On the Complexity of Or-Parallelism

Desh RANJAN and Enrico PONTELLI and Gopal GUPTA

Laboratory for Logic, Databases and Advanced Programming
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
http://www.cs.nmsu.edu/lldap

{dranjan,epontell,gupta}@cs.nmsu.edu

Received February 15, 1999

Abstract We formalize the implementation mechanisms required to support or-parallel execution of logic programs in terms of operations on dynamic data structures. Upper and lower bounds are derived, in terms of the number of operations n performed on the data structure, for the problem of guaranteeing correct semantics during or-parallel execution. The lower bound $\Omega(\lg n)$ formally proves the impossibility of achieving an ideal implementation (i.e., parallel implementation with constant time overhead per operation). We also derive an upper bound of $\tilde{O}(\sqrt[3]{n})$ per operation for or-parallel execution. This upper bound is far better than what has been achieved in the existing or-parallel systems and indicates that faster implementations may be feasible.

Keywords Or-parallelism, Logic Programming, Complexity, Pointer Machines.

§1 Introduction

Logic programming (LP) is a popular computer programming paradigm that has been used in a wide variety of applications, ranging from Artificial Intelligence, Genetic Sequencing, Database programming, Expert Systems, Natural Language Processing, Constraint based Optimization, to general purpose programming and problem solving. The most popular logic programming language is *Prolog* [35]. A nice property of logic programming languages is that parallelism can be automatically extracted from logic programs by the compiler or the run-time system without requiring any effort from the programmer. However, the implementation of a parallel logic programming system poses many interesting and challenging problems. Several solutions have been proposed for these problems and parallel implementations have been realized in the past (e.g., [20, 19, 1, 24]). To date, however, little attention has been paid to analyzing the complexity of the various operations involved in these implementations.

The execution of a logic program can be abstracted as the process of *building a dynamic tree*. An implementation of logic programming must support the process of (explicitly or implicitly) building these execution trees. The operational semantics of the logic language determines how this tree is built and what operations on the tree are of interest. As execution proceeds according to the operational semantics, this tree grows and shrinks—in a parallel implementation the tree can grow and shrink in parallel. Various operations are needed during parallel execution to guarantee the correct behaviour. For example, at runtime we may need to determine at a particular moment if a given node is in the leftmost branch of the tree that exists at that moment; or, given two nodes, if one node is an ancestor of another; etc. Although dynamic data structures have been studied extensively [5, 10, 38, 16, 34, 37, 9], to the best of our knowledge the specific problems that arise in parallel logic programming have not been formally studied.

In this paper we focus on *Or-Parallelism* [11], a form of parallelism peculiar to logic programming and non-deterministic languages. The overall goal of this research is to create an abstraction of the problem which allows us to study the inherent complexity of the basic operations required to support or-parallelism. We abstract the problems in or-parallel implementation of logic programs as that of supporting certain dynamic data types of interest. This is one of the major contributions of the paper, and allows us to study the complexity of these problems formally, and to state lower and upper bounds for this particular problem. In the following let us denote by N the maximum number of nodes in the dynamic tree (corresponding to the execution) and M the total number of operations. We prove a lower bound of $\Omega(\log N)$ (per operation) on the time-complexity of the operations required to implement even a restricted version of or-parallelism (without *aliasing* of variables). We prove that the *aliasing problem* in or-parallel implementation of Prolog is at least as hard as the union-find problem. This gives us an amortized lower bound of $\Omega(N + M\alpha(M,N))$ for the aliasing problem on pointer machines, due to the results of [22, 37]. We also give a scheme to support a restricted version of or-parallelism in time $\tilde{O}(\sqrt[3]{N})$ per operation. To our knowledge this is the best scheme known to date, and is a substantial improvement over the previous schemes, most of which have a worst-case time complexity of $\tilde{O}(N)$ per operation (see table 1 at the end of the paper). This result indicates that there is scope for producing practical implementation schemes which will perform better than those currently in use. Let us remark that the focus of this work is on determining the cost of the basic operations required to support execution of or-parallelism—i.e., in this work we will not deal with issues like scheduling and control of speculative work [3].

All the results presented are novel, as these problems have never been considered before. The only work that comes close is that of Gupta and Jayaraman [15] that establishes some results on the complexity of realizing an or-parallel implementation of Prolog. Our work advances the results in [15] by providing more precise bounds (which subsume the generic non-constant time results presented in [15]) and providing a more rigorous treatment of the problem. Our work can be viewed in the same spirit as that of Pippenger for Lisp, where he studies the *complexity* of pure vs impure Lisp [26].

The work presented here leaves some open problems [28] (e.g., a wide gap between upper and lower bounds). One of the goals of this paper is to bring these problems to the attention of the programming language theory and data structures communities. We hope this paper will initiate further research in the area of data-structures for dynamic graphs that arise in parallel implementation of declarative languages[*1] and parallel AI systems. It should be pointed out that, while our results have been developed in the context of or-parallelism in logic programming, they are also applicable to other domains such as parallel search in AI, constraint programming, implementation of inheritance in object-oriented programming, etc.

The paper is organized as follows. In the rest of this section we give a brief overview of logic and parallel logic programming. Section 2 describes the major issues in supporting or-parallelism and guaranteeing the correct semantics of execution. Section 3 explains how the problems arising in or-parallelism can be abstracted as that of supporting certain dynamic data types of interest. Typ-

[*1] Dynamic data structures also arise in functional languages; for example, in graph reduction based execution models.

ically these data types are dynamic trees with some logic programming specific operations. The remaining sections present various lower and upper bounds for the problems described.

1.1 Logic Programming

Logic Programming (LP) [23] is a programming paradigm based on a subset of first order logic (Horn Logic). A logic program P consists of a set of (Horn) *clauses* of the form $(A\text{:-}B_1, \ldots, B_n)$, where A and B_i are atomic formulae, and $n \geq 0$. Execution starts with an initial *goal* of the form $(B_1, \ldots, B_k)$ and aims to prove that there exists a substitution, σ, for the variables in the goal such that $(B_1, \ldots, B_k)\sigma$ is a logical consequence of the program P, i.e., $P \models (B_1 \wedge \ldots \wedge B_k)\sigma$. The B_i's in a goal are called *subgoals*. Execution of a logic program is typically based on *SLD-resolution* [31].

In SLD-resolution, a derivation is constructed starting from the goal and using the clauses of the program. At each stage there is a goal (called the resolvent) that needs to be proved. The initial goal is the initial resolvent. At every stage, SLD-resolution attempts to reduce (prove) one subgoal of the resolvent using one clause of the program [23] to produce the new resolvent. Execution terminates when an empty resolvent is obtained.

In pure logic programming, given a goal, we have considerable latitude in choosing which subgoal in the goal to solve next. This is termed *don't-care non-determinism* [23]. Once a subgoal is selected, we also have considerable freedom in choosing which clause to use for reducing it, as heads of many clauses may be unifiable with a given subgoal. This is termed *don't know non-determinism.* The operational semantics of an LP language is determined by the selection strategy chosen for picking a subgoal in the resolvent and the clause to reduce it. The most widely used logic programming language, *Prolog* [8], uses the following selection strategy: at each resolution step the leftmost subgoal in the resolvent is picked, and among the multiple matching clauses, the one that occurs textually first (in the file containing the program) is chosen. Prolog also includes several *extra-logical* features [12] such as built-in side-effect predicates (for input/output, etc.) and meta-programming predicates (cuts, assert, retract, etc.). One of the most important aspects of these features is their strong dependence on the order of execution—i.e., the adoption of different strategies in the selection of clauses and subgoals during resolution will possibly lead to different and incorrect results.

1.2 Parallel Logic Programming

The presence of non-determinism is a peculiar feature of logic programming. The non-determinism present in logic programs can be used as a source of parallelism. Two major forms of parallelism can be identified [11]:

- *And-Parallelism (AP)* [19, 27], which arises from don't care non-determinism. Given a goal, several subgoals can be selected for simultaneous reduction.
- *Or-Parallelism (OP)* [2, 24]: which arises from don't know non-determinism. Given a subgoal G that has been selected from the goal, the multiple clauses whose heads unify with G can be solved in parallel.

A major research direction in parallel logic programming has been the design of systems that exploit parallelism from Prolog programs *implicitly* [20, 11, 19, 2, 24]. This implies that the parallel execution mechanisms have to be developed in such a way that a user sees the same externally observable behaviour during *parallel* execution as is observed during *sequential* Prolog execution. We say that such a parallel system *preserves Prolog semantics*. Thus, in a parallel system that preserves Prolog semantics: (i) the same solutions are produced, in the same order, as in sequential execution; (ii) the same side-effects are observed, in the same order, as in sequential execution.

In this paper, we mainly consider those parallel implementations that aim to preserve Prolog semantics. The most attractive feature of this approach is that Prolog programs can be written for sequential machines and then automatically run in parallel on parallel machines. Parallel implementations of other logic programming languages, and of extensions and modifications of Prolog [17, 33], that require parallelism to be programmed explicitly by the user have also been considered. Nevertheless, most of the implementation issues that arise in the context of implicit exploitation of parallelism also arise during its explicit exploitation—thus the results presented here are also applicable to logic programming systems based on explicit parallelization.

§2 Problems in Parallel Logic Programming

Various parallel implementations of Prolog have been proposed and realized [20]. Most of the current systems are not very efficient. Many of these inefficiencies arise from the need to guarantee Prolog semantics during parallel execution. Other inefficiencies, such as those arising in or-parallelism, are present due to the complexity entailed in implementing don't know non-determinism. Preservation of Prolog semantics, in fact, requires various operations (e.g., execution of side-effects) to be ordered correctly; this creates dependences between the concurrent executions. Only executions that respect all such dependences are acceptable.

2.1 Or-Parallelism

An or-parallel execution can be described as the dynamic development of a tree, called the *or-tree*. Each node contains a goal obtained from the computation. The root node is labeled with the initial goal. To expand a node further, the leftmost subgoal in the goal of that node is selected and the matching clauses are found. For each matching clause, a child node is created. If $B_1, \ldots, B_n$ is the goal at the node, for each clause $H_j : -D_j^1, \ldots, D_j^m$ such that θ_j is the most general unifier of H_j and B_1, a child node labeled with the goal $(D_j^1, \ldots, D_j^m, B_2, \ldots, B_n)\theta_j$ is created. Note also that the child node v_i corresponding to a matching clause C_i that textually precedes another matching clause C_j is placed to the left of v_j, where v_j is the child node corresponding to C_j. Sequential execution corresponds to building the or-tree one node at the time, in depth-first order. OP executions correspond to concurrently growing the or-trees at multiple sites (nodes).

In the or-tree, each branch maintains a local view of the substitution computed. This is essential because, during a reduction like the one mentioned above, the substitutions θ_j produced by unification may potentially conflict and must be kept separate in different branches in the or-tree. For example, if a node containing a resolvent `p(X)` is expanded into two branches due to the two matching clauses,

```
p(1) :- ....                p(2) :- ....
```

then the variable `X` receives a different binding from each of the two clauses. These bindings need to be represented and maintained separately, as they will each lead to a different solution (`X=1` and `X=2`) for the initial resolvent. `X=1` and `X=2` represent different instances of `X` and they need to be separately maintained and properly associated to the correct thread of execution (i.e., the correct branch in the or-tree).

One of the main issues, thus, in designing an implementation scheme capable of efficiently supporting OP is the development of an efficient way of associating the correct set of bindings to each branch of the or-tree. The naive approach of keeping θ_j for each branch is highly inefficient, since it requires the creation of complete copies of the substitution (which can be arbitrarily large) every time a branching takes place [11, 15].

2.2 Side-Effects

Preserving Prolog semantics during or-parallel execution requires also that side-effects are processed in the correct order. The problem of performing parallel execution of logic programs in presence of side-effects has been thoroughly investigated [6, 18, 12]. The various schemes proposed differ in the amount of parallelism that can be exploited even in presence of side-effects. Nevertheless, all the schemes have to address a common basic problem: the need to verify that a given node is *leftmost* with respect to the whole or-tree.

Verifying whether a node is in the leftmost branch of the tree is needed due to the following reason: in order to execute side-effects in the correct order, we need to delay the execution of a side-effect in a branch of the or-tree until all the "preceding" side-effects have been completed ("preceding" in the sense of Prolog's sequential execution order). Unfortunately, the problem of deciding whether all the side-effects which precede a given one have been performed is akin to the problem of deciding termination—thus it is undecidable. Since all the preceding side-effects are guaranteed to lie in the branches of the computation tree on the left of the current one, then the simplest and most effective solution to the problem is to delay the side-effect until the corresponding node in the computation tree is in the leftmost branch of the tree—i.e. until all the preceding computations have been completed. However, this leads to the following problem: how does one efficiently determine at runtime that a particular branch has become leftmost?

§3 Abstraction of the Problems

In this section we formalize the problems arising in OP. As stated earlier execution of a program can be abstracted as building a (rooted, labeled) tree. We will assume that the trees are binary (this assumption doesn't lead to any loss of generality because for a given program, the number of branches at any given node is bounded by some constant). We can then abstract the process of building the tree through the following three operations:

1. $\mathsf{create\_tree}(\gamma)$ which creates a tree containing only the root, with label γ;
2. $\mathsf{expand}(u, \gamma_1, \gamma_2)$ which, given one leaf u and two labels γ_1 and γ_2, creates two new nodes (one for each label) and adds them as children

of u (γ_1 as left child and γ_2 as right child);

3. remove(u) which, given a leaf u of the tree, removes it from the tree.

We assume these three operations are the only ones available to modify the "physical structure" of the tree. During or-parallel execution, multiple processor may execute the expand or remove operations at different nodes concurrently. For our purposes, The tree imposes a partial ordering on its nodes. Given two nodes u and u', we write $u \preceq u'$ if u is an ancestor of u'; $u \prec u'$ additionally says that $u \neq u'$. Additionally, $u \succeq u'$ iff $u' \preceq u$. We will be often referring to the notion of *leftmost branch*. Given a node u, the leftmost branch of the subtree rooted in u can be defined inductively:

- if u does not have any children, then u is the leftmost branch;
- if u has children, and v is the leftmost child, then the leftmost branch is u followed by the leftmost branch of v.

We can define another partial order $\trianglelefteq$ on the nodes of the tree:

$$u \trianglelefteq v \Leftrightarrow u \textit{ is a node in the leftmost branch of the subtree rooted at } v$$

Given a node u, let $\mu(u)$ indicate the node $min_{\preceq}\{v \in N | u \trianglelefteq v\}$. Intuitively, $\mu(u)$ indicates the highest node v in the tree (i.e., closest to the root) such that u is in the leftmost branch of the subtree rooted at v. $\mu(u)$ is also known in the logic programming community as the *subroot node* of u.

3.1 Abstracting Side-effects

During OP execution nodes are added and deleted from the the or-tree. The execution steps can be directly expressed using the tree operations described in the previous section. In particular:

- each or-parallel computation is described by a branch in the tree. The tip of each branch represents the current execution "point" of that computation.
- whenever a computation (which is currently located at a certain leaf u of the tree) generates a new node, then an expand operation is executed to expand u with the new branches created[*2].
- whenever a computation either terminates or fails, it immediately enters into backtracking; each step of backtracking will delete the current leaf of the branch (remove operation).

It must be noted that this represents just an *abstraction* of the actual execution behaviour. An actual implementation, for example, may not remove nodes

[*2] Following the assumption made before, we assume that each subgoal has at most two matching clauses.

during backtracking.

Whenever a branch encounters a side-effect, it must check if it can execute it. This check boils down to verifying that the branch containing the side-effect is currently the leftmost active computation in the tree. Thus, if u is the current leaf of the branch where the side-effect is encountered, its computation is allowed to continue only if $\mu(u) = \mathsf{root}$. Thus, in OP, checking if a side-effect can be executed is as complex as the problem of performing the operation $\mathsf{find\_subroot}(u)$ which, given a leaf u, computes the node $\mu(u)$. We let $\mathcal{SE}$ denote the problem of supporting the operations create, expand, remove, and find_subroot for a dynamic tree.

We will not deal with the complexity of the $\mathcal{SE}$ problem here. It is possible to prove the following result [28]:

Theorem 3.1 (Complexity of $\mathcal{SE}$)
The $\mathcal{SE}$ problem can be solved with worst case time complexity of $O(1)$ per operation.

3.2 Abstracting Or-Parallelism

The abstraction of an OP execution should account for the various issues present in OP, (e.g., management of variables and of their bindings, creation of tasks etc.). Previous research as well as development of practical implementations [11, 1, 24] have demonstrated that variable management is indeed one of the key issue in an OP system. Actual or-parallel systems have experimentally demonstrated the impact of variable management [36]; furthermore, concurrent exploitation of and/or-parallelism exacerbates the problem [14, 25, 13, 39], making variable management one of the basic problems, if not the fundamental one.

The development of the OP computation takes place as described in the previous section on side-effects. Variables that arise during execution, whose multiple bindings have to be correctly maintained, can be modeled as attributes of nodes. For the purposes of this paper, the only attribute of nodes that is important for us are the variables. We can assume that a set Γ of M variables is available. If the computation tree has size N, then we can assume that $M = O(N)$. At each node u, three operations are possible:

- assign a variable X to a node u. We will assume we have a function function $access : \Gamma \times Nodes \rightarrow \{true, false\}$ to test whether a given variable has been associated to a node—i.e., $access(X, u) = true$ iff $\mathsf{assign}(X)$ has been executed at node u.

- dereference a variable X at node u—that is identify the ancestor v of u (if any) which has been assigned X. This is equivalent to finding the node $v(= \mathsf{dereference}(X,u))$ such that $access(X,v) = true \;\wedge\; v \preceq u$.
- alias two variables X_1 and X_2 at node u; this means that for every node v such that $v \succeq u$ every reference to X_1 in v will produce the same result as X_2 and vice-versa.

The single assignment semantics of Prolog impose the following restriction: for each pair of distinct nodes u_1, u_2 belonging to the same branch (i.e., $u_1 \prec u_2$ or $u_2 \prec u_1$) and for each variable X, we have that $access(X,u_1) = true$ implies $access(X,u_2) = false$. This means that each variable X can be assigned at most once in each branch of the tree.

In the previous abstraction we have assumed the presence of one variable binding per node. This restriction can be made without loss of generality (we can always assume that the number of bindings in the node is bound by a program dependent constant). Let us refer to the problem of supporting these dynamic tree operations as the $\mathcal{OP}$ problem. Later we are going to discuss the problems of *aliasing* and *dereferencing* separately. Figure 1 illustrates an example of an or-tree with various operations performed on it.

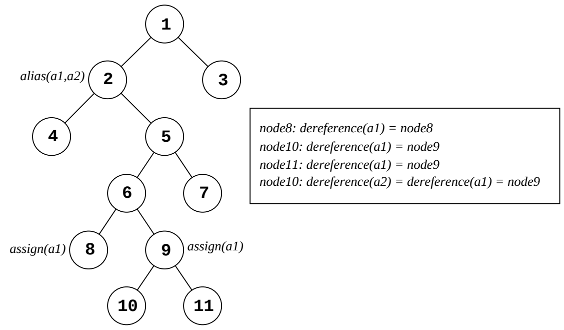

Fig. 1 Operations on or-tree

In [15] Gupta and Jayaraman describe or-parallelism in terms of three basic problems: *task creation* (i.e., creation of a node in the or-tree), *task switching* (i.e., activation of the computation at a given node), and *variable access/binding* (the problem of correct ordering of side-effects is not considered there). All the three operations need to be implemented efficiently to guarantee adequate performance. In the context of this work, the task-switching operation and the task-creation operation of [15] are subsumed by our expand operation. That is, the costs incurred at the time of task-switching and task-creation are incurred in our model at the time of the expand operation (after all, after task-switching to a site where an untried alternative is available, the very next step performed is to expand the untried alternative). Thus, even schemes which move part of the cost of managing to the task switching phase are subsumed by our abstraction—as these costs will be accounted for in the our abstraction during the expansion of a node. For example, a scheme like *Binding Arrays* [24] supports constant-time variable access via the use of tables associated to each branch of the or-tree. The cost of creating the new tables is paid during task switching—i.e., when a given leaf is about to be expanded. In our abstraction, this cost will be associated to the expand operation, giving a complexity of $O(1)$ per dereference and $O(N)$ per expand, where N is the maximum number of nodes in the tree. Some naive schemes for solving $\mathcal{OP}$, and their respective complexities, are presented in section 5.2.

Before we can analyze the complexity of $\mathcal{OP}$, and derive lower and upper bounds, we need to fix a model of computation. We choose to work with the *pointer machine* model as it is close enough to reality and simple enough to allow us to prove non-trivial and meaningful bounds. The pointer machine model has been used before for similar purposes [37, 32, 7, 22].

§4 Pointer Machines

A *Pure Pointer machine* consists of a finite but expandable collection R of *records*, called *global memory*, and a finite collection of *registers*. Each record is uniquely identified through an address (let $\mathcal{N}$ be the set of all the addresses). A special address *nil* is used to denote an invalid address. Each record is a finite collection of named fields. All the records in the global memory have the same structure, i.e. they all contains the same number of fields, with the same names. Each field may contain either a *data* (from a generic data domain $Data$) or an address (i.e., and element from $\mathcal{N}$).

The pointer machine is also supplied with two finite collections of registers, $d_1, d_2, \ldots$ (data registers) and $r_1, r_2, \ldots$ (pointer registers). Each register d_i can contain an element from $Data$, while each register r_i can contain one element of $\mathcal{N}$. The machine can execute *programs*. A program P is a finite, numbered sequence of instructions (where each instruction in P is uniquely identified by one number). The instructions allow to move addresses and data between registers and between registers and records' fields. The only "constant" which can be explicitly assigned to a register is *nil*. Special instructions are used to create a new record (returning its address) and to perform conditional jumps. The only conditions allowed in the jumps are true and equality comparison between two pointer registers. Observe that the content of the data fields will never affect the behaviour of the computation. In terms of analysis of complexity, it is assumed that each instruction has a unit cost. For further details on the structure and behaviour of the pure pointer machine the reader is referred to [4, 21, 37, 32, 7, 22].

Even though RAM is the most commonly used reference model in studies of complexity of sequential algorithms, the pointer machine model has received increasing attention as a viable model to study complexity of algorithms. The Pointer Machine model is simpler, thus making it more suitable for analysis of lower bounds of time complexity [37]. Furthermore, RAM commonly hides the actual cost of arithmetic operations, by allowing operations on numbers of size up to $\lg n$ (n being the size of the input to the problem) to be treated as constant time operations. Pointer Machines instead make these (relevant) costs explicit. Indeed, many lower bound results have been developed using the Pointer Machine model [37, 5, 22].

§5 Complexity on Pointer Machines

In this section we will explore the lower and upper bounds for some of the most important operations required to support OP. As mentioned earlier, the only previous work that deals with the complexity of the mechanisms required in parallel logic programming was in the context of or-parallelism [11, 15]. This previous work gives an intuitive and informal argument to show that a generic $\mathcal{OP}$ problem with N variables and M operations has a lower bound which is strictly worse than $\Omega(N+M)$. The section is organized in two parts: the first one describes the computation of the lower bound for $\mathcal{OP}$, while the second one focuses on the development of an upper bound for the problem.

5.1 Lower Bound for $\mathcal{OP}$

It is quite easy to observe that along each branch of the tree, variables are managed in a way very similar to the way in which elements are managed in the union-find problem. Initially all variables are distinct; given a variable X, let us denote with $\phi(X)$ the set of all the variables that are currently (on the branch at hand) aliased to X. If a new aliasing between X_1 and X_2 takes place, then the set ϕ is modified as $\phi(X_1) = \phi(X_1) \cup \phi(X_2)$. If we assume that the value of a variable X is always associated to the representative of $\phi(X)$, then any dereference operation is based on a find operation in order to detect the representative.

The observation above can be used to provide a new proof of the non-constant time nature of the management of or-parallel computations.

Theorem 5.1
The amortized time complexity of $\mathcal{OP}$ is $\Omega(N + M\alpha(M, N))$, where N is the number of variables and M is the number of tree operations.

Proof *(Sketch)* The proof is based on the idea that $\mathcal{OP}$ subsumes the union-find problem. Let us consider a generic union-find problem with N elements and M union-find operations. From the results in [37, 22)] we know that this problem has a lower bound of $\Omega(N + M\alpha(M, N))$.

Let us show how we can use the mechanisms of $\mathcal{OP}$ to implement a solution to the union-find problem. Each of the N elements is encoded as one of the variables. Each time a create(X) operation is performed in the union-find problem, an expand operation is applied on the leftmost leaf and the operation assign(X) is performed on the leftmost leaf. Each union(X, Y) operation is translated into an aliasing between the two variables X and Y; the aliasing is performed to the leftmost leaf of the tree. Each find(X) operation in the union-find problem corresponds to the dereferencing of the variable X, starting from the leftmost leaf in the tree. Thus a set is represented by a collection of variables which have been aliased to each others; the fact that the dereference operation returns the same result for aliased variables guarantees the existence of a unique representative for the set. Clearly each operation in the union find problem can be encoded using the $\mathcal{OP}$ support and a constant number of additional operations. Thus, each solution to $\mathcal{OP}$ cannot be better than the lower bound for the union-find problem. □

This result, as can be seen from the proof, relies mainly on the execution of aliasing operations. Nevertheless, it is possible to show that the result holds even when we disallow aliasing during OP execution. In fact, the inability to achieve constant-time behaviour is inherent in the management of the variables in the tree.

Theorem 5.2
On pointer machines, the worst case time complexity of $\mathcal{OP}$ is $\Omega(\lg N)$ per

operation even without aliasing.

Proof The basic idea of the proof is that since there is no direct addressing in the pointer machines starting from a particular node only a "small" number of nodes can be accessed in a small number of steps. Without loss of generality, assume that each record has at most two pointers. Assume that we have a pointer machine program that performs the dereference operation in worst–case $c(N)$ time for some function c. The dereference procedure receives only two pointers as arguments, one to the record representing the variable being searched, and the other to the record representing the leaf node for which this variable is being searched. Starting from either of these records, we can access at most $2^{c(N)+1}$ different records within $c(N)$ steps. We use this fact to prove that $c(N) = \Omega(\lg N)$. Suppose $c(N)$ is not $\Omega(\lg N)$. Then for large enough $N, c(N) < \frac{1}{8}\lg N$. Consider the tree with N nodes that has the following structure (see figure 2). It has depth $t_1(N)+t_2(N)$. The first $t_1(N)$ levels constitute a complete binary tree (let us indicate with Γ_1 the first $t_1(N)$ levels of the tree); the remaining $t_2(N)$ levels consist of $2^{t_1(N)}$ lists of nodes, each of length $t_2(N)$ and each attached to a different leaf of Γ_1. Clearly, we must have that $2^{t_1(N)+1} - 1 + t_2(N) * 2^{t_1(N)} = N$ or that $t_2(N) = (N+1)/2^{t_1(N)} - 2$.

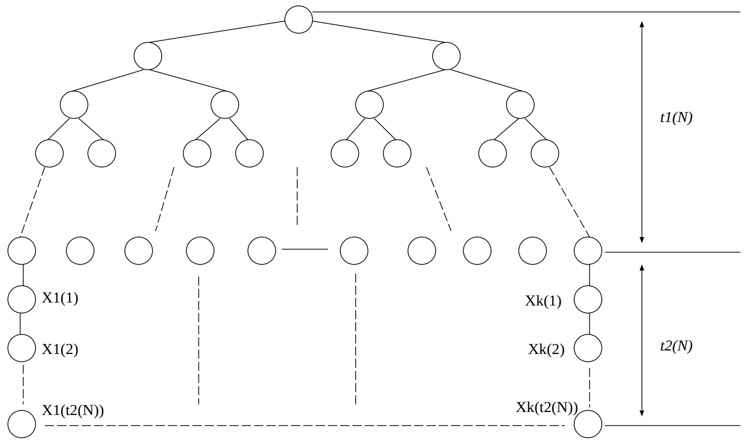

Fig. 2 Complex Tree

Let us also assume that we have $t_2(N)$ different variables, $\{X_1, \ldots, X_{t_2(N)}\}$. The nodes in each branch segment going from depth $t_1(N)+1$ to $t_2(N)$ are arbitrarily assigned the variables $\{X_1, \ldots, X_{t_2(N)}\}$. The tree structure is represented in figure 2; in the figure, $k = 2^{t_1(N)}$, and $\langle Xi(1), \ldots, Xi(t_2(N))\rangle$ is a permutation

of $\langle X_1, \ldots, X_{t_2(N)} \rangle$.
Suppose we choose $t_1(N)$ and $t_2(N)$ such that

$$2^{c(N)+1} t_2(N) < 2^{t_1(N)} \tag{1}$$

We should observe that, since the procedure can perform at most $c(N)$ steps, then starting from the records representing the variables we can explore at most

$$2^{c(N)+1} t_2(N)$$

records—since there are $t_2(N)$ variable records. From the choice of $t_1(N)$ and $t_2(N)$ in the formula (1), and from the fact that there is a total of

$$2^{t_1(N)} t_2(N)$$

records to be searched, then there exists a leaf node v in the tree such that none of the last $t_2(N)$ nodes on the path from root to v is accessed starting from the pointer to the variable record in any of the computations $\mathsf{dereference}(X_i, v)$, $i = 1 \ldots t_2(N)$. This implies that for any of the $t_2(N)$ different $\mathsf{dereference}(X_i, v)$ calls, the appropriate node on the path must be accessed starting from the record for the leaf node v. Since the dereference procedure works in time $c(N)$ we must have that

$$2^{c(N)+1} \geq t_2(N)$$

otherwise we will not be able to access all the nodes with variables starting from the leaf v.

This gives us that $c(N) \geq \lg t_2(N) - 1$. Since, $c(N) < \dfrac{1}{8} \lg N$, then $2^{c(N)+1} < 2N^{1/8}$. Let us choose $t_1(N) = \dfrac{3}{4} \lg N$. Then $2^{t_1(N)} = N^{3/4}$ and $t_2(N) = (N+1)/N^{3/4} - 2 \approx N^{1/4}$. These $t_1(N), t_2(N)$ satisfy the inequality (1) above as $2^{c(N)+1} t_2(N) \approx 2N^{3/8} < N^{3/4}$. Hence we can say that $c(N) > \lg t_2(N) - 1$, i.e. $c(N) > 1/4 \lg N - 1$ which contradicts our assumption that $c(N) < \dfrac{1}{8} \lg N$. Hence $c(N) = \Omega(\lg N)$. □

The above result is much stronger than the result in theorem 5.1, and gives us a better idea of why it is impossible to support or-parallelism with a constant overhead. The result essentially states that, no matter how smart is the implementation scheme selected, there will be cases which will lead to a non-constant time cost. This proof confirms the result conjectured in [15)]. This non-constant time nature is also evident in all the implementation schemes presented in the literature—e.g., the creation of the shared frames and the copying of the choice points in Muse [1], the installation of the bindings in Aurora [24], management of time-stamps in various other models [11].

5.2 Upper Bound for $\mathcal{OP}$

The relevant research on complexity of the $\mathcal{OP}$ problem has been limited to showing that a constant time cost per operation cannot be achieved in any implementation scheme. No work has ever been attempted, to the best of our knowledge, to supply a tight upper bound to this problem. Most of the implementation schemes proposed in the literature can be shown to have a worst case complexity of $O(N)$ per operation. Currently, the best result we have been able to achieve is the following:

Theorem 5.3
The $\mathcal{OP}$ problem with no aliasing can be solved on a pointer machine with a single operation worst–case time complexity of $O(\sqrt[3]{N}(\lg N)^k)$ for a small k.

Let us start by simplifying the description of the problem. The $\mathcal{OP}$ problem can be cast in the following terms. The computation is described in terms of operations on a dynamic tree. For the sake of simplicity we will focus on a growing only tree. Thus, the only operation allowed on the tree is $\mathsf{expand}(v, X_1, X_2)$ which expands the leaf v by appending two successors u_1 and u_2, respectively labeled with X_1 and X_2. Also, for simplicity we assume that X_1 and X_2 are the actual variables associated to such nodes—i.e., we avoid the explicit use of assign. Only leaves can be expanded. The tree is *labeled*; thus a tree is coupled with a labeling function:

$$l\colon \mathit{Nodes} \rightarrow \Gamma$$

where *Nodes* is the set of all the nodes in the tree and Γ is a finite set of labels. Additionally, the labeling function l always satisfies the following requirement: if $u \prec v$ denotes the fact that u is an ancestor of v, then

$$\forall u, v \in \mathit{Nodes}\, (v \prec u \rightarrow l(u) \neq l(v))$$

i.e., each label may occur at most once along each branch of the tree. Given this requirement, we can define the function

$$\mathsf{dereference}\colon \Gamma \times \mathit{Nodes} \rightarrow \mathit{Nodes} \cup \{\perp\}$$

which, given a label γ and a node u, returns the node v (if any) on the path (from the root) ending in u which is labeled γ. The problem at hand is the efficient implementation of the two operation expand and $\mathsf{dereference}$. For the sake of simplicity we can assume that the function dereference is applied only to leaves of the tree (which is actually the case for or-parallelism, as the parallel agents expands always the bottommost choice points). Let us start developing a solution under the assumption that the number N of nodes to be inserted in the tree is known in advance.

For our purposes, we can think the labels to be stored directly in the nodes of the tree. Let us first consider the naive scheme. The expand operation simply expands a leaf attaching the two new successors. The dereference operation starts with a label and a leaf, and walks back towards the root searching for the desired label. Clearly, in this naive implementation, the expand operation is $O(1)$, but the dereference operation can take time $O(N)$ in the worst case, since the tree may be completely unbalanced. Alternatively, on the other extreme, we could require expand to create a complete summary the branch up to that point ($\tilde{O}(N)$), allowing a very fast dereference operation ($\tilde{O}(1)$). In both cases, a sequence of M operations will lead to a worst–case complexity of $\tilde{O}(M^2)$.

The general idea in our improved schemes is to balance the amount of work done during the expand and the dereference operations, by making the expand operation more complex in order to make the search more efficient. We first start by providing a simple $\tilde{O}(\sqrt{N})$ solution; we later improve the solution to $\tilde{O}(\sqrt[3]{N})$. This last solution is considerably more involved than the first one, but uses some of the ideas in the first one.

[1] An $\tilde{O}(N^{\frac{1}{2}})$ scheme

The first algorithm that we are describing will lead to a complexity[*3] $O(\sqrt{N}(\lg N)^k)$.

The behaviour of dereference can be improved by creating at various points of each branch a "table" aimed at summarizing the actions performed on that segment of the branch. This is illustrated in figure 3. Table *T1* summarizes the assignment of labels to nodes for the part of the branch between nodes `n1` and `n2`; table *T2* summarizes the assignment of labels to nodes for the part of the branch between nodes `n3` and `n4`. Nodes that are associated to tables are linked together. Note that, although we talk conceptually about tables, these tables are actually implemented as *balanced binary search trees* on pointer machines. We can do this because we can assume the existence of a total order between the

[*3] For the sake of simplicity in most of the following analysis we will ignore the $(\lg N)^k$ factor.

labels—a reasonable assumption considering that variables are indeed ordered in their internal memory representation in any Prolog implementation (we can remove this assumption by paying an $O(\lg N)$ penalty).

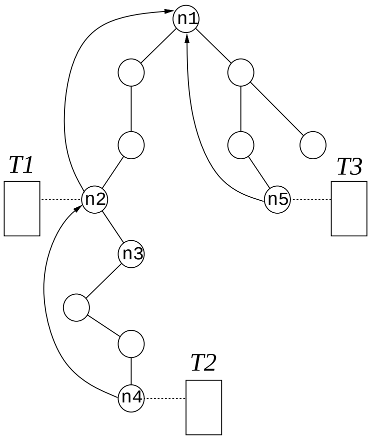

Fig. 3 Use of Tables

The execution of dereference requires scanning linearly the branch from the leaf up to the first table node. If the desired label is not found in such part of the branch, then the table is examined. If the label is not found, then the search is repeated in the preceding table, and so on. If the root is reached, then the searched label has not been assigned to any node on that branch.

The complexity of this process is the following: let us assume that the tables are created every $t(N)$ nodes.

- the initial linear search will not visit more than $t(N)$ nodes;
- searching inside a table will require $O(\lg t(N))$;
- since there are at most $O(N)$ nodes on the branch, then the search process will visit at most $O(\frac{N}{t(N)})$ tables.

Combining these elements, we can state that the complexity of this implementation of dereference is

$$O(t(N) + \lg t(N)\frac{N}{t(N)})$$

The expand operation, on the other hand, has a worst-case time complexity $\tilde{O}(t(N))$ per operation (occurring at the time of table creation).

Choosing $t(N) = \sqrt{N}$ the total cost[*4] is $O(\sqrt{N} \lg N)$.

[2] An $\tilde{O}(N^{\frac{1}{3}})$ Solution:

The above scheme can be improved by making use of the solution to the ancestor problem—the problem of determining if one node is ancestor of the other in a dynamically growing tree. This is a well-studied problem with very efficient solutions [10, 16, 21, 34, 38]. Let $\beta(N)$ denote the complexity of solving the ancestor problem for a dynamic tree with N nodes. We know from the literature [38, 16] that it is possible to solve the ancestor problem for two nodes x, y in time $\approx O(\lg \min(depth(x), depth(y)))$. Thus $\beta(N) = O(\lg N)$.

The intuition behind this second solution is the following: if a certain label γ has been assigned to very few nodes—let's say k nodes $m_1, \ldots, m_k$—it is possible to solve the problem dereference(γ,n) by simply considering each of the k nodes m_i and verifying whether m_i is an ancestor of n. This is possible only if k is sufficiently small. Observe that, in the worst case, the number of nodes associated with γ could be $\Theta(N)$. On the other hand, only "few" labels will be used very frequently. We can try to take advantage of these facts by collecting the assignments of labels to nodes into *groups*. Each group is supposed to have sufficiently small cardinality to allow its exhaustive search. The general idea in the search scheme (dereference) would be to first determine the group of the variable being searched, and then perform an exhaustive search within the group, using the solution to the ancestor problem. Let $t(N)$ be the cardinality of a group. Thus, we associate to each label γ a set of groups, containing all the nodes u in the tree such that $l(u) = \gamma$. During the construction of the tree, every time we create a node v with label γ, we insert v in the current group for γ. If the current group is full—i.e., it contains $t(N)$ elements—then a new group for γ is created, and v is added to such group.

A variable is *heavily used* if it has been assigned to more than $t(N)$ nodes. A node is called a *heavy* node if, at the time of its creation, it is assigned a heavily used variable.

During the creation of the tree, each heavy node is marked, and contains a pointer to the previous heavy node on the path to the root (if any). Also, for each variable, all the unmarked nodes are maintained as a linked list.

Figure 4(i) schematically illustrates these concepts with a simple example. The example assumes $t(N) = 2$. The numbers indicate the order in which

[*4] We choose to ignore the polylog factors in making an optimum choice for $t(N)$.

the indicated nodes have been inserted. It is easy to see that nodes 4, 6, and 7 are heavy nodes.

Observe that, since there are N nodes in the tree, and each node has one variable, we can have at most $\frac{N}{t(N)}$ heavily used variables. Since each variable can occur at most once on each path, this means that along each branch of the tree we can have at most $O(N/t(N))$ nodes that have been marked.

A strategy to implement the $\mathsf{dereference}(X, v)$ operation would be to simply visit the marked nodes along the path from v to the root of the tree. If a node with label X is found, then we have an immediate solution—determined in $O(N/t(N))$. Otherwise, for each node u in the linked list of unmarked nodes for X an ancestor (u, v) operation is performed. If the answer is positive for any u, then we have determined a solution. Otherwise, there is no node on the path from v to the root which is assigned the variable X. We argue the correctness of the above procedure next. There are only three possibilities—the variable X occurs on the path to the root on a marked node, the variable X occurs on the path to the root on an unmarked node, or it does not occur at all. In the first case the variable would be found during the search of the marked nodes. In the second case, the node is contained in the list of unmarked nodes for X, and hence X is discovered in the second phase of the search. In the third case, obviously X is never discovered. The complexity of the above procedure is $O(t(N)\beta(N) + N/t(N))$.

The search can be improved by adding, as in the scheme described in the previous section, tables to summarize the marked nodes in the previous section of the branch. Noting that we need to keep track of at most $N/t(N)$ nodes per branch—the maximum number of marked nodes—using the results from the previous section we can determine that the best behaviour can be achieved by placing a table at every $\sqrt{N/t(N)}^{th}$ marked node along a branch. The effect of the table will be to summarize the previous $\sqrt{N/t(N)}$ marked nodes generated along the current branch. Figure 4(ii) illustrates the situation. Nodes with a "*" represent marked nodes. Using this scheme, a search for a label starting from a given leaf will proceed as follows:

- starting from the leaf, the various marked nodes are accessed up to the first table. If the label is found within these nodes, then we have immediately an answer to the problem. Considering the assumption made before, this first phase will traverse at most $O(\sqrt{N/t(N)})$ nodes.

- if the label is not found in the marked nodes at the end of the branch, then the first table is searched for the considered label. This search will cost $O(\lg(N/t(N)))$.
- if the label is found in the table, then we can assume that the entry will directly refer to the correct node. Otherwise the search has to be repeated in the preceding table. At most $\sqrt{N/t(N)}$ tables are going to be searched, leading to a total time of $O(\sqrt{N/t(N)}\lg(N/t(N)))$.
- finally, if the label is not found in any of tables, then a final search has to be performed in the list of unmarked nodes for the label. For every unmarked node, we may have to perform an ancestor operation. This final search costs $O(t(N)\beta(N))$.

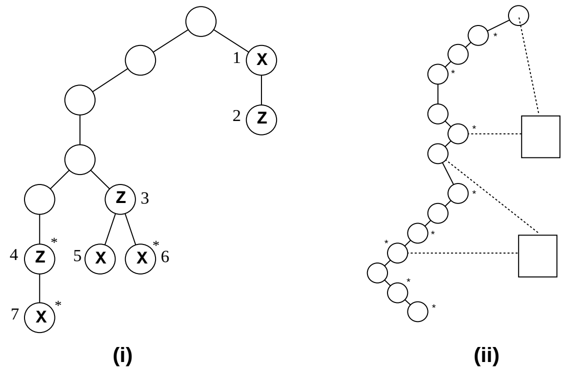

Fig. 4 Examples of Trees

Thus, we obtain a total complexity for the dereference operation of:

$$O(\sqrt{N/t(N)} + \sqrt{N/t(N)}\lg(N/t(N)) + t(N)\beta(N))$$

Choosing $t(N) = \sqrt[3]{N}$, noting that $\beta(N)$ is $O(lgN)$ we get that the complexity of dereference is $\tilde{O}(\sqrt[3]{N})$.

To complete our argument, we need to tackle the two assumptions made at the beginning of the proof:

1. we assumed growing-only trees, i.e., we ignored the remove operation. It is relatively easy to observe that the remove operation does not affect the validity of our results since it is applied only to leaves of the tree.
2. we assumed at the beginning that the number of nodes N is known beforehand. One can show that the use of the *current* number of nodes in place of the *total* number of nodes is going to produce variations

only in the polylog part of the complexity. We made this assumption to keep the presentation of the proof simple and easy to follow.

□

It is interesting to observe that the average case behaviour can be improved by adopting a better structuring of the nodes in the unmarked list. In particular, the nodes in such list can be organized as a tree which reflects the ancestor relationship present in the original dynamic tree—which would, on average, improve the final search.

[3] Insertion

The previous discussion focused on improving the cost of the dereference operation. In order to make sure that the model has an acceptable low complexity we need to guarantee that also the other main operations for manipulating the tree structure, i.e., expand/assign and remove can be performed at low cost.

In all the schemes described above, removal of a node can be performed in constant time without endangering the behaviour of the scheme. The execution of expand/assign requires some more attention. Let us focus on the last scheme described. One of various cases may occur during creation of a new node:

1. the node is created and is assigned a variable which is not heavily used: in this case no special action is required (the cost is $O(1)$);
2. the node is created and assigned an heavily used variable, but the node is not going to contain a summarizing table. As in the previous case, no special action is required, except for marking the node and linking it to its predecessor marked node (the cost is $O(1)$);
3. the node is created, assigned an heavily used variable, and the node requires the creation of a table of marked nodes. The cost for the construction of the table is $\tilde{O}(\sqrt[3]{n})$.

Thus, the combined expand and assign operation can be performed with a worst case time complexity $\tilde{O}(\sqrt[3]{n})$, which is consistent with the cost of the other operations.

It is also interesting to observe that the amortized complexity of the combined expand/assign operation is $\tilde{O}(1)$, since the non-constant time cost ($\tilde{O}(\sqrt[3]{n})$) is paid only once every $\sqrt[3]{n}$ operations.

§6 Conclusions

Method	*Complexity*
This Paper	$\tilde{O}(M \times N^{\frac{1}{3}})$
Stack Copying [2]	$\tilde{O}(M \times N)$
Directory Tree Method [11]	$\tilde{O}(M \times N \lg N)$
Binding Arrays [24]	$\tilde{O}(M \times N)$
Environment Closing [11]	$\tilde{O}(M \times N)$

Table 1 Worst-case Complexity of Some Or-parallel Schemes (m operations)

In this paper we studied the complexity of supporting or-parallel execution of Prolog programs. We focused on the complexity of the basic operations associated with parallel execution of Prolog programs—i.e., those operations needed to create and maintain the computation tree and the variable bindings. We developed lower and upper bounds for the $\mathcal{OP}$ problem, i.e., the problem of implementing the basic operations required for correctly managing variable bindings in presence of or-parallelism.

The lower bound produced, $O(\lg N)$ per operation, is a confirmation and refinement of the results proposed by Gupta and Jayaraman [15], and a further proof that an ideal or-parallel system (where all the basic operations are realized with constant-time overhead) cannot be realized. The upper bound, $\tilde{O}(\sqrt[3]{N})$, even if far from the lower bound, is of great importance, as it suggests that current implementation mechanisms are more complex than necessary, and better implementation schemes are feasible. Furthermore, the scheme proposed achieves a worst-case time complexity which is better than all the or-parallel schemes reported in the literature (some of which have also been implemented); Table 1 compares the *worst case* time complexity of performing a sequence of M operations, on an N node tree, for some of the most well known schemes for or-parallelism [11]. The scheme we present handles the $\mathcal{OP}$ without the use of the alias operation. Nevertheless, from [5] we can infer that the alias operation should be solvable with worst–case complexity $O(\frac{\lg N}{\lg \lg N})$ per operation, thus adding just a lower order component to $\sqrt[3]{N}$.

The results presented in this work are mainly related to worst–case analysis of the problem. Nevertheless, all the worst-case situations depicted are reasonable and we expect the average-case complexity (which is, as a matter of fact, heavily dependent on the specific application) to be sufficiently close to the results presented here. We believe these results offer valuable insights regarding the *inherent* complexity of the problem at hand.

A direct implementation of the scheme proposed in section 5.2 may not lead to optimal results, e.g., due to the presence of global operations to produce the marking of the nodes. Nevertheless, we believe the scheme proposed is of great practical value as it suggests a way to design new and more efficient implementation schemes—possibly as modifications of existing models. Furthermore, it is interesting to observe that the $\mathcal{OP}$ problem shares deep similarities with the inheritance problem in dynamic object-oriented systems [29], and the solution scheme used in section 5.2 is a generalization of the caching and tabling mechanisms commonly used in such framework.

An open problem of interest is the development of better upper bounds for the $\mathcal{OP}$ in case of no aliasing between variables. This problem has been shown to have a non-constant time lower bound, but an optimal upper bound is still a subject of research. Our conjecture is that the problem admits a solution with a worst case time complexity $O((\lg N)^k)$ (for a small k) per operation.

Finally, it is important to observe that the $\mathcal{OP}$ problem is actually a very general problem, which occurs in almost the same form in various different frameworks. For example, [30, 29] illustrates how the same problem arises in the context of dynamic and multiple inheritance for object oriented programming systems. Thus, the results presented here are very general and can be successfully applied to analyze and support research also outside of parallel logic programming.

§7 Acknowledgements

The authors wish to thank the anonymous referees for their comments and suggestions. This work has been partially supported by NSF grants CDA 97-29848, CDA-98-02251, EIA 98-10732, CCR 96-25358, INT 95-15256, and HRD 9628450, and by a grant from the Fullbright's US-Spain Program.

References

1) Ali, K.A.M. and Karlsson, R. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.*, pages 757–776. MIT Press, 1990.

2) Ali, K.A.M. and Karlsson, R. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1991. 19(6):445–475.

3) Beaumont, T. and Warren, D.H.D. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *10th Int'l Conference on Logic Programming*, pages 135–149. MIT Press, 1993.

4) Ben-Amram, A.M. What is a Pointer Machine? Technical report, DIKU, University of Copenhagen, 1995.

5) Blum, N. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4):1021–1024, 1986.

6) Chang,S-E. and Chiang, Y-P. Restricted AND-Parallelism Execution Model with Side-Effects. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 350–368, 1989.

7) Chazelle, B. A Functional Approach to Data Structures and its Use in Multidimensional Searching. *SIAM Journal of Computing*, 17(3):427–462, 1988.

8) Clocksin, W.F. and Mellish, C.S. *Programming in Prolog*. Springer-Verlag, 1981.

9) Fredman, M.L. and Saks, M.E. The Cell Probe Complexity of Dynamic Data Structures. In *Procs. of 21st ACM Symposium on Theory of Computing*, pages 345–354. ACM, 1989.

10) Gabow, H.N. Data structures for weighted matching and nearest common ancestor. In *ACM Symp. on Discrete Algorithms*, pages 434–443, 1990.

11) Gupta, G. *Multiprocessor Execution of Logic Programs*. Kluwer Academic Press, 1994.

12) Gupta, G. and Santos Costa, V. Cuts and Side-effects in And/Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, 1996.

13) Gupta, G., Santos Costa, V. and Pontelli, E. Shared Paged Binding Arrays: A Universal Data-structure for Parallel Logic Programming. Proc. NSF/ICOT Workshop on Parallel Logic Programming and its Environments, CIS-94-04, University of Oregon, Mar. 1994.

14) Gupta, G., Pontelli, E., Hermenegildo, H. and Santos Costa, V. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.

15) Gupta, G. and Jayaraman, B. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.

16) Harel, D. and Tarjan, R.E. Fast Algorithms for Finding Nearest Common Ancestor. *SIAM Journal of Computing*, 13(2):338–355, 1984.

17) Haridi, S. and Janson, S. Kernel Andorra Prolog and its Computation Model. In *Proc. 7th International Conf. on Logic Programming*, pages 31–46. MIT Press, 1990.

18) Hausman, B., Ciepielewski, A. and Calderwood, A. Cut and Side-Effects in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems*, pages 831–840. Tokyo, November 1988.

19) Hermenegildo, M. and Greene, K. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 Int'l Conf. on Logic Prog.*, pages 253–268. MIT Press, June 1990.

20) Kergommeaux, J.C. and Codognet, P. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

21) Knuth, D.E. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

22) LaPoutré, H. Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines. *Journal of Computer and System Sciences*, 52:87–99, 1996.

23) Lloyd, J.W. *Foundations of Logic Programming*. Springer-Verlag, 1987.

24) Lusk, E. et. al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2/3), 1990.

25) Montelius, J. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. PhD thesis, Uppsala University, 1997.

26) Pippenger, N. Pure versus Impure Lisp. *ACM Transactions on Programming Languages and Systems*, 19(2):223–238, 1997.

27) Pontelli, E., Gupta, G. and Hermenegildo, M. &ACE: A High-performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–571. IEEE Computer Society, Santa Barbara, CA, April 1995.

28) Pontelli, E., Ranjan, D. and Gupta, G. On the Complexity of Parallel Implementation of Logic Programs. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 123–137. Springer Verlag, 1997.

29) Pontelli, E., Ranjan, D. and Gupta, G. The Complexity of Late-binding in Dynamic Object-Oriented Languages. In *Proceedings of the International Symposium on Programming Languages, Implementations, Logic, and Languages (PLILP)*, pages 213–229. Springer Verlag, 1998.

30) Ranjan, D., Pontelli, E. and Gupta, G. Dynamic Data Structures in Advanced Programming Languages Implementation. Technical report, Dept. Computer Science, New Mexico State University, 1997.

31) Robinson, J.A.. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

32) Schönhage, A. Storage Modification Machines. *SIAM Journal of Computing*, 9(3):490–508, August 1980.

33) Shapiro, E.Y. editor. *Concurrent Prolog: Collected Papers*. MIT Press, Cambridge MA, 1987.

34) Sleator, D.D. and Tarjan, R.E. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

35) Sterling, L. and Shapiro, E.Y. *The Art of Prolog*. MIT Press, Cambridge MA, 1994.

36) Szeredi, P. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

37) Tarjan, R.E.. A Class of Algorithms which Require Nonlinear Time to Mantain Disjoint Sets. *Journal of Computer and System Sciences*, 2(18):110–127, 1979.

38) Tsakalidis, A.K. The Nearest Common Ancestor in a Dynamic Tree. *ACTA Informatica*, 25, 1988.

39) Westphal, H. and Robert, P. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *IEEE Int'l Symp. on Logic Prog.*, pages 436–448, 1987.