# Analysis of Or-Parallel Execution Models†

**Gopal Gupta**
*Department of Computer Science*
*New Mexico State University*
*Las Cruces, NM 88003-0001*
*E-Mail: gupta@nmsu.edu*
*Tel: 505-646-6236*

**Bharat Jayaraman**
*Department of Computer Science*
*State University of New York at Buffalo*
*Buffalo, NY 14260, U.S.A.*
*E-Mail: bharat@cs.buffalo.edu*
*Tel: 716-645-3194*

## Abstract

We discuss fundamental limitations of or-parallel execution models of nondeterministic programming languages. Or-parallelism corresponds to executing the different nondeterministic computational paths in parallel. A natural way to represent the state of (parallel) execution of a nondeterministic program is by means of an or-parallel tree. We identify three important criteria that underlie the design of or-parallel implementations based upon the or-parallel tree: constant-time access to variables, constant-time task creation, and constant-time task switching, where the term 'constant-time' means that the time for these operations is independent of the number of nodes in the or-parallel tree, as well as the size of each node. We prove that all three criteria cannot be simultaneously satisfied by any or-parallel execution model based on a finite number of processors but unbounded memory. We discuss in detail the application of our result to the class of logic programming languages, and show how our result can serve as a useful way to categorize the various or-parallel methods proposed in this field. We also discuss the suitability of different or-parallel implementation strategies for different parallel architectures.

† This paper generalizes and expands the results in our paper, "Criteria for Or-Parallel Execution Models of Logic Programs," in *Proc. North American Conference on Logic Programming*, Austin, TX, November 1990.

## 1. Introduction

This paper explores fundamental limitations of or-parallel execution models of nondeterministic programming languages. We use the term 'nondeterminism' in the automata-theoretic sense of "don't know" nondeterminism, rather than in the sense of "don't care" nondeterminism. Or-parallel execution of nondeterministic programs involves the parallel exploration of the multiple search paths arising due to nondeterministic choices. An important subclass of these languages is the class of logic programming languages. In these languages, execution involves the solution of goals with respect to a set of clauses, and or-parallelism arises whenever a goal unifies with more than one clause head, in which case the corresponding clause bodies are executed in parallel. In general, exploiting or-parallelism is a useful way to speed up the solution to search problems in artificial intelligence and applications of symbolic computation and reasoning, such as theorem proving, natural language analysis, etc.

A natural way to represent the state of (parallel) execution of a nondeterministic program is by means of an or-parallel tree. In this paper, we analyze execution models that are based on the or-parallel tree. We base our analysis upon the following three criteria for an ideal or-parallel execution model:

1. the cost of *environment creation* should be constant-time;
2. the cost of *variable access and binding* should be constant-time; and
3. the cost of *task switching* should be constant-time.

We use the term 'constant-time' to mean that the time for these operations is independent of the number of nodes in the or-parallel tree, as well as the size of each node. While the connection between these criteria might appear remote at first, they actually are derived from the three principal operations in any nondeterministic programming system (sequential or parallel): procedure call (which involves environment creation), assignment statements and parameter transmission (which involves variable access and binding), and resumption after success or failure (which involves task switching). The criteria essentially state that the principal operations in an or-parallel execution model should be performed efficiently, i.e., in constant-time. While it would be ideal if one could satisfy all three criteria, we show in this paper that this ideal cannot be achieved by any or-parallel model using a finite number of processors and constant-time addressable memory.

To appreciate this result, consider the sequential execution of logic programs. A sequential execution model, which can be viewed as a special case of an or-parallel execution model, provides constant-time environment creation and constant-time variable access and binding, but not constant-time task switching. The latter operation, also known as *backtracking*, requires undoing the bindings of variables, but the number of such variables is not a constant in general since it depends upon the depth of the search path. It might appear that an or-parallel model does not have to perform backtracking, and hence all three criteria can be satisfied. But this reasoning is not correct because of the finite processor assumption; the or-parallel tree can have more or-parallel tasks available than processors, and hence there is a need for task switching. One can make task switching a constant-time operation (as in the directory tree method [HC86]), but, as we will see, this can be achieved only at the expense of the other two operations.

An important difference from a sequential implementation is that an or-parallel implementation must support multiple bindings (or values) for the same variable. Such multiple bindings are possible for a variable that is passed *by reference* in a procedure call, and the the different nondeterministic search paths arising from this call assign different values to this variable. (For logic programs, unification plays the role of parameter transmission.) These multiple bindings are the main cause for non-constant time variable access, because it is not obvious, without additional bookkeeping, which binding is applicable when a variable is accessed from an arbitrary node of the or-parallel search tree. Attempts to make variable access a constant-time operation lead to non-constant overheads in the other operations. For example, in order to facilitate constant-time variable access, if we record the binding of a variable applicable to a given node in the node itself, then environment creation is affected because there are in general a non-constant number of variables for which this information must be recorded.

The fact that at least one of the criteria must be sacrificed explains, in retrospect, why so many or-parallel execution models of logic programs have been proposed in the literature [A88, AK90, B84, C87, DLO87, HC83, HC86, H87, K86, L84, T89, W84, W87a]. The three criteria provide a natural scheme for classifying the various or-parallel methods proposed in the literature, namely, according to which (proper) subset of the criteria they satisfy. Our treatment is more general than that of D.H.D. Warren [W87b], who examined a few or-parallel methods, especially those that are variants of the Argonne Model [DLO87] and the SRI model [W87a], as well as that of Hausman and Ciepielewski [HC86], who examined several variations of their directory tree method [HC83]. Our study encompasses all or-parallel methods that are based on the or-parallel search tree. While the three criteria have been briefly mentioned in works of other researchers [W87a, H87, HC86], the main contributions of our work lie in: (i) stating these criteria in their present form, (ii) proving that these criteria cannot all be satisfied together, and (iii) using them as a basis for classification.

The remainder of the paper is organized as follows: section 2 introduces the essential concepts and terminology in or-parallel models in the form of definitions and remarks; section 3 proves the main result with aid of a few lemmas, by considering all possible ways of solving the problem and showing that one or more of the criteria fail in each case; section 4 discusses in detail the result in the context of logic programs; section 5 uses the result as a basis for classifying several or-parallel execution models; and, finally, section 6 presents a brief summary and conclusions.

## 2. Notation and Terminology

To keep this paper self-contained, we provide in this section definitions for all technical terms, and also introduce notation that will be used in later sections.

**Definition:** *nondeterministic program, query*

A *nondeterministic program* is a set of procedure definitions, each of which is of the form *head; body*, where *head* has the form *id(pars)* and the same procedure *id* may appear in the head of more than one procedure definition. The syntactic category *pars* stands for a possibly empty list of formal parameters, each of which is assumed, without loss of generality, to be a *reference* parameter (*call-by-reference*).

A *query* has the form of a procedure *body*, which consists of a sequence of statements, each of which is either a procedure *call* or an *assignment* to a formal parameter.

**Remarks:**

The following assumptions are made to simplify the discussion, but they do not affect our technical results.

i. No *type* declarations for variables are given. The domain of data values is also left unspecified.
ii. There are no nested procedure definitions or free (non-local) variables appearing in procedure definitions.
iii. Control structures such as *if-then-else*, *while*, etc. are absent.

**Definition:** *single-assignment property*

A variable can be bound at most once in any nondeterministic computational path.

**Remarks:**

This is the generalization of the well-known single-assignment property to nondeterministic languages. It suffices to restrict attention to this case, since our results are no easier to prove without this restriction.

### 2.1. Or-parallel Tree

In the following definitions, $N$ stands for the possibly infinite set of nodes in the or-parallel tree, $V$ the possibly infinite set of all variables, $T$ the set (domain) of terms or values, $P$ the set of processors, and $M$ the set of memory locations in the multiprocessor system. Let $\mathcal{P}(S)$ denote the powerset of a set $S$, and $|S|$ its cardinality.

**Definition:** *or-parallel tree, local environment*

An *or-parallel tree* for a given nondeterministic program and query is a tree of nodes, each with an associated *continuation* and *local environment*, such that:

1. the root node of the tree has the query as its associated *continuation*, and the set of variables appearing in the query as its *local environment*; and
2. each non-root node $n$ is created as a result of choosing a different procedure definition to execute the first call in the body of $n$'s parent node, and:
   a. the *continuation* of node $n$ consists of the statements in the body of the chosen procedure definition followed by statements after the first call in the parent of $n$ (the statements are assumed to be interpreted in the environment of $n$).
   b. the *local environment*, $l(n)$, of $n$ is the set of all variables present in the chosen procedure definition, where $l : N \to \mathcal{P}(V)$.

**Remarks:**

i. All nodes of an or-parallel tree need *not* have the same number of children nodes, i.e., the *branching factor* is not the same for all nodes.

ii. The branching factor is bounded by a constant, determined by the program.

iii. We assume there is an unbounded supply of variables, and variables in a definition are uniquely renamed before the call so that the sets of variables in any two local environments are disjoint from one another.

**Definition:** $\leq$

For two nodes $n_1$ and $n_2$ of an or-parallel tree, we say $n_1 \leq n_2$ iff $n_1$ and $n_2$ lie on the same path from the root of the tree and either $n_1 = n_2$ or $n_1$ is closer to the root than $n_2$. In the latter case, we say $n_1 < n_2$.

**Remarks:**

i. $\leq$ is a partial order.

ii. We will sometimes write $m \geq n$ instead of $n \leq m$.

**Definition:** *global environment*

The *global environment*, $g(n)$, of a node $n$ in an or-parallel tree is the union of the set of variables in all local environments lying on the path from the root of the tree to $n$. That is, $g : N \to \mathcal{P}(V)$, and $g(n) = \{v \mid (\exists x)\ x \leq n$ and $v \in l(x)\}$.

**Remarks:**

i. We assume that $n_1 \neq n_2 \Rightarrow g(n_1) \neq g(n_2)$, since we may introduce one distinct dummy variable in the local environment of each node.

**Definition:** *owner node, access node, binding node*

For each variable $v$ in the local environment of a node $n$:

1. There is a unique *owner node* for $v$, defined as $o(v) = n$, where $o : V \rightarrow N$.
2. There is a subset of $N$ called *access nodes* for $v$, where each access node $m$ is such that $m \geq n$.
3. There is a subset of access nodes called *binding nodes* for $v$, which can be characterized in terms of a partial function $b : V \times N \rightarrow N$ that specifies the binding node of a variable $v$ in the *global environment* of a node $n$. Due to the *single-assignment* property stated earlier, the function $b$ satisfies the requirement that if there is a node $m$ such that $b(v, m) = m$, then $b(v, y)$ is undefined for all $y < m$ and $b(v, y) = m$ for all $y \geq m$. The set of binding nodes for $v =$

$$\{m \mid b(v, m) = m\}.$$

**Remarks:**

i. At each access node of a variable $v$, the value of $v$ is accessed; and at each binding node of a variable $v$, a value is assigned to $v$.

ii. The characterization of the or-parallel tree in terms of owner, binding and access nodes is based on the observation that there are three basic operations performed on variables: environment creation, variable binding, and variable access.

iii. Every binding node is an access node since a variable must be accessed before being bound; i.e., variable binding is a special case of variable access.

**Proposition 1:** *uniqueness property of binding nodes*

If a variable $v$ has two distinct binding nodes $n_1$ and $n_2$, then $n_1 \not< n_2$ and $n_2 \not> n_1$, i.e., $n_1$ and $n_2$ do not lie on the same path from the root.

**Proof:** Let $n_1$ and $n_2$ be two distinct binding nodes, i.e., $n_1 \neq n_2$. Suppose $n_1 \leq n_2$, to derive a contradiction. By the definition of a binding node, for all $y \geq n_1$, we have $b(v, y) = n_1$. Hence $b(v, n_2) = n_1$. But $b(v, n_2) = n_2$, since $n_2$ is a binding node. Hence $n_1 = n_2$, a contradiction. **End of Proof.**

**Corollary:** The binding node of a variable in a given global environment is unique, if it exists.

**Definition:** *value cell, value*

For each variable $v$ in the local environment of a node $n$:

1. There is a set of *value cells* for $v$, where each value cell is a memory location that is used for storing the value of $v$ created by the corresponding binding node. We use a partial function $vc : V \times N \rightarrow M$, which defines the value cell for $v$ in the global environment of a node $n$, where the function $vc$ satisfies the requirement that $vc(v, y)$ is undefined for all $y < o(v)$.

2. There is a one-to-one correspondence† between the set of binding nodes for $v$ and a set of values. This set of values can be specified by a partial function $val : V \times N \rightarrow T$, which defines the value for $v$ in the global environment of a node $n$. For each binding node $m$ for $v$, we have (a) for all $y < o(v)$, $val(v, y)$ is undefined; (b) for all $o(v) \leq y < m$, $val(v, y)$ is unbound; and (c) for all $y \geq m$, $val(v, y) = val(v, m)$.

**Remarks:**

i. Note that the value cell does not always refer to the cell allocated for $v$ in the local environment of the owner node of $v$; rather, the value cell is the memory cell allocated for holding the binding generated by a binding node.

ii. The function $val(v, n)$ cannot be described algorithmically because then we would be able to predict the value of a given variable in a particular environment even before that variable gets bound.

**Definition:** *conditional binding, conditional variable*

A binding for a variable $v$ is said to be *unconditional* iff either $b(v, o(v)) = o(v)$ or the branching factor of every node from $o(v)$ down to and excluding the binding node of $v$ is one. In this case, the variable $v$ is said to be *unconditionally bound*; otherwise, the binding is said to be *conditional* and $v$ is referred to as a *conditionally bound variable* or a *conditional variable.*

**Definition:** *complete or-parallel tree*

A *complete or-parallel tree* is one where:

1. for each leaf node $l$, the continuation of $l$ is either empty (*success leaf node*) or the first call in $l$ cannot be processed due to an undefined procedure (*failure leaf node*); and
2. for each non-leaf node $n$, a child node is present in the or-parallel tree corresponding to each procedure definition in the program that can be used execute the first call in $n$'s continuation.

**Remarks:**

i. An *incomplete or-parallel tree* is one in which some leaf or non-leaf node can be extended by creating a child node.

ii. It is possible that there is no complete or-parallel tree for a given query and program (because of nontermination).

### 2.2. Computational Model

We assume a model of computation in which there is an unbounded number of memory locations, each of which can be accessed in constant-time. We assume that a finite number of processors (i.e. $|P|$ is finite) cooperatively

† We assume that two identical bindings generated by different binding nodes are distinguishable from each other, say, through the use of tags.

construct the or-parallel tree, by creating children nodes as long as the tree is incomplete. In order to create a child node at some node $n$, a processor must first "move" from its current node to the location of node $n$. All processors may be assumed to be located "at" the root of the or-parallel tree initially. The following definitions focus on three important operations in this model of computation: task creation, task switching, and variable access.

**Definition:** *task site*

A *task site* in an incomplete or-parallel tree is a node $n$ for which there is at least one untried procedure definition for the first call in $n$'s continuation.

**Definition:** *task creation*

If $n$ is a task site, a processor is said to *create a task* at node $n$ when it constructs the global environment and continuation for a child node corresponding to an untried procedure definition at $n$. The time taken to construct the *global environment* of the child node of $n$ is called the *cost of task creation.*

**Remarks:**

i. A lower-bound on the cost of task creation is the cost of creating the local environment for the child node of $n$.

ii. In certain parallel schemes [C87, L84], variable bindings are copied from a child node's local environment to the parent node's local environment after completion of a call. We regard this work as part of task creation for the next call.

**Definition:** *current environment*

The *current environment* of a processor located at node $n$ is $g(n)$.

**Definition:** *task switching*

A processor is said to *task-switch* when it moves from node $n_1$ to node $n_2$ in an or-parallel tree. The time taken by the processor to change its current environment to that at $n_2$ is called the *cost of task-switching.*

**Definition:** *access cell*

For every variable $v$ accessible to a processor $p$ at a node $n$, there is a memory location called an *access cell* that is used for looking up the binding of $v$ during variable access. A partial function $ac : V \times P \times N \rightarrow M$ defines the access cell for $v$ in the environment of a processor $p$ at node $n$. The function $ac$ satisfies the requirement that $ac(v, p, y)$ is undefined for all $y < o(v)$.

**Remarks:**

i. The difference between an access cell and a value cell is that the former is used for *looking up* the binding of $v$ during variable access, whereas the latter is used for *storing* the binding of $v$ during variable binding.

ii. Each value cell of a variable has to be accessible to all processors since it contains the binding generated for the variable. Hence, value cells cannot be maintained privately by any processor. On the contrary, each processor can maintain its own access cell for any particular binding of any variable $v$. Hence $vc$ is not a function of $p$, but $ac$ is.

iii. For every $v$, $n$, and $p$ the value associated with the access cell $ac(v,p,n)$ is identical to that of value cell $vc(v,n)$. Thus, any or-parallel model has to ensure that the contents of value cells are reflected in the access cells in a given environment. Some methods ensure this by having the value cell and access cell share the same physical memory location [HC83, B84]; other methods keep them physically separate but copy the binding in both the value cell and access cell during variable binding generation [W87, H87].

**Definition:** *variable access*

The process of locating the access cell $ac(v,p,n)$ by a processor $p$ at node $n$ for a variable $v$ in $g(n)$ is called *variable access.* The time taken by the processor for this operation is called the *cost of variable access.*

**Remarks:**

i. Locating the final binding of a variable is a two-step process: (a) variable access, in which the applicable access cell is located, and (b) dereferencing, in which the contents of the binding cell are used to obtain the final binding. Dereferencing may involve indirection and thus in general would be non-constant time. Our definition of variable access does not include dereferencing.

We conclude this section by re-stating our three criteria for an ideal or-parallel implementation in the light of the discussion above:

1. the cost of task creation should be constant-time;
2. the cost of variable access should be constant-time; and
3. the cost of task switching should be constant-time.

By constant-time, we mean that the time taken by these operations do not depend either on the size of the or-parallel tree or the size of the continuation. Similarly, we consider the size of a set to be non-constant if its cardinality is proportional to the depth or size of the or-parallel tree, the size of the continuation or size of terms created during execution. For example, the size of the set of conditional variables in a global environment of a node is non-constant, since its cardinality is proportional to the depth of the tree.

## 3. The Main Result

In this section we prove that the three criteria cannot be simultaneously satisfied. The most conventional strategy for this proof would be to construct a constant-time reduction of a known problem having the desired property to the problem at hand, such that the known problem is solved if and only if the problem at hand is solved. However, we are not aware of any problem that could serve as a basis for reduction. We therefore present an alternative approach: we essentially consider all possible ways in which the problem could be solved and show that one or more of our criteria is violated. The proof can be divided into two main cases:

1. *Access cells and value cells are identical.* Since there is a one-to-one correspondence between values and binding nodes, accessing a value reduces to that of determining binding nodes (this will be proved in corollary 1). For such methods we show that, in general, no binding node can be determined without violating one of the criteria. Lemma 1 considers the case when the information needed to locate a binding node is kept somewhere in the or-parallel tree, i.e., in some combination of owner nodes, binding nodes, and access nodes; and lemma 2 considers the case when the information is maintained in data-structures independent of these nodes. Theorem 1 combines lemmas 1 and 2 together.

2. *Access cells and value cells are separate.* For such methods we show in Lemma 3 that task-switching is always a non-constant time operation.

Finally, the conclusions of Lemmas 1, 2 and 3, and Theorem 1 are used to prove Theorem 2, our main result.

Our constant-time arguments hinge on the following basic proposition, the proof of which is obvious.

**Proposition 2:** *Given an arbitrary set $S$ and a property $P$ such that $P(x)$ is true for at most one $x \in S$, determining which $x$ satisfies $P(x)$ cannot be done in constant-time, i.e., in time independent of $|S|$.*

In the lemmas below we assume that a variable is treated uniformly in all environments. For example, if a particular binding of a variable in a particular environment is stored in a value cells allocated in the binding node, then all other bindings for $v$ in other environments are also stored in value cells allocated in respective binding nodes and so on. It is easy to extend the proofs to include non-uniform treatment of variables in different environments, but we do not include it here for the sake of simplicity and brevity.

**Lemma 1** : *Let access cells and value cells be identical. If the information regarding binding node $b(v, n)$ (of a variable $v$ in the global environment of a node $n$) is associated with either the owner node, the binding node or the access nodes, then either $b(v, n)$ cannot be determined in constant-time by a processor at node $n$ or non-constant overhead is incurred during task-creation.*

**Proof:** We consider all possible places where the information about $b(v, n)$ can be recorded in the or-parallel tree. Since there are only three operations that can be performed on a variable—creation, binding/unbinding, and accessing its value—we consider first the three cases where the information is maintained exclusively in either the owner (or creation) nodes, binding nodes, or access nodes, and then consider the case when the information is maintained in some combination of these three types of nodes:

i. *The association between $v, n$, and $b(v, n)$ is kept in the binding node:* Since there is only one binding node for $v$ in the global environment $g(n)$ of $n$ and since there are a non-constant number of nodes in $g(n)$, it follows from proposition 2 that determining $b(v, n)$ is not a constant-time operation. If there is no binding node for $v$ in $g(n)$, this fact also cannot be determined in constant-time, by proposition 2, since the number of nodes between $o(v)$ and $n$ is not a constant in general.

ii. *The association between $v, n$, and $b(v, n)$ is kept in the owner node:* In this case, the owner node of $v$ maintains the set of binding nodes for $v$, $B(v) = \{m \mid b(v, m) = m\}$. In order for a processor at $n$ to determine the applicable binding node $b(v, n)$ for $n$, it must determine an $x \in B(v)$ such that $x \leq n$. Since $B(v)$ is a set of non-constant size, searching for an $x$ such that $x \leq n$ is a non-constant time operation, by proposition 2.

iii. *The association between $v, n$, and $b(v, n)$ is kept in the access nodes:* Since access nodes cannot be determined in advance, the association would have to be maintained in every node, i.e., each node $k$ would need to store the binding pair $\langle w, b(w, k)) \rangle$ for each (conditional) variable $w$ accessible from it. Since the number of conditional variables accessible from a particular $k$ would not in general be constant, creation of $k$ would become a non-constant time operation, because all the binding pairs of $k$'s parent would have to be copied into $k$. If, instead of copying these pairs, node $k$ shares them with its parent node, then access to these pairs (and hence access to the binding nodes) would become a non-constant time operation, since $k$'s parent might share the binding pairs of its parent, etc. In the worst case, to determine the binding node of a variable, all ancestors of $k$ would also have to be searched, and thus this case degenerates to case (i). Thus, either determining the binding node becomes a non-constant time operation or non-constant overhead is incurred during task-creation.

iv. *The association between $v, n$ and $b(v, n)$ is stored in some combination of binding, owner, and access nodes:* Let $C$ be the set of all conditional variables in the global environment of a node $n$. In general $C$ would be a set of a non-constant size. Let $C_1$ be the set (possibly empty) of conditional variables for which case (i) is true, $C_2$ the set (possibly empty) of conditional variables for which case (ii) is true, and $C_3$ the set (possibly empty) of conditional variables for which case (iii) is true. Thus, $C = C_1 \cup C_2 \cup C_3$. Assuming $C$ is non-empty, at least one of $C_1, C_2$ or $C_3$ is non-empty too. Since $C$ is of non-constant size, at least one of $C_1, C_2$ or $C_3$ is of non-constant size, and thus the reasoning of cases (i), (ii) or (iii) can be used to prove this case. **End of Proof**.

**Lemma 2:** *Let access cells and value cells be identical, and let $C$ be the* *set of conditional variables in the global environment of a node n. If the information regarding binding node b(v, n) of a variable $v \in C$ is recorded in data-structures that are independent of the or-parallel tree, then either accessing the binding node is a non-constant-time operation or non-constant-time overhead is incurred during task-creation or task-switching.*

**Proof:** Suppose that a processor records in some data-structures the value for $b(v, n)$ for all variables $v(\in C)$ accessible from $n$. We make no assumption about these structures; they could be an array, hash table, etc. In the most general case there will be an arbitrary number of such data-structures, each of which is independent of the or-parallel tree and is global to some subset of the processors. There are two extreme cases of this general case: one where there is a single such data-structure that is common to all processors, and the other where there is one such (private) data-structure for each processor. We will show that in the former case either accessing a binding node or task-creation is a non constant-time operation, while in the latter case task-switching is a non-constant time operation. We will then show how the general case is a combination of both these extreme cases.

i. *A single global data-structure common to all the processors:* A processor that is about to create a node $n$ must ensure that $b(v, n)$ for all $v \in g(n) \cap C$ is recorded in the global data-structure. There are only two occasions when this can be done: (i) the $b(v, n)$ values are recorded *either* at the time when $n$ is created, *or* (ii) incrementally when each individual $b(v, n)$ value is needed in one of the descendent nodes of $n$. We consider each case in turn: If the $b(v, n)$ values for all variables in $g(n) \cap C$ accessible from node $n$ are recorded in the global data-structure during the creation of node $n$, then task-creation is a non-constant time operation since in general a non-constant sized subset of variables in $C \cap g(n)$ would be accessible from $n$. If the $b(v, n)$ values are recorded incrementally as they are needed, then accessing them would be a non-constant time operation: We would have to find an ancestor node $m$ of $n$ (i.e., $m \leq n$) such that $b(v, n) = b(v, m) = m$ and an entry for $b(v, m)$ exists in the global data-structure, and this cannot be done in constant-time by proposition 2 since the size of $g(n)$ and that of the global data-structure are not a constant.

ii. *A data-structure private to each processor:* Here, task-switching will be a non-constant time operation, since each time a processor switches task, it has to record in its private data-structure the entries for all variables accessible from the new nodes that are now in processor's path. The number of new nodes accessible to a processor on a task-switch, as well as the number of variables accessible from these newly accessible nodes, is dependent on the depth of the node corresponding to the new task, and thus is not constant. Hence task-switching is a non-constant time operation.

The general case reduces to a combination of cases (i) and (ii). Let the set of processors $P$ be divided into $s$ disjoint subsets $P_1, \ldots, P_s$, where each

$P_i$ has its own data-structure for recording $b(v,n)$. By reasoning similar to that in case (i), either task-creation or accessing the binding node will be a non-constant time operation. Similar to case (ii), if a processor $p \in P_i$ switched to another branch that has been created by processors $q \notin P_i$, then task-switching will be a non-constant time operation, since information about the new binding nodes in $p$'s path will have to be recorded in the data-structure for set of processors $P_i$.

It is also possible to split the data-structure into a part that is global and a part that is private to each processor. Since $C$ is of non-constant size, either the set of variables recorded in the global data-structure or the set of variables recorded in the private data-structures would be of non-constant size. Hence again by case i and ii, one of task-creation, binding node access, or task-switching will be non-constant time. **End of Proof**.

**Theorem 1:** *Let access cells and value cells be identical and let $C$ be the set of conditional variables in the global environment of a node $n$. The binding node $b(v,n)$ of a variable $v \in C$ cannot be determined by a processor at node $n$ in constant time without incurring non-constant overhead during task-creation or task-switching.*

**Proof:** The function $b$ cannot be described algorithmically since doing so would allow us to predict the binding node of a given variable in a particular environment even before that variable gets bound. This means that to evaluate the function $b(v,n)$ the association between a variable $v$ and its binding node in environment $g(n)$ has to be explicitly maintained. There are two ways to maintain this association: either in the or-parallel tree (in some combination of owner, binding and access nodes) or independent of the or-parallel tree (in data-structures independent of the tree).

Let $C_{in}$ be the set of variables for which the association information is stored in the or-parallel tree and let $C_{out}$ be the set of variables for which the association information is stored independent of the or-parallel tree. Since $C$ is a set of non-constant size, and $C = C_{in} \cup C_{out}$, at least one of $C_{in}$ or $C_{out}$ is also of non-constant size. If $C_{in}$ is of non-constant size, by lemma 1 either binding nodes cannot be accessed in constant time or non-constant overhead is incurred during task-creation. If $C_{out}$ is of non-constant size, by lemma 2 either binding nodes cannot be accessed in constant time or non-constant overhead is incurred during task-creation or task-switching. Thus, in either case non-constant overhead is incurred. **End of Proof.**

**Corollary 1**: *Let access cells and value cells be identical and let $C$ be the set of conditional variables in the global environment of a node $n$. It is not possible for a processor at an access node $n$ to access $val(v,n)$ (for $v \in C$) in $vc(v,n)$ in constant time without incurring non-constant overhead during task-creation or task-switching.*

**Proof:** Since the relation between functions $b$ and $val$ is one-one, finding $val(v,n)$ in constant time without incurring non-constant time task-creation overheads means $b(v, n)$ can also be found in constant time without incurring non-constant-time task-creation overheads†. Clearly this is not possible according to theorem 1. **End of Proof**.

**Lemma 3**: *If access cells are distinct from value cells, task-switching is a non-constant time operation.*

**Proof:** Let $C$ be a set of conditional variables. Its size is therefore of not a constant. During task-switch by an arbitrary processor $p$, the bindings applicable to conditional variables in $C$ may change due to a change of current global environment of $p$. Hence, $p$ should ensure that the access cells are correctly established for all such conditional variables. Suppose $p$ switches from node $k_1$ to node $k_2$. Let $k$ be the least common ancestor node of $k_1$ and $k_2$. Let $K$ be the set of all nodes in the path from $k_1$ to $k_2$ via $k$. Let $C' = \{v \mid v \in C \ \wedge \ b(v, k_1) \in K\} \cup \{v \mid v \in C \ \wedge \ b(v, k_2) \in K\}$. A variable $w$ is *affected* by a task switch if $w \in C'$.

The set $C$ can be partitioned into two subsets $C_1$ and $C_2$ such that:

i. $C_1 = \{v \mid ac(v, p, k_1) = ac(v, p, k_2)\}$. That is, for conditional variables in $C_1$, the processor uses the same physical memory location for the access cell irrespective of the environment it is in (i.e., it does not allocate a new one).

ii. $C_2 = \{v \mid ac(v, p, k_1) \neq ac(v, p, k_2)\}$. That is, a new physical memory location is allocated in the environment at $k_2$ for the access cell for each conditional variable in set $C_2$.

Since $C$ is of non-constant size, at least one of $C_1$ or $C_2$ is also of non-constant size. In case (i) the access cells have to be *updated* for each conditional variable in $C_1$ (at the very least for *affected* conditional variables in $C' \cap C_1$); similarly, in case (ii) the access cells have to be *created and updated* for each conditional variable in $C_2$ (at the very least for *affected* conditional variables in $C' \cap C_2$). Since either $C_1$ or $C_2$ is of non-constant size and in general $C'$ would be of non-constant size, task-switching will be a non-constant time operation. **End of Proof**.

**Theorem 2**: *There is no implementation method for a bounded number of processors which satisfies all three criteria.*

**Proof** : Let $C$ be the (non-constant sized) set of conditional variables in the or-parallel tree at any given moment. Partition the set $C$ into two disjoint sets $C_a$ and $C_b$, where $C_a$ is the subset of $C$ for which access cells and value cells are distinct, and $C_b$ the subset of $C$ for which these cells are identical. Clearly, either $C_a$ or $C_b$ must have a non-constant number of variables. By lemma 3, if $C_a$ were of non-constant size, task-switching will

† For example, when a binding is generated we can record the address of its binding node with it in the value cell. Then, access to a variable's binding in the value cell means automatic access to corresponding binding node at no extra cost.

be a non-constant time operation. Hence, consider the case that $C_b$ is of non-constant size. $C_b$ corresponds to the set where access cells and value cells are identical. For such variables, accessing variables amounts to accessing the values in value-cells. By corollary 1, if $C_b$ is of non-constant size then the values of variables in $C_b$ cannot be accessed in constant time without incurring non-constant overhead during task-creation **End of Proof.**

## 4. Or-Parallel Execution of Logic Programs

We now apply the results of the previous section to the or-parallel execution of logic programs. As noted earlier, nondeterminism in logic programs arises because multiple clause heads may unify with some procedure call (or goal). Since unification is essentially "two-way pattern-matching," unbound variables in the calling environment may be bound through unification. This aspect of unification coupled with nondeterminism effectively makes possible multiple bindings for some variable in a local environment.

As noted in lemmas 1 and 2, and theorem 1, the main task in an or-parallel execution model, is to determine the binding node during variable access. It can now be seen why it suffices to restrict attention to single-assignment languages for the purpose of deriving our results. In imperative languages—where a variable can be bound more than once in any nondeterministic path—determining the applicable binding node is further complicated since there can be several binding nodes for a variable in a given global environment. For a given access node of a variable, the applicable binding node would be the one that is closest to it on the path from the access node to the root of the or-parallel tree.

Different methods differ in how they identify the binding node, but a central aim of all methods is to identify the binding node efficiently, so that the value cells can be accessed in constant time during variable access or for environment-update during task-switching. For example, some methods simplify the operation of finding the binding node of a variable by allocating the value cells in the binding node itself (e.g. binding arrays method [W84, W87a], version vectors method [H87], hashing windows method [B84]). Some schemes assign a unique id to the binding node and record this id with each binding. For example, in the time stamping method [T89], the variable bindings are organized in a fashion that once the binding node is known the corresponding binding can be efficiently accessed.

In most methods proposed in the literature on or-parallel execution of logic programs, either set $C_a$ or $C_b$ (described in theorem 2) is an empty set. However, the proof above is more general and subsumes even those methods which may have both $C_a$ and $C_b$ non-empty. We consider a few methods and see why they satisfy theorem 2; in the process, we will see an illustration of the proof of the lemmas and the theorem.

*Binding Arrays Method*

In the binding arrays method [W84, W87], value cells and access cells are allocated memory space separately. Access cells are allocated space in the binding array private to the processor while value cells are allocated space in the binding lists in the binding node. Since the slot used in the access cell for a particular conditional variable is fixed irrespective of which global environment it is in, its content has to be updated on a task-switch. This results in task-switching time becoming non-constant in accordance with lemma 3. Thus, theorem 2 holds for the binding arrays method. (By a similar reasoning, we can see that the version vectors [H87] method also has non-constant task-switching time and thus satisfies theorem 2.)

For the binding arrays method, the value of sets $C_a$ and $C_b$ described in theorem 2 are: $C_a = C_v$ and $C_b = \phi$, where $C_v$ is the set of all conditional variables in the or-parallel tree at a given time. Also, the value of sets $C_1$ and $C_2$ described in lemma 3 are: $C_1 = C_v$ and $C_2 = \phi$. Note that lemma 3 is parameterized by $p$, $k_1, k_2$, $C_1, C_2$. Thus, a variation of the standard binding arrays method can be conceived where for certain pair of nodes $k_1$ and $k_2$ a processor decides to update its binding array in place ($C_1 \neq \phi$), whereas for certain other pair of nodes $k_3$ and $k_4$ it leaves its old binding array stationed at $k_3$ and constructs a new one for $k_4$ ($C_2 \neq \phi$). This corresponds to the case where there are more binding arrays than the number of processors and the ownership of a binding array can change from time to time. In this case, in order to optimize task-switching time, a processor may build a new binding array for a branch when it sees that the or-parallel tree has become too big. Obviously this modified binding array method satisfies lemma 3 and thus theorem 2.

Note that it is not necessary for the binding arrays and versions vectors method to record the value cells in the binding nodes. Value cells can be recorded using a different scheme, say, for example, as in the time stamping method [T89]. This will give us a variation of the binding arrays and the versions vector method. More variations of the binding arrays method (Manchester-Argonne and Argonne-SRI models), which also satisfy theorem 2, are given in [W87b].

*Muse, Kabu-Wake, Delphi, and Randomized Parallel Backtracking*

In the Muse system [AK90] the binding environment of each processor is distinct, i.e. it is not shared with others. When a processor switches to another node in the or-parallel tree, it constructs the whole stack from that node to the root node in its memory. The copy of the stack can be regarded as the private data-structure (alluded to in lemma 2) for accessing the variable bindings. As a result, by case ii of lemma 2 and corollary 1, task-switching becomes a non-constant time operation. Task-switching is also a non-constant time operation for the Kabu-Wake method [K86], the Delphi method [CA88] and the Randomized Parallel Backtracking method [JA88, L89]. In the latter two cases, on a task-switch, a processor constructs the data-structure mentioned in lemma 2 by *recomputing* all nodes from the root of the or-parallel tree to the node where the task-site is located (thus the stack is reconstructed rather than copied). Although this recomputation results in less interaction between processors, overall this strategy can incur more overhead because, in general, recomputing would involve more work than copying stacks. To avoid repeated recomputing, the Delphi method uses an optimization that is based on comparing so-called *oracles*, where oracles are essentially node identifiers indicating the path taken to reach the node from the root. Since the oracles themselves are of size proportional to the depth of the nodes in the or-parallel tree, comparing them is a non-constant time operation; hence, even with this optimization, task-switching still remains a non-constant time operation.

*Directory Tree Method*

In the directory tree method [HC83], the binding environment is represented using directories and contexts. Each node has its own directory which contains a number of contexts, one for each of its ancestor nodes. A context corresponds to a clause and has space for variables in the clause. When a node branches it has to create new directories (with copies of the node's contexts) for each of its children nodes.

In this method value cells and access cells are identical and are allocated space in access nodes. Each node has its own copy of all the contexts. Hence, by lemma 1 and corollary 1, task creation is a non-constant time operation. Thus, theorem 2 holds for the directory tree method. Note that for this method the value of sets $C_a$ and $C_b$ described in theorem 2 are: $C_a = \phi$ and $C_b = C_v$.

For similar reasons, theorem 2 holds for the environment closing method [C87] and the environment representation scheme used in ROPM [RK89]. They both have non-constant task-creation time.

*Hashing Window Method*

In the hashing windows method [B84], the access cell and value cell are identical and allocated space in the binding node. Hence, by lemma 1 and corollary 1, variable access is a non-constant time operation. (To insure fast access to variable bindings within a binding node, a value cell is allocated a cell in a *hash window* whose address is obtained by hashing on the variable address). Thus, the hashing window method also satisfies theorem 2.

For similar reasons, theorem 2 holds for the Argonne Model [DLO87, W87b]. However, in the Argonne Model, the hash window is of constant size (= 32) and the contents of the hash window of a node are shared with its children, which results in each bucket in the hash-window having a large number of variables mapped to it (which are stored as a list). Thus, the non-constant cost of traversing the tree upwards (towards the root) and finding the binding node is reflected as the cost of locating the value cell in the bucket. The further up the tree the binding node of $v$ is, the more towards the end of the list in the bucket would be the value cell of $v$.

Theorem 2 also holds for the variation of the hashing windows method adopted for representing environments in the PEPSys model [WR87]. In this model, when a conditional variable is accessed from a node, its binding is copied into the hashing window of the access node, so that future accesses to this variable from the descendent nodes have to traverse fewer nodes. Thus, this scheme also satisfies theorem 2 by virtue of lemma 1 (case iii) and corollary 1.

*Time Stamping Method*

The time stamping method [T89] records the id of the processor and a unique time stamp with each variable binding. The ⟨pid, time-stamp⟩ pair can be regarded as the id of the binding node which created the binding. The id is later used in accessing the bindings. The value cells (which are identical to access cells in this method) are organized in a data-structure independent of the or-parallel tree. Thus, in accordance with lemma 2 and corollary 1, variable access is a non-constant time operation. Actually, in the time stamping method, task-switching too is a non-constant time operation because the *ancestor stack* (which can be viewed as representation of the association between variables and their binding nodes) has to be constructed during task-switching from information recorded in the nodes. Thus, due to lemma 2 and corollary 1 task-switching too is a non constant time operation in this method.

*And-Or parallel Methods*

Theorem 2 also holds for the or-parallel component of and-or parallel systems. For example, consider the following three methods for incorporating independent and-parallelism and or-parallelism in a single framework: (i) the Reduce-Or Parallel Model [RK87] (ii) the PEPSys model [WR87] and (iii) the authors' own and-or parallel model [GJ89, G91]. Note that the environment representation scheme used in the three methods sacrifice the task-creation, variable-access and task-switching criteria respectively.

## 5. Classification of Or-Parallel Models of Logic Programs

The fact that all three criteria cannot be simultaneously satisfied is partly the reason why so many methods have been proposed in the literature for or-parallel execution of logic programs. Based on which criteria one chooses to satisfy one has a different execution model for or-parallelism. One can also use this as a basis for classifying the various or-parallel execution models. Since there are three criteria to be satisfied, in principle there are $2^3 = 8$ classes of methods according to this classification (see Figure 1). We have proved that no methods exists in Class 1, which corresponds to methods providing constant-time for all three operations, i.e. task creation, variable access and task switching. Class 8 methods are uninteresting since all three operations are non-constant time. None of the methods proposed in the literature fall in class 7, which corresponds to non-constant task creation as well as non-constant task-switching time. Although it is theoretically possible to create an or-parallel method which would fall in this class, this is unlikely in practice, because, if task creation is non-constant time, it is very likely that the environment has already been replicated during node-creation;

thus there is no need for any environment updating during task-switch since each branch in the or-parallel tree has its own private environment.

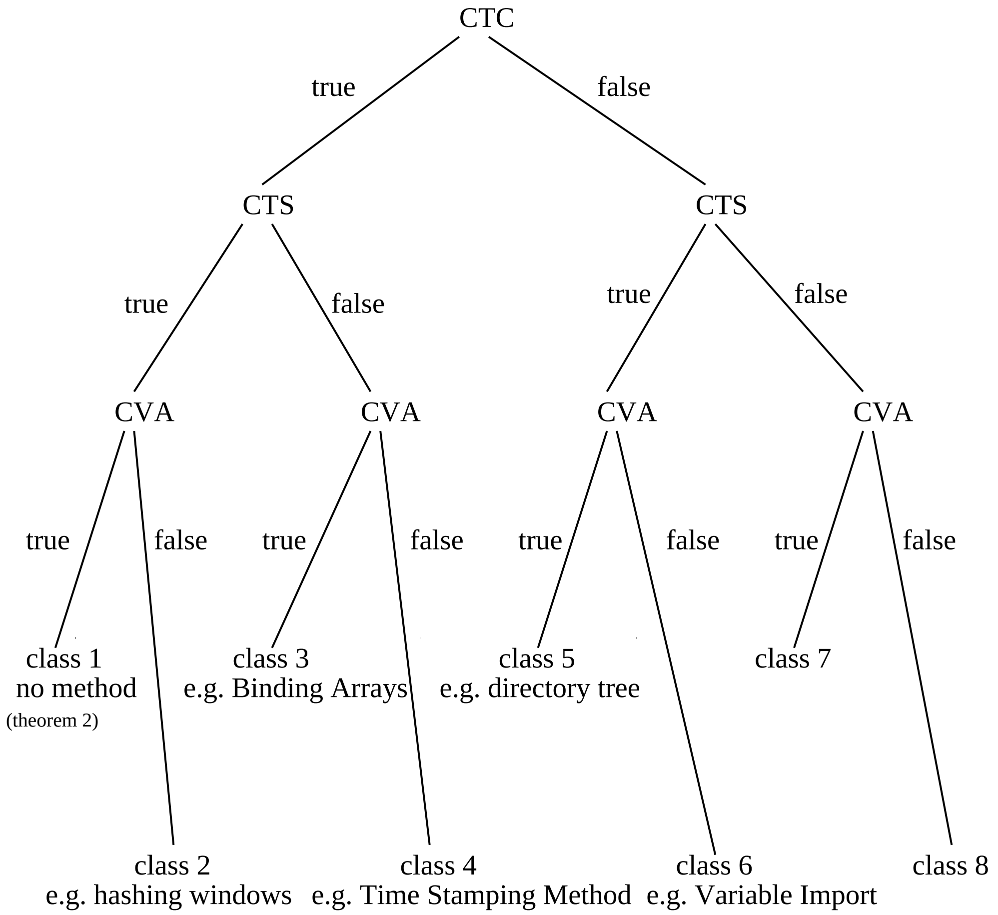


Figure 1: Classification of Or-parallel Methods

Theoretically, there is no reason why any method should sacrifice more than one criterion, although some methods do, e.g., the variable importation method [L84] and the time stamping method [T89]. This is because they distribute the cost of multiple environment representation over two operations instead of one. The overall performance may eventually remain unchanged or may even improve. For example, in the time stamping method [T89], in the worst case variable bindings would have to be accessed in the global data-structure by searching the entire data-structure. However, by keeping the ancestor stack, which results in non-constant time task-switching, only a fraction of the global data-structure has to be accessed. Thus, although two criteria may be sacrificed by a particular method, it does not necessarily mean that it is inferior to a method which sacrifices only one criterion.

| Method | CTC | CVA | CTS |
|---|---|---|---|
| Version Vectors [H87] | $\surd$ | $\surd$ | |
| Binding Arrays [W84, W87a] | $\surd$ | $\surd$ | |
| Argonne-SRI Model [W87b] | $\surd$ | $\surd$ | |
| Manchester-Argonne [W87b] | $\surd$ | $\surd$ | |
| Time-Stamping [T89] | $\surd$ | | |
| Hashing Windows [B84] | $\surd$ | | $\surd$ |
| Naive Model [W87b] | $\surd$ | | $\surd$ |
| Argonne Model [DLO87] | $\surd$ | | $\surd$ |
| Directory Tree [HC83] | | $\surd$ | $\surd$ |
| Environment Closing [C87] | | $\surd$ | $\surd$ |
| Variable Import [L84] | | | $\surd$ |
| Kabu-Wake [K86] | $\surd$ | $\surd$ | |
| BC-Machine [A88] | $\surd$ | $\surd$ | |
| Muse [AK90] | $\surd$ | $\surd$ | |
| Abstract Model [W87b] | | $\surd$ | $\surd$ |
| Delphi [CA88] | $\surd$ | $\surd$ | |
| Randomized Method [JA88] | $\surd$ | $\surd$ | |

CTC = Constant-time Task Creation; CVA = Constant-time Variable Access; CTS = Constant-time Task Switch

Table 1: Criteria Satisfied by Various Methods

In table 1 we give a table showing which particular criteria are satisfied by different methods. A $\surd$ in a column indicates that the criterion corresponding to that column is satisfied by the method. Methods falling in the same class can be directly compared with each other, and the best one among them determined. It makes more sense to compare methods within the same class, rather than methods belonging to different classes. For example, both the directory tree method and the environment closing method duplicate the environment during task-creation. However, environment closing duplicates a smaller fraction of the environment since it does not duplicate variables which are already bound, while the directory tree method does. Thus, although two or-parallel methods might be theoretically equivalent, in practice one may be more efficient than the other. Similarly, the binding arrays method is superior to versions vectors methods as far as executions costs are concerned (the reasons are given in [W87a]).

For shared-memory multiprocessors, methods that sacrifice constant time task-switching tend to be superior to those that sacrifice constant-time task-creation and variable-access. This is because the number of task-creation and variable-access operations is program dependent while the number of task-switching operations is dependent on the scheduling policy employed by the run-time system. Thus, the cost of task-switching can be carefully controlled by the implementor (by optimizing granularity, for instance), while this is not possible for task-creation and variable access. This has been demonstrated by the Aurora and the Muse system [LWH90] which employ sophisticated schedulers [AK90, CS89, S89, B90] to minimize the cost of task-switching.

For distributed memory systems, methods with non-constant time task-creation are likely to perform better. This is because the relevant part of the environment that is needed for carrying on parallel execution in another processor is made available through explicit copying during task creation in that processor (for example, in the Environment Closing method [C87] and the Directory Tree method [HC83]). All accesses to variables in the other processor use the copy that is available locally. However, the advent of *distributed shared memory systems* [LH89, WH88] may make methods sacrificing constant-time task-switching as competitive on distributed systems as those sacrificing constant-time task-creation. And, as mentioned earlier, since performance of the methods in the former class is less dependent on the behavior of the logic programs, they might even out-perform the methods in the latter class on distributed memory systems.

## 6. Conclusions

We have presented three criteria that an ideal or-parallel system should satisfy: constant-time access to all variables, constant-time task creation, and constant-time task-switching. The significance of these criteria is that they correspond naturally to three fundamental operations of any nondeterministic programming language: procedure call (which involves environment creation), assignments and parameter transmission (which involves variable access and binding), and resumption after success or failure (which involves task switching). We showed that, given a finite number of processors, there is no or-parallel execution method, based upon the or-parallel tree method, that can satisfy all three criteria simultaneously. While we believe that our result also applies to or-parallel models that are not based upon the or-parallel tree, we leave the formal proof of this conjecture as an open question. Notwithstanding the restriction to models based upon the or-parallel tree, we believe our result is of significance, since all the well-known or-parallel implementations of logic programs are based upon this approach.

It was sufficient to derive our results for the class of nondeterministic, *single-assignment* languages, since it is no easier to satisfy the three criteria when destructive assignments are present. Although our results were first formulated for logic programs [GJ90], by reformulating them for general nondeterministic languages, we hope they are accessible to researchers interested in parallelizing applications of symbolic computation and reasoning that exhibit substantial or-parallelism, such as symbolic integration, program synthesis, natural language analysis, theorem-proving, game-tree searching, etc. While the technology of parallel execution of logic programs has advanced to the point that there are several practical parallel Prolog implementations [LWH90, AK90, T92], not much has been done to organize the subject in a systematic way. This paper provides such a framework, and could serve as a valuable reference for implementors of logic languages. Compared with existing work, our results provide greater insight into the nature of or-parallel execution models and facilitate more meaningful comparisons between them.

**Acknowledgements:** Thanks to David H.D. Warren and Feliks Kluźniak for their comments and suggestions on an earlier version of this paper. Thanks are also due to the anonmyous referees whose comments spurred us to generalize our earlier result and broaden the scope of our paper. This work was supported in part by NSF grant DCR 8603609. Gopal Gupta was also supported by U.K. Science and Engineering Research Council grant GR/F 27420, and performed some of this work at the University of Bristol.

**References**

[A88] K. Ali, "Or-parallel execution of Prolog on BC-machine," In *Proceedings of 5th International Conference and Symposium on Logic Prog.*, Seattle, 1988.

[AK90] K. Ali, R. Karlsson, "The Muse Or-parallel Prolog Model and its performance," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press.

[B84] P. Borgwardt, "Parallel Prolog using stack segments on shared memory multiprocessors," In *Proc. 1984 Int. Symp. on Logic Prog.,* Atlantic City, NJ, 1984, pp. 2-11.

[B90] A. J. Beaumont, et. al., "The Bristol Scheduler for Aurora," Technical Report TR-90-04, University of Bristol, 1990.

[C87] J. S. Conery, "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors," *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco, pp. 457-467

[CA88] W. F. Clocksin and H. Alshawi, "A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors," In *New Generation Computing, 5(1988)*, 361-376.

[HC83] B. Hausman and A. Ciepielewski, "A Formal Model for Or-parallel Execution of Logic Programs," In *IFIP 83, North Holland, P.C. Mason (ed.).*

[HC86] B. Hausman and A. Ciepielewski, "Performance Evaluation of an Or-parallel Execution Model for Logic Programs," In *Symposium on Logic Prog.*, 1986, pp. 246-255.

[CS89] A. Calderwood and P. Szeredi, "The Manchester Scheduler," *International Conference on Logic Programming '89.*

[DLO87] T. Disz, E. Lusk, and R. Overbeek, "Experiments with OR-parallel Logic Programs," In *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco.

[G91] G. Gupta, "Parallel Execution of Logic Programs on Shared Memory Multiprocessors," Ph.D. Thesis, Univ. of N. Carolina at Chapel Hill, 1991.

[GJ89] G. Gupta and B. Jayaraman, "Compiled And-Or Parallel Execution of Logic Programs," In *Proceedings of the North American Conference on Logic Programming '89*, MIT Press, pp. 332-349.

[H87] B. Hausman, et. al., "Or-Parallel Prolog Made Efficient on Shared Memory Multiprocessors," in *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco, CA.

[HG92] M. Hermenegildo, K. J. Green, "&-Prolog System: Exploiting Independent And-parallelism," In *New Generation Computing,* **9**(3,4), pp 233-256.

[JA88] V. Janakiram, D. Agarwal and R. Malhotra, *et. al.*, "A Randomized Parallel Backtracking Algorithm," *In IEEE Transactions on Computers,* 37(12), Dec. '88.

[K86] K. Kumon *et al*, "Kabu-wake: A New Parallel Method and Its Evaluation," Proceedings of CompCon '86, 1986, pp. 168-172.

[L84] G. Lindstrom, "Or-Parallelism on Applicative Architectures," in *2nd International Logic Prog. Conf.,"* Uppsala, Sweden, 1984.

[L89] Z. Lin, "Expected Performance of the Randomized Parallel Backtracking Method," In *Proceedings of the North American Conference on Logic Programming '89*, MIT Press, pp. 677-696.

[LH89] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," In *ACM Transactions on Computer Systems*, **7**(4), November 1989, pp. 321-359.

[LWH90] E. Lusk, S. Haridi, D.H.D. Warren et. al. "The Aurora Or-Prolog System", In *New Generation Computing*, Vol. 7, No. 2,3, 1990 pp. 243-273.

[M79] D. Maier, "An Efficient Method for Storing Ancestor Information in Trees," In *SIAM J. Computing*, Vol 8, No. 4, Nov. 1979.

[RK89] B. Ramkumar and L. V. Kale, "Compiled Execution of the REDUCE-OR Process Model," In *Proceedings fo the North American Conference on Logic Programming '89*, MIT Press, pp. 313-331.

[S89] P. Szeredi, "Performance Analysis of the Aurora Or-Parallel Prolog System," In *Proceedings fo the North American Conference on Logic Programming '89*, MIT Press, pp. 713-734.

[SV88] B. Scheiber and U. Vishkin, "On Finding Lowest Common Ancestors: Simplification and Parallelization," In *SIAM Journal of Computing*, Vol. 17, No. 6, Dec. 1988.

[T89] P. Tinker, "Performance of an OR-parallel Logic Programming system," In *Int. J. of Parallel Programming*, **17**(1), pp. 59-92.

[T92] H. Tanaka, ed., Proceedings of the International Conference on Fifth Generation Computer Systems, 1992.

[WH88] D.H.D. Warren and S. Haridi, "The data diffusion machine – a shared virtual memory architecture for parallel execution of logic programs," In *Proceedings of FGCS '88*, pp. 943-952.

[W87a] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog – Abstract Design and Implementation Issues," *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco.

[W87b] D. H. D. Warren, "Or-Parallel Execution Models of Prolog," TAPSOFT '87, Springer Verlag, LNCS 250.

[W84] D. S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategies," In *The 1984 Int. Symp. on Logic Prog.,* Atlantic City, pp. 198-202.

[WR87] H. Westphal, P. Robert, J. Chassin and J. Syre, "The PEPSys Model: Combining Backtracking, AND- and OR-parallelism," In *1987 IEEE Int'l Symp. in Logic Prog.*, San Francisco, pp. 436-448.