottom-up approach to parallel evaluation of logic programs is very promising. A more careful analysis of the relative overheads in the top-down and bottom-up evaluation paradigms is needed, however, and we discuss some of the issues.

The abstract model allows us to establish several results comparing other proposed parallel evaluation methods in the logic programming and deductive database literature, thereby showing some natural, and sometimes surprising, connections. We consider the limitations of the abstract model and of the proposed bottom-up evaluation method, including the inability of sips to describe certain evaluation methods, and the effect of resource constraints. Our results shed light on the limits of the sip paradigm of computation, which we extend in the process.

1 Introduction

We consider the parallel evaluation of logic programs. This has been the subject of much research in the logic programming and, recently, the deductive database communities. We review this work, and observe that there is a commonly used measure of parallelism based on a top-down evaluation paradigm of identifying subgoals and answers. To formalize this intuition, we propose a simple abstract model of computation that underlies much of the work on parallel evaluation of logic programs. There is a tension between the objectives of restricting the computation on the one hand and extracting parallelism on the other. This precedence is reflected in our model of computation by the choice of *sideways information propagation graphs*, or sips, which, informally, describe the order in which the literals in the body of a rule are to be solved.

Our thesis is that parallel evaluation methods can be viewed as implementing a choice of sips, a choice that determines the set of goals and facts that must be evaluated. Two evaluation methods that implement the same sips can then be compared to see which obtains a greater degree of parallelism, and we provide a formal measure of parallelism to do this. It is important to understand what is and — more importantly, perhaps — what is not implied by the statement that evaluation method $\mathcal{M}$ is "more parallel in our model " than evaluation method $\mathcal{N}$. First, our model only allows comparison of methods that fit the sip paradigm of computation, which is that some choice of sips for the rules in the logic program is implemented by the evaluation method. In Section 3, we show that most proposed methods for parallel evaluation of logic programs do fit this paradigm; in Section 7 we consider some methods that do not. Second, we compare the parallelism obtained by methods when they use the same sips. Thus, informally, $\mathcal{M}$ is more parallel than $\mathcal{N}$ if for every choice of sips, it succeeds in obtaining as much or more parallelism. Similarly, when we say that an evaluation method is "most parallel in our model", this does not mean that a faster parallel method cannot be found for a given problem. It does mean that once we choose to represent a problem — any problem — as a particular logic program and make a choice of sips, then the evaluation method obtains as much or more parallelism than any other method for evaluating the program according to the sips. Third, our model implicitly assumes that there are enough resources to work on all identified subcomputations in parallel, and therefore ignores implementation overheads and resource constraints.

Using this metric, we prove that transforming a program using the Magic Templates algorithm and then evaluating the fixpoint bottom-up provides a "most parallel" implementation for a given choice of sips, provided that there are no resource constraints. We emphasize a fundamental difference between this approach and top-down, process-oriented evaluation methods: whereas a top-down evaluation method proceeds by creating processes to solve subgoals, the bottom-up approach proceeds by applying rules to facts to produce new facts. Indeed, the bottom-up method has no inherent notion of a "process", nor of a "goal", although we will establish a correspondence between certain facts generated in the bottom-up evaluation of a rewritten program (as per the Magic Templates algorithm) and goals generated in top-down evaluation methods, and refer to these facts as goals. This distinction is significant in terms of implementation overhead, and we discuss this point in Section 8.

This work has many limitations. First, the class of sips described here cannot describe certain parallel evaluation methods in the literature, and there seem to be some fundamental difficulties in extending the definition of sips to cover these methods. Second, the measure of parallelism is rather strong — two evaluation methods that implement the same sips may be incomparable under the definition, because each does better than the other on some program. Further, it does not allow us to compare two executions that use different choices of sips, and this could indeed be seen as a limitation of our separation of concerns in restricting the computation and in parallelizing it. Finally, ignoring implementation overheads and resource constraints is hardly realistic. Any real evaluation method must contend with the problem of mapping a computation onto the available resources, and in doing so must often sacrifice either restriction or parallelism. This aspect of the computation is not captured by our abstract model; however, it clearly affects results obtained using the model. We discuss this issue further in Section 8.

In spite of the limitations, we believe that the results provide strong evidence in favor of program transformation and bottom-up fixpoint computation as a parallel evaluation method, especially for non-deterministic goals to which all solutions are required. We view this work as a first step towards a fuller understanding of parallel bottom-up evaluation of logic programs. Recent work in deductive databases has addressed the specific problem of mapping a bottom-up fixpoint computation onto a network of processors; we discuss this work and place it in the context of our approach.

The paper is organized as follows. Following preliminary definitions in Section 2, we survey some proposed parallel evaluation methods for logic programs in Section 3. In Section 4, we develop a model of computation that allows us to view a class of evaluation methods based on sips at an abstract level, and a measure 5f parallelism that can be used to compare them. The class includes all the methods surveyed in Section 3, and several others as well. In Section 5, we present a bottom-up evaluation method based on rewriting a program, according to the Magic Templates algorithm, and evaluating the fixpoint of the rewritten program bottom-up. In Section 6, we compare the parallelism obtained using several proposed parallel evaluation methods. We consider the limitations of sips in Section 7, and discuss possible extensions. In Section 8, we discuss several pragmatic considerations that must be taken into account in deciding the relative merits of parallel evaluation methods for logic programs, and then present our conclusions in Section 9.

2 Preliminaries

The language considered in this paper is that of Horn logic. Such a language has a countably infinite set of variables and countable sets of function and predicate symbols, these sets being mutually disjoint. It is assumed, without loss of generality, that with each function symbol f and each predicate symbol p, is associated a unique natural number n, referred to as the *arity* of the symbol; f and p are then said to be n-ary symbols A 0-ary function symbol is referred to as a constant. A *term* in a first order language is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms. A tuple of terms is sometimes denoted simply by the use of an overbar, e.g., $\bar{t}$.

A substitution is an idempotent mapping from the set of variables of the language under consideration to the set of terms, that is, the identity mapping at all but finitely many points. A substitution σ is *more general* than a substitution θ if there is a substitution φ such that $\theta = \varphi \circ \sigma$. Substitutions are denoted by lower case Greek letters θ, σ, ϕ, etc. Two terms t_1 and t_2 are said to be *unifiable* if there is a substitution σ such that $\sigma(t_1) = \sigma(t_2)$; σ is said to be a *unifier* of t_1 and t_2. Note that if two terms have a unifier, they have a most general unifier that is unique upto renaming of variables.

A clause is the disjunction of a finite number of literals, and is said to be Horn if it has at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause*. The positive literal in a definite clause is its *head*, and the remaining literals, if any, constitute its *body*. A predicate definition consists of a set of definite clauses, whose heads all have the same predicate symbol; a goal is a set of negative literals. We consider a logic program to be a pair $\langle P, Q\rangle$ where P is a set of predicate definitions and Q is the *input*, which consists of a query, or goal, and possibly a set of facts for "database predicates" appearing in the program. We follow the convention in deductive database literature of separating the set of rules with non-empty bodies (the set P) from the set of facts, or unit clauses, which appear in Q and are called the *database*. P is referred to as the *program*, or the set of rules. The motivation is that the rewriting algorithms to be discussed are applied only to the program, and not to the database. This is important in the database context since the set of facts can be very large. However, the distinction is artificial, and we may choose to consider (a subset of) facts to be rules if we wish. The meaning of a logic program is given by its least Herbrand model [vEK76].

Following the syntax of Edinburgh Prolog, definite clauses (rules) are written as

$$p :\!-\ q_1, \ldots, q_n.$$

read declaratively as q_1 and q_2 and ... and q_n implies p. Names of variables begin with upper case letters, while names of non-variable (i.e. function and predicate) symbols begin with lower case letters.

We will use *derivation trees* in several proofs:

Definition 2.1 Given a program P and input Q, derivation trees in $\langle P, Q\rangle$ are defined as follows:

- Every fact h in Q is a derivation tree for itself, consisting of a single node with label h.
- Let r be a rule: $h :- b_1, b_2, \ldots, b_k$ in P, let $d_i,\ i = 1 \ldots k$ be atoms with derivation trees t_i, and let θ be the mgu of $(b_1, \ldots, b_k)$ and $(d_1, \ldots, d_k)$. Then, the following is a derivation tree for $h\theta$: The root is a node labeled $h\theta$, and each $t_i,\ i = 1 \ldots n$, is a child of the root. Each arc from the root to a child has the label r.

3 A Survey of Proposed Parallel Evaluation Methods

We discuss several proposed parallel evaluation methods, focusing on the parallelism that is realized, that is, what subgoal computations are allowed to proceed in parallel. The survey in this section motivates our development of an abstract model of computation to compare the parallelism in different methods. We develop the model in the next section; it abstracts the behaviour of a class of methods called "sip-methods". The methods discussed in this section all fall into this class, unless otherwise noted.

One of the objectives of this paper is to identify the similarities and differences in proposed parallel evaluation methods, both top-down and bottom-up, and to this end, we provide a uniform and sufficiently detailed description of the major approaches. While the relationship between bottom-up and top-down evaluation methods has recently been studied widely in the deductive database community, the more complicated nature of parallel evaluation methods has made the connections harder to see. Indeed, it has been remarked that the work on parallel evaluation in the logic programming community, typically top-down methods, are not likely to be useful in the context of bottom-up parallel evaluation [CW89]. We think that on the contrary much can be gained by a careful study of the literature of both top-down and bottom-up approaches. There is a strong relationship between the structure of top-down and bottom-up computations, as demonstrated in [Ra88, Se89] and also [BMSU86, BeR87, Br89, Ul89b, Ul89a, Vi89, KL86, KL88], etc. While the details of an implementation of a top-down method would differ considerably from that of a bottom-up method, we believe that many ideas, such as schemes for structure-sharing, are likely to work in either approach. (See [NR89] for a closer look at this issue.)

The parallelism in logic programs is often broadly classified into *And-*, *Or-* and *Stream* parallelism. And-parallelism refers to the parallel solution of subgoals generated from literals in the same rule body. Or-parallelism refers to the parallel evaluation of different rules that unify with a given goal. Stream-parallelism refers to the eager processing by a subgoal (the "consumer") of an argument value, such as a list, that is being constructed by another subgoal (the "producer"). We will restrict our attention to the first two, since the last typically forces us to consider additional properties of the computation such as determinacy and structure-sharing in some detail. Most of the methods that we discuss in this section proceed by identifying subgoals and creating processes to solve them. However, there has been some work on achieving similar results through bottom-up fixpoint evaluation, and we discuss this work as well.

The discussion in [Ka87b] supplements the survey presented in this section, and we suggest that the reader consult it for more details.

3.1 The And-Or Tree Model

An And-Or tree for a logic program has the query as the root node, which is an Or-node. An Or-node is always labeled with a goal, and has one child And-node per rule whose head unifies with this label. The label of a child And-node is the corresponding rule with the unifying substitution applied to it. The unifying substitution is used to label the arc from the parent Or-node to this And-node. An And-node has at most one child Or-node per body literal in its label. The label of a child Or-node is a variant of the corresponding body literal.

The And-Or model presented in [Co83] builds And-Or trees by generating a process for each node in a top-down order. The query is the root node. The children of an Or-node are generated as described above. We now describe how the children of an And-node are generated: A child Or-node is created for the left-most body literal in the label of the And-node. The arc to the Or-node is labeled with the identity substitution. For each answer, which can be viewed as a substitution σ, to the Or-node corresponding to a body literal, an Or-node is generated for the next body literal. If the path from the And-node to the first Or-node is labeled with θ, the label of the second Or-node is the corresponding body literal with the substitution $\sigma\theta$ applied to it. This substitution is used to label the arc to it.

At any time, an And-node has at most one child Or-node per body literal in the label. Solutions to Or-nodes are saved as they are generated, and And-nodes are solved by generating all combinations of children through backtracking.

Much work has been done on this model; in particular, the ordering could be a partial order and sibling Or-nodes corresponding to different body literals could be generated simultaneously. In general, this creates problems if these children share variables. Therefore sibling Or-nodes are generated simultaneously only if they do not share variables. Since a variable that is shared between the corresponding literals could be bound to a ground term by a preceding Or-node, detecting such opportunities for solving the children of an And-node in parallel is a difficult problem. Several researchers have addressed this issue, e.g., [De84, CDD85]. Another area of research has been to identify intelligent ways to backtrack past predecessor nodes when a node fails (i.e., to recognize that alternative solutions to these predecessors would not enable the given node to succeed, and thus avoid generating further solutions to them.) Conery also suggested schemes for dynamically re-ordering the nodes in the And-Or tree [Co83]; these cannot always be described as sip-methods, and this is discussed further in Section 7.

An important restriction of the And-Or model is to simply avoid Or-parallelism by generating the children And-nodes of an Or-node one at a time. This restriction ensures the property that every variable instance in the computation has a unique binding at any time. (With Or-parallelism, recall that an Or-node saves multiple answers; these provide multiple bindings for the variables that appear in it.) This typically results in the loss of much parallelism, but reduces implementation overhead (see e.g., [De84, CDD85, HR89]).

3.2 Full Or-Parallelism

Full Or-parallelism is best understood in terms of *SLD-trees*. The SLD-tree for a logic program has the query as the root node. Every node in the tree is a conjunction of goals. A node has one child for each resolvent obtained by resolving one of the goals in the node with some rule in the program. The leaves are empty nodes. The conjunction of substitutions along a path from the root to a leaf, applied to the query, yields an answer.

With full Or-parallelism, each branch is explored in parallel, as initially proposed in [CH83]. Thus, if we have a node "$p1(5, X), p2(X, Y)$" and two rules "$p1(U, V) :- q1(U, V)$." and "$p1(W, Z) :- q2(W, Z)$.", there are two children for this node: "$q1(5, V), p2(V, Y)$" and "q2(5,Z),p2(Z,Y)". This leads to an unnecessary duplication of effort — with no real gain in parallelism — in the repeated solution of the goal $p2(Z, Y)$?.

A solution to this problem is to solve $q1(5, V)$? and $q2(5, Z)$? in parallel, and to then solve the $p2$ goal for each binding of the first argument in parallel. We describe the solution in the general case in terms of a modified And-Or tree, with the only difference being that at any time, an And-node could have more than one child Or-node corresponding to a given body literal. As before, for the left-most body literal in its label, an And-node has one child Or-node per rule whose head unifies with it, and the arc to this Or-node is labeled with the unifying substitution. The Or-nodes for every other body literal are generated as follows: When a child Or-node for the ith body literal returns an answer, which can be viewed as a substitution σ for the variables in it, this is composed with the substitution, say θ, on the arc to this Or-node and the resulting substitution $\theta\sigma$ is applied to the $i + 1$st body literal in the label of the And-node. This results in a goal, generated from the $i + 1$st literal, and one child Or-node is created with this label. The arc from the And-node to this Or-node is labeled with the substitution $\theta\sigma$.

This is indeed how the Or-parallel model proposed in [CH83] is implemented, as described in [CH84]. In essence, rules are solved left to right, and for each goal, all rules with which it unifies are solved in parallel. Notice that, except for the root, each Or-node is created in response to the answer to another Or-node; the creation of And-nodes can be avoided by directly creating tokens for all the Or-nodes that are its left-most children. Thus, with each Or-node in the tree, we can associate a set of Or-nodes that were generated because of answers to it. Let us call this the set of successors. The computation proceeds by creating "tokens" in a top-down order for each Or-node in the modified And-Or tree. A token contains enough information to generate tokens for all its successors. (In particular, this includes information about the label of the parent And-node; this is achieved by means of a "continuation", and we refer the reader to [CH84] for details.)

Note that there is no And-parallelism; a rule is always solved from left to right.

The above realization of the fully Or-parallel model qualifies as a "sip-method", but the proposal in its original form does not.

3.3 The Reduce-Or Model

Kalé observed that many of the proposed evaluation methods were either incomplete or did not extract all available parallelism, or both. This was the motivation for the development of the Reduce-Or evaluation model. It is in effect a combination of the And-Or and the fully Or-parallel models as we have described them.

The model is essentially the fully Or-parallel model extended to solve And-nodes according to a partial order, rather than a total left to right order, thereby also exploiting And-parallelism. The only change concerns the generation of the children Or-nodes of an And-node. A partial order is associated with each rule (and thus, any And-node that it labels). Consider an And-node, and the associated partial order over the body literals of the label. A node with no predecessors is treated like a left-most literal in the fully Or-parallel model — one Or-node is generated for each rule that unifies with it, and the arc to this Or-node is labeled with the unifying substitution. Consider a body literal p with predecessors $p_1, \ldots, p_k$ in the partial order. Let θ_i,$i = 1, \ldots, k$ be an answer substitution for p_i, and let the composition $\theta = \theta_1, \ldots, \theta_k$ be consistent. Then, an Or-node is generated for the goal $p\theta$, and the arc from the And-node to this Or-node is given the label $p\theta$. Kalé does not insist that sibling Or-nodes that correspond to different body literals and that are generated in parallel should contain no shared variables. Instead, any conflicts are resolved by explicitly composing answer substitutions for all the body literals, one per literal. In effect, this corresponds to taking a join on the body of a rule to generate an answer fact for the head predicate.

We conclude this discussion of top-down methods by formally defining And- and Or- parallel steps in terms of And-Or trees.

Definition 3.1 An *And-parallel step* is the concurrent generation of two goals that correspond to different body literals in the label of an And-node.

Definition 3.2 An *Or-parallel step* is the concurrent generation of goals $g_1, \ldots, g_k$ from a given goal g by unifying g with the heads of two different rules and generating $g_1, \ldots, g_k$ by instantiating body literals with no predecessors. The generated goals must not all be obtained from just one of the rules.

We assume that once a goal is "generated", it can be processed immediately. (In effect, a goal is considered to be generated when its processing begins.)

3.4 Bottom-Up Methods

The literature on bottom-up evaluation is extensive, and we do not propose to cover it in detail here. We refer the reader to surveys and expositions presented in [BaR86, Br89, NR89, Ul89a]. We note that while most of this literature deals with the implementation of *Datalog*, which is a subset of logic programs without function symbols, and also does not deal with non-ground terms, recent proposals treat full logic programs [Ra88, Se89]. We will examine one of these proposals ([Ra88]) in detail later. The following brief discussion should be supplemented by consulting Section 5.

The fundamental operation in bottom-up approaches is the application of a rule to a set of facts to generate new facts, which is similar to the use of the T_P operator to construct the least fixpoint model [vEK76]. An obvious drawback is that all consequences of the program are generated, not just the facts relevant to processing the given query. From our presentation of the top-down methods, it is clear that these methods restrict the computation by propagating bindings from the query as the construct the And-Or trees in top-down order. The essential idea in most bottom-up methods is to combine a top-down generation of goals with a bottom-up generation of facts. In general, this requires that all generated goals and facts should be retained and the process repeated iteratively until no new goals and facts are generated.

Most of the proposed methods use a top-down control strategy to generate goals, e.g., [DW87, Lo85, Vi89]. Some use a graph structure over the rules of the program for this purpose, e.g., [KL86, KL88, vG86]. It has been shown however, that this can be achieved through source-to-source program transformations, and this is the approach that we will pursue [BMSU86, RLK86, BeR87, Ra88, Se89]. We believe that this has significant advantages to offer in terms of uniformity, overheads, and implementation alternatives.

4 An Abstract Model of Computation

We consider how the evaluation of a logic program can be formalized at an abstract level in a way that allows us to make precise the degree of parallelism. We emphasize that the model we develop in this section is not an execution model, in that it does not specify how to evaluate a program, and should not be confused with execution models such as And-Or models or the Reduce-Or Process Model. Rather, it is a formal model in which we can abstractly represent computations that correspond to execution of a logic program using some execution model (i.e., evaluation method). In the next section, we propose the Magic Templates evaluation method for parallel evaluation, and subsequently analyze it in terms of our abstract model. This evaluation method and the abstract model are independent; the distinction should be borne in mind.

We begin by observing that while the semantics of a logic program is purely declarative in that it does not depend on how the program is evaluated or on any concept of a program state, there is a natural notion of state associated with any execution of a logic program.

As a first step, we could try to define an abstract model of computation as follows: The *state* of a program execution is a pair $\langle \mathcal{F}, \mathcal{G} \rangle$, where $\mathcal{F}$ and $\mathcal{G}$ are sets of facts and goals, respectively. The *initial state* is defined by $\mathcal{F} =$ set of given facts in the program (the EDB, in database terminology, or the set of rules with empty bodies), and $\mathcal{G} =$ the initial query. A *computation* is a progression from the initial state to a final state, in which $\mathcal{F}$ contains all facts in the answer to the query, through a sequence of transitions from one state to another.

To complete our model of computation, we must define the notion of a state transition. Intuitively, we seek to describe a single step of computation. The class of evaluation methods that we consider proceed by identifying subgoals and obtaining solutions to them, and thus, a natural candidate for a state transition is the addition of a single new goal or fact.

There are two complications, however. First, if possible, a parallel computation seeks to identify new goals and facts simultaneously. Thus, from a given state, it might be possible to simultaneously infer a set of new goals and facts. We will therefore allow a state transition to add a set of new goals and facts, rather than just a single new goal or fact.

A second complication arises from the fact that in several top-down evaluation schemes based on the AND-OR tree model, the computations of subgoals do not share their results, and indeed, the facts and goals generated in the computation of a subgoal are discarded when this computation is completed, even though the main computation is still in progress. In effect, different subcomputations see different sets of facts and goals, and the sets of facts and goals are not monotonically increasing since facts and goals are sometimes discarded. In our model, we will always refer to a single set of facts $\mathcal{F}$ and a single set of goals $\mathcal{G}$; these are the (monotonically increasing) sets of all facts and goals generated in any part of the computation. The subsets of visible facts or goals in a subcomputation influences the set of allowed transitions. These subsets, however, are not uniquely determined by $\mathcal{F}$ and $\mathcal{G}$ — that is, there is more to the state of a computation than just the sets of previously derived facts and goals. We will model this by augmenting the state vector with a third component, $\mathcal{H}$ (for "hidden"). Of the possible transitions based on $\mathcal{F}$ and $\mathcal{G}$, an evaluation method may only allow a subset, based on the third component $\mathcal{H}$. We will assume that this third component is suitably updated as part of a state transition; this depends on the specific evaluation method, and we will not go into this in further detail in this paper. (For a class of methods that always retain all deduced facts and goals, and in which these sets are visible to all subcomputations, $\mathcal{H}$ plays no role, and we will take advantage of this fact.)

We have introduced the notion of a state transition informally as the addition of a set of facts and goals to the current state, accompanied by an update to the hidden component of the state. We have not yet specified how transitions depend on the current state. To do this, we must make explicit certain assumptions about the class of evaluation methods that we consider, and we do this in the following subsection.

4.1 Sip-Method

The class of evaluation methods that we consider proceed by generating subgoals and facts (that are solutions to some of the subgoals), using the logic program $\langle P, Q\rangle$. Initially, the set of facts $\mathcal{F}$ consists of the EDB facts in Q. The set of goals $\mathcal{G}$ contains the given query, also in Q. (The hidden state $\mathcal{H}$ is assumed to be properly initialized.)

At any point in the computation, the state is a triple $\langle \mathcal{F}, \mathcal{G}, \mathcal{H}\rangle$. A new fact or goal can be generated by the use of a rule in P on $\mathcal{F} \cup \mathcal{G}$.

> Consider a rule:
> $r : p \coloneq q_1, \ldots, q_n$.
> We can generate a new fact $p\theta$ by applying a substitution θ such that for $i = 1, \ldots, n$:
>
> 1. there is a fact d_i in $\mathcal{F}$ and a substitution σ_i such that $q_i\theta = d_i\sigma_i$, and
> 2. there is a goal $c?$ in $\mathcal{G}$ and a substitution σ_0 such that $p\theta = c\sigma_0$.

In most evaluation methods, $\phi = mgu(\langle p, q_1, \ldots, q_n\rangle, \langle c, d_1, \ldots, d_n\rangle)$ is the only substitution applied, since applications of other substitutions only generate facts that are subsumed by the fact $p\phi$. We will assume this in the rest of this paper, and also make a similar assumption in the following description of how goals can be generated. The effect of the hidden state $\mathcal{H}$ and the evaluation method $\mathcal{M}$ — which we do not specify in further detail — is to allow only a subset of the above new facts to be generated. For example, in methods that do not memo generated facts, the device of a hidden state can be used to simulate the effect of generated facts (which are therefore in $\mathcal{F}$) that have been discarded at some point in a computation (and thus cannot be used to make a new inference). Generated facts are added to $\mathcal{F}$; further, a (newly generated or previously known) fact $f \in \mathcal{F}$ can be discarded if it is subsumed by another fact in $\mathcal{F}$. The cost of detecting that a fact is subsumed may sometimes override the gains, and some methods do not discard such facts; we will not require this as part of our definition of sip methods. However, not discarding subsumed facts may lead to unnecessary derivations of new facts.

To specify how goals can be generated, we must introduce the notion of a *sideways information passing*, or sip, graph. We define a sip graph for a rule to be a partial ordering of the body literals. New goals are generated by invoking a rule, in a top-down sense, with some known goal. Further, the literals in the body of a rule are solved in some order, more generally a partial order. Each literal is solved by generating a subgoal from it and then obtaining solutions to this subgoal. In generating a subgoal from a literal, the goal with which the rule was invoked and the solutions obtained to literals that precede the given literal in the sip partial order are all used to bind variables and thereby restrict the new subgoal. Thus, to generate a subgoal from a literal q_k, we need the goal with which the rule was invoked, and the facts (solutions) corresponding to literals that precede it in the sip order.

Formally:

> Let the predecessors of q_k be the literals $q_i, \ldots, q_j$, let $c \in \mathcal{G}$ and $\{d_i, \ldots, d_j\} \subseteq \mathcal{F}$, and let $\theta = mgu(\langle p, q_i, \ldots, q_j\rangle, \langle c, d_i, \ldots, d_j\rangle)$. Then, we can generate the goal $q_k\theta?$.

Generated goals are added to $\mathcal{G}$, and as for facts, subsumed goals can be discarded. The effect of the hidden state $\mathcal{H}$ and the evaluation method $\mathcal{M}$ is again to allow only a subset of the new goals to be generated. Henceforth, we will refer to the above operations as simply "applying a rule" (in a given state, according to a given evaluation method) to generate a fact or a goal. In a given state, we will in general be able to apply several rules simultaneously to produce new goals and facts. Indeed, the same rule could be applied to produce several new goals and facts from the sets $\mathcal{F}$ and $\mathcal{G}$. Thus, a state transition can add a set of facts or goals, each of which can be generated by a single application of a rule to $\mathcal{F} \cup \mathcal{G}$.

We remark that the sip can be more sophisticated — for example, the new subgoal may be restricted using only a subset of the bindings made by its predecessors in the sip. (This may be motivated by the overhead of propagating the bindings in some evaluation schemes.) We do not consider such refinements of sips in this paper, and instead we refer the reader to [BeR87, Ra88]. Further, it is possible to choose a different sip for the same rule when it is invoked using different goals. For example, we may wish to solve it left-to-right when the first argument of the goal is bound to a ground term, and right-to-left otherwise. We consider this point in Section 7, but for now we assume that a single sip is chosen for a rule. That is, a rule is solved in the same way regardless of the goal with which it is invoked. *We will assume that the choice of sips is made for us — making a good choice is a hard problem, and orthogonal to the results in this paper.*

We now summarize our description of sip-methods.

Definition 4.1 Sip-Method
Consider a logic program $\langle P, Q \rangle$ and a choice of sips for the rules in P. Consider the sets of facts (say $\mathcal{F}$) and goals (say $\mathcal{G}$) that can be generated from Q by applying the rules in P in some order according to the chosen sips under the assumption of a hidden state that disallows the generation of no fact or goal. If subsumed facts and goals are discarded as soon as possible, then let the sets of facts and goals be denoted $\mathcal{F}_1$ and $\mathcal{G}_1$.
A *sip-method* is defined to be an evaluation method that generates only facts and goals in $\mathcal{F} \cup \mathcal{G}$.
A *subsumption-checking sip-method* is defined to be an evaluation method that generates only facts and goals in $\mathcal{F}_1 \cup \mathcal{G}_1$.
A *complete (resp. subsumption-checking) sip-method* is one that computes maximal sets of facts and goals, as per the definition of a (resp. subsumption-checking) sip-method.

The maximal sets of goals and facts that may be computed are independent of the details of the evaluation method (and the associated encoding of the hidden state), and are determined by the program and the sips. The maximal sets are not unique for subsumption-checking sip-methods, due to effect of the non-determinism in the order of rule applications on the discarding of subsumed facts and goals. If the evaluation method is to guarantee all answers that follow from the least Herbrand model semantics, all these goals and facts must be generated, since it is otherwise possible to construct inputs such that some answer is not generated. This motivates the definition of complete sip-methods; we note that not all proposed evaluation methods are complete. On the other hand, techniques such as intelligent backtracking or the use of abstract interpretation can enable an evaluation method to avoid generating some of these goals and facts while retaining completeness. Since such methods would not qualify as complete sip-methods — although they do qualify as sip-methods — our abstract model is unable to capture their completeness.

While a broad class of evaluation methods can be viewed as sip-methods, it is important to note that methods that allow "coroutining" — the computation of two goals is interleaved, and typically, the bindings generated by each are used to restrict the other — cannot be considered sip-methods. We pursue this point further in Section 7.

The property that we will study in this paper is the effect of the choice of the evaluation method on when goals and facts are identified, and this provides the basis for our measure of parallelism. The reader who is familiar with [Ra88] will notice a difference in our definition of a sip-method. In [Ra88], a sip-method was required to generate the facts and goals that it is possible to generate by our definition of complete sip-methods, but it was possible for the method to generate other facts and goals as well. Methods that generated precisely the sets of goals and facts that it is possible to generate by our definition of complete subsumption-checking sip-methods were said to be *sip-optimal*. We have chosen to study this class in order to focus on the issue of parallelism; in [Ra88], the primary concern was the restriction of the computation to relevant facts and goals. We discuss methods that are not "sip-optimal" in Section 6.1.

Several examples of sip-methods were presented in Section 3. There are others that we have not considered; see e.g., [Po81, EKM82].

4.2 A Summary of Our Model of Computation

We now present the formal definitions of states, transitions and computations.

Definition 4.2 Consider a program $\langle P, Q \rangle$.

- The *state* of a program execution is a triple $\langle \mathcal{F}, \mathcal{G}, \mathcal{H} \rangle$, where $\mathcal{F}$ and $\mathcal{G}$ are sets of facts and queries, respectively, and $\mathcal{H}$ denotes a hidden component of the state.

- The *initial state* is defined by $\mathcal{F}$ = set of given facts in the program (the EDB, in database terminology, or the set of rules with empty bodies), and $\mathcal{G}$ = the initial query.

- A *state transition* according to evaluation method $\mathcal{M}$ in state $\mathcal{S}_1 = \langle \mathcal{F}_1, \mathcal{G}_1, \mathcal{H}_2 \rangle$ changes the state to $\mathcal{S}_2 = \langle \mathcal{F}_2, \mathcal{G}_2, \mathcal{H}_2 \rangle$, and is denoted as $\mathcal{S}_1 \vdash_{\mathcal{M}} \mathcal{S}_2$.

  $\mathcal{F}_2 = \mathcal{F}_1 \cup$ $\{f | f$ is a fact that can be generated from $\mathcal{F}_1 \cup \mathcal{G}_1$ in hidden state $\mathcal{H}_1$ according to method $\mathcal{M}$ by a single rule application.$\}$

  $\mathcal{G}_2 = \mathcal{G}_1 \cup$ $\{g | g$ is a goal that can be generated from $\mathcal{F}_1 \cup \mathcal{G}_1$ in hidden state $\mathcal{H}_1$ according to method $\mathcal{M}$ by a single rule application.$\}$

  Note that $\cup$ may result in some facts or goals being discarded because they are now subsumed. Further, we assume that the hidden state $\mathcal{H}_2$ is obtained by suitably updating $\mathcal{H}_1$ to reflect the behaviour of $\mathcal{M}$.

- A *final state* is a state such that no new facts or goals can be generated and no rule applications change the hidden state.

- A *computation* according to method $\mathcal{M}$ is a progression from the initial state to a final state, through a sequence of state transitions according to $\mathcal{M}$ from one state to another.

  The *length* of a computation sequence is the number of state transitions in it.

According to our model, in a given state, there is a unique transition according to a given evaluation method, and thus a unique computation sequence for a given program and choice of sips. This is essentially reflects the most optimistic situation, where all possible generations of new goals and facts are carried out simultaneously at each step, and makes the assumption that there are no resource constraints. It is worth remarking that the sets $\mathcal{F}$ and $\mathcal{G}$ may not change in a state transition, and only the hidden state $\mathcal{H}$ is updated. This corresponds to the situation that all the facts and goals that can be generated are previously known, and the only effect of generating them is to possibly make them visible in some subcomputations where they were not visible earlier. These details are germane to how $\mathcal{H}$ is to be updated; we do not consider this updating process in our abstraction of a computation.

In subsequent sections, we denote the hidden state as `T` for evaluation methods in which all goals and facts are visible to all computations. However, in the following example, we simply omit the hidden state, for simplicity, with the understanding that it is manipulated appropriately by the evaluation method and influences the generation of the computation sequence.

Example 4.1 We now present an example that illustrates our model of computation by listing the computation sequences in our model for execution according to several different evaluation methods. We use the following program; the only rule with a body that contains more than one literal is the first, and we assume that the chosen sip leaves the first two literals relatively unordered but before the third.

$p(X, Z) \;{:}{-}\; b1(X), b2(Y), b3(X, Y, Z).$
$p(X, Y) \;{:}{-}\; b4(X, Y).$
$b1(5).$
$b2(6).\ \ b2(7).$
$b3(5, 6, 8).\ \ b3(5, 7, 9).$
$b4(1, 2).$
$p(U, V)?$

We mark goals by a terminal "?", and represent the sets $\mathcal{F}$ and $\mathcal{G}$ as a single set of goals and facts. For brevity we use the notation "$\cup\{\ldots\}$" to denote a state in a computation sequence that is obtained by adding the set between $\{$ and $\}$ to the set in the previous state in the sequence (and updating the hidden state, which is not shown).

Prolog Prolog is a left-to-right evaluation method that does not exploit any parallelism. Its computation sequence is:
$\{p(U, V)?\} \vdash \cup\{b1(X)?\} \vdash \cup\{b1(5)\} \vdash \cup\{b2(Y)?\} \vdash \cup\{b2(6)\}$
$\vdash \cup\{b3(5, 6, Z)?\} \vdash \cup\{b3(5, 6, 8)\} \vdash \cup\{p(5, 8)\} \vdash \cup\{b2(7)\} \vdash \cup\{b3(5, 7, Z)?\}$
$\vdash \cup\{b3(5, 7, 9)\} \vdash \cup\ \{p(5, 9)\} \vdash \cup\{b4(X, Y)?\} \vdash \cup\{b4(1, 2)\} \vdash \cup\{p(1, 2)\}$

Note that the goal $b2(Y)?$ is generated a second time after backtracking. We do not see this in the above sequence since its only effect in our model is to affect the hidden state; the set of known facts and goals is unaffected by the re-derivation of a previously known goal.

Ciepielewski-Haridi This is a fully Or-parallel method proposed in [CH83]. It does not exploit any And-parallelism.
$\{p(U,V)?\} \vdash \cup\{b1(X)?, b4(X,Y)?\} \vdash \cup\{b1(5), b4(1,2)\}$
$\vdash \cup\ \{p(1,2), b2(Y)?\} \vdash \cup\{b2(6), b2(7)\} \vdash \cup\{b3(5,6,Z)?, b3(5,7,Z)?\}$
$\vdash \cup\{b3(5,6,8), b3(5,7,9)\} \vdash \cup\{p(5,8), p(5,9)\}$

Observe that the parallelism has resulted in a much shorter computation sequence. There is no And-parallelism since in no one transition do we add goals corresponding to different body literals.

DeGroot This is an And-parallel method that exploits no Or-parallelism, and was proposed in [De84].
$\{p(U,V)?\} \vdash \cup\{b1(X)?, b2(Y)?\} \vdash \cup\{b1(5), b2(6)\} \vdash \cup\{b3(5,6,Z)?\}$
$\vdash \cup\{b3(5,6,8)\} \vdash \cup\{p(5,8)\} \vdash \cup\{b2(7)\} \vdash \cup\{b3(5,7,Z)?\} \vdash \cup\{b3(5,7,9)\}$
$\vdash \cup\ \{p(5,9)\} \vdash \cup\{b4(X,Y)?\} \vdash \cup\{b4(1,2)\} \vdash \cup\{p(1,2)\}$

Conery This is a method that attempts to realize both And- and Or-parallelism, and is one of the methods proposed in [Co83].
$\{p(U,V)?\} \vdash \cup\{b1(X)?, b2(Y)?\} \vdash \cup\{b1(5), b2(6), b2(7)\} \vdash \cup\{b3(5,6,Z)?\}$
$\vdash \cup\{b3(5,6,8)\} \vdash \cup\{p(5,8)\} \vdash \cup\{b3(5,7,Z)?\} \vdash \cup\{b3(5,7,9)\} \vdash \cup\ \{p(5,9)\}$
$\vdash \cup\{b4(X,Y)?\} \vdash \cup\{b4(1,2)\} \vdash \cup\{p(1,2)\}$

Notice that in this method, the two $b3$ goals are sequentialized.

Reduce-Or This is also a method that exploits both And- and Or- parallelism, and is proposed in [Ka87a]. It identifies all the available parallelism in this example.
$\{p(U,V)?\} \vdash \cup\{b1(X)?, b2(Y)?, b4(X,Y)?\} \vdash \cup\{b1(5), b2(6), b2(7), b4(1,2)\}$
$\vdash \cup\{p(1,2), b3(5,6,Z)?, b3(5,7,Z)?\} \vdash \cup\{b3(5,6,8), b3(5,7,9)\}$
$\vdash \cup\{p(5,8), p(5,9)\}$ $\square$

4.3 A Measure of Parallelism

We now describe how the parallelism allowed by two evaluation methods can be compared.

Definition 4.3 Given two evaluation methods $\mathcal{M}_1$ and $\mathcal{M}_2$, we say that $\mathcal{M}_1$ is *more parallel than* $\mathcal{M}_2$ if and only if for every choice of a program $\langle P, Q\rangle$ and a set of sips S, the computation sequence according to $\mathcal{M}_1$ is no longer than the computation sequence according to $\mathcal{M}_2$.

By definition, our measure of parallelism will not allow us to compare computations that use different choices of sips, since the measure is defined in terms of a property that must hold for every choice of sips (and programs).

We remark that the length of a computation sequence corresponds to the time taken by the algorithm. For example, if we consider bottom-up fixpoint computation of a (rewritten) program, this length is equal to the number of *stages* or *iterations*.

We now present a result that is useful for proving that one method is more parallel than another.

Theorem 4.1 *$\mathcal{M}_1$ is more parallel than $\mathcal{M}_2$ if the following holds for every program $\langle P, Q\rangle$ and choice of sips $\mathcal{S}$:*

> *Let $\mathcal{F}_{1_i}$ and $\mathcal{G}_{1_i}$ denote the set of facts and goals in the state of the computation sequence S_1 according to $\mathcal{M}_1$ at Step i, and let $\mathcal{F}_{2_i}$ and $\mathcal{G}_{2_i}$ denote the corresponding sets for the computation sequence S_2 according to $\mathcal{M}_2$. For all i less than or equal to the length of S_1, $\mathcal{F}_{2_i} \subseteq \mathcal{F}_{1_i}$ and $\mathcal{G}_{2_i} \subseteq \mathcal{G}_{1_i}$.*

Proof Let the length of computation sequence S_1 be k and that of S_2 be l. It follows that $\mathcal{F}_{1_k} = \mathcal{F}_{2_l}$ and $\mathcal{G}_{1_k} = \mathcal{G}_{2_l}$; also, $\mathcal{F}_{1_{k-1}} \subset \mathcal{F}_{1_k}$ or $\mathcal{G}_{1_{k-1}} \subset \mathcal{G}_{1_k}$. From the conditions of the theorem, this implies that $\mathcal{F}_{2_{k-1}} \subset \mathcal{F}_{1_k}$ or $\mathcal{G}_{2_{k-1}} \subset \mathcal{G}_{1_k}$. That is, $\mathcal{F}_{2_{k-1}} \subset \mathcal{F}_{2_l}$ or $\mathcal{G}_{2_{k-1}} \subset \mathcal{G}_{2_l}$. Thus, $\mathcal{M}_2$ cannot terminate at Step $k-1$. $\square$

The theorem does not hold in the only-if direction because we can choose arbitrary hidden states. Typically, considering methods in the literature, the only-if direction also holds. However, it is difficult to identify abstract conditions on hidden states that allow us to prove the claim in the only-if direction.

We identify two extreme classes of methods.

Definition 4.4 An evaluation method is said to be *maximally parallel* if no other method that implements the same choice of sips is more parallel in our abstract model of computation.

Definition 4.5 An evaluation method is said to be *sequential* if it is not more parallel in our abstract model of computation than any other method that implements the same choice of sips.

5 Bottom-Up Evaluation

The bottom-up approach that we consider is to take the program $\langle P, Q\rangle$, rewrite P according to the choice of sips, and to then evaluate the fixpoint by a bottom-up iteration.

To keep this paper self-contained, we present brief descriptions of the rewriting and iteration phases in this section.

5.1 The Magic Templates Rewriting Algorithm

We present a simplified version of the algorithm, tailored to the case that sips are just partial orderings of the body literals in a rule, and that a single sip is associated with a rule, for all goals that invoke this rule. The reader is referred to [Ra88] for a more general algorithm capable of implementing more sophisticated sip choices, and also for a detailed discussion of bottom-up fixpoint computation in the presence of non-ground facts.

The idea is to compute a set of auxiliary predicates that contain the goals. The rules in the program are then modified by attaching additional literals that act as filters and prevent the rule from generating irrelevant tuples.

Definition 5.1 The Magic Templates Algorithm
We construct a new program P^{mg}. Initially, P^{mg} is empty.

1. Create a new predicate *magic_p* for each predicate p in P. The arity is that of p.

2. For each rule in P, add the *modified version* of the rule to P^{mg}. If rule r has head, say, $p(\bar{t})$, the modified version is obtained by adding the literal $magic\_p(\bar{t})$ to the body.

3. For each rule r in P with head, say, $p(\bar{t})$, and for each literal $q_i(\bar{t}_i)$ in its body, add a *magic rule* to P^{mg}. The head is $magic\_q_i(\bar{t}_i)$. The body contains all literals that precede q_i in the sip associated with this rule, and the literal $magic\_p(\bar{t})$.

4. Create a *seed* fact $magic\_q(\langle\bar{c}\rangle)$ from the query.

Example 5.1 Consider the following program.

$$sg(X,Y) \coloneq flat(X,Y).$$
$$sg(X,Y) \coloneq up(X,U), sg(U,V), down(V,Y).$$
$$sg(john, Z)?$$

For a choice of sips that orders body literals from left to right, as in Prolog, the Magic Templates algorithm rewrites it as follows:

$$sg(X,Y) \coloneq magic\_sg(X,Y), flat(X,Y).$$
$$sg(X,Y) \coloneq magic\_sg(X,Y), up(X,U), sg(U,V), down(V,Y).$$
$$magic\_sg(U,V) \coloneq magic\_sg(X,Y), up(X,U).$$
$$magic\_sg(john, Z).$$

□

We have the following results characterizing the transformed program P^{mg} with respect to the original program P, from [Ra88].

Theorem 5.1 [Ra88] *$\langle P, Q\rangle$ is equivalent to $\langle P^{mg}, Q\rangle$ with respect to the set of answers to the query.*

Definition 5.2 Let us define the *Magic Templates Evaluation Method* as follows:

1. Rewrite the program $\langle P, Q\rangle$ according to the choice of sips using the Magic Templates algorithm.

2. Evaluate the fixpoint of the rewritten program.

We hope that the slight abuse of notation in having the same name for the evaluation method and the rewriting algorithm will not lead to confusion; the distinction should be should be clear from the context. The second step is presented in more detail in the next subsection.

Theorem 5.2 [Ra88] *The Magic Templates Evaluation Method is a complete sip-method.*

The careful reader will notice that some joins are repeated in the bodies of rules defining magic predicates and modified rules. The *supplementary* version of the rewriting algorithm essentially identifies these common subexpressions and stores them (with some optimizations that allow us to delete some columns from these intermediate, or supplementary, relations). We refer the reader to [BeR87] for details, with the remark that the variant is similar to the basic Magic Templates algorithm with respect to parallelism.

5.2 Seminaive Iteration

We describe Seminaive iterative fixpoint evaluation, which is a refinement of ordinary bottom-up fixpoint evaluation. The main difference is that derivations are not repeated in subsequent iterations, through the use of duplicate elimination. We follow the presentation in [MR89] in the rest of this section, with some simplification. [1]

Let us first define a binary operator W_P, whose role is similar to that of the well-known T_P operator of [vEK76]:

$$\begin{aligned} W_P(X,Y) = \{h\theta \mid \quad & h \ :-\ b_1, \ldots, b_k \text{ is a rule of } P, \\ & \theta \text{ is mgu of } (b_1, \ldots, b_k) \text{ and } (d_1, \ldots, d_k), \\ & Y \subseteq X, \{d_1, \ldots, d_k\} \subseteq X, \text{ and} \\ & \text{if } k \neq 0 \text{ then } \{d_1, \ldots, d_k\} \cap Y \neq \emptyset.\} \end{aligned}$$

Intuitively, W_P only allows deductions from the set of facts X that use the "new" facts Y. We now define Seminaive iteration. In the following definition, *set* is an operator that takes a multiset and returns a set.

Definition 5.3 Seminaive Iteration

Let $S_{-1} = S_0 = \delta_0 = \emptyset$.

$$\begin{aligned} \delta_{n+1} &= set(W_P(S_n, \delta_n - S_{n-1})) \\ S_{n+1} &= set(S_n \cup \delta_{n+1}) \\ S &= dup\_elim(\lim_{n\to\infty} S_n) \\ GC &= set(S) \end{aligned}$$

The set GC is the set of *generated consequences* of the program.

[1] In particular, the operator W_P is defined in [MR89] as a multiset constructor. For our purposes here, the cardinality of the elements is not important, and we treat W_P as a set constructor.

The set of facts produced in iteration n (δ_n) is compared with the set of known facts (S_n) to identify the new facts produced ($\delta_n - S_{n-1}$). Only derivations that use one of these new facts are carried out in iteration $n+1$. This avoids generating many duplicate facts by avoiding repeated derivations; remaining duplicates are handled by the set data structure. The algorithm terminates (at step $n+1$) when $S_{n+1} = S_n$; we must test whether $\delta_{n+1} \subseteq S_n$. Consequently Seminaive Iteration terminates iff S is finite.

Let *ground* be an operator that takes a set of possibly non-ground facts and returns the set of ground facts that are instances of the input set. the following well-known result shows that Seminaive Iteration is consistent with the usual least Herbrand model semantics of [vEK76].

Proposition 5.3 *The set of generated consequences GC of a program computed using Seminaive Iteration is such that $ground(GC) = M$.*

We note that ordinary fixpoint evaluation corresponds to the case where the second argument of W_P in the definition of Seminaive Iteration is replaced by S_n. In the sequel, when we refer to bottom-up fixpoint evaluation, it is not essential that the Seminaive variant be used; our results hold with either Seminaive or ordinary fixpoint evaluation. We have presented Seminaive Iteration, rather than just ordinary fixpoint evaluation, for two reasons. First, it illustrates an important advantage that is obtainable with computations that retain generated facts, and also makes explicit the associated operation of eliminating duplicates, which could be expensive. Second, it allows us to discuss some difficulties with this refinement in the context of parallel computations (Section 8).

6 Comparing Methods

We now present some results characterizing the parallelism obtained by some proposed evaluation methods, using the abstract model of computation and measure of parallelism developed in Section 4.

We remark that in this section, positive results, of the form that one method is more parallel than another, are typically proved by an induction on the height of derivation trees for the program. Negative results, of the form that some degree of parallelism cannot (always) be achieved by a method, are typically established by considering an example and proving that the claim holds on this program.

Our first result provides strong evidence in favor of the Magic Templates approach to parallel evaluation. We show that rewriting a program using the Magic Templates algorithm and then computing the fixpoint bottom-up realizes all the parallelism allowed by the choice of sips.

Theorem 6.1 Parallelism of Magic Templates
The Magic Templates evaluation method is maximally parallel.

Proof Let us denote the bottom-up fixpoint evaluation of the rewritten program as $\mathcal{M}$. Let $\mathcal{F}_{1_i}$ and $\mathcal{G}_{1_i}$ denote the set of facts and goals in the state of the computation sequence S_1 at Step i, according to any other sip method that uses the same choice of sips. Let $\mathcal{F}_i$ and $\mathcal{G}_i$ denote the corresponding sets for the computation sequence S according to $\mathcal{M}$. For all i less than or equal to the length of S, we will prove that $\mathcal{F}_{1_i} \subseteq \mathcal{F}_i$ and $\mathcal{G}_{1_i} \subseteq \mathcal{G}_i$. The proof proceeds by induction on i. As a basis, the sets of known facts and goals in the initial state are identical. The induction step relies on the structure of rules defining "magic" predicates, and the fact that the hidden state for a bottom-up computation is always `T` since all goals and facts are visible (in the form of facts, the distinction no longer being significant) to all subcomputations.

Let the claim be true for $i < n$, and consider a computation sequence of length n in which the nth step is of the form $\langle \mathcal{F}_{1_{n-1}}, \mathcal{G}_{1_{n-1}}, \mathcal{H}_{1_{n-1}} \rangle \vdash_{\mathcal{S}_1} \langle \mathcal{F}_{1_n}, \mathcal{G}_{1_n}, \mathcal{H}_{1_n} \rangle$. From the induction hypothesis, $\mathcal{F}_{1_{n-1}} \subseteq \mathcal{F}_{n-1}$ and $\mathcal{G}_{1_{n-1}} \subseteq \mathcal{G}_{n-1}$. By the definition of rule application, every fact in $\mathcal{F}_{1_n} - \mathcal{F}_{1_{n-1}}$ can be derived by a single application of a rule in the original program using the facts and goals in $\mathcal{F}_{n-1}$ and $\mathcal{G}_{n-1}$. By construction of the rewritten program, the modified rules are identical to the original rules except that they contain an additional magic literal. This literal unifies with the goal used in deriving one of these new facts. A similar argument based on the definition of a rule application for generating goals, and the structure of magic rules, shows that every goal in $\mathcal{G}_{1_n} - \mathcal{G}_{1_{n-1}}$ can be derived in a single application of a magic rule. This completes our induction, and the proof of the theorem. $\square$

Next, we consider similar results for other proposed evaluation methods. Let us define a *memoing* method to be one that maintains a copy of all generated goals and facts. The next theorem indicates that memoing reduces the length of the computation sequence (and hence improves the parallelism, by our measure) because it avoids recomputation of goals. In essence, the result tells us that the effect of redundant work done by non-memoing methods on the length of the computation cannot be compensated for simply through the use of parallelism, even if unlimited resources are available (enabling us to exploit all available parallelism).

Theorem 6.2 Power of Memoing
No non-memoing method is maximally parallel.

Proof The bottom-up evaluation of $\langle P^{mg}, Q \rangle$, a memoing method, obtains more parallelism on the following program $\langle P, Q \rangle$:

$p(X,Y) \coloneq q1(X,Z), q2(Z,Y).$
$q1(X,Y) \coloneq b(X,Y).$
$q2(X,Y) \coloneq q1(X,Z), q3(Z,Y).$
$q3(X,Y) \coloneq b(X,Y).$
$b(5,5).$
$p(5,U)?$

That is, the length of the computation sequence according to a non-memoing method must be greater than the length of the computation sequence according to Magic Templates. Intuitively, when the goal $q1(5, Z)$? is generated a second time, in Rule 3, the solution $q1(5, 5)$ has already been generated. Bottom-up evaluation can use this solution directly at the next step to identify the goal $q3(5, Y)$?. On the other hand, without memoing, we must resolve this goal (generating the subgoal $b(5, Z)$? and the facts $b(5, 5), q1(5, 5)$ in subsequent steps) before we can identify the goal $q3(Z, Y)$?. $\Box$

The difference in the program used in the above proof can be significant if the computation of the goal $q1(5, Z)$? is expensive. The following result shows that the length of the computation sequence according to a non-memoing method may not even be polynomial in the length of the computation sequence according to bottom-up (memoing) evaluation of the rewritten program. This is not surprising if we require that both methods use the same sip: Consider the well-known Fibonacci program. It is easy to see that the bottom-up method is polynomial and that the non-memoing method is exponential in terms of the number of inferences. If we choose a left-to-right sip for the recursive rule, the computation is made sequential, and the difference in the number of inferences directly translates into a difference in the length of the computation sequence. The following result is stronger in that it is independent of the choice of sips. That is, there are programs such that the difference cannot be bridged by any choice of sips for the non-memoing method.

Theorem 6.3 *Consider a logic program $\langle P, Q\rangle$. Let the length of the computation sequence (in our abstract model, for some choice of sips) of the bottom-up evaluation of $\langle P^{mg}, Q\rangle$ be m, and let the length for computation according to a non-memoing method for some choice of sips be n. In general, the function g such that $n = g(m)$ is at least exponential, independent of the choice of sips for the non-memoing method.*

Proof Consider the following refinement of the Fibonacci program:

$f(1, N, X, W)$:- $N \geq 2, f(1, N-1, X1, V), f(V, N-2, X2, W), X = X1 + X2$.
$f(0, N, X, 0)$:- $g(N^N, X)$.
$f(1, 1, 1, 1)$.
$f(1, 0, 0, 1)$.
$g(N + 1, X + 1)$:- $g(N, X)$.
$g(0, 0)$.
$f(1, n, Z, T)$?

The second argument of the query, n, denotes a constant. Thus, the first two arguments are bound to constants, and the last two are free. If we consider the left-to-right sip for the recursive rule, we can show by induction that the length of the computation sequence for bottom-up evaluation of the rewritten program using Magic Templates is linear in n. The key observation is that when the first f literal is solved, V is always bound to 1, and so the first rule for f is never invoked. On the other hand, the length of the computation according to a non-memoing left-to-right evaluation is exponential, because the number of calls is exponential, and there is no parallelism since each transition identifies a single goal or a fact. This is essentially the behaviour of a method like Prolog. The only other possible sips are right-to-left and parallel evaluation of the two body f-literals. Both of these are exponential in n, because the first argument of the second call is always free, and the call can be matched with the second rule. This leads to a sequential computation that has an exponential number of steps. $\Box$

Since evaluation under the Reduce-Or model does not do memoing, the previous theorems show that it is not maximally parallel, and that the length of a computation may be exponentially longer than that of a computation according to the Magic Templates method.

Kalé discusses the parallelism obtained by several methods in [Ka87b], but without reference to a precise measure of parallelism, and the following theorems may be viewed as formalizations of the discussion in that paper. A method is said to do no memoing if no generated goal or fact is ever saved, except answers to the query. Thus, if we view the And-Or tree of computation, a new goal or fact is used to further instantiate a rule and then discarded.

Theorem 6.4 *Evaluation according to the Reduce-Or Model is maximally parallel relative to the class of methods that do not do any memoing.*

Proof Let us denote evaluation according to the Reduce-Or model as $\mathcal{R}$. Let $\mathcal{F}_{1_i}$ and $\mathcal{G}_{1_i}$ denote the set of facts and goals in the state of the computation sequence S_1 at Step i, according to any other non-memoing sip method that uses the same choice of sips. Let $\mathcal{F}_i$ and $\mathcal{G}_i$ denote the corresponding sets for the computation sequence S according to $\mathcal{R}$. For all i less than or equal to the length of S, we will prove that $\mathcal{F}_{1_i} \subseteq \mathcal{F}_i$ and $\mathcal{G}_{1_i} \subseteq \mathcal{G}_i$. The proof proceeds by induction on i. As a basis, the sets of known facts and goals in the initial state are identical.

Let the claim be true for $i < n$, and consider a computation sequence of length n in which the nth step is of the form $\langle \mathcal{F}_{1_{n-1}}, \mathcal{G}_{1_{n-1}}, \mathcal{H}_{1_{n-1}} \rangle \vdash_{\mathcal{S}_1} \langle \mathcal{F}_{1_n}, \mathcal{G}_{1_n}, \mathcal{H}_{1_n} \rangle$. From the induction hypothesis, $\mathcal{F}_{1_{n-1}} \subseteq \mathcal{F}_{n-1}$ and $\mathcal{G}_{1_{n-1}} \subseteq \mathcal{G}_{n-1}$. By the definition of rule application, every fact in $\mathcal{F}_{1_n} - \mathcal{F}_{1_{n-1}}$ can be derived by a single application of a rule in the original program using the facts and goals in $\mathcal{F}_{n-1}$ and $\mathcal{G}_{n-1}$. However, we must consider the effect of the hidden state, which may preclude some of these derivations. Indeed, some of the facts and goals in $\mathcal{F}$ and $\mathcal{G}$ cannot be used in rule applications at Step n because they have been discarded.

However, since S_1 is a computation sequence for a non-memoing method, the sets of facts and goals that are available for rule applications at Step n

— i.e. rule applications that are permitted by the hidden state $\mathcal{H}_{1_{n-1}}$ — are subsets of $\mathcal{F}_{1_{n-1}}$ and $\mathcal{G}_{1_{n-1}}$. In terms of the extended And-Or Tree model of the computation (Sections 3.1, 3.2, 3.3), the effect of the hidden state is most easily stated in terms of the And-Or tree underlying the computation:

1. Facts are generated at And-nodes that have at least one child Or-node per body literal.
   The subset of $\mathcal{F}_{1_{n-1}}$ that is generated at (the children And-nodes of) the children Or-nodes of an And-node a is available for rule applications at Step n, using the (instantiated) rule that labels node a. (The instantiation of the rule corresponds to the goal that is used; this goal must be in $\mathcal{G}_{1_{n-1}}$.)

2. Goals are generated at And-nodes, and result in the creation of children Or-nodes.
   Consider a goal g that is generated at an And-node a, from a body literal l. The subset of $\mathcal{F}_{1_{n-1}}$ that is generated at (the children And-nodes of) the children Or-nodes of predecessors of l — in the partial order associated with the rule by the chosen sip — is used to further instantiate the rule labelling a in order to generate g.

The subsets of $\mathcal{G}_{1_{n-1}}$ that are available at Step n are identified implicitly in the above conditions through the rule instantiations that label And-nodes. It is straightforward to show that the Reduce-Or method allows the above subsets of $\mathcal{F}_{1_{n-1}}$ and $\mathcal{G}_{1_{n-1}}$ to be used in derivations at Step n. (The details of the proof of this claim follow from the full description of the Reduce-Or method; the reader is asked to consult [Ka87a].) This completes our induction, and the proof of the theorem. □

Theorem 6.5 *Evaluation according to a non-memoing method that exploits only And- or Or- parallelism, but not both, is strictly less parallel than evaluation according to the Reduce-Or process model.*

Proof The proof is similar to that of Theorem 6.2; we present an example on which the computation sequence of the Reduce-Or method is shorter than that of any non-memoing method that does not exploit both And- and Or-parallelism, i.e., generate both And- and Or- parallel steps. Consider the following program, with a sip for the first rule that orders the $gen1$ and $gen2$ literals before the $test$ literal, and the goal $p(X)?$:

$p(Z)$:- $gen1(X), gen2(Y), test(X,Y,Z)$.
$test(X,Y,Z)$:- $f1(X,Y,Z)$.
$test(X,Y,Z)$:- $f2(X,Y,Z)$.
$gen1(5)$.
$gen2(6)$.
$f1(5,6,7)$.
$f2(5,6,8)$.

The Reduce-Or method generates the following computation sequence:
$\{p(X)?\} \vdash \cup\{gen1(X)?, gen2(Y)?\} \vdash \cup\{gen1(5), gen2(6)\}$
$\vdash \cup\{test(5,6,Z)?\} \vdash \cup\ \{f1(5,6,Z)?, f2(5,6,Z)?\}$
$\vdash \cup\{f1(5,6,7), f2(5,6,8)\} \vdash \cup\{test(5,6,7)?, test(5,6,8)?\} \vdash \cup\{p(7), p(8)\}$
The only parallelism available is in the And-parallel step that concurrently identifies the $gen1$ and $gen2$ goals, and the Or-parallel step that concurrently identifies the $f1$ and $f2$ goals. Any method that does not generate either And- parallel or Or- parallel steps must sequentialize one of these steps into two transitions, and must therefore have a computation sequence that is longer than the above sequence by at least one transition. $\square$

From the proof of the above theorem, it is easy to see that there are programs on which the computation sequence for a purely Or-parallel method must be longer than the computation sequence for a method that utilizes And- parallelism. (For example, the program in the proof with the following changes: delete the *test* literal from the first rule, and delete all rules used to define the predicate *test*.) Similarly there are programs on which a purely And-parallel method must have a longer computation sequence than a method that utilizes Or-parallelism. (For example, the subprogram that defines the predicate *test*, with the goal $test(5,6,Z)?$.) This observation leads to the following proposition, which illustrates a limitation of our measure of parallelism, in that it does not allow us to compare certain pairs of evaluation methods.

Proposition 6.6 *Consider a method whose allowed transitions contain no Or-parallel steps, and one whose allowed transitions contain no And-parallel steps. Let both methods be more parallel than a sequential method. Then, neither method is more parallel than the other.*

6.1 Methods That Sacrifice Restriction for Parallelism

We present a result that indicates why we chose a definition of a sip-method that differs from the definition in [Ra88]. It also illustrates the trade-off between restricting search and parallelizing the computation.

Let us relax our definition of a sip-method in this subsection to also include methods that compute a set of the facts and goals, say $\mathcal{F}_1$ and $\mathcal{G}_1$, such that $ground(\mathcal{F} \cup \mathcal{G}) \subseteq ground(\mathcal{F}_1 \cup \mathcal{G}_1)$, where $\mathcal{F}$ and $\mathcal{G}$ are the sets that must be computed according to the definition of a sip-method in Section 4. This allows us to consider methods that are not sip-optimal, in that they do not eliminate all computation that is irrelevant according to the sips. As an extreme example, the bottom-up evaluation of the original program can be seen to implement any choice of sips (extremely inefficiently), since we can view it as generating a goal containing a vector of n distinct variables for each n-ary predicate, and obtaining all solutions. Thus, every possible goal with predicate name p is an instance of this most general goal for p. Intuitively, this allows us to work on all relevant goals immediately, but at the cost of additionally working on irrelevant goals. From the proof of Theorem 6.3, it is easy to see that any irrelevant computation can be made arbitrarily complex, even non-terminating, and thus the unrestricted computation sequence could be much longer than a restricted computation sequence. Thus, bottom-up evaluation of the original program is not necessarily more parallel than another evaluation method, by our measure of parallelism. This is pertinent when we wish to compute all answers and terminate, or if (as is likely) resources are limited. However, termination is in general undecidable, and even the restricted computation may not terminate. If resources are (effectively) unlimited, and we are only interested in obtaining answers as soon as possible, then, it might be worth evaluating the fixpoint of the original program without rewriting it to restrict the computation. This is justified by the following simple proposition.

Proposition 6.7 *Consider a logic program $\langle P, Q \rangle$. Let C_1 be the computation sequence in our abstract model for bottom-up fixpoint evaluation of this program, and let C_2 be the computation sequence for some other evaluation method, for some choice of sips. If a goal or fact appears at Step n in C_2, then it is subsumed by some fact that appears in C_1 at Step m, for some $m \leq n$.*

6.2 A Remark About Magic Templates

The above results lead us to the following observation.

Remark: A claim such as Theorem 6.1 cannot be made for any other evaluation method that we are aware of. (It is possible to extend some of the methods so that such a claim holds for them.)

Such a remark is tedious to prove given the number of proposed methods, and so we simply offer an informal justification. First, from Theorem 6.2 it follows that we need only consider memoing methods as candidates. Of these, Alexander Templates [Se89] is the only one (other than Magic Templates) that is capable of dealing with non-ground facts. Examples are readily found where dealing with such facts is necessary to restrict search as per the sips we consider. Alexander Templates, like Magic Templates, rewrites the program and then evaluates the fixpoint, but it cannot deal with And-parallelism since it only allows left-to-right sips.

We note that this remark should be read with all the limitations of sip-methods and our measure of parallelism in mind. In particular, some methods, e.g., implementations of committed-choice languages, cannot be described as sip-methods, and our measure is an abstract metric that does not take several implementation overheads and resource constraints into account. However, limiting ourselves to the class of sip-methods (a broad class that includes the methods surveyed in Section 3), we believe that this observation is significant. First, as Kalé observes [Ka87a], identifying the available parallelism is a useful first step; it remains to consider efficient realizations.

In this, we believe that the Magic Templates method offers considerable flexibility since it frees us from the constraints imposed by maintaining a network of processes and associated binding environments. We consider this point further in Section 8.

7 A Closer Look at Sip-Methods

We have restricted out attention to evaluation methods that are sip-methods. This has allowed a fundamental separation of concerns: the sips specify the order in which rules are to be evaluated, that is, how bindings are to be propagated in order to restrict the computation, and the evaluation method implements this decision (a step that includes some choice of a control strategy). Not all proposed evaluation methods qualify as sip-methods. We now consider behaviour that cannot be captured by sip-methods, and attempt to extend our definition of sips, simultaneously indicating the necessary changes to the Magic Templates method. These extensions preserve the essential separation of concerns in the sip paradigm of computation. There are certain evaluation methods, however, whose behaviour we cannot capture even with the extended definitions of sips. We examine this and observe that there are some fundamental limitations to the sip paradigm; this implies that certain top-down methods cannot be mimicked by rewriting followed by fixpoint evaluation.

7.1 Multiple Sips Per Rule

Let us return to the survey in Section 3, and the discussion of And-Or trees. We made the assumption that for each And-node, there was a unique partial order that determined by the associated label. That is, each rule in the program has a unique partial ordering according to which the body literals are to be solved in any invocation of the rule. We could relax this assumption in several ways. Consider the set of possible goals with predicate name p. We could partition this set into several — preferably, but not necessarily, non-intersecting — subsets. For each rule defining p, for each such subset, we could choose a sip that indicates the order in which body literals are to be solved when the this must be done for each subset of goals.

One way to partition the set of goals is by means of a compile-time analysis that indicates which argument positions we expect to be bound. This leads to a notion of "bound" and "free" arguments, similar to "input" and "output" modes, that has been proposed and used by a number of researchers. (More sophisticated compile-time classification has been explored in [MFPR89].) We note that [Ra88] incorporates such an analysis into the Magic Templates algorithm. Recall that the algorithm adds a modified rule and a set of magic rules for each rule in the original program. If we wish to use a different ordering of body literals for goals in different subsets, in essence a modified rule and magic rules must be added for each subset.

All of the methods in Section 3 choose sips at compile time.

7.2 Dynamic Sips

It is possible that the choice of the order in which the body literals are to be solved is made at run-time when the rule is invoked. We briefly outline one way to incorporate this into the Magic Templates algorithm. The crux of the problem is that for each rule, we may wish to choose a different partial order at run-time for each goal. Noting that there are only a finite number of different partial orders over a finite set, we could simply generate modified and magic rules corresponding to each partial order. Now, we must determine which group of modified and magic rules is to be used for solving a given goal. To do this, we observe that the goal is described in these rules by a magic literal in the body, say $mp(\bar{t})$. We now add an additional literal $classify_{r\_s}(\bar{t})$ to the body. The s subscript denotes the subset of goals, and the corresponding choice of sips or partial ordering, for which this (modified or magic) rule was generated. If $p(\bar{t})$? is a generated goal, $classify_{r\_s}(\bar{t})$ must be true for some s (since it must be a member of one of the subsets of goals that we consider).

In effect, we have taken advantage of the finite number of partial orders to rewrite the program at compile time. However, we have abandoned a static classification of goals based on a compile-time analysis, such as "bound" and "free" arguments, in favor of a dynamic classification. We remark that this is not necessarily a win; our objective here is to examine the limits of the sip paradigm of computation, which we believe is essentially reached with the above formulation of dynamic sips.

7.3 Limitations of the Sip Paradigm

These limitations are seen when we examine evaluation methods that reorder goals in And-Or trees dynamically, but they can also be observed with a static ordering. Let us return to the discussion of And-Or trees, and consider And-nodes again. Let p and q be two body literals in the label of an And-node. Let $p1$ and $p2$ be body literals in a descendant And-node of p, and similarly $q1$ and $q2$ for q. A sip-method, even one that uses the dynamic sip selection of the previous subsection, must either order both $p1$ and $p2$ ahead of both $q1$ and $q2$, order the q's ahead of the p's, or leave the p's unordered relative to the q's. In particular, an evaluation method that requires the following solution order is not a sip-method: $p1, q1, p2, q2$.

This limitation arises, of course, because the sip mechanism only allows us to order goals that arise as body literals in a single rule. All subgoals of these goals must respect the above order. The sip formalism does not allow us to consider the resolvent that is the set of all subgoals and then pick an arbitrary order.

This is precisely what committed choice languages such as Parlog [CG86] and Concurrent Prolog [Sh86], the *freeze* primitive in Nu-Prolog [Na87], and some other proposed methods, e.g. in [Co83], achieve by dynamically suspending and starting goals. The ordering is controlled typically by variable annotations that, for example, suspend a goal until one of its variables is instantiated [CG86, Sh86, Na87]; it can also be controlled by a sophisticated run-time scheduler [Co83].

Methods that use annotations typically sacrifice completeness. Completely unrestricted dynamic re-ordering carries a high run-time overhead. Nevertheless, there may be situations where such approaches perform better than any sip-method.

Returning to our example above and the ordering of goals "$p1, q1, p2, q2$", we remark that this amounts to *bi-directional* binding propagation between the goals p and q. Indeed, the solution of $p1$ could bind some variables in $q1$ and this in turn could generate bindings for variables in $p2$. It may be possible to capture some limited amount of bi-directionality through a program rewriting algorithm, at some cost.

Example 7.1 Consider the following program:

$$\begin{array}{l} p(X,Z) \text{ :- } q1(X,Y,Z), q2(X,Y). \\ q1(X,Y,Z) \text{ :- } b1(X), b2(Y,Z). \\ q2(X,Y) \text{ :- } b3(X,Y). \\ p(U,V)? \end{array}$$

If we build the And-Or tree for this program, we obtain a frontier with nodes $b1(X)?$ and $b2(Y,Z)?$, generated from node $q1(X,Y,Z)?$, and node $b3(X,Y)?$, which is generated from node $q2(X,Y)?$. An ordering that cannot be realized by a sip-method is "$b1(X)?, b3(X,Y)?, b2(Y,Z)?$". The following transformed program achieves this ordering:

$$\begin{array}{l} p(X,Z) \text{ :- } q1(X,Y,Z), q2(X,Y). \\ q1(X,Y,Z) \text{ :- } r_{q1}(Y), b1(X), b2(Y,Z). \\ q2(X,Y) \text{ :- } r_{q2}(X), b3(X,Y). \\ r_{q2}(X) \text{ :- } b1(X), b2(\_,\_). \\ r_{q1}(Y) \text{ :- } q2(X,Y). \\ p(U,V)? \end{array}$$

The above program can be systematically derived from the following program, using optimizations presented in [RBK88], but we do not pursue this point here.

$$\begin{array}{l} p(X,Z) \text{ :- } q1(X,Y,Z), q2(X,Y). \\ q1(X,Y,Z) \text{ :- } r_{q1}(Y), b1(X), b2(Y,Z). \\ q2(X,Y) \text{ :- } r_{q2}(X), b3(X,Y). \\ r_{q2}(X) \text{ :- } q1(X,Y,Z). \\ r_{q1}(Y) \text{ :- } q2(X,Y). \\ p(U,V)? \end{array}$$

□

8 Pragmatics

We briefly discuss several practical considerations.

8.1 Overheads

There are a number of important differences in the overheads associated with top-down and bottom-up evaluation. Top-down evaluation uses a recursive control strategy. A sequential implementation such as Prolog uses stacks to manage goals. Parallel methods generate a new process each goal, which carries a significant overhead on most systems. (Token based methods, e.g. [CH84], have their own additional overheads such as managing shared environments.) Bottom-up methods do not create a process per goal, but they recover the connections between facts and goals by explicit additional joins. This is typically also done by top-down methods that do memoing and aim to exploit both And- and Or- parallelism; however, significant optimization is possible in methods that only exploit a limited form of And-parallelism that results in a single binding for each variable at any point in the execution. A full discussion of these issues is beyond the scope of this paper. Some of these issues are considered in more detail in [NR89].

We note that the results in this paper depend upon the availability of sufficient resources to exploit all available parallelism. In the case that resources are limited, as is likely, the actual parallelism obtained will be curtailed by how efficiently the computation can be mapped onto the resources. For example, in evaluating the fixpoint of a program, a widely used refinement is *Seminaive* evaluation, which avoids repeating inferences. The usual formulation of Seminaive evaluation proceeds in phases, in that facts produced in one iteration are not used until the next iteration. While it is possible to use some facts in the same iteration that they are produced, it is in general a useful restriction to maintain a clear separation between phases. If resources are limited, and we cannot assume that an iteration is completed in one step, then the mapping of the computation onto the available resources affects the parallelism that is obtained.

8.2 Load Sharing in Bottom-up Evaluation

The problem of mapping a bottom-up fixpoint computation onto a fixed set of processors has received attention lately. While considering this work is beyond the scope of this paper, we remark that the interactions of the techniques used in this work and the Magic Templates algorithm remain little understood and suggest an area for further study. We direct the interested reader to [WS88, CW89, Do89, GST89].

8.3 Some Added Advantages of Memoing

The remarks in this subsection apply equally to top-down and bottom-up methods that do memoing.

As we have already seen, memoing offers gains in terms of avoiding redundant computation and increased parallelism. It also offers other important advantages:

1. *Multiple Query Optimization* Multiple queries may be seen as providing multiple "seeds" for the Magic Templates algorithm. Redundancy is avoided as before, whether it arises in the computation of one of the queries alone, or whether it arises due to common subcomputations in different top-level queries. In either case, some goal is generated more than once, and can be discarded after the first time, as before, if we have memoed the goal and solutions to it.

2. *Incremental Evaluation* If we wish to re-evaluate a query after adding some facts or rules to the program, the memoed results of the previous evaluation naturally enable us to avoid much recomputation. In the context of the Magic Templates algorithm, all memoed results can be taken to be assertions. Re-evaluation after deletions is more difficult, but some analysis of the affected predicates may allow us to retain many of the memoed relations.

3. *Improved Termination Properties* It is possible that memoing makes the difference between termination and non-termination. For example, consider the following program:

   $t(X, Y) :\!-\ t(X, Z), b(Z, Y).$
   $t(X, Y) :\!-\ b(X, Y).$
   $t(5, Y)?$

   This is a program on which Prolog will not terminate, repeatedly generating the goal t(5,Z)?, but memoing enables us to recognize that the goal has been generated before, and thereby devise modifications to Prolog that do terminate (e.g., see [Vi89]). In fact, this causes Prolog to be incomplete. We note that memoing is not essential for completeness; the Reduce-Or model [Ka87a] is complete, although it does not memoing. This is essentially because all paths are explored in parallel, and so even if some paths are non-terminating — and will never produce new solutions — all paths that do produce solutions are considered. However, the Reduce-Or computation will not terminate on this program. Memoing methods, including Magic Templates, terminate on it.

9 Conclusions

The main contributions of this paper are: (1) an abstract model of computation that allows us to make precise the degree of parallelism that is obtained by several proposed evaluation methods, (2) comparisons between methods based on this model, including the result that the Magic Templates algorithm is maximally parallel in this model, and (3) a discussion of the limitations of the abstract model, and in particular, the limitations of the sip paradigm on which the model is based.

In summary, we believe that the results provide strong motivation for a careful study of parallel evaluation of logic programs based on rewriting and subsequent fixpoint evaluation, as well as a sound basis for comparisons of parallelism in various logic program evaluation methods.

Acknowledgements

Conversations with Catriel Beeri, Sanjay Kalé, Michael Kifer, Jeff Naughton, Divesh Srivastava and S. Sudarshan have influenced this paper. I thank them for their generous input.

A preliminary version of this paper appeared in the proceedings of POPL 90. This work was supported in part by an IBM Faculty Development Award, a David and Lucile Packard Foundation Fellowship in Science and Engineering and NSF grant IRI-8804319.

References

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman, Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. ACM Symposium on Principles of Database Systems, pages 1–15, Boston, Massachusetts, March 1986.*

[BaR86] F. Bancilhon and R. Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. ACM SIGMOD International Conference on Management of Data, pages 16-53, Washington, D.C., 1986.* Revised and reprinted in *Readings in AI and Databases, Eds. M. Brodie and J. Mylopoulos, pages 376-430, 1988.*

[BeR87] C. Beeri and R. Ramakrishnan, On the Power of Magic. In *Proc. ACM Symposium on Principles of Database Systems, pages 269–283, San Diego, California, March 1987.*

[Br89] F. Bry, Query Evaluation in Recursive Databases: Bottom-up and Top-Down Reconciled. *ECRC TR IR-KB-64, April 1989.*

[CDD85] J.-H. Chang, A.M. Despain and D. DeGroot, AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis. In *Digest of Papers, Compcon 85, IEEE Computer Society, Feb. 1985.*

[CH83] A. Ciepielewski and S. Haridi, A Formal Model for Or-Parallel Execution of Logic Programs. In *Information Processing 83, pages 299-305, North-Holland, Sept. 1983.*

[CH84] A. Ciepielewski and S. Haridi, Control of Activities in the Or-Parallel Token Machine In *Proc. IEEE Symposium on Logic Programming, Atlantic City, Feb. 1984.*

[CG86] K.L. Clark and S. Gregory, Parlog: Parallel Programming in Logic. In *Transactions on Programming Languages, pages 1–49, January 1986.*

[CW89] S.R. Cohen and O. Wolfson, Why a Single Parallelization Strategy is Not Enough in Knowledge Bases. In *Proc. ACM Symposium on Principles of Database Systems, pages 200–216, Philadelphia, Pennsylvania, March 1989.*

[Co83] J. Conery, The And-Or Process Model for Parallel Interpretation of Logic Programs. *Ph.D. thesis, TR 204, Univ. of California, Irvine, June 1983.*

[De84] D. DeGroot, Restricted And-Parallelism. In *Proc. Intl. Conf. on Generation Computer Systems, ICOT, 1984.*

[DW87] S.W. Dietrich and D.S. Warren, Extension Tables: Memo Relations in Logic Programming. In *Proc. IEEE Symposium on Logic Programming, San Francisco, Sept. 1987.*

[Do89] G. Dong, On Distributed Processing of Datalog Queries by Decomposing Databases. In *Proc. ACM SIGMOD International Conference on Management of Data, pages 26-35, Portland, 1986.*

[EKM82] N. Eisinger, S. Kasif, J. Minker, Logic Programming: A Parallel Approach. In *Proc. First Logic Programming Conference, 1982.*

[GST89] S. Ganguly, A. Silberschatz, S. Tsur, A Framework for the Parallel Processing of Datalog Queries. *Manuscript.*

[HR89] M. Hermenegildo and F. Rossi, On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *Proc. N. American Conference on Logic Programming, pages 369-389, Cleveland, 1989.*

[Ka87a] L.V. Kalé, Parallel Execution of Logic Programs: The Reduce-Or Process Model. In *Proc. Intl. Conference on Logic Programming, pages 616-632, Melbourne, May 1987.*

[Ka87b] L.V. Kalé, Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Proc. IEEE Symposium on Logic Programming, pages 125-133, San Francisco, Sept. 1987.*

[KL86] M. Kifer and E. Lozinskii, A Framework for an Efficient Implementation of Deductive Databases. In *Proc. Advanced Database Symposium, Tokyo, 1986.*

[KL88] M. Kifer and E. Lozinskii, Sygraf: Implementing Logic Programs in a Database Style. In *Trans. on Software Engineering, pages 922-935, July 1988.*

[Lo85] E. Lozinskii, Evaluating Queries in Deductive Databases by Generating. In *Proc. Intl. Joint Conf. on Artificial Intelligence, pages 173-177, 1985.*

[MR89] M. Maher and R. Ramakrishnan, Déjà Vu in Fixpoints of Logic Programs. In *Proc. North American Conference on Logic Programming, pages , 1989.*

[MFPR89] I.S. Mumick, S. Finkelstein, H. Pirahesh and R. Ramakrishnan, Magic Conditions. *In preparation.*

[Na87] L. Naish, Parallelizing Nu-Prolog. *Dept. of Computer Science, Univ. of Melbourne, TR 17, 1987.*

[NR89] J. Naughton and R. Ramakrishnan, A Unified Approach to Logic Program Evaluation. *Technical Report, University of Wisconsin-Madison, 1989.*

[Po81] G.H. Pollard, Parallel Execution of Horn Clause Programs. *Ph.D. thesis, Imperial College of Science and Technology, Univ. of London, 1981.*

[Ra88] R. Ramakrishnan, Magic Templates: A Spellbinding Approach to Logic Programs. In *Proc. Intl. Conference on Logic Programming, pages 140-159, Seattle, Washington, August 1988.*

[RBK88] R. Ramakrishnan, C. Beeri and R. Krishnamurthy, Optimizing Existential Datalog Queries, In *Proc. 7th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1988.*

[RLK86] J. Rohmer , R. Lescoeur and J.M. Kerisit, The Alexander Method, a Technique for the Processing of Recursive Axioms in Deductive Databases. In *New Generation Computing, 4, 3, pages 273–285, 1986.*

[Se89] H. Seki, On the Power of Alexander Templates. In *Proc. 8th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, pages 150-159, 1989.*

[Sh86] E. Shapiro, Concurrent Prolog: A Progress Report. In *IEEE Computer, pages 44–58, August 1986.*

[Ul89a] J.D. Ullman, Principles of Database and Knowledge-Base Systems, Volumes 1 and 2. *Computer Science Press, 1989.*

[Ul89b] J.D. Ullman, Bottom-Up Beats Top-Down for Datalog, In *Proc. 8th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, pages 140-149, 1989.*

[vG86] A. van Gelder, A Message Passing Framework for Logical Query Evaluation. In *Proc. ACM SIGMOD International Conference on Management of Data, pages 16-53, Washington, D.C., 1986.*

[vEK76] M. van Emden and R. Kowalski, The Semantics of Predicate Logic as a Programming Language. In *JACM 28, no. 4, pages 733-742, Oct. 1976.*

[Vi89] L. Vieille, Recursive Query Processing: The Power of Logic To appear in *Theoretical Computer Science, 1989.*

[WS88] O. Wolfson and A. Silberschatz, Sharing the Load of Logic Program Evaluations. In *Proc. 7th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1988.*