On the Complexity of Parallel Implementation of Logic Programs*
(Extended Abstract)

E. Pontelli, D. Ranjan, G. Gupta

Dept. of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
{epontell,dranjan,gupta}@cs.nmsu.edu

Abstract. We study several data-structures and operations that commonly arise in parallel implementations of logic programming languages. The main problems that arise in implementing such parallel systems are abstracted out and precisely stated. Upper and lower bounds are derived for several of these problems. We prove a lower bound of $\Omega(\log n)$ on the overhead incurred in implementing even a simplified version of *or-parallelism*. We prove that the aliasing problem in parallel logic programming is at least as hard as the union-find problem. We prove that an *and-parallel implementation* can be realized on an *extended pointer machine* with an $O(1)$ overhead.

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. A nice property of logic programming languages is that parallelism can be automatically extracted from logic programs by the compiler or the runtime system. 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. To date, however, little attention has been paid to analyzing the complexity of the operations involved in this implementation.

The problem of implementing a logic programming language can be abstracted as the process of maintaining a dynamic tree. The operational semantics of the logic language determine how this tree is built and what operations on the tree are of interest. As execution proceeds according to the operational semantics, this trees grows and shrinks. In a parallel implementation the tree can

* This work has been partially supported by NSF grants CCR 96-25358, INT 95-15256, and HRD 9628450, and by NATO Grant CRG 921318.

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 [2, 5, 23, 8, 20, 21, 4], to the best of our knowledge the specific data structures needed to support parallel logic programming have not been studied formally. In this paper we derive upper bounds and lower bounds for some of these operations. We prove a lower bound of $\Omega(\log n)$ on the overhead incurred in implementing even a restricted version of or-parallelism (no *aliasing*). We prove that the *aliasing problem* in parallel implementation of Prolog is at least as hard as the union-find problem. This gives us a lower bound of $\Omega(n + m\alpha(m,n))$ for the aliasing problem on pointer machines, due to the results of [18, 21]. We prove that an and-parallel implementation can be realized on an *extended pointer machine* with an $O(1)$ overhead. 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. We also give a tight relationship between the and-parallelism problem and the problem of "time-stamping" on pointer machines. Elsewhere, we also show that the *side-effect problem* in parallel logic programming can be solved with a *constant* time overhead per operation [16].

We believe that abstracting out the problems in parallel implementation of logic programs as that of building data structures to support certain operations of interest is one of the major contributions of the paper. 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 [7] that establishes some results on complexity of realizing an or-parallel implementation. The work presented here is far from complete and numerous open problems remain. One of the goals of this paper is to bring these problems to the attention of the complexity theory and data structures community. 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 and parallel AI systems.

The reader is assumed to be familiar with the general concepts and terminology of logic programming and Prolog [12].

1.1 Parallel Logic Programming

The presence of non-determinism is a peculiar feature of logic programming. This non-determinism in logic programs can be used as a source of *implicit* parallelism. Two major forms of parallelism can be identified [3]:

- *And-Parallelism (AP)* [9, 15], which arises from don't care non-determinism; given a goal, several subgoals can be selected for simultaneous reduction.
- *Or-Parallelism (OP)* [1, 13]: which arises from don't know non-determinism. Given a subgoal, the multiple clauses whose heads unify with it can be solved in parallel.

A major research direction in parallel logic programming has been to design parallel implementations of *Prolog* [3, 9, 1, 13]. This implies that the parallel execution mechanisms have to be designed 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 only consider parallel implementations that preserves Prolog semantics.

2 Problems in Parallel Logic Programming

Various parallel implementations of Prolog have been proposed and realized [3, 10]. Most of the current systems suffer from severe efficiency problems. Many of these inefficiencies, especially in and-parallel systems, arise from the need of guaranteeing 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 can be accepted. It is possible to combine both and-parallelism and or-parallelism together [3, 6], however, for the sake of simplicity, we consider the two forms of parallelism independently.

In the rest of this work we will also not deal with the issue of implementing the correct ordering of side-effects: this problem has been studied elsewhere [16] and shown to have a worst-case time complexity of $O(1)$ per operation.

2.1 And-Parallelism

Given a resolvent $B_1, \ldots, B_n$, multiple subgoals in the resolvent can be concurrently reduced in and-parallel execution. And-parallel execution can be visualized through the *and-tree*. The root of the and-tree is labeled with the initial goal. If a node contains a conjunction $B_1, \ldots, B_n$, then it will have n children: the ith child of the node is labeled with the body of the clause used to solve the B_i.

The main problem in the implementation of and-parallelism is how to efficiently manage the unifiers produced by the concurrent reduction of different subgoals. Two subgoals B_i and B_j $(1 \leq i < j \leq n)$ in the resolvent $B_1, \ldots, B_n$ should agree in the bindings of all the variables that are common to them (such variables are termed *dependent variables* in parallel logic programming terminology). In sequential Prolog execution, usually, B_i, the goal to the left, binds the common variable and B_j works with the binding produced by B_i. In and-parallel execution, when B_j is started, the common variable will be unbound. B_j may attempt to instantiate this variable, violating Prolog semantics. The key problem is to ensure that bindings to common variables are made in the same order as in a sequential execution. This requirement is much stronger compared to just requiring that Prolog semantics be preserved (i.e., externally observable behavior during parallel execution be the same as in the sequential execution). If a parallel system satisfies this stronger requirement, we say that it *preserves strong Prolog semantics*. Preserving strong Prolog semantics is important, as otherwise considerable amount of redundant computation may be performed [14]. Note that if strong Prolog semantics is preserved, then Prolog semantics is preserved, but not vice versa. And-parallel implementations handle the requirement of preserving strong Prolog semantics by assigning *producer* or *consumer* status to each subgoal that shares the dependent variable [14]. The leftmost goal in the resolvent that has access to the dependent variable is designated as the producer subgoal for that variable, all others are consumers. A consumer subgoal is not allowed to bind the dependent variable, it is only allowed to read its binding. If a consumer subgoal attempts to unify against the unbound dependent variable, it has to suspend until the producer goal binds it. If the producer subgoal finishes execution without binding the dependent variable, then the producer status is transferred to the leftmost consumer subgoal for that variable. The producer subgoal for a dependent variable, therefore, can change dynamically during execution. Thus, a major problem in an and-parallel implementation is to keep track of the leftmost subgoal that can (directly or indirectly) access each variable.

2.2 Or-Parallelism

In case of OP the parallel computation can be described as 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. 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 $\theta_j = mgu(H_j, B_1)$, 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 n_i corresponding to a matching clause C_i that textually precedes another matching clause C_j is placed to the left of n_j, where n_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. In the or-tree, each branch is forced to maintain a local view of the substitution computed. This requirement emerges from the fact that, 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.These conflicting bindings need to be maintained separately, as they will each lead to a different solution for the initial resolvent. The main issue, 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 clearly highly inefficient, since it requires the creation of complete copies of the substitution (which can be arbitrarily large) every time a branching takes place [7].

3 Abstraction of the Problems

Let us consider a labeled tree $\mathcal{T}$ with n nodes and a root root. Without loss of generality we will focus exclusively on binary trees. Trees are manipulated through three instructions: *(i)* $\mathsf{create\_tree}()$ which creates a tree containing only the root; *(ii)* $\mathsf{expand}(n, b_1, b_2)$ which, given one leaf n and two labels b_1 and b_2, creates two new nodes (one for each label) and adds them as children of n (b_1 as left child and b_2 as right child); *(iii)* $\mathsf{remove}(n)$ which, given a leaf n of the tree, removes it from the tree. These three operations are assumed to be the only ones available for modifying the "physical structure" of the tree.

The tree implements a partial ordering between the node. Given two nodes n and n', we write $n \preceq n'$ if n is an ancestor of n'; $n \prec n'$ additionally says that $n \neq n'$. We will be often referring to the notion of *leftmost branch*. Given a node n, the leftmost branch of the subtree rooted in n can be defined inductively: *(i)* if n does not have any children, then the branch containing only n is the leftmost branch; *(ii)* if n has children, and l is the leftmost child, then the leftmost branch is n followed by the leftmost branch of l. We can define in particular the following partial order $n \trianglelefteq m$:

$(n \trianglelefteq m \Leftrightarrow n$ *is a node in the leftmost branch of the subtree rooted at* $m)$

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

3.1 Abstracting Or-Parallelism

The main issue in dealing with OP is the management of the variables and their bindings. 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. We can assume that a number m of attributes are available (i.e., there are m variables). If the computation tree has size n, then we can assume that $m = O(n)$. At each node n three operations on attributes are possible:

- assign a value v to an attribute α; as a consequence, every reference to α in a node $m \succeq n$ will produce v. For the sake of simplicity we can assume that, given the node n where the assignment is performed and the attribute α we can uniquely infer the value v. Let us use the function $access(n, \alpha)$ to obtain the value v if $\mathsf{assign}(\alpha, v)$ was performed in n, $\perp$ otherwise.
- alias two attributes α_1 and α_2; This means that every reference to α_1 (α_2) in a node $m \succeq n$ will produce the same result as α_2 (α_1);
- $\mathsf{dereference}$ an attribute α—that is identify the value (if any) of α. This is equivalent to finding the node $k = max_{\preceq}\{m \mid access(m, \alpha) \neq \perp \wedge m \preceq n\}$.

Let us refer to the problem of finding this node n as the $\mathcal{OP}$ problem. Later we are going to discuss the problems of *aliasing* and *dereferencing* separately.

3.2 Abstracting And-Parallelism

The main issue in implementing AP is the management of shared variables. Informally, each node may create a variable, try to bind a variable, or alias two variable to each other. More precisely, we need to support the following operations on a variable v:

- *creation*: for our purposes, we can assume that creation is implicit with the expand operation. Let us denote with $\epsilon(v)$ the node where v is created;
- *binding*: assign a value to the variable v;
- *aliasing*: given two unbound variables v and v', make them aliases of each other. Once two variables are aliased, they will reference the same value.

A variable v will be directly accessed or used only in the subtree rooted at $\epsilon(v)$. Aliasing of variables, in general, does not create an immediate problem for and-parallel computation (it will not directly create violations of the Strong Prolog semantics—as long as the subgoal which performs the aliasing is a producer of at least one of the two aliased variables). On the other hand, binding of an aliased variable does create a problem. If two variables v and v' are aliased, and later an attempt to bind one of them is made, then the binding should be allowed only if it satisfies the binding requirements. More precisely, let us use the notation $\alpha(v)$ to denote the set of variables that are aliased to the variable v. For an UN-aliased variable we have $\alpha(v) = \{v\}$.

The binding condition can then be expressed as follows: a binding taking place in a leaf n of the tree can be safely performed (w.r.t. Strong Prolog semantics) iff the node n lies on the leftmost branch of each node in $\alpha(v)$. Equivalently, $\forall w \in \alpha(v)\,(n \trianglelefteq \epsilon(w))$. In the following we will denote with $\mathsf{verify\_leftmost}(l, n)$ the operation which given a leaf node l and a node n (guaranteed to be on the path from the root to l) verifies that l is leftmost in the subtree rooted in n. Thus, apart from the general operations on the tree (creation, expansion, and removal), we need to perform two additional (and, in our opinion, orthogonal) operations to realize and-parallelism. Given a variable v for which a binding is attempted at leaf n of the tree, then in order to successfully bind v we need to:

- identify in $\alpha(v)$ the "oldest" variable, w, in the set (a variable v is older than another variable w, if $\epsilon(v)$ is an ancestor of $\epsilon(w)$). This is necessary in order to determine the reference node for the leftmost test and to locate a memory cell to store the binding.
- verify whether the node n is leftmost with respect to the node identified in the previous step ($\mathsf{verify\_leftmost}(n, w)$).

Let us refer to the first problem (identification of the "oldest" node in the set and memory location) as the $\mathcal{AP}^1$ problem, and the second problem (determining if a node is leftmost in the subtree rooted in a given node) as the $\mathcal{AP}^2$ problem. Later, we will discuss simpler versions of these problems, where one or more operations are not considered (e.g., the $\mathcal{AP}$ problem without the alias operation).

4 Pointer Machines and Extended 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 $\mathcal{N}$). If $\mathcal{F}$ is the finite collection of field names, then the following function $\mathsf{field\_access} : \mathcal{F} \times \mathcal{N} \rightarrow Data \cup \mathcal{N}$ allows to access the value stored in a field of a record. In the rest, for the sake of simplicity we will use the notation $n(r)$ to denote the value returned by $\mathsf{field\_access}(n, \mathrm{address}(r))$.

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 [11, 21, 19].

As evident from the previous description, the pure pointer machine model is quite limited in its ability. It provides a good base for modeling implementation of linked data structures, like trees and lists. But it fails to capture some computational aspects of a real computer architecture. Two major drawbacks of the pure pointer machine model are: *(1)* the lack of arithmetic—numbers have to be explicitly represented as linked lists of bits. This is realistic in the sense that, for arbitrarily large problems requiring arithmetic operations, the time to compute such operations will be a function of the size of the problem. Nevertheless models which assume the uniform cost criterion (like the RAM model)—i.e., cost of an operation is independent from the word size—have been frequently used. *(2)* the lack of pointer arithmetic—the only operation allowed on pointers is dereferencing (following the pointer). Real machines allows arithmetic operations on pointers, allowing implementations of arrays and other direct access structures.

Of the two possible extensions, we consider in this framework the first one. That is, we allow data registers to contain non-negative natural numbers and we allow for some additional machine instructions, which allow to initialize to 0 a data field/register, to increment the content of a data field/register, and two compare ($\leq$) two data registers. Unitary cost of the operations is assumed only on numbers whose maximum size in bits is $\lceil \lg n \rceil$, where n is the size of the problem at hand.

4.1 Complexity of And-Parallelism

The Problem $\mathcal{AP}^1$ The Union-Find problem involves maintaining a collection of disjoint dynamic sets. Sets can be created and merged, and each set is uniquely identified through a representative element. The following operations are allowed: *(1)* Make_set(x): given an element x, this operation creates a singleton set containing x. x is clearly used as representative element for this set. Disjointness requires that x does not belong to any other set of the collection; *(2)* Union(x, y): given two elements x and y belonging to two disjoint sets A_x and A_y, this operation destroys the two sets and creates a new set containing the elements $A_x \cup A_y$. An element in $A_x \cup A_y$ is selected as representative of the newly created set; *(3)* Find_set(x): given an element x which belongs to one set of the collection, the function returns the representative of the set containing x.

The general problem we are considering is that of performing an arbitrary (correct) sequence of the above operations, containing n Make_set operations and m Union and Find_set operations. The operations in the sequence are assumed to be performed *on-line*, that is each operation is not started until the immediately preceding one has been completed. Various researchers have investigated the complexity of this problem. The major result (originally [21] and then extended in [18]) is that the this problem has a lower bound of $\Omega(n + m\alpha(n, m))$, where α is the inverse of the Ackermann function (thus a very slowly growing function). Optimal algorithms achieving this complexity have been proposed [21]. Tarjan also was the first one to prove the upper bound complexity of the problem [22], which happens to be $O(n + m\alpha(n, m))$. We can easily show that the $\mathcal{AP}^1$ problem has a complexity greater or equal to that of union-find. Let us consider an arbitrary solution of the $\mathcal{AP}^1$ problem. This means that we have means to represent variables, keep track of aliasing, retrieve a selected representative (the oldest variable). Given a set of objects $\{a_1, \ldots, a_n\}$ and a sequence of union and find operations, we can implement them on top of the $\mathcal{AP}^1$. Each element a_i is associated to a distinct variable v_i (e.g. the memory location associated to the variable). Each time a union is required, the corresponding variables are aliased. A find operation for an element a_i can be realized by searching for the representative variable in $\alpha(v_i)$. The mapping between elements a_i and variables does not require more than a finite number of steps. This implies that the amortized complexity of $\mathcal{AP}^1$ is $\Omega(n + m\alpha(n, m))$, where n is the number of variables and m is the number of alias/find operations. Moreover, from the results of [2], the single operation worst case complexity for this problem is $\Omega(\frac{\lg n}{\lg \lg n})$ for a large class of algorithms on pointer machines.

An optimal upper bound for this problem is still unknown. In general, $\mathcal{AP}^1$ seems to have a complexity greater than that of union-find. This arises from the fact that when a variable v is about to be bound, and we are performing a find on $\alpha(v)$, then we want to obtain as result not any arbitrary element of the set but the oldest variable in $\alpha(v)$. This requires the use of some sort of time stamping which may result in time complexity worse than that of union-find.

A simple (amortized) upper bound is $O((n + m\alpha(n, m)) \lg n)$. This is due to the fact that we can "time-stamp" each variable with its creation time and, amongst the aliased variables, we keep the time stamp of the oldest variable. A naive scheme to maintain time-stamps requires an order $O(\lg n)$ overhead on pointer machines. This scheme for time-stamping is not optimal. Using the results in [17], where it is shown that the time-stamping can be performed with complexity $O((\lg\lg n)^k)$ for some small k, we can show that the $\mathcal{AP}^1$ problem can be solved in time $O((n + m\alpha(n,m))(\lg\lg n)^k)$ on pointer machines.

The problem $\mathcal{AP}^2$ In this section we will argue that the problem $\mathcal{AP}^2$ has an upper bound of $O(n + m\alpha(n,m))$, where n is the number of nodes created and m are the remaining operations.

The upper bound for the $\mathcal{AP}^2$ problem on a pure pointer machine can be derived from the results known for the *union-find* problem [22, 21, 18, 4]. Let us assume that we have an implementation of the union-find problem, i.e. we have a notion of disjoint set and the three operations Make_set, Find_set, and Union. We show that this can be adopted to give an efficient implementation of the tree manipulation operations required for $\mathcal{AP}^2$. We assume that we are working on a pointer machine. Trees are represented in the traditional way. Each node is encoded in a record, and each record is assumed to have at least three pointers available, one to identify the parent node (*nil* if the node is the root of the tree), and two to identify the two children. The intuitive idea is to use the union-find operations to keep track of the "left branches" in the tree. The following operations on the tree are performed (assumption is that we are processing binary trees):

- the creation of the tree does not require any special action. A new singleton set containing the node is created through the Make_set operation.
- if we expand one node n to the right, adding a node m below it, then we have two possible cases:
  - if the node n has a left child, then nothing else is required.
  - if the node n does not have a left child, then a Union operation is performed between the set containing n and the one containing m.
- if we expand one node n to the left, then we should perform a union operation between the new node m added and the set containing n.
- let us consider the various cases that may occur when a leaf n is removed. Let m be the predecessor of n in the tree.
  - if n is the right child of m, then no special action is required upon removal of n;
  - if n is the left child of m and m does not have any right child, then again no special action is required;
  - if n is the left child of m and m has a right child v, then upon removal of n it is necessary to apply a Union operation between the set containing m and the one containing v.
- the operation of determining if a node n is leftmost in the subtree rooted at a certain node m can be solved by applying the Find_set operation to both n and m and verifying whether they fall in the same set.

From the above discussion we can infer that for $\mathcal{AP}^2$, given a sequence of n create operations and m operations of all other types, the problem can be solved with a time complexity of $O(n + m\alpha(n,m))$. This provides an amortized upper bound to the problem. Regarding lower bounds on pure pointer machines, clearly $\Omega(n+m)$ is an obvious lower bound. We conjecture that this is not tight. In section 6 we propose some related conjectures, proved for special cases, which indeed show that the lower bound is $\omega(n+m)$. Summarizing, the complexity $C_{\mathcal{AP}^2}(n,m)$ of $\mathcal{AP}^2$ is

$$\Omega(m+n) \;\stackrel{?}{<}\; C_{\mathcal{AP}^2}(n,m) \;\leq\; O(n+m\alpha(n,m))$$

Our conjecture is that the problem indeed has the same complexity as Union-find. This is also substantiated by the ability to show a lower bound of $\Omega(m + n\alpha(n,m))$ for a slightly different problem [14].

We would like to mention that, using the "time-stamping" scheme of [17], the $\mathcal{AP}^2$ problem can be solved with single operation worst-case time complexity of $O(\lg\lg n)$ per operation. This is better than the $O(\frac{\lg n}{\lg\lg n})$ scheme that will result if we use the union-find scheme from [2].

Complexity on Extended Pointer Machines The extended pointer machines add a fundamental component to the design, the ability to perform basic increment and comparison operations between natural numbers. This feature is sufficient to modify some of the lower bounds previously discussed. In particular, the presence of counters and the ability to compare them in constant time allows us to solve the $\mathcal{AP}^2$ problem in time $\Theta(1)$ per operation.

It is possible to show that keeping track of the $\mu(n)$ information for each leaf of the tree can be realized with complexity $O(1)$ per operation [16]. This scheme can be modified on an extended pointer machine to solve the $\mathcal{AP}^2$ problem. Let us assume that each record in the memory of the pointer machine contains a data field capable of maintaining a counter, called age. During the create_tree operation, the root of the tree is created and the value 0 is stored in the age field of such record. Every time a node n is expanded, the two new nodes created (n_1 and n_2) receive a value in their age field obtained by incrementing by one the age of n.

The remove operation leaves unaffected the age fields of the various nodes. We can assume that each variable v itself is represented by a record, which contains a pointer to the node $\epsilon(v)$ (the node where v was created). For each group of aliased variables we also maintain the information $max(v) = min_{\preceq}\{m \mid m = \epsilon(w) \wedge w \in \alpha(v)\}$. When the binding for a variable v is attempted, then we will need to verify whether the current computation point (represented by the leaf n in the tree) is leftmost in the subtree rooted at $max(v)$. At this point, the operation verify_leftmost can be easily realized. The intuitive idea is that the counter can be used to compare the depth of nodes. The sub field points to the highest node in the tree in which such node is on the leftmost branch. If such node is below the expected node, then the test should fail, otherwise it will succeed. On the other hand, it is also quite easy to see that, as consequence of the uniform cost assumption on the management of counter (addition and comparison), we can propose an upper bound for $\mathcal{AP}^1$. It is relatively straightforward to add a simple comparison in the implementation of the union operation in Tarjan's optimal algorithm [21] to force the root of the tree to always contain the older variable. This allows direct use of the union-find algorithm to support the $\mathcal{AP}^1$ problem, resulting in a complexity of $O(n + m\alpha(n, m))$.

The two results combined, reflect exactly the "practical" results achieved in [14], where a "constant-time" practical implementation scheme for handling the $\mathcal{AP}$ problem without aliasing is described.

5 Complexity of Or-Parallelism

The problem of Or-parallelism is considerably more complex than those we have explored earlier. 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 [7]. The results expressed in this work show that a generic $\mathcal{OP}$ problem with n attributes and m operations per branch generates a lower bound which is strictly worse than $\Omega(n + m)$.

It is quite easy to observe that along each branch of the tree, attributes are managed in a way very similar to the way in which elements are managed in the union-find problem. Initially all attributes are distinct; given an attribute α, let us denote with $A(\alpha)$ the set of all the attributes that are currently (on the branch at hand) aliased to α. If a new aliasing between α_1 and α_2 takes place, then the set A is modified as $A(\alpha_1) = A(\alpha_1) \cup A(\alpha_2)$. If we assume that the value of an attribute α is always associated to the representative of $A(\alpha)$, 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 1. *The amortized time complexity of $\mathcal{OP}$ is $\Omega(n + m\alpha(n, m))$, where n is the number of attributes 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 [21, 18] we know that this problem has a lower bound of $\Omega(n+m\alpha(n, m))$. 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 attributes. Each time an operation is performed a new node is added along the left spine of the tree (we can ignore in this context the nodes that will be added to the right during the expansion). Each union operation is translated into an aliasing between the two attributes. Each find operation corresponds to the dereferencing of the corresponding attribute. 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 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. Thus, the inability to achieve constant-time behaviour is inherent in the management of the attributes in the tree.

Theorem 2. *On pointer machines, the worst case time complexity of $\mathcal{OP}$ is $\Omega(\lg n)$ per operation even without aliasing.*

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 find operation in worst–case $c(n)$ time for some function c. The find procedure receives only two pointers as arguments, one to the record that contains the name of the attribute whose value is being searched, and the other to the leaf node for which this attribute value 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. 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 $t_2(n) = (n+1)/2^{t_1(n)} - 2$. Let us also assume that we have $t_2(n)$ different attributes, $\{\alpha_1, \ldots, \alpha_{t_2(n)}\}$. All the nodes at level $t_1(n) + i$ of the tree perform a bind operation on α_i, and each node assigns a different value to α_i. Suppose we choose $t_1(n)$ and $t_2(n)$ such that

$$2^{c(n)+1} t_2(n) < 2^{t_1(n)} \qquad (1)$$

Then there exists a leaf node l in the tree such that none of the last $t_2(n)$ nodes on the path from root to l is accessed starting from the pointer to the attribute node in any of the computations $\mathsf{find}(\alpha_i, l)$, $i = 1 \ldots t_2(n)$. This implies that for any of the $t_2(n)$ different calls $\mathsf{find}(\alpha_i, l)$ the appropriate node on the path must be accessed from starting from the leaf node. Since the find procedure works in time $c(n)$ we must have that $2^{c(n)+1} \geq t_2(n)$.

This gives us that $c(n) \geq \lg t_2(n) - 1$. Since, $c(n) < \frac{1}{8}\lg n$, $2^{c(n)+1} < 2n^{1/8}$. Let us choose $t_1(n) = \frac{3}{4}\lg n$. 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 $c(n) < \frac{1}{8}\lg n$. Hence $c(n) = \Omega(\lg n)$. □

Upper Bound for $\mathcal{OP}$ The related 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 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 some small constant k.*

Proof. *(Sketch)* the $\mathcal{OP}$ problem can be cast in the following simplified version. Each node in the tree is assigned a label. The same label can be used in various nodes of the tree, but no label can occur more than once in each branch of the tree. The only two operations allowed are $\mathsf{insert}(n, \alpha_1, \alpha_2)$ which creates two new nodes under node n (required to be a leaf) and assigns respectively the label α_1, α_2 to the newly created nodes, and $\mathsf{label}(n, \alpha)$ which retrieves (if any) the node with label α present on the path between n and the root of the tree. Observe that we are simplifying the problem by disallowing deletions of nodes and restricting the attention to binary trees. To obtain the worst-case time complexity $\tilde{O}(\sqrt[3]{n})$ per operation we maintain the collection of nodes assigned the same label α in tables of size $\sqrt[3]{n}$, and record in the tree the nodes where one table gets filled (*fill nodes*). Furthermore, every $\sqrt[3]{n}$ fill nodes a table is created summarizing the effect of the previous $\sqrt[3]{n}$ fill nodes. A label operation is performed by first scanning the fill nodes up to the last created table, and then visiting table by table. Either an entry is found in one of the initial nodes or one of the tables (which will directly point to the group containing the node of interest), or the node of interest is in the first table for such label (i.e., the first table has not been filled yet), or the label has never been used along the considered branch. Testing each entry in a table requires the checking for nearest common ancestor [8, 5, 23]. □

6 Open Problems

Various problems are still open and are currently being researched. First of all, for various problems we have proposed lower and upper bounds which are rather distant; these gaps need to be closed. The gap is particularly evident for the $\mathcal{OP}$ problem (even without aliasing), where the single operation worst–case lower bound is $\Omega(\lg n)$, while the upper bound is $\tilde{O}(\sqrt[3]{n})$ per operation in the worst case. We still lack a lower bound for the $\mathcal{AP}^2$ problem. The problem, as illustrated in [17], seems to have very strong ties with the mentioned *time stamping* problem.

Let us consider the following problem. Suppose we use a pointer machine to maintain an ordered sequence of elements. The only two operations allowed are insert_element, which adds a new element to the sequence, and precedes, which tests whether one element has been inserted before another. Let us consider the problem of performing a sequence of n insertions followed by a precedes test between two arbitrary elements of the sequence (we will refer to this problem as the $\mathcal{IP}$ problem). Our conjecture is the following:

Conjecture 4. *On a pure pointer machine, an arbitrary (correct) sequence of n operations belonging to the $\mathcal{IP}$ problem has a complexity strictly worse than*

$\Omega(n)$—in particular, there is no solution where both insertion and test use a constant number of steps per operation.

In [17] we show that the $\mathcal{IP}$ problem has a single operation worst–case upper bound of $O(poly(\lg\lg n))$. We can prove this conjecture for a special class of algorithms, but this is not sufficient to establish the above conjecture. We are confident that the result holds and can be proved. The relevance of this conjecture can be seen from the following:

Lemma 5. *The problem $\mathcal{AP}^2$ has a single operation worst–case lower bound of $\omega(1)$ iff conjecture 4 is true.*

Proof. if we can solve $\mathcal{IP}$ in constant time (i.e., conjecture 4 is false), then $\mathcal{AP}^2$ can also be solved with single operation worst–case time complexity of $\Theta(1)$. This follows from the scheme described in section 4.1. Let us assume by contradiction that we are capable of implementing all the operations required by the $\mathcal{AP}^2$ problem in constant time. These operations can be used to provide a solution to the $\mathcal{IP}$ problem. In fact, the operation of inserting a new element can be translated directly in an expand operation applied on the leftmost leaf of the tree; the first argument of the expand is the node inserted by the $\mathcal{IP}$ operation, while the second node inserted is a dummy node. The precedes operation can be directly translated into an is_leftmost operation. In fact, given two nodes n_1 and n_2 which have been previously inserted, we know by construction that both lie on the leftmost branch of the tree. Thus, n_1 is leftmost in the tree rooted in n_2 iff n_2 has been inserted before n_1 in the tree. □

7 Conclusions

In this paper we studied the complexity of several operations associated with parallel execution of Prolog programs. The following results were achieved:

- on both the pure and extended pointer machines, execution of and-parallelism requires a non-constant execution time due to the need of managing aliased variables. If aliasing is disallowed, a constant time behaviour is achievable on extended pointer machines. An upper bound for the problem was found. We showed that managing and-parallelism has an upper bound equivalent to the union-find problem.
- On both models, execution of or-parallelism was showed to have a non-constant time lower bound. In contrast to and-parallelism, the non-constant time holds even when we consider extended pointer machines and we disallow aliasing of variables.

Some problems are still open, the main one being showing that the problem $\mathcal{AP}^2$ has a non-constant time lower bound on pure pointer machines. This result is currently based on the conjecture about the $\mathcal{IP}$ problem. Such conjecture has been proved to hold in special cases, but a general proof is still missing.

Another open problem of interest is the development of an upper bound 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.

References

1. K.A.M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1991. 19(6):445–475.
2. N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4), 1986.
3. M. Carlsson,G. Gupta, K.M. Ali and M.V. Hermenegildo. Parallel execution of prolog programs: a survey. *Journal of Logic Programming*, 1998.. to appear.
4. M. Fredman, M. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Procs. of 21st ACM Symposium on Theory of Computing*. ACM, 1989.
5. H.N. Gabow. Data structures for weighted matching and nearest common ancestor. In *ACM Symp. on Discrete Algorithms*, 1990.
6. G. Gupta and V. Santos Costa. Cuts and Side-effects in And/Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, 1996.
7. G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.
8. D. Harel and R.E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestor. *SIAM Journal of Computing*, 13(2), 1984.
9. M. Hermenegildo and K. Greene. &-Prolog and its Performance. In *Int'l Conf. on Logic Prog.*, MIT Press, 1990.
10. J.C. Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3), 1994.
11. D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.
12. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
13. E. Lusk and al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3), '90.
14. E. Pontelli and G. Gupta. Implementation mechanisms for dependent and-parallelism. In *International Conference on Logic Programming*. MIT Press, 1997.
15. E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-performance Parallel Prolog System. In *IPPS 95*. IEEE Computer Society, 1995.
16. E. Pontelli, D. Ranjan, and G. Gupta. On the complexity of the Parallelism in Logic Programming. Technical report, New Mexico State University, 1997.
17. E. Pontelli, D. Ranjan, and G. Gupta. On the complexity of the insertion/precedes problem. Technical report, New Mexico State University, 1997.
18. H. La Poutré. Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines. *Journal of Computer and System Sciences*, 52:87–99, 1996.
19. A. Schönhage. Storage Modification Machines. *SIAM Journal of Computing*, 9(3):490–508, August 1980.
20. D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26, 1983.
21. R.E. Tarjan. A Class of Algorithms which Require Nonlinear Time to Mantain Disjoint Sets. *Journal of Computer and System Sciences*, 2(18), 1979.
22. R.E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF, 1983.
23. A.K. Tsakalidis. The Nearest Common Ancestor in a Dynamic Tree. *ACTA Informatica*, 25, 1988.