Parallel Execution of Prolog Programs: A Survey

(Preliminary Version)

Gopal Gupta

Department of Computer Science
Box 30001/CS
New Mexico State University
Las Cruces, NM 88003, USA
gupta@nmsu.edu

Khayri A. M. Ali and Mats Carlsson

Swedish Institute of Computer Science, SICS
Box 1263, S-164 28
Kista, Sweden
{khayri,matsc}@sics.se

Manuel V. Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid
28660-Boadilla del Monte, Madrid, Spain
herme@fi.upm.es

0. Introduction

An important property of logic programming languages is that they are single assignment languages. Unlike conventional programming languages they disallow destructive assignment and explicit control information. Not only does this allow cleaner (declarative) semantics for logic programs, and hence a better understanding of them by their users, its also permits an evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can be executed in any order without affecting the meaning of the program. In particular, these operations can be performed by the evaluator *in parallel*.

Parallel execution of the different operations in a logic program results in *implicit* exploitation of parallelism, i.e., no input is required from the user to exploit parallelism, rather, parallelism is exploited by the evaluator at run-time itself. It should be pointed out that there are many proposals for extending a logic programming language with explicit constructs for concurrency. They can be largely put into three categories:

1. those that add explicit message passing primitives to Prolog, *e.g.*, Delta Prolog [9] and CS-prolog [56]. Multiple Prolog processes are run in parallel that communicate with each other via messages.

2. those that add *blackboard* primitives to Prolog, *e.g.*, Shared Prolog [36]. These primitives are used by multiple Prolog processes running in parallel to communicate with each other via the blackboard.

3. those based on guards, committed choice, and data-flow synchronization, *e.g.*, Parlog, GHC, and Concurrent Prolog [39, 136, 128].

We will not consider such approaches in this survey since our focus is primarily on approaches that exploit parallelism from logic programs implicitly. Clearly, the approach to exploit parallelism implicitly has significant advantages over the "explicitly parallel programming" approach, because the burden on the programmer is greatly reduced. Four principal kinds of (implicitly exploitable) parallelism can be identified in logic programs:

1. *Unification parallelism* arises when arguments of a goal are unified with those of a clause head with the same name and arity. The different argument terms can be unified in parallel as can the different subterms in a term [14]. Unification parallelism is very fine-grained and is best exploited by building specialized processors with multiple unification units [118]. Unification parallelism has not been the major focus of research in parallel logic programming, so we will not consider it any further.

2. *Or-parallelism* arises when more than one rule defines some relation and a procedure call unifies with more than one rule head—the corresponding bodies can then be executed in parallel with each other, giving rise to or-parallelism. Or-parallelism is thus a way of efficiently searching for solutions to the query, by exploring alternative solutions in parallel.

3. *Independent and-parallelism* arises when more than one goal is present in the query or in the body of a procedure, and the runtime bindings for the variables in these goals are such that two or more goals are *independent* of one another, i.e., their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables. Parallel execution of such goals gives rise to (independent) and-parallelism.

4. *Dependent and-parallelism* arises when two or more goals in the body of a procedure have a common variable and are executed in parallel. Dependent and-parallelism can be exploited in two ways: (i) The two goals can be executed independently until one of them accesses/binds the common variable. Note that it is also possible to continue executing the two goals independently in parallel (i.e., executing each without regard to the other goal) even after the common variable has been accessed; in such a case, after the two goals finish, the bindings produced by each will have to be checked for compatibility (this compatibility check at the end is called *back unification*). (ii) Once the common variable is accessed by one of the goals, if it is bound to a structure, or *stream* (the goal generating this binding is called the *producer*), and this structure is read as an input argument of the other goal (called the *consumer*) then parallelism can be further exploited by having the consumer goal compute with one element of the stream while the producer goal is computing the next element. Case (i) is very similar to independent and-parallelism. Case (ii) is sometimes also referred to as *stream-parallelism* and is useful for speeding up producer-consumer interactions (e.g., those found in system programs). Note that stream-parallelism can also introduce a form of co-routining. Stream-parallelism forms the basis for Committed Choice Languages (e.g., Parlog [39], GHC [136], and Concurrent Prolog [128]).

Loosely speaking, the four different kinds of parallelism are largely orthogonal to each other, i.e., one can be exploited without affecting the exploitation of the other. Thus, it is feasible to exploit all of them simultaneously. However, no *efficient* parallel system has been built yet that achieves this. A system that exploits maximal parallelism from logic programs while achieving the best possible performance is the ultimate goal of researchers in parallel logic programming.

It should be noted that research on parallel logic programming started almost with the inception of logic programming itself. The earliest published work is the Ph.D. thesis of Pollard [114]. Ever since then there has been a healthy interest in parallel logic programming, as is obvious from the number of papers that have been published in conferences and journals devoted to logic programming and parallel processing, and the number of advanced tutorials and workshops organized on this topic in these conferences. Although parallel logic programming systems are still being researched, very mature and robust systems such as MUSE [4], Aurora [99], &-Prolog [110], Andorra-I [120], and various implementations of committed choice languages [55, 46], are available today. Parallel logic programming got a big boost when it was selected by the Japanese for their 5th generation project as a vehicle for obtaining fast and intelligent systems based on parallel processing [103] (one should, however, note that the Fifth Generation Project focused on exploiting parallelism through committed choice logic languages rather than through traditional logic languages such as Prolog).

In the rest of the paper, we discuss or-parallelism, independent and-parallelism and dependent and-parallelism in greater detail, describing the problems that arise in exploiting them. We describe the various solutions that have been proposed for overcoming these problems, followed by description of actual parallel logic programming systems that have been built. We discuss the efficiency issues in parallel logic programming, and current and future research in this area. We assume that the reader is familiar with sequential implementation techniques for logic programming languages. An excellent description to these can be found in [100, 2].

It should be pointed out that the literature on parallel logic programming is vast, indicative of the richness of the field, and therefore it is not possible to cover every aspect of it in a paper of this size. Therefore, in this paper our approach has been to summarize the main ideas of this area, describing in greater detail topics to which researchers have devoted more attention. More details can be found in the references that have been provided.

1. Data parallelism vs. Control parallelism

Parallelism present in logic programs can be exploited both in SIMD architectures as well as in MIMD architectures. However, as is true for general programs, the class of logic programs that can benefit from SIMD architectures is smaller compared to the class of logic programs that can benefit from MIMD architectures[1]. Majority of the research has focussed on exploiting parallelism only on MIMD architectures, viewing or-parallelism and and-parallelism as forms of control-parallelism. An alternative view has been to treat or- and and-parallelism as data-parallelism and exploit them on an SIMD architecture. We call such parallelism data-or parallelism and data-and parallelism respectively.

Data Or-parallelism

In a data-or parallel system, exemplified by the MultiLog system [119], or-parallelism of highly regular nature is exploited on an SIMD architecture. There is one control thread but multiple environments. Data-or parallelism as exploited in MultiLog is useful in many generate-and-test kind of applications, where the generator binds a variable to different values taken from a set. Consider the following program:

```
member(X, [X|T]).
member(X, [Y|T]) :- member(X, T).
?- member(Z, [1,2,..,100]), process(Z).
```

The `member` goal will be identified as the generator in the MultiLog system. For such a goal, a subcomputation is begun, and all solutions are collected and turned into a disjunction of substitutions for

[1]Note that any program that can be speeded-up on an SIMD architecture can also be speeded-up on an MIMD architecture but not *vice versa*.

variable Z. The process goal is then executed in data-parallel for each binding received by Z. Note that execution of the various process goals differs only in the value of variable Z. Therefore, only one control thread is needed which operates on data that is different on different processors. For a (restricted) set of applications that conform to a certain pattern, a data-or parallel system such as MultiLog can get good speed-ups.

Data And-parallelism

The idea of data-parallel execution can also be applied to and-parallel goals: clauses that contain recursive calls can be unfolded and the resulting goals executed in data-parallel. This approach has been taken in Reform Prolog [22]. Consider the following program:

```
rev([], []).
rev[H|T],R) :- rev(T,Tr), append(Tr,[H],R).
?- rev[1, 2, 3], Z).
```

Unfolding this goal we obtain:

```
rev([],Tr"), append(Tr",[3],Tr'), append(Tr',[2],Tr), append(Tr,[1],Z).
```

Note that the three append goals are identical except for the data-values and can be executed in data-parallel (i.e., with a single thread of control and multiple data values). Thus, the answer to the above query can be executed in two data-parallel steps. Data-and parallelism, as exploited in Reform Prolog, also will get good speed-ups in a (restricted) set of applications that conform to a certain pattern.

Exploitation of data-and parallelism explicitly through *bounded quantification* has also been proposed [15]. In the rest of the paper we will only consider control-parallelism exploited on MIMD multiprocessors.

2. Or-parallelism

Or-parallelism arises when a subgoal can unify with heads of more than one clause. In such a case the bodies of these clauses can be executed in parallel with each other giving rise to or-parallelism. To take an example, consider the program for solving the 8-queens problem:

```
queens(Qs) :- queens(Qs, [], [1,2,3,4,5,6,7,8]).

queens([],_,[]).
queens([X|Xs], Placed, Values):-
            delete(X, Values, New_values),
            noattack(X, Placed),
            queens(Xs,[X|Placed],New_values).

delete(X, [X|Xs], Xs).
delete(X, [Y|Ys], [Y|Zs]) :- delete(X, Ys, Zs).

noattack(X, Xs) :- noattack(X, Xs, 1).
noattack(_, [], _).
noattack(X, [Y|Ys], Nb) :-
     X =\= Y-Nb,
     X =\= Y+Nb,
```

```
        Nb1 is Nb + 1,
        noattack(X,Ys,Nb1).
```

and the query:

```
?- queens(Z).
```

The call to `delete/3` in second clause of `queens/3`, acts as a generator of values and creates a choicepoint. The multiple alternatives in this choicepoint can be executed in parallel. The predicate `delete/3` will be called again in the recursive invocations of `queens/3`, creating yet more choicepoints and yet more untried alternatives that can be picked up by processors for or-parallel processing.

An alternative way to visualize or-parallelism is through the *or-parallel search tree*. Informally, an or-parallel search tree (or simply an or-parallel tree or a search tree) for a query Q and logic program LP is a tree of nodes, each with an associated *goal-list*, such that:

1. the root node of the tree has Q as its associated goal-list;

2. each non-root node n is created as a result of successful unification of the first goal in (the goal-list of) n's parent node with the head of a clause in LP, H `:-` $B_1, B_2, \ldots, B_n$. The goal-list of node n is $(B_1, B_2, \ldots, B_n, L_2, \ldots, L_m)\theta$, if the goal-list of the parent of n is $L_1, L_2, \ldots, L_m$, where $\theta = mgu(H, L_1)$.

The or-parallel tree for a (simple) example program is shown in Figure 1. Note that each node of the or-parallel tree contains the variables found in its corresponding clause, i.e., it holds that clause's *environment*. During sequential execution the or-parallel tree is searched in a depth first manner. However, if more than one processors are available, then multiple branches of the tree can be searched simultaneously giving rise to or-parallelism. Note that we haven't addressed how the multiple bindings that different processor executing in parallel may impart to a variable are maintained. This is discussed in the next subsection.

Or-parallelism manifests itself in a number of applications [92]. It arises while exercising rules of an expert systems where multiple rules can be fired simultaneously to achieve a goal. It also arises in some applications that involve natural language sentence parsing. In such applications the various grammar rules can be applied in or-parallel to arrive at a parse for a sentence. If the sentence is ambiguous then the multiple parses would be found in parallel. Or-parallelism also frequently arises in database applications where there are large numbers of clauses. Or-parallelism also arises in generate and test kind of problems—the various alternatives to be tested can be generated in or-parallel (e.g., N-queens).

2.1. Problems in Implementation of Or-parallelism

In principle, or-parallelism should be easy to implement since various branches of the or-parallel tree are independent of each other, thus requiring little communication between processors. However, in practice, implementation of or-parallelism is difficult because of the sharing of nodes in the or-parallel tree. That is, given two nodes in two different branches of or-tree, all nodes above (and including) the least common ancestor node of these two nodes would be shared between the two branches. A variable created in one of these ancestor nodes might be bound differently in the two branches. The environments of the two branches have to be organized in such a fashion that in spite of the ancestor nodes being shared the correct bindings applicable to each of the two branches are easily discernible.

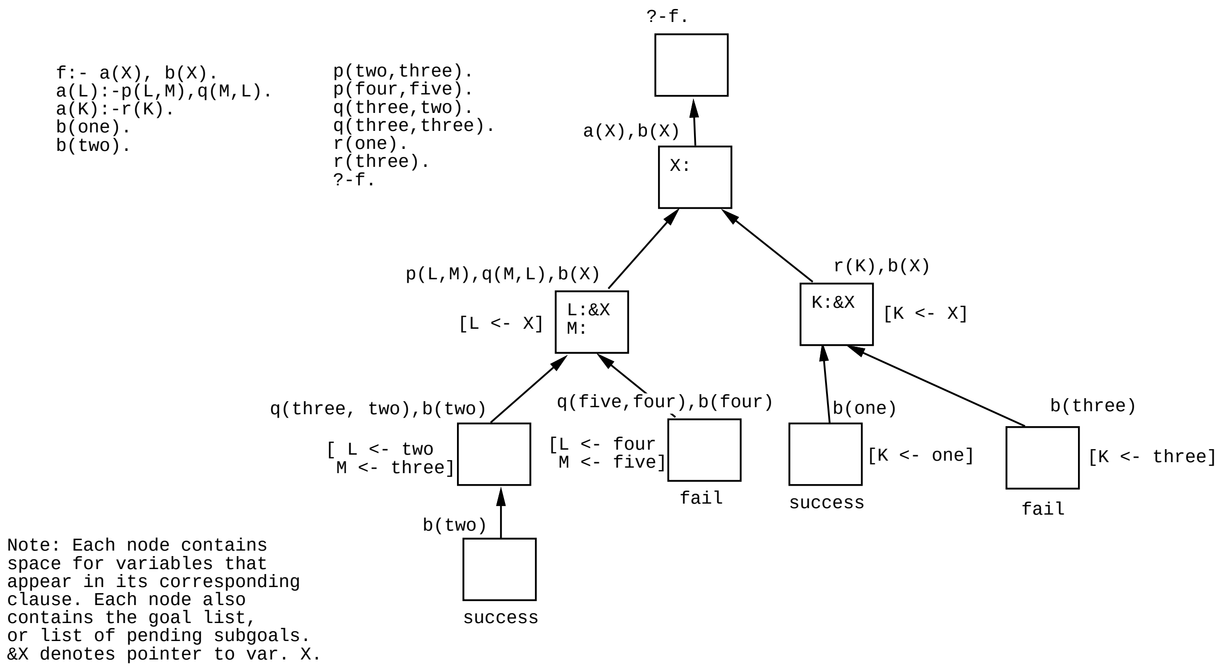


Figure 1: An Or-parallel Tree

If a binding for a variable, created in one of the common ancestor nodes, is generated above (or at) the least common ancestor node then this binding is the same for both branches and hence should be used as such (such a binding is known as an unconditional binding and such a variable as an unconditional variable). However, if a binding to such a variable is generated by a node below the least common ancestor node then that binding should be visible only to the branch to which the binding node belongs (such a binding is known as a conditional binding and such a variable as a conditional variable). Thus, the main problem in implementing or-parallelism is the efficient representation of multiple environments that co-exist simultaneously in the or-parallel tree corresponding to a program. Note that the main problem in management of multiple environments is that of efficiently representing and accessing the conditional bindings; the unconditional bindings can be treated as in normal sequential execution of logic programs (i.e., they can be stored in-place).

Essentially, the problem of multiple environment management has to be solved by devising a mechanism where each branch has some private area where it stores conditional bindings applicable to itself. There are many ways of doing this [143, 58]. For example:

- storing the conditional binding created by a branch in an array or a hash table private to that branch, from where the binding is accessed whenever it is needed.
- keeping a separate copy of the environment for each branch of the tree, so that every time branching occurs at a node the environment of the old branch is copied to each new branch.
- recording all the conditional bindings in a global data-structure and attaching a unique identifier with each binding which identifies the branch a binding belongs to.

Each approach has its associated cost. This cost is non-constant time and is incurred either at the time of variable access, or at the time of node creation, or at the time a processor begins execution of a new branch. In [58] three criteria were derived for an ideal or-parallel system, namely:

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*[2] should be constant-time.

It has been shown that it is impossible to satisfy these three criteria simultaneously [58]. In other words, the non-constant time costs in managing multiple or-parallel environments cannot be avoided. Although this non-constant cost cannot be avoided in supporting or-parallelism, it can be significantly reduced by a careful design of the *scheduler*, whose function is to assign *work* to processors (where *work* in an or-parallel setting will mean an unexplored branch of the or-parallel tree represented as an untried alternative in a choicepoint). The design of the scheduler is very important in an or-parallel system, and is discussed later; we first briefly describe the various models that have been proposed for exploiting or-parallelism.

2.2. Or-parallel Execution Models

A number of execution models have been proposed in the literature for exploiting or-parallelism (a listing of about 20 of them can be found in [58]). These models differ in the technique they employ for solving the problem of environment representation. For instance, the following models employ an environment representation technique that satisfies criteria 1 and 2 above (constant-time task creation and variable access): Versions Vectors Scheme [74], Binding Arrays Scheme [140, 142], Argonne-SRI Model [143], Manchester-Argonne Model [143], Delphi Model [40], Randomized Method [85], BC-Machine [3], MUSE [4], Virtual Memory Binding Arrays model [139], Kabu-Wake Model [93], etc.; while the following models employs an environment representation technique that satisfies criteria 2 and 3 above (constant-time variable access and task switch): Directory Tree Method [72], Environment Closing Method [43], etc.; and the following models employs an environment representation technique that satisfies criteria 1 and 3 above (constant-time task-creation and task-switch): Hashing Windows Method [23], Argonne Model [52], Virtual Memory Hashing Windows model [139], etc. Likewise, example of a model that only satisfies criteria 1 (constant time task-creation) is the Time-Stamping Model [135], while the example of a model that only satisfies criteria 3 (constant-time task switching) is the Variable Import Scheme [96]. We describe some of these techniques for implementing or-parallelism in greater detail below. A detailed study and derivation of some of the or-parallel models has also been done in [143].

Directory Tree Method

In the directory tree method [72], developed by Ciepielewski and Haridi in the early 80s for their or-parallel token machine [38], each branch of the or-tree has an associated process. The binding environment of a process consists of *contexts*. A new context is created for each clause invoked. Each process has a separate binding environment but allows sharing of some of the contexts in its environment by processes of other branches. To access its environment efficiently a process uses *directories*. A directory of a process is an array of references to contexts. The environment of the process consists of contexts pointed to by its directory. The ith location in the directory contains a pointer to the ith context for that process.

[2]That is, the cost associated with updating the state of a processor when it switches from one node of the tree to another.

When branching occurs, a new directory is created for each child process. For every context in the parent process which has at least one unbound variable, a new copy is created, and a pointer to it is placed at the same offset in the child directory as in the parent directory. Contexts containing no unbound variable (called *committed context*) can be shared and a pointer is simply placed in the corresponding offset of the child's directory pointing to the committed context.

A conditional variable is denoted by the triple ⟨directory address, context offset, variable offset⟩ where the directory address is the address of the base of the directory, context offset is the offset in the directory array and variable offset is the offset within the context. Notice that in this method all variables are accessed in constant time, and process switching does not involve any state change.

Hashing Windows Method

The hashing windows scheme, proposed by Borgwardt [23], maintains separate environments by using *hashing windows*. The hashing window is essentially a hash table. Each node in the or-tree has its own hashing window where the conditional bindings of that particular node are stored. The hash function is applied to the address of the variable to compute the address of the bucket in which the conditional binding would be stored in the hash window. Unconditional bindings are not placed in the hash window, rather they are stored in-place in the nodes. Thus, the hash window of a node records the conditional bindings generated by that node. During variable access the hash function is applied to the address of the variable whose binding is needed and the resulting bucket number is checked in the hash-window of the current node. If no value is found in this bucket, the hash-window of the parent node is searched recursively until either a binding is found, or the node where the variable was created is reached. If the creator node of the variable is reached then the variable is unbound. Hash windows need not be duplicated on branching since they are shared.

Time Stamping Method

The time-stamping method, developed by Tinker and Lindstrom [135] uses time stamps to distinguish the correct binding for an environment. All bindings for a variable are visible to all the processors. All bindings are stamped with the time at which they were created. The bindings also record the process-id of the process which created them. The branch points are also stamped with the time at which they were created. An *ancestor stack*, which stores the ancestor-process/binding-time pairs to disambiguate variables, is also kept with each process. The ancestor stack records the binding spans during which different processes worked on a branch. The ancestor stack is copied when a new process is created for an untried alternative.

To access the value of a variable, a processor has to examine all its bindings until the correct one is found, or none qualify, in which case the variable is unbound for that process. To check if a particular binding is valid, the id of the process, say P, which created it and the time stamp are examined. Next, one checks if the time stamp falls in the time span of the process P in any one of its entries in the ancestor stack. If such a P/binding-span entry is found then the binding is valid, else the next binding is examined until there are none left in which case the variable is not bound.

Binding Arrays Method

In the binding arrays method [140, 142] each processor (rather than a process) has an auxiliary data structure called the binding array. (Note that the description that follows is largely based on [142] rather than on [140]. The binding arrays technique in [140] was not primarily concerned with or-parallelism, rather with (primarily sequential) non-depth-first search.) Each conditional variable along a branch is numbered sequentially outward from the root.

To do this numbering, each branch maintains a counter (when branching occurs each branch gets a copy of the counter). When a conditional variable is created it is marked as one (by setting a tag), and the value of the counter recorded in it (this value is known as the offset value of the variable). The counter is then incremented. When a conditional variable gets bound, the binding is stored in the binding array of the processor at the offset location given by the offset value of that conditional variable. In addition, the conditional binding together with the address of the conditional variable is stored in the trail (thus, the trail is extended to include bindings as well). If the binding of this variable is needed later, then the offset value of the variable is used to index into the binding array to obtain the binding (Figure 2). Note that bindings of all variables, whether conditional or unconditional, are accessible in constant time.

To ensure consistency, when a processor switches from one branch (say $\mathtt{b}_i$) of the or-tree to another (say $\mathtt{b}_j$), it has to update its binding array by de-installing bindings from the trail of the nodes that are in $\mathtt{b}_i$ and installing the correct bindings from the trail of the nodes that are in $\mathtt{b}_j$ (Figures 2.(i) and 2.(ii)).

The binding arrays method has been used in the Aurora Or-parallel system, which is described in detail later.

Versions Vectors Method

The versions vectors method [74] is very similar to the binding arrays method except that instead of a conditional variable being allocated space in the binding array each one is associated with a *versions vector*. A versions vector stores the vector of bindings for that variable such that the binding imparted by a processor with processor-id i (processor ids are numbered from 1 to n, where n is the total number of processors) is stored at offset i in the vector. The binding is also recorded in the trail, like in the Binding Arrays method. Like in the binding arrays method, on switching to another branch a processor with pid j has to update the jth slots of versions vectors of all conditional variables that lie in the intervening nodes to reflect the correct bindings corresponding to the new site.

Environment Closing Method

The environment closing method was proposed by Conery [43] and is primarily meant for message-passing systems. The idea behind *closing* an environment is to make sure that all accesses are only to variables owned by search tree nodes that reside locally. A node in the search tree (Conery refers to nodes as frames) `A` is closed with respect to another node `B` by eliminating all pointers from the environment of node `A` to the environment of node `B` (changing them from node `B` to node `A` instead). The process involves traversing all the structures in node `B` which can be reached through the environment of node `A`. For each unbound variable V in such a structure a new variable V' is introduced in `A`. The unbound variable is made to point to this new variable. The structure is copied into `A`, with the variable V in that structure being replaced by the new variable V'. Note that multiple environments for each clause matching a goal are represented in this method through explicit copying of all unbound variables that are accessible from the terms in the goal.

During execution, each new node introduced is closed with respect to its parent node after the unification is done. After the body of the clause corresponding to the node is solved the parent node is closed with respect to its child node so that the child's sibling can be tried. If the child node corresponds to a unit clause the parent node is immediately closed with respect to its child after unification. Closing the child

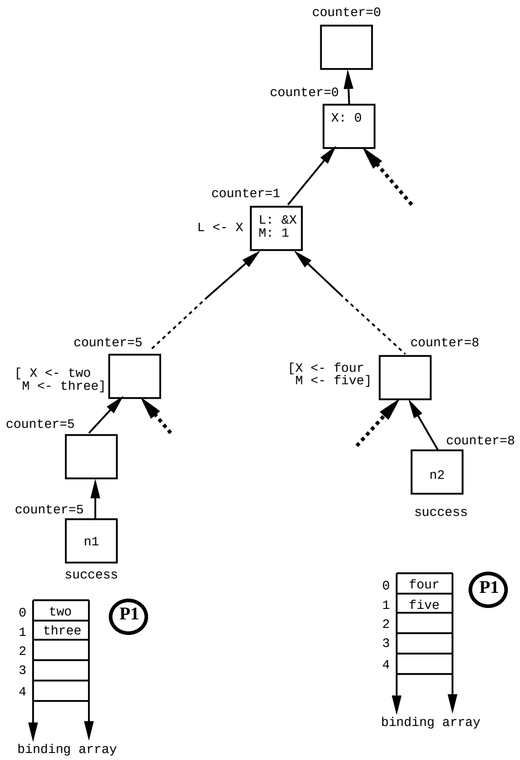


Fig (i): Processor P1 is at the success node labeled n1. The bindings for X and M are stored in location 0 and 1 respectively of the binding array. Since L is aliased to X, L is made to point to X.

Note: The trail entries are shown enclosed in square brackets.

Fig (ii): Processor P1 switches from node labeled n1 to the node labeled n2. It has to update its binding array so that the correct bindings for X, L, and M are reflected. To update its binding array the processor travels up the tree to the common ancestor node of n1 and n2 and removes the binding for L and M when it encounters them in the trail. It then moves down from the common ancestor node to node n2, installing the correct binding of X, L and M from the trail.

Figure 2: The Binding Arrays Method

node ensures that no variables in ancestor nodes would be accessed. Closing the parent node ensures that the variable bindings produced by the execution of its children are imported back into the parent node's environment.

Favored-Bindings Method

The favored binding method [52] proposed by researchers at Argonne National Laboratory is very similar to the hash-window method. In this method the or-parallel tree is divided into *favored*, *private*, and *shared* sections. Bindings imparted to conditional variables by favored section are stored in-place in the node. Bindings imparted by other sections are stored in a hash table containing a constant number of buckets (32 in the Argonne implementation). Each bucket contains a pointer to the linked list of bindings which map to that bucket. When a new binding is inserted, a new entry is created and inserted at the beginning of the linked list of that bucket as follows: (i) The next pointer field of the new entry records the old value of the pointer in the bucket. (ii) The bucket now points to this new entry. At a branch point each new node is given a new copy of the buckets (but not a new copy of the lists pointed to by the buckets).

When a favored branch has to look up the value of a conditional variable it can find it in-place in the value-cell. However, when a non-favored branch accesses a variable value it computes the hash value using the address of the variable and locates the proper bucket in the hash table. It then traverses the linked list until it finds the correct value. Notice how separate environments are maintained by sharing the linked list of bindings in the hash tables.

MUSE Method

In the MUSE method [4, 5] a separate environment is maintained by each processor in which it can write without causing any binding conflicts. In MUSE even unconditional bindings are not shared, as they are in other methods described above. When an idle processor P2 picks an untried alternative from a choicepoint created by another processor P1, it copies all the stacks of P1. This copying of stacks is made more efficient through the technique of *incremental copying*. The idea of *incremental copying* is based on the fact that the idle processor could have already traversed a part of the path from the root node of the or-parallel tree to the least common ancestor node, thus it does not need to copy this part of stacks. As a result of copying, each processor can carry out execution exactly like a sequential system, requiring very little synchronization with other processors. The implementation of MUSE is described in greater detail later.

2.3. Support for Full Prolog

Most of the models described above consider only pure logic programs (pure Prolog) for parallel execution. However, to make logic programming practical many extralogical, metalogical and side-effect predicates have been incorporated in Prolog, the most commonly used logic programming language. Some researchers have taken the view that a parallel logic programming system should transparently execute Prolog programs in parallel [99, 77]. That is, the same effect should be seen by a user during parallel execution of a Prolog programs, as far as side-effects etc. are concerned (including printing of the final solutions), as in its sequential execution with left-to-right, top-to-bottom selection rule. Such a system is said to support *sequential Prolog semantics*. The advantage of such an approach is that existing Prolog programs can be taken and executed in parallel without any modifications. Two prominent or-parallel systems that have been built, namely MUSE and Aurora, do support sequential Prolog semantics by executing an extralogical predicates only when the branch containing it becomes the leftmost in the search tree (however, support for extra-logical predicates that can be executed asynchronously is also present (see later)).

An issue that arises in the presence of pruning operators such as cuts and commits during or-parallel execution is that of *speculative work* [8, 20, 75, 76, 129]. Consider the following program:

```
p(X, Y) :- q(X), !, r(Y).
p(X, Y) :- g(X), h(Y).
...
```

and the goal:

```
?- p(A, B).
```

Executing both branches in parallel, corresponding to the two clauses that match this goal, may result in unnecessary work, because following sequential Prolog semantics would mean that if `q(X)` succeeds then the second clause for `p` will never be tried. Thus, in or-parallel execution, execution of the second clause is *speculative*, in the sense that its usefulness depends on the success/failure outcome of goal `q`.

It is a good idea for a scheduler designed for an or-parallel system that aims to support sequential Prolog semantics to take speculative work into account. Essentially, such a scheduler should bias all the processors to pick work that is within the scope of a cut from branches to the left in the corresponding subtree rather than from branches to the right [8, 18, 20, 129].

A detailed survey on scheduling and handling of speculative work for or-parallelism is beyond the scope of this paper, and can be found in [37]. One must note that the efficiency and the design of the scheduler has the biggest bearing on the overall efficiency of an or-parallel system (or any parallel system for that matter). We describe two such systems next where a significant amount of effort has been invested in designing and fine-tuning the or-parallel system and its schedulers.

2.4. The Aurora Or-parallel Prolog System

Aurora is a prototype or-parallel implementation of the full Prolog language. It was first developed for UMA (Uniform Memory Access) shared-memory multiprocessors such as the Sequent Symmetry and was subsequently adapted [104] for NUMA (Non-Uniform Memory Access) architectures such as the BBN TC-2000, a current scalable architecture with Motorola 88000 processors.

Aurora was developed as part of an informal research collaboration known as the "Gigalips Project" with research groups at Argonne National Laboratory, the University of Bristol (previously at the University of Manchester), the Swedish Institute of Computer Science, and IQSOFT SZKI Intelligent Software Co. Ltd., Budapest as the main implementors.

Aurora is based on the SRI model, as originally described in [142] and refined in [99]. The SRI-model employs binding arrays, described earlier, for representing multiple environment. In the SRI model, a group of processing agents called *workers* cooperate to explore a Prolog *search tree*, starting at the root (the topmost point). A worker has two conceptual components: an *engine*, which is responsible for the actual execution of the Prolog code, and a *scheduler*, which provides the engine component with work. These components are in fact independent of each other, and a clean *interface* between them has been designed [132, 32] allowing different schedulers and engines to be plugged in. To date, Aurora has been run with five different schedulers, and the same interface has been used to connect one of the schedulers with the Andorra-I engine [120] to support both and- and or-parallelism. The Aurora engine and compiler [32] were constructed by adapting SICStus Prolog 0.6. Garbage collection for Aurora has been investigated by Weemeeuw [146].

In the SRI model, the search tree, defined implicitly by the program, is explicitly represented by a *cactus stack* generalizing the stacks of sequential Prolog execution. Workers that have gone down the same branch *share* the data on that branch. Bindings of shared variables must of course be kept private, and are recorded in the worker's private *binding array*. The basic Prolog operations of binding, unbinding, and dereferencing are performed with an overhead of about 25% relative to sequential execution (and remain fast, *constant-time* operations). There is also a significant overhead associated with task switching—the worker has to update its binding array by deinstalling bindings as it moves up the tree and installing bindings as it moves down another branch. The overhead, called *migration cost* (or task-switching cost), is proportional to the number of bindings that are deinstalled and installed. Aurora divides the or-parallel search tree into a *public* region and a *private* region. The public region consists of those nodes from which other workers can pick up untried alternatives. The private region consists of nodes private to a worker that cannot be accessed by other workers. Execution within private region is exactly like sequential Prolog execution. Nodes are transferred from private region of a worker P to the public region by the scheduler, that does so when another idle worker Q requests work from worker P.

One of the principal goals of Aurora has been the support of the *full* Prolog language. Preserving the semantics of built-in predicates with side effects is achieved by *synchronization*: whenever a non-leftmost branch of execution reaches a side-effect predicate, the given branch is suspended until it becomes leftmost [76]. This technique ensures that the side-effect predicates are executed in the same left-to-right order as in a sequential implementation, thus preserving compatibility with these implementations.

It is often the case that this strict form of synchronization is unnecessary, and slows down parallel execution. Aurora therefore provides non-synchronized variants for most side-effect predicates which come in two flavors: the *asynchronous* form respecting the cut pruning operator, and the completely relaxed *cavalier* form. Notably, non-synchronized variants are available for the dynamic database update predicates (`assert`, `retract` etc.).

A systematic treatment of pruning operators (cut and commit) and of speculative work has proved to be of tremendous importance in or-parallel implementations. Algorithms for these aspects have been investigated by Hausman [75, 76] and incorporated into the interface and schedulers.

Graphical tracing packages have turned out to be essential for understanding the behavior of schedulers and parallel programs and finding performance bugs in them [53, 83, 33].

Several or-parallel applications for Aurora were studied in [92] and [98]. The non-synchronized dynamic database features have been exploited in the implementation of a general algorithm for solving optimization problems [131].

Three schedulers are currently operational. Two older schedulers were written [29, 25], but have not been updated to comply with the scheduler-engine interface:

The Manchester Scheduler. The Manchester scheduler [31] tries to match workers to available work as well as possible. The matching algorithm relies on global arrays, indexed by worker number. One array indicates the work each worker has available for sharing and its migration cost, and the other indicates the status of each worker and its migration cost if it is idle. The Manchester scheduler was not designed for handling speculative work properly. A detailed performance analysis of the Manchester scheduler was done in [130].

The Bristol Scheduler. The Bristol scheduler [19] tries to minimize scheduler overhead by extending the public region eagerly: sequences of nodes are made public instead of single nodes, and work is taken from the bottommost live node of a branch. A more recent version [20] addresses the problem of efficiently scheduling speculative work. It actively seeks the least speculative work available and uses one strategy if all the work is speculative and another if non-speculative work is available.

The Dharma Scheduler. The Dharma scheduler [129] is also designed for efficiently scheduling speculative work. It addresses the problem of quickly finding the leftmost, thus least speculative, available work, by directly linking the tips of each branch.

Good speed-ups have been obtained by all schedulers of Aurora for many benchmark programs. Some of these benchmarks contain lots of speculative work in which speed-ups are measured for finding the first (leftmost) solution. The degree of speedup for such benchmark programs depends on where in the Prolog search tree the first solution is and on the frequency of moving workers from right to left to pick less speculative work. There are other benchmarks with little or no speculative work that are used for producing all solutions. The degree of speedup for such benchmark programs depends on the amount of parallelism present. More on the Aurora system, and a detailed discussion of its performance results, can be found in [31, 130, 19, 20, 129]

2.5. The MUSE Or-parallel Prolog System

The MUSE or-parallel Prolog system has been designed and implemented on a number of NUMA and UMA computers (Sequent Symmetry, Sun Galaxy, BBN Butterfly II, etc.) [4, 5, 6, 7, 8, 91]. It supports the full Prolog language and programs run on it with almost no user annotations. It is based on a simple extension of the state-of-the-art sequential Prolog implementation (WAM [141]).

The MUSE model assumes a number of extended WAMs (called *workers*), each with its own local address space, and some global space shared by all workers. The model requires copying parts of the WAM stacks when a worker runs out of work or suspends its current branch. The copying operation is made efficient by utilizing the stack organization of the WAM. Workers make a number of choicepoints sharable, and they get work from those shared choicepoints (nodes) by the normal backtracking of Prolog. A worker consists of two components: the *engine*, which performs the actual Prolog work, and the *scheduler*. The schedulers, working together, schedule the work between engines and support the sequential semantics of Prolog.

The first MUSE engine has been produced by extending the SICStus Prolog version 0.6. Extensions are carefully added to preserve the high efficiency of SICStus leading to a negligible overhead which is significantly lower than in other or-parallel models.

The MUSE scheduler supports efficient scheduling of speculative work [8]. It has two different scheduling strategies: one for busy workers actively seeking the leftmost available work, and the other for idle workers looking for work. The idea of the first strategy is based on dividing the Prolog tree into two sections: the right section contains voluntarily suspended work and the left section contains active work. The available workers concentrate on the available work in the left section. When the amount of parallelism in the left section is not enough for the workers, some of the leftmost part of the right section will be resumed.

The other strategy is based on the principle that when a worker is idle, its next piece of work will be taken from the bottommost (i.e., youngest) node in the richest branch (i.e., the branch with maximum or-parallel work) of a set of active branches. When the work at the youngest node is exhausted, that worker will find more work by backtracking to the next youngest node. If the idle worker cannot find work in the system, it will resume the leftmost part of the suspended section of the tree.

The MUSE system controls the granularity of jobs at run-time by avoiding sharing very small tasks. The idea is that when a busy worker reaches a situation at which it has only one private parallel node, it will make its private load visible to the other workers only when that node is still alive after a certain number of Prolog procedure calls. Without such a mechanism the gains due to parallel execution can be lost as the number of workers is increased.

A clean interface between the MUSE engine and the MUSE scheduler has been designed and implemented. It has improved the modularity of the system and preserved its high efficiency.

Tools for debugging and evaluating the MUSE system have been developed. The evaluation of the system on Sequent Symmetry and on BBN Butterfly machines I and II shows very promising results in absolute speed and also in comparison with results of the other similar systems. The speed-ups obtained are near linear for programs with large amounts of or-parallelism. For programs that do not have enough or-parallelism to keep all available workers busy the speed-ups are (near) linear upto the point where all parallelism is exploited. The speed-up does not increase or decrease thereafter with increase in number of workers. For programs with no or very low or-parallelism, the speed-ups obtained are close to 1 due to very low parallel overheads. More details of the MUSE system and a discussion of its performance results can be found in references cited earlier [6, 7, 8, 91].

A useful result of the MUSE work has been that the first commercial parallel Prolog system on Sun Galaxy will be produced by extending the *ProLog by BIM* according to the MUSE approach.

The next MUSE system will be based on the latest version of SICStus (version 2.1) with native code generation and constraint solvers.

3. Independent And-parallelism

Independent and-parallelism arises when two or more *independent* subgoals can be executed in parallel. Given two subgoals, they either have a data-dependency between them (occurrence of the same variable in the arguments of two subgoals) or they don't. If they don't have any data-dependencies then they can be freely executed in parallel. This kind of and-parallelism is termed as *independent and-parallelism.* If they do have data-dependencies then they can still be executed independently in parallel, although only up to a certain point because unrestricted parallel execution of two dependent subgoals can be very inefficient, as will be discussed later.

To take a simple example, consider the naïve *fibonacci* program shown below:

```
fib(0, 1).
fib(1, 1).
fib(M, N) :- [ M1 is M - 1, fib(M1, N1) ],
             [ M2 is M - 2, fib(M2, N2) ],
             N is N1 + N2.
```

The two list of goals, each enclosed within square brackets above, have no data-dependencies among themselves and hence can be executed independently in parallel with each other. But the last subgoal

`N is N1 + N2` depends on the outcomes of the two and-parallel subgoals, and should start execution only after `N1` and `N2` get bound.

And-parallelism manifests itself in a number of application—those in which a given problem can be divided into a number of independent sub-problems. For example, it appears in divide and conquer algorithms, where the independent recursive calls can be executed in parallel (e.g., matrix multiplication, quicksort, etc.).

Or-parallelism arises only in non-deterministic programs, whereas and-parallelism arises in non-deterministic as well as deterministic programs. Thus, if one wants to speed up deterministic programs, one has to exploit and-parallelism.

3.1. Problems in Implementing Independent And-parallelism

In this section we examine the problems associated with implementing and-parallelism. We discuss the various phases of an and-parallel system and examine the problems encountered in each.

An independent and-parallel system can be divided into three phases [42]:

i. Ordering Phase: deals with detection of dependencies among goals.

ii. Forward Execution Phase: deals with steps to be taken to select the next subgoal for execution and initiating its execution.

iii. Backward Execution Phase: deals with steps to be taken when a goal fails, i.e., the operation of backtracking.

3.1.1. Ordering Phase

The ordering phase in independent and-parallel system is concerned with detecting data dependencies between subgoals. Once it is determined that two (or more) subgoals do not have any data-dependencies they can be executed in parallel. If and-parallel execution is initiated without caring for data-dependencies, then it may lead to wasteful computation. Consider the rules:

```
p(X) :- r(X), s(X).
```

and the goal

```
?- p(Z).
```

Suppose `X` is an "output argument" for `r` and "input argument" for `s`, i.e., there is a data dependency between the two subgoals. Suppose we initiate their execution in parallel and assume that `r` produces the binding `X = a` while `s` concurrently searches for values of `X`. The goal `s` might generate a number of bindings for `X` after a great deal of computation, very few of which match with `a`. There are two places here where wasteful computation takes place. Firstly, `s` computes bindings for `X` which will eventually be discarded, hence these computations are wasted. If `s` had known that the only permissible value of `X` is `a`, its search space would have been narrowed. Secondly, every time `s` produces a binding for `X` the binding value has to be unified with the binding value produced by `r` to determine that they are identical. This unification, termed *back-unification* [148], can introduce some extra overhead.

Data dependencies cannot be detected at compile time, because in many cases they arise at runtime. Consider the clause:

```
p(X, Y) :- r(X), s(Y).
```

It may appear that goals `r` and `s` are independent in the clause for `p` and hence can be executed in parallel. However, it is possible that the variables `X` and `Y` may get aliased at runtime, making them dependent on each other. For instance, if the query was `?- p(Z,Z)`, both `X` and `Y` would get aliased to each other via `Z`.

The example above clearly shows that syntactic data-dependency checks are not sufficient for exploiting and-parallelism. We have to check for independence of subgoals at runtime. However, the cost incurred in incorporating checks at runtime will slow down program execution.

A number of approaches have been proposed for detecting data-dependencies. They range from purely compile-time techniques to purely runtime ones. There is a trade-off between the amount of and-parallelism exploited and data-dependency analysis overhead incurred at runtime—purely compile time techniques lose a lot of and-parallelism but incur very little run-time overhead while purely run time techniques capture maximal and-parallelism at the expense of costly overhead. The various approaches are briefly described below:

i. Input Output Modes: One way to overcome the data-dependency problem is to require the user to specify the 'mode' of the variables, i.e., whether an argument of a predicate is an input variable or an output variable. Input variables of a subgoal have to become bound before the subgoal can start executing. This approach has been taken in Committed Choice Languages [128] to control execution of dependent goals.

ii. Static Data Dependency Analysis: In this technique the goal and the program clauses are globally analyzed at compile time, assuming worst cases for subgoal dependencies [35]. No checks are done at runtime. Since the analysis is done at compile-time, assuming worst case scenario, a lot of parallelism may be lost. The advantage is, of course, that no overhead is incurred at run-time.

iii. Run-time Dependency Graphs: Another approach is to generate the *dependency graph* at runtime. This involves examining bindings of relevant variables every time a subgoal finishes executing. This approach has been adopted by Conery in his AND/OR model [41, 42, 44]. This approach has prohibitive runtime cost, since variables may be bound to large structures with embedded variables. The advantage of this scheme is that maximal and-parallelism is exploited.

iv. A fourth approach, which is midway between (ii) and (iii), encapsulates the dependency information in the code generated by the compiler, along with addition of some extra information to the variables. In this way simple runtime checks can be done to check for dependency. This technique was first devised by DeGroot and is called Restricted (or Fork/Join) And-Parallelism (RAP) [49], and was further enhanced by Hermenegildo and Nasr [81]. It does not capture all the and-parallelism, although it does manage to exploit a substantial portion of it (see later).

Approach (i) differs from the rest in that the programmer has to explicitly specify the dependencies, using annotations. Approach (iv) is a nice compromise between (ii), where extensive compile time analysis is done to get sub-optimal parallelism, and (iii), where a costly runtime analysis is needed to get maximal parallelism. Moreover recent research has shown that these annotations (CGEs) can be generated via compile-time analysis [105, 106, 86, 110] based on abstract interpretation [45].

3.1.2. Forward Execution Phase

The forward execution phase follows the ordering phase. It selects independent goals that can be executed in and-parallel, and initiates their execution. The execution continues like normal sequential execution until either failure occurs, in which case the backward execution phase is entered, or a solution is found. It is also possible that the ordering phase might be entered again during forward execution; for example in the case of Conery's scheme when a non-ground term is generated. Implementation of the forward execution phase is relatively straightforward; the only major problem is efficient determination of goals that are ready for and-parallel execution.

3.1.3. Backward Execution Phase

The backward execution phase ensues when failure occurs, or more solutions to the top-level query are sought after one is reported. The subgoal to which execution should backtrack is determined, the machine state is restored, and forward execution of the selected subgoal is initiated. The two major problems with backward execution are: (i) efficiently terminating execution of sibling and-parallel subgoals if one of them fails, and; (ii) efficiently and correctly selecting the appropriate subgoal to backtrack to. These problems are complicated by the fact that and-parallel subgoals may have nested and-parallel subgoals currently executing which have to be terminated or backtracked over. In the past, backtracking algorithms have been proposed which later turned out to be incomplete [149].

3.2. Criteria for Independent And-parallel Execution

We developed three criteria for ideal or-parallel systems. Likewise, we present some criteria for ideal and-parallel systems:

i. An ideal and-parallel system should avoid wasteful computation.

ii. It should avoid complex runtime analyses (such as runtime data dependency analysis).

iii. It should take advantage of intelligent backtracking that is possible with independent and-parallelism.

The motivation for the first two criteria is clear from the preceding discussion. To see why and-parallel systems should support intelligent backtracking consider the goal:

```
?- a, b, c, d.
```

Suppose `b` and `c` are independent subgoals and can be executed in and-parallel. Suppose that both `b` and `c` are non-determinate and have a number of solutions. Consider what happens if `c` fails. In normal sequential execution we would backtrack to `b` and try another solution for it. However, since `b` and `c` do not have any data dependencies, retrying `b` is not going to bind any variables which would help `c` to succeed. So if `c` fails, we should backtrack and retry `a`. This kind of backtracking, based on the knowledge of data dependence, is called *intelligent backtracking*. As should be obvious, knowledge about data dependencies is needed for both intelligent backtracking as well as and-parallel execution. Thus, if an and-parallel system performs data-dependency analysis for parallel execution, it should take further advantage of it for intelligently backtracking as well. Note that intelligent backtracking achieved may be limited, since, in the example above, `a` may not be able to cure failure of `c`. Execution models for independent and-parallelism that exploit limited intelligent backtracking [81] as well as those that employ fully intelligent backtracking [94] have been proposed and implemented.

3.3. Support for Full Prolog

Like or-parallel systems, some researchers have favored supporting Prolog's sequential semantics in independent and-parallel systems [111, 50, 34]. This imposes some constraints on how backtracking as well as forward execution take place. Essentially, the approach that has been taken is that if two independent goals are being executed in parallel, both of which lead to a side-effect, then the side-effect in the right goal can only be performed after the last side-effect in the goal to the left has been executed. It also means that intelligent backtracking has to be sacrificed, because considering again the previous example, if `c` fails and we backtrack directly into `a`, without backtracking into `b` first, then we may miss executing one or more side-effects that would be executed had we backtracked into `b`.

The issue of speculative computation also arises in and-parallel systems. Given two independent goals `a(X), b(Y)` that are being executed in and-parallel, if `a` eventually fails, then work put in for solving `b` will go wasted (in sequential Prolog the goal `b` will not ever get executed). Therefore, not too many resources (processors) should be invested on goals to the right. Once again, it should be stressed, the design of the work-scheduler is very important for a parallel logic programming system.

3.4. Independent And-parallel Execution Models

In this section we briefly describe some of the methods that have been proposed for realizing an and-parallel system. These are:

i. Conery's abstract parallel implementation model [41, 42];

ii. And-Parallel Execution (APEX) Model of Lin and Kumar [95]; and,

iii. Restricted And-parallel (RAP) model, introduced by DeGroot [49], and further refined by Hermenegildo and Nasr [81, 79].

Conery's Model

In this method [42] a data-flow graph is constructed in the ordering phase making the producer-consumer relationships between subgoals explicit. If a set of subgoals have an uninstantiated variable `V` in common, one of the subgoals is designated as the producer of the value of `V` and is solved first. Its solution is expected to instantiate `V`. When the producer has been solved, the other subgoals, the consumers, may be scheduled for evaluation. The execution order of the subgoals is expressed as a data-flow graph, in which an arc is drawn from the producer of a variable to all its consumers.

Once the data-flow graph is determined, forward execution phase ensues. In this phase and-parallel execution of subgoals which do not have any arcs incident on them in the data-flow graph is initiated. When a subgoal is resolved away from the body of a clause (i.e., it is successfully solved), the corresponding node and all of the arcs emanating from it are removed from the data-flow graph. If a producer creates a non-ground term during execution, the ordering algorithm must be invoked again to redraw the data-flow graph.

When execution fails, some previously solved subgoal must be solved again to yield a different solution. The backward execution phase picks the last parent (as defined by a linear ordering of subgoals, obtained by a depth first traversal of the data-flow graph) for the purpose of re-solving.

Note that in this method data dependency analysis for constructing the data-flow graph has to be carried out every time a non-ground term is generated, making its cost prohibitive.

APEX Model

The APEX (And Parallel EXecution) model has been devised by Lin and Kumar [95]. In this method forward execution is implemented via a token passing mechanism. A token is created for every new variable that appears during execution of a clause. A subgoal `P` is a producer of a variable `V` if it holds the token for `V`. A newly created token for a variable `V` is given to the leftmost subgoal `P` in the clause which contains that variable. A subgoal becomes executable when it receives tokens for all the uninstantiated variables in its current binding environment. Parallelism is exploited automatically when there are more than one executable subgoals in a clause.

The backward execution algorithm performs intelligent backtracking at the clause level. Each subgoal $\mathtt{P}_i$ dynamically maintains a list of subgoals (denoted as *B-list*($\mathtt{P}_i$)) consisting of those subgoals in the clause which may be able to cure the failure of $\mathtt{P}_i$, if it fails, by producing new solutions. When a subgoal $\mathtt{P}_i$ starts execution, *B-list*($\mathtt{P}_i$) consists of those subgoals that have contributed to the bindings of the variables in the arguments of $\mathtt{P}_i$. When $\mathtt{P}_i$ fails, $\mathtt{P}_j$ = head(*B-list*($\mathtt{P}_i$)) is selected as the subgoal to backtrack to. The tail of *B-list*($\mathtt{P}_i$) is also passed to $\mathtt{P}_j$ and merged into *B-list*($\mathtt{P}_j$) so that if $\mathtt{P}_j$ is unable to cure the failure of $\mathtt{P}_i$, backtracking may take place to other subgoals in B-list($\mathtt{P}_i$).

This method also has significant runtime costs since *B-lists* are created, merged and manipulated at runtime. APEX has been implemented on shared memory multiprocessors for pure logic programs [95].

RAP Model

In this method program clauses are compiled into Conditional Graph Expressions (CGEs). Conditional Graph Expressions are expressions of the form

$$(\mathtt{condition} \ \rightarrow \ \mathtt{goal}_1 \ \& \ \mathtt{goal}_2 \ \& \ \ldots \ \& \ \mathtt{goal}_\mathtt{n}),$$

meaning that, if *condition* is true, goals $goal_1 \ldots goal_n$ should be evaluated in parallel, otherwise they should be evaluated sequentially. The *condition* can be either $ground(v_1, \ldots, v_n)$, which checks whether all of the variables $v_1, \ldots, v_n$ are bound to ground terms or $independent(v_1, \ldots, v_n)$, which checks whether the set of variables reachable from each of $v_1 \ldots v_n$ are mutually exclusive of one another. The *condition* can also be the constant *true*, which means the goals can be executed in parallel unconditionally. The groundness and independence conditions are evaluated at runtime. (A simple technique which keeps track of groundness and independence properties of variables through tags is presented in [49].) The method is conservative in that it may type a term as nonground even when it is ground—one reason why this method is regarded as "restricted." Another way in which CGEs are restrictive is that they cannot capture all the and-parallelism present in a program, because of their parenthetical nature (the same reason why parbegin-parend expressions are less powerful than fork-join expressions in exploiting concurrency; see Section 9.2.3, p. 317, of [113]).We think that among all the models the RAP model comes closest to realizing the criteria mentioned in the previous section. This model has been extended by Hermenegildo and Nasr, and has been efficiently implemented using WAM-like instructions [79, 110] as the &-Prolog system [110]. The CGEs are generated at compile time [105, 106, 86] using the technique of abstract interpretation [45]. CGEs generated through this analysis at compile-time manage to capture a substantial amount of and-parallelism [107].

3.5. The &-Prolog AND-Parallel Prolog System

&-Prolog is a prototype Prolog implementation, fully compatible with the "Edinburgh" standard, and capable of exploiting independent and-parallelism automatically by means of a parallelizing compiler. Explicit parallelization of programs by the user is also supported through the &-Prolog language extensions, which allow expressing arbitrary and-parallelism (i.e., not only independent). The same language is used to make the result of the automatic parallelization visible to the user if so desired. The parallelizing compiler has been integrated into the Prolog run-time environment in the standard way so that a familiar user interface with on-line interpreter and compiler is provided. Normally, users are unaware (except for the increase in performance) of any difference with respect to a conventional Prolog system. Compiler switches (implemented as "prolog flags") determine whether or not user code will be parallelized and through which type of analysis. If the user chooses to parallelize some of the code the compiler still helps by checking the supplied annotations for correctness, and providing the results of global analysis to aid in the dependency analysis task.

&-Prolog was originally designed for global addressing space systems and it has been implemented on a number of shared memory multiprocessors, including Sequent Balance, Sequent Symmetry, and Sun Galaxy systems. The &-Prolog system comprises a parallelizing compiler aimed at uncovering the parallelism in the program and an execution model/run-time system aimed at exploiting such parallelism. There is also an on-line visualization system (based on the X-windows standard) which provides a graphical representation of the parallel execution and has proven itself quite useful in debugging and performance tuning [33]. The first version of the &-Prolog system was developed collaboratively between The University of Texas and MCC. Newer versions have been developed at the Technical University of Madrid (UPM).

3.5.1. &-Prolog Parallelizing Compiler

Input code is processed by several compiler modules as follows: The *Annotator*, or "parallelizer", performs a (local) *dependency* analysis on the input code. It receives information from the *Side-Effect Analyzer* on whether or not each non-builtin predicate and clause of the given program is *pure*, or contains or calls a *side-effect*. This information is used to correctly sequence such side-effects [111]. If the appropriate option is selected, the annotator gets information about the possible run-time substitutions ("variable bindings") at all parts in the program as well as other types of information from the *Global Analyzer* (described below). Finally, it also receives information from the *Granularity Analyzer* regarding the size of the computation associated with a given goal [48]. This information is used in an additional pass aimed at introducing granularity control, implemented using dynamic term size computation techniques [97]. The annotator uses all available information to rewrite the input code for parallel execution. Its output to the next stage is an annotated &-Prolog program. Some of the techniques and heuristics used in the annotator are described in [107, 11, 30]. A final pass (an extension of the SICStus compiler) produces PWAM code from an already parallelized &-Prolog program.

The global analysis mentioned above is performed by using the technique of "abstract interpretation" [45] to compute safe approximations of the possible run-time substitutions at all points in the program. Two generations of analyzers have been implemented, namely the "MA^3" and "PLAI" analyzers. "MA^3" [82] uses the technique of "abstract compilation" and a domain which is currently known as "depth-K" abstraction. Its successor, PLAI, is a generic framework based on that of Bruynooghe [26] and the specialized fixpoint algorithms described in [105, 108, 109]. PLAI also includes a series of abstract domains and unification algorithms specifically designed for tracking variable dependence information.

Other concepts and algorithms used in the global analyzer, the rest of the &-Prolog compiler, and the MA^3 and PLAI systems are described in [106, 82, 11].

3.5.2. &-Prolog Run-Time System

The &-Prolog run-time system is based on the Parallel WAM (PWAM) model [110], an evolution of RAP-WAM [80, 79, 133], itself an extension of the Warren Abstract Machine (WAM) [141]. The actual implementation has been performed by extending the SICStus-Prolog abstract machine.

The philosophy behind the PWAM design is to achieve similar efficiency to a standard WAM for sequential code while minimizing the overhead of running parallel code. Each PWAM is similar to a standard WAM. The instruction set includes all WAM instructions (some of which have been modified for the PWAM) and several additional instructions related to parallel execution. The storage model includes a complete set of WAM registers and data areas, called a *stack set*, with the addition of a *goal stack* and two new types of stack frames: *parcall frames* and *markers*.[3] Goals which are ready to be executed in parallel are pushed on to the goal stack. The goals are then available to be executed on any PWAM (including the PWAM which pushed them). Parcall frames are used for coordinating and synchronizing the parallel execution of the goals inside a parallel call, both during forward execution and during backtracking. Markers are used to delimit *stack sections* (horizontal cuts through the stack set of a given abstract machine, corresponding to the execution of different *parallel* goals) and they implement the storage recovery mechanisms during backtracking of parallel goals in a similar manner to choice-points for sequential goals [78, 126]. In practice, the stack is divided into separate *control* (choice point and markers) and *local* stacks (environments, including parcall frames) for reasons of locality and locking.

The &-Prolog run-time system architecture comprises a ring of stack sets, a collection of agents, and a shared code area. The agents (unix processes) run programs from the code area on the stack sets. All agents are identical (there is no "master" agent). In general, the system starts allocating only one stack set. Other stack sets are created dynamically as needed upon appearance of parallel goals. Also, agents are started and put to "sleep" as needed in order not to overload the system when no parallel work is available. Several scheduling and memory management strategies have been studied for the &-Prolog system. For more details the reader is referred to [78, 110, 126].

3.5.3. Performance results

Experimental results for the &-Prolog system are available in the literature illustrating the performance of both the parallelizing compiler and the run-time system. The cost and influence of global analysis in terms of reduction in the number or run-time tests using the "MA^3" analyzer was reported in [82]. Number of CGEs generated, compiler overhead incurred due to the global analysis, and the result both in terms of number of unconditional CGEs and of reduction of the number of checks per CGE were studied for some benchmark programs. These results suggested that, even for this first generation system, the overhead involved in performing global analysis is fairly reasonable while the results can be quite precise.

Early experimental results regarding the performance of the second generation parallelizing compiler in terms of attainable program speedups were reported in [11] both without global analysis and also with

[3] While the PWAM uses conventional *environment sharing* for sequential goals, it uses a combination of *goal stacking* and environment sharing for parallel goals: goal descriptors are stacked in the goal stack, but their associated storage is in shared environments in the stack.

sharing and sharing+freeness analysis running in the PLAI framework [86, 105, 108, 106]. Speedups were obtained from simulations based on actual sequential execution of threads using the IDRA system [33].[4] A much more extensive study covering numerous domains and situations, a much larger class of programs, and the effects of the three annotation algorithms described in [107] (UDG/MEL/CDG), can be found in [27]. Although work still remains to be done, specially in the area of detecting non-strict independence, results compared encouragingly well with those obtained from studies of theoretical ideal speedups for optimal parallelizations, such as those given in [125].

Finally, experimental results regarding the run-time system can be found in [110]. Actual speedups obtained on the Sequent Balance and Symmetry systems were reported for the parallelized programs for different numbers of processors. Results were also compared to the performance of the sequential programs under both &-Prolog, SICStus Prolog, and Quintus Prolog. Attained performance was substantially higher than that of SICStus for a significant number of programs, even if running on only two processors. For programs showing no speedups, the sequential speed was preserved to within 10%. Furthermore, substantial speedups could even be obtained with respect to commercial systems such as Quintus, despite the sequential speed handicap of &-Prolog due to the use of a C-based bytecode interpreter.[5]

4. Dependent And-parallelism

As mentioned earlier, given two subgoals with data dependencies between them, they can be executed in parallel. However, this parallel execution has to be carried out under certain constraints since unrestricted parallel execution of two dependent subgoals can result in redundant computation being performed. This form of parallelism is known as *dependent and-parallelism.* The constraint that is usually laid down is that only one of the subgoals can bind the dependent variable (the producer subgoal), the other can only read its value (the consumer subgoal). Thus, the dependent variable becomes a one way communication channel (or a *stream*). Hence, dependent and-parallelism is also known as *Stream parallelism.* Two way communication between the two subgoals is also possible through the dependent variable. For example, given the goals `a(X), b(X)`, if `a` binds the first subgoal to a structure, say `f(Y)`, where `Y` is an unbound variable and `f(Y)` is then consumed by `b` which then binds the variable `Y` to some ground value. The ground value that `Y` gets bound to can in turn be consumed by goal `a`, completing the two way communication. This two way communication gives rise to a *coroutining* effect [55] which can lead to substantial improvement in efficiency in program execution [120].

As mentioned earlier, dependent and-parallelism can be exploited with and without coroutining. When exploited without coroutining, for example in the DDAS (Dynamic Dependent And-Parallel System) model [123, 124], it is very similar to independent and-parallelism and can be used to extract more and-parallelism from logic programs. This is further illustrated with a simple example. Consider the following program taken from [55]:

```
produce(0, Out) :- Out = [].
produce(N, Out) :-
     N > 0,
     N1 is N - 1,
```

[4]Note that simulations are better than actual executions for evaluating the amount of *ideal* parallelism generated by a given annotation, since the effects of the limited numbers of processors in actual machines can be factored out.

[5]Performance of such systems ranges from about the same as SICStus to to about twice the speed, depending on the program.

```
     Out = [ferrari | Ms],
     produce(N1, Ms).

consume([]).
consume([ferrari | Ms]) :-
     go_ride_ferrari,
     consume(Ms).

?-produce(10, X), consume(X).
```

When this goal is executed, as soon as `X` is partially bound to `[ferrari | Ms]` by the `produce` goal, the `consume` goal begins execution. Thus, while `produce` is busy generating more elements of stream `X`, `consume` can start processing the elements already generated, thereby giving rise to dependent and-parallelism. The `consume` goal will suspend if it attempts to access the binding of `Ms` and `Ms` is unbound. DDAS attempts to mimic Prolog semantics in execution as much as possible. It *dynamically* keeps track of the producer goal—the leftmost executing goal for a variable is the producer (hence the name Dynamic Dependent And-parallel System (DDAS)). In DDAS bindings can be communicated from goals to the left to those in the right, the reverse is not possible. Also, if the producer goal is non-deterministic, the consumer goal will be executed in its entirety for each alternative of the producer.

When exploited with coroutining (i.e., bindings are communicated not only from left to right but also from right to left), stream parallelism can be used for exploiting parallelism in reactive computations. Stream parallelism also arises in many problems that involve constrained search. Stream parallelism with coroutining is further illustrated via an example:

```
produce(0, Out) :- Out = [].
produce(N, Out) :-
     N > 0,
     N1 is N - 1,
     Out = [ferrari(Envelope) | Ms],
     spend(Envelope),
     produce(N1, Ms).

consume([]).
consume([ferrari(Envelope) | Ms]) :-
     Envelope = big_bucks,
     go_ride_ferrari,
     consume(Ms).

spend(big_bucks).
spend(small_bucks).

?-produce(10, X), consume(X).
```

This example is very similar to the previous example, except that now each element in the stream `X` is a structure that contains an unbound variable called `Envelope`. `Envelope` variable is bound by the consume goal, this value is then processed by produce in the goal spend. In traditional Prolog as well as DDAS, spend will create a choicepoint since the argument of spend gets bound only in the goal to the right of the produce goal in the top-level query. Coroutining based systems will, however, execute produce and consume concurrently, feeding the binding of Envelope from consume goal into produce goal in the backward direction, thus permitting a bidirectional communication of bindings. Stream parallel execution with coroutining can constrain the search space of both the producer and the consumer goals.

4.1. Problems in Implementation of Dependent And-parallelism

The main problem in implementing dependent and-parallelism is controlling the parallel execution of the consumer goal. That is, given two subgoals that share a variable X, one a producer of X and another a consumer, the execution of both can be initiated in parallel. However, one must make sure that the consumer goal computes only as far as it doesn't instantiate the dependent variable X to a non-variable binding. If it attempts to do so, it must suspend. It will be woken up only after a non-variable binding has been produced for X by the producer goal. Thus, the two main concerns in implementing dependent and-parallelism are:

1. Determining which instance of a given dependent variable is a producer instance and which instances are consumer instances;

2. Efficient mechanism for waking up suspended subgoals when the variables on which they were suspended are instantiated.

Dependent and-parallelism without coroutining is relatively easy to implement since in addition to the two problems mentioned above most other problems are similar to those faced by independent and-parallel systems [124] [6]. For instance, in the DDAS system [123] based on this idea, two subgoals that share a dependent variable are executed in parallel, independent of each other, until one of them accesses that variable. If the goal that accessed the variable is the consumer of the binding, and the variable is unbound, then it immediately suspends. It is woken up when the producer goal binds the variable to a non-variable value. Dependent and-parallelism exploited in this form is very similar to independent and-parallelism discussed earlier, because goals are executed in and-parallel as long as their activities are not in conflict with each other.

In the presence of coroutining, however, implementation of dependent and-parallelism becomes quite cumbersome, primarily due to the presence of non-determinism and interaction of the goals via the dependent variable. This is because if both the producer and the consumer goal of a binding of a variable are non-deterministic, then before any cooperation can take place between the producer and the consumer through the shared variable, the non-determinism should be somehow eliminated. There are three possible options:

(i). Employing "don't care" non-determinism, i.e., arbitrarily eliminating all but one choices for both the producer and the consumer goal;

(ii). letting the producer and consumer goals execute, as long as at most one branch will succeed for each; and,

[6] Note that the criteria for independent and-parallel systems also apply to those for dependent and-parallel systems.

(iii). creating a copy of the consumer goal for every alternative of the producer and vice versa (akin to computing a join) and letting the computation proceed in each such combination.

4.2. Committed Choice Languages

Option **(i)** is the simplest, and amounts to throwing out non-determinism from the logic programming language. This approach has been adopted in *Committed Choice Languages* [128, 136, 39, 55, 133]. Committed Choice Languages have some nice properties and are interesting in their own right. However, we will not discuss them any further over here. A detailed account can be found in [127] and also in a companion article by Saraswat and Tick in this volume.

4.3. The Basic Andorra Model

Option **(ii)** has been adopted in the Basic Andorra Model [69, 144] (and its implementation in the Andorra-I system [120]), Pandora [10], and P-Prolog [151][7]. In the Basic Andorra Model, goals can be executed ahead of their turn ("turn" in the sense of Prolog's depth first search) in parallel *if they are determinate*, i.e., if at most one clause matches the goal (the determinate phase). These determinate goals can be dependent on each other. If no determinate goals can be found for execution, a branch point is created for the leftmost goal in the goal list (non-determinate phase) and parallel execution of determinate goals along each alternative of the branch point continues. Dependent and-parallelism is obtained by having determinate goals execute in parallel. The different alternatives to a goal may be executed in or-parallel. Executing determinate goals (on which other goals may be dependent) eagerly also provides a coroutining effect which leads to the narrowing of the search space of logic programs. The Basic Andorra Model can be seen as adding non-determinism to a committed choice language with flat guards and thus subsumes both a non-deterministic logic programming language as well as a committed choice language such as FGHC [136]. A similar approach has been adopted in Pandora [10], which extends Parlog [39] by adding non-determinism to an otherwise committed choice language. In Pandora, clauses are classified as either *don't-care* or *don't-know*. Like the Basic Andorra Model, execution alternates between the *and-parallel phase* and the *deadlock phase*. In the and-parallel phase, all goals in a parallel conjunction are reduced concurrently. A goal for a don't-care clause may suspend on input matching if its arguments are insufficiently instantiated as in normal Parlog execution. A goal for a don't-know clause is reduced if it is determinate, like in the Basic Andorra Model. When none of the don't-care goals can proceed further and there are no determinate don't-know goals, the deadlock phase is activated (Parlog would have aborted the execution in such a case) that chooses one of the alternatives for a don't-know goal and proceeds. If this alternative were to fail, backtracking would take place and another alternative will be tried (potentially, the multiple alternatives could be tried in or-parallel).

The Andorra-I System

The Andorra-I system is an implementation of the Basic Andorra Model. Andorra-I exploits determinate dependent and-parallelism together with or-parallelism. Implementation of or-parallelism is very similar to that in Aurora and is based on Binding Arrays [140, 142]. Due to its similarity to Aurora as far as or-parallelism is concerned, Andorra-I is able to use the schedulers that have been built for Aurora. The current version of Andorra-I is compiled [152] and is a descendent of the earlier interpreted version [120].

[7] It should be noted that some of the ideas of the Basic Andorra Model come from P-Prolog [151].

As a result of exploitation of determinate dependent and-parallelism and the accompanying coroutining, not only Andorra-I can exploit parallelism from logic programs it can also reduce the number of inferences performed to compute a solution. This is because execution is divided into two phases—determinate and non-determinate—execution of non-determinate phase is begun only after all unconditional choices have been made in the determinate phase, i.e., after all determinate goals in the current goal list, irrespective of their order in this list, have been solved. Any goal that is non-determinate (that is, has more than one potentially matching clauses) will be suspended in the determinate phase. Solving determinate goals early constrains the search space much more than if one used the standard sequential Prolog execution order (for example, for the 8-queen's program the search space is reduced by 44%, for the zebra puzzle by 70%, etc.). Note that execution of a determinate goal to the right may bind variables which in turn may make non-determinate goals to their left determinate. The Andorra-I compiler performs an elaborate determinacy analysis of the program and generates code so that the determinate status of a goal is determined as early as possible at runtime [121].

The Andorra-I system supports full Prolog, in that execution can be performed in a way that sequential Prolog semantics is preserved [121]. This is achieved by analysing the program at compile-time and preventing early (i.e., out of turn) execution of those determinate goals that may contain extralogical predicates. These goals will be executed only after all goals to the left of them have been completely solved.

The Andorra-I system speeds-up execution in two ways: (i) by reducing the number of inferences performed at run-time; and, (ii) by exploiting dependent and-parallelism and or-parallelism from the program. Very good speed-ups have been obtained by Andorra-I for a variety of benchmark programs. The Andorra-I engine [122, 152] combines the implementation techniques used in implementing Parlog, namely the JAM system [46], and the Aurora system [99]. The Andorra-I system had to overcome many problems before an efficient implementation of its engine could be realized. Chief among them was a backtrackable representation of the goal list (since goals are solved out of order, they should be inserted back in the goal list if backtracking were to take place; recall that there is no backtracking in Parlog so this is not a problem in JAM). The Andorra-I system was the first one to employ the notion of teams of processors, where available processors are divided into teams, and each team shares all the data-structures (except the queue of ready-to-run goals). Or-parallelism is exploited at the level of teams (i.e., each team behaves like a single Aurora worker). Determinate dependent and-parallelism is exploited by processors within a team, i.e., processors within a team will co-operatively solve a goal along the or-branch that the team has picked up. There are separate schedulers for or-parallel work and dependent and-parallel work, and overall work balancing is achieved by a top-scheduler [54]. The notion of teams of processors was also adopted by the ACE [60] and the PBA [62, 61] models that combine or-parallelism with independent and-parallelism while preserving sequential Prolog semantics. A parallel system incorporating the Basic Andorra Model has also been implemented by Palmer and Naish [112].

4.4. The Extended Andorra Model

Option **(iii)** has been adopted in the Extended Andorra Model [144, 70, 68], and in the Andorra Kernel Language (AKL) [70]. (Note that the Extended Andorra Model (EAM) and the Andorra Kernel Language are very similar in spirit to each other, the major difference being that while the EAM strives to keep the control as implicit as possible, AKL gives the programmer complete control over parallel execution through *wait guards*. In the description below we use the term Extended Andorra Model in a generic sense, to include models such as AKL as well.)

The Extended Andorra Model is an extension of the Basic Andorra Model. The Extended Andorra Model goes a step further and removes the constraint that goals become determinate before they can execute ahead of their turn. However, goals which do start computing ahead of their turn must compute only as far as the (multiple) bindings they produce for the uninstantiated variables in their arguments are consistent with those produced by the "outside environment." If such goals attempt to bind a variable in the outside environment, they suspend. Once a state is reached where execution cannot proceed, then each suspended goal which is a producer of bindings for one (or more) of its argument variables "publishes" these bindings to the outside environment. For each binding published a copy of the consumer goal is made and its execution continued. (This operation of "publication" and creation of copies of the consumer is known as a "non-determinate promotion" step.) The producer of bindings of a variable is typically the goal where that variable occurs first. However, if a goal produces only a single binding (i.e., it is determinate) then it doesn't need to suspend, it can publish its binding immediately, thus automatically becoming the producer for that goal irrespective of whether it contains the left most occurrence of that variable or not (as in Basic Andorra Model). An alternative way of looking at the EAM is to view it as an extension of the Basic Andorra model where non-determinate goals are allowed to execute locally so far as they do not influence the computation going on outside them. This amounts to including in the Basic Andorra Model the ability to execute independent goals in parallel.

There are many manifestations of the Extended Andorra Model, but the essential ideas are summarized below. Consider the following very simple program:

```
p(X, Y) :- X = 2, m(Y).
p(X, Y) :- X = 3, n(Y).
q(X, Y) :- X = 3, t(Y).
q(X, Y) :- X = 3, s(Y).
r(Y) :- Y = 5.
?- p(X, Y), q(X, Y), r(Y).
```

When the top-level goal begins execution, all three goals will be started concurrently. Note that variables `X`, and `Y` in the top-level query are considered to be in the environment "outside" of goals `p`, `q`, and `r` (this is depicted by existential quantification of `X` and `Y` in figure 3). Any attempt to bind these variables from inside these goals will lead to the suspension of these goals. Thus, as soon as these three goals begin execution, they immediately suspend since they try to constrain either `X` or `Y`. Of these, `r` is allowed to proceed and constrain `Y` to value 5, because it binds `Y` determinately. Since `p` will be reckoned the producer goal for the binding of `X`, it will continue as well and publish its binding. The goal `q` will, however, suspend since it is neither determinate nor the producer of bindings of either `X` or `Y`. To resolve the suspension of `q` and make it active again, the *non-determinate* promotion step will have to be performed. The non-determinate promotion step will match all alternatives of `p` with those for `q`, resulting in only two combination remaining active (the rest having failed because of non-matching bindings of `X`). These steps are shown in figure 3.

The above is a very coarse description of the Extended Andorra Model, a full description of the model is beyond the scope of this paper. More details can be found elsewhere [144, 70, 68]. The EAM is a very general model, more powerful than the Basic Model, since it can narrow down the search even further by local searching. It also exploits more parallelism since it exploits all major forms of parallelism present in logic programs: or-, independent-and, and dependent-and parallelism, including both determinate and non-determinate dependent-and parallelism. A point to note is that the EAM does not distinguish between independence and dependence of conjunctive goals: it tries to execute

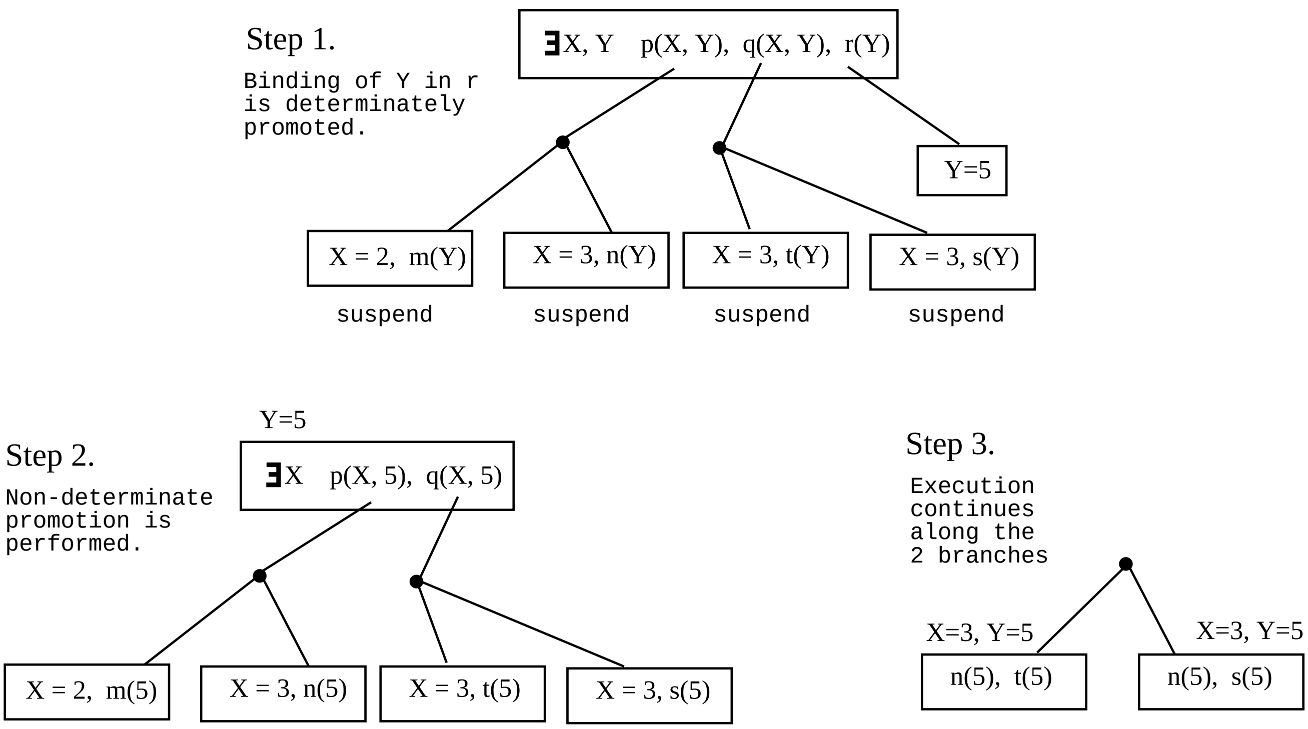


Figure 3: Execution in EAM

them in parallel whenever possible. Also note that the Extended Andorra Model subsumes both the committed choice logic programming (with non-flat as well as flat guards) and non-deterministic logic programming.

The generality and the power of the Extended Andorra Model makes its efficient implementation quite difficult. A sequential implementation of one instance of the EAM (namely, the Andorra Kernel Language or AKL) has been implemented at Swedish Institute of Computer Science [87]. A parallel implementation has also been undertaken by Moolenaar and Demoen [102]. Although AKL includes non-determinism, it differs from Prolog both in syntax and semantics. However, automatic translators that transform Prolog program into AKL programs have been constructed [28]. The Extended Andorra Model and its implementation is still being researched.

5. Combining Or-parallelism and And-parallelism

As one can gather, parallel systems that exploit only one form of parallelism from logic programs have been efficiently implemented and reached a mature stage. Roughly speaking, that sums up the state-of-the-art of parallel logic systems today. Although, very general models for parallel execution of logic programs have been proposed, e.g., the Extended Andorra Model, they have not yet been efficiently realized. A compromise approach that many researchers have been pursuing, long before the EAM was conceived, is that of combining techniques that have been effective in single-parallelism systems to obtain efficient systems that exploit more than one sources of parallelism in logic programs[8]. The

[8]Simulations have shown that indeed better speed-ups will be achieved if more than one sources of parallelism are exploited [124, 125].

implementation of the Basic Andorra Model [69, 144], namely, Andorra-I [122] can be viewed in that way since it combines (determinate) dependent and-parallelism, implemented using techniques from JAM [46], with or-parallelism, implemented using Binding Arrays technique [99, 142]. Likewise, the PEPSys model [147, 16], the AO-WAM [57], ROPM [89, 115, 90], ACE [60, 61], and the PBA models [62, 61] have attempted to combine independent and-parallelism with or-parallelism; these models differ with each other in the environment representation technique they use for supporting or-parallelism. One should also note that, in fact, Conery's model described earlier is an and-or parallel model [44] since solutions to goals may be found in or-parallel. Models combining independent and-parallelism, or-parallelism and (determinate) dependent and-parallelism have also been proposed [63]. The abstract execution models that these systems employ (including those that only exploit a single source of parallelism) can be viewed as subsets of the EAM with some restrictions imposed, although this is not how they were conceived. In subsequent subsections, we review these various systems that have been proposed for combining more than one sources of parallelism.

The problems faced in implementing combined and- and or-parallel system are essentially the sum of problems faced in implementing and-parallelism and or-parallelism individually. However, sometimes in the combined system the problems faced in one may worsen those faced in the other, especially those regarding control of execution, representation of environment, and memory management.

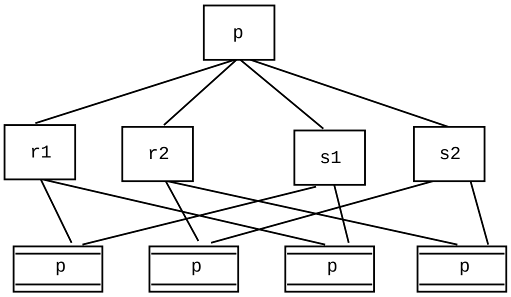
An issue that combined systems also have to face is whether they should support sequential Prolog semantics. The alternatives to supporting Prolog semantics are: (i) consider only pure Prolog programs for parallel execution; this was the approach taken by many early proposals, e.g., AO-WAM [57] and ROPM [89]; or, (ii) devise a new language that will allow extra-logical features but in a controlled way, e.g., PEPSys [17, 147]. The disadvantage with both these approaches is that existing Prolog programs cannot be parallelized. Recently, however, newer approaches have been developed that allow support for Prolog's sequential semantics even during parallel execution [63, 60, 62, 121, 64].

Another issue that arises in systems that exploit independent and-parallelism is whether to *recompute* solutions of independent goals, or to *reuse* them. For example, consider the following program for finding "cousins at the same generation" taken from [137]:

```
sg(X, X) :- person(X).
sg(X, Y) :- parent(X, Xp), parent(Y, Yp), sg(Xp, Yp).
```

In executing a query such as `?- sg(fred, john)` under a (typical) purely or-parallel or a purely and-parallel or a sequential implementation, the goal `parent(john, Yp)` will be recomputed for every solution to `parent(fred, Xp)`[9]. This is clearly redundant since the two parent goals are independent of each other. It would be better to compute their solutions separately, take a crossproduct (join) of these solutions, and then try the goal `sg(Xp, Yp)` for each of the combinations. In general, for two independent goals G_1 and G_2 with m and n solutions respectively, the cost of the computation can be brought down from $m * n$ to $m + n$ by computing the solutions separately and combining them through a crossproduct (assuming the cost of computing the crossproduct is negligible). However, for independent goals with very small granularity, the gain from solution sharing may be overshadowed by the cost of computing the cross-product etc., therefore, such goals should either be executed serially, or they should be recomputed instead of being shared [14]. Independent goals that contain side-effects and extra-logical predicates should also be treated similarly [14, 16]. This is because the number of times, and the order in which, these side-effects will be executed in the solution sharing approach will be different from that in sequential Prolog execution, altering the meaning of the logic program. Thus,

[9] A purely and-parallel system can avoid recomputation of independent goals but most existing ones do not.



Clause p consists of the AND-parallel goals r and s with two solutions each. The join cells are marked by double horizontal bars and their last-common-hash-window.

Figure 4: Join Cells

if we were to support Prolog's sequential semantics in such parallel systems, independent goals will have to be recomputed. This is indeed the approach adopted by systems such as ACE [60] and the PBA model [61], which are based on an abstraction called Composition-tree that represents Prolog's search tree in a way that or-parallelism and independent and-parallelism become explicitly apparent in the structure of the tree itself [62, 61].

We now briefly describe the systems that combine more than one sources of parallelism in logic program.

The PEPSys Model

The PEPSys model [147, 16] combines and- and or-parallelism using a combination of techniques of time-stamping and *hashing windows* for maintaining multiple environments. In PEPSys, each node in the execution tree has a process associated with it. Each process has its own hash-window. All the bindings of conditional variables generated by a process are time-stamped and stored in that process' hash-window. Any PEPSys process can access the stacks and hash-windows of its ancestor processes. The time-stamp associated with each binding permits it to distinguish the relevant binding from the others in the ancestor processes' stacks and hash-windows.

Independent and-parallel goals have to be explicitly annotated by the programmer. The model can handle only two and-parallel subgoals at a time. If more than two subgoals are to be executed in and-parallel, the subgoals are nested in a right associative fashion. If or-parallelism is nested within and-parallelism then and-parallel branches can generate multiple solutions. In this case the cross-product (join) of the left-hand and right-hand solution sets has to be formed. For each combination of solutions in the cross-product set a process is created. Each such process can communicate with its two ancestor processes (one corresponding to the left and-branch and other corresponding to the right and-branch) that created the corresponding solution. Access to the bindings of these ancestor processes is handled by *join cells*. A join cell contains a pointer to the hash-window of the left and-branch process and to the hash-window of the right and-branch process. It also contains a pointer to the hash-window that was current at the time of the and-parallel split (Figure 4). Looking up a variable binding from a goal after the and-parallel join works as follows: the linear chain of hash-windows is followed in the usual way until a join cell is reached. Now a branch becomes necessary. First the right-hand process is searched by following the join-cell's right hand side hashed window chain. When the last-common-hash-window is encountered control bounces back to the join-cell and the left branch is searched.

The basic scheme for forming the cross-product, gathering the left-hand solutions and the right-hand solutions in solution-lists and eagerly pairing them, relies on the fact that all solutions to each side are computed incrementally and co-exist at the same time in memory to be paired with newly arriving solutions to the other side. However, if all solutions to the and-parallel goal on the right have been found and backtracked over, and there are still more solutions for the and-parallel goal to the left remaining to be discovered, then the execution of the right goal will be restarted after discovery of more solutions of the goal to the left (hence PEPSys uses a combination of goal-reuse and goal-recomputation).

The PEPSys model uses time-stamping and hash windows for environment representation. This doesn't permit constant time access to conditional variables. Therefore, access to conditional variables is expensive. However, environment creation is a constant time operation. Also a processor doesn't need to update any state when it switches from one node to another since all the information is recorded with the or-tree. In PEPSys sharing of and-parallel solutions is not complete because the right hand and-parallel subgoal may have to be recomputed again and again. Although recomputing leads to economy of space, its combination with cross-product computation via join cells makes the control algorithm very complex. Due to this complexity, and-parallelism was not incorporated in implementation of the PEPSys system. PEPSys was later modified and evolved into the ElipSys System [139]: the hashed windows have been replaced with Binding Arrays and it has also been extended to handle constraints.

The ROPM Model

ROPM (Reduce-Or Parallel Model) was devised by Kalé in his Ph.D. Thesis [89]. The model is based on a modification of the And-Or tree, called the Reduce-Or Tree. There are two types of nodes in the a Reduce-Or tree, the Reduce-nodes and the Or-nodes. The Reduce nodes are labeled with a query (i.e., a set of goals) and the or-nodes are labeled with a single literal. To prevent global checking of variable binding conflicts every node in the tree has a *partial solution set* (PSS) associated with it. The PSS consists of a set of substitutions for variables that make the subgoal represented by the node true. Every node in the tree contains the bindings of all variables that are either present in the node or are reachable through this node. The Reduce-Or tree is defined recursively as follows [89]:

1. A Reduce node labeled with the top level query and with an empty PSS is a Reduce-Or tree.

2. A tree obtained by extending a Reduce-Or tree using one of the rules below is a Reduce-Or tree:

   a. Let Q be the set of literals in the label of a Reduce node R. Corresponding to any literal L in Q, one may add an arc from R to a new Or-node O labeled with *an instance* of L. The literal must be instantiated with a consistent composition of the substitutions from the PSS of subgoals preceding L in Q.

   b. To any Or-node, labeled with a goal G, one may add an arc to a new REDUCE node corresponding to some clause of the program, say C, whose head unifies with G. The body of C with appropriate substitutions resulting from the head unification becomes the label of the new Reduce node (say) R. If the query is empty, i.e., the clause is a 'fact', the PSS associated with R becomes a singleton set. The substitution that unifies the goal with the fact becomes the only member of the set.

   c. Any entry from the PSS of the Reduce node can be added to the PSS of its parent Or-node. A substitution can be added to the PSS of a Reduce node R representing a composite goal Q if it is a consistent composition of the substitutions, one for each literal of Q, from the PSS's of the children (Or-nodes) of R.

```
quicksort(L, Sorted) :- partition(L, L1, L2),
                        quicksort(L1, Sorted1), quicksort(L2, Sorted2),
                        append(Sorted1, Sorted2, Sorted).
```

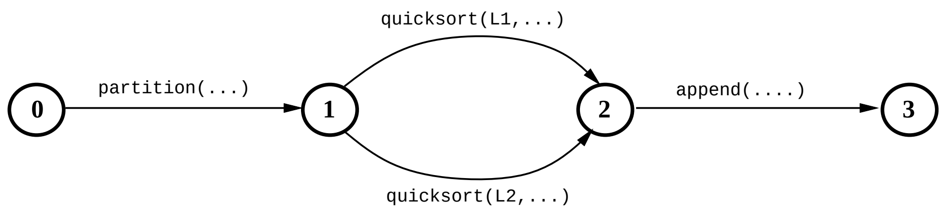


Figure 5: An Example Data Join Graph

ROPM associates a Reduce Process with every Reduce node and an Or Process with every Or-node. The program clauses in ROPM are represented as Data Join Graphs (DJGs), in which each arcs of the graph denotes a literal in the body of the clause (Figure 5).

DJGs are a means of expressing and-parallelism and are similar in spirit to Conery's data-flow graph. A set of variable binding tuples, called a relation (PSS), is associated with each arc and each node of the DJG. The head of a clause is matched with a subgoal by an Or process. A reduce process is spawned to execute the body of the clause. In the reduce process, whenever a binding tuple is available in the relation of a node k, subgoals corresponding to each of the arcs emanating from k will be started, which leads to the creation of new Or processes. When a solution for any subgoal arrives, it is inserted in corresponding arc relation. The node relation associated with a node n is a join of the arc-relations of all its incoming arcs. So when a solution tuple is inserted in an arc-relation, it is *joined* with all the solution tuples in the arc relations of its parallel arcs that originated from the same tuple in the lowest common ancestor node of the parallel arcs [116]. A solution to the top level query is found, when the PSS of the root-node becomes non-empty.

In ROPM multiple environments are represented by replicating them at the time of process creation. Thus each Reduce- or Or-process has its own copy of variable bindings (the Partial Solution Set above) which is given to it at the time of spawning. Thus process creation is an expensive operation. ROPM is process based model rather than a stack based one. As a result, there is no backtracking, and hence no memory reclamation that is normally associated with backtracking. Computing the join is an expensive operation since the actual bindings of variables have to be cross-produced to generate the tuple relations of the node (as opposed to using symbolic addresses to represent solutions, as done in PEPSys [147] and AO-WAM [57]), and also since the sets being cross-produced have many redundant elements. Much effort has been invested in eliminating unnecessary elements from the constituent sets during join computation [116]. However, the computation of the join is made a little efficient by using structure sharing. One advantage of the ROPM model is that if a process switches from one part of the reduce-or tree to another, it doesn't need to update its state at all since the entire state information is stored in the tree.

ROPM model has been implemented in the ROLOG system on a variety of platforms. However, although ROLOG yields very good speed-ups, its absolute performance does not compare very well with other parallel logic programming systems, chiefly because it is a process based model and uses the expensive mechanism of environment closing [115, 43] for multiple environment representation.

The AO-WAM model

The AO-WAM model [57, 59] combines or-parallelism and independent and-parallelism. Independent and-parallelism is exploited via CGEs of &-Prolog and solutions to independent goals are reused. To represent multiple or-parallel environments in the presence of independent and-parallelism, the AO-WAM extends the binding arrays technique [140, 142].

The model works by constructing an *Extended And-Or Tree*. Execution continues like a standard or-parallel system until a CGE is encountered, at which point a *cross-product node* that keeps track of the control information for the and-parallel goals in the CGE is added to the or-parallel tree. New or-parallel sub-trees are started for each independent and-parallel goals in the CGE. As solutions to goals are found, they are combined with solutions of other goals to produce their cross-product. For every tuple in the cross-product set, the continuation goal of the CGE is executed (i.e., its tree is constructed and placed as a descendent of the cross-product node).

As far as maintenance of multiple environments is concerned, each processor has its own binding array. In addition, each processor has a *base array*. Conditional variables are bound to a pair of numbers consisting of an offset in the base array and a relative offset in the binding array. Given a variable bound to the pair `<i, v>`, the location `binding_array[base_array[i] + v]` will contain the binding for that variable. For each and-parallel goal in a CGE, a different base-array index is used. Thus the binding array contains a number of smaller binding arrays, one for each and-parallel goal, that are accessible through the base array. When a processor produces a solution for an and-parallel goal and computes its corresponding cross-product tuples, then before it can continue execution with the continuation goal of the CGE, it has to load all the conditional bindings made by other goals in the CGE that are present in the selected tuple (See Figure 6). Also, on switching nodes, a processor must update its binding array and base array with the help of the trail, like in Aurora.

The ACE Model

ACE (And/Or-parallel Copying-based Execution of logic programs) is another model that has been proposed for exploiting or- and independent and-parallelism simultaneously. ACE employs stack-copying developed for MUSE to represent multiple environments. And-parallelism is exploited via CGEs. ACE employs goal recomputation and thus can support sequential Prolog semantics. ACE can be considered as subsuming &-Prolog and MUSE. The implementation can be envisaged as multiple copies of &-Prolog [110] running in parallel with each other, where each copy corresponds to a different solution to the top-level query (analogous to the view of MUSE as multiple sequential Prologs running in or-parallel). When there is only and-parallelism or or-parallelism, ACE behaves exactly like &-Prolog and MUSE respectively. When there is or-parallelism and independent and-parallelism present together, both are simultaneously exploited.

Multiple environments are maintained by stack-copying as in MUSE. In ACE, available processors are divided into teams like Andorra-I, where different teams execute in or-parallel with each other while different processors within a team execute in independent and-parallel with each other. A team executes the top level query in and-parallel like &-Prolog until a choicepoint is created, at which point other teams may steal the untried alternatives from this choicepoint. Before doing so, the stealing team has to copy the appropriate stacks from the team from which the alternative was picked. When the choicepoint from which the alternative is picked is not in the scope of any CGE, all the operations are very similar to those in MUSE. However, the situation is slightly more complex when an alternative from a choicepoint in the scope of a CGE is stolen by a team. To illustrate this, consider the case where a team selects an untried alternative from a choice point created during execution of a goal g_i inside the

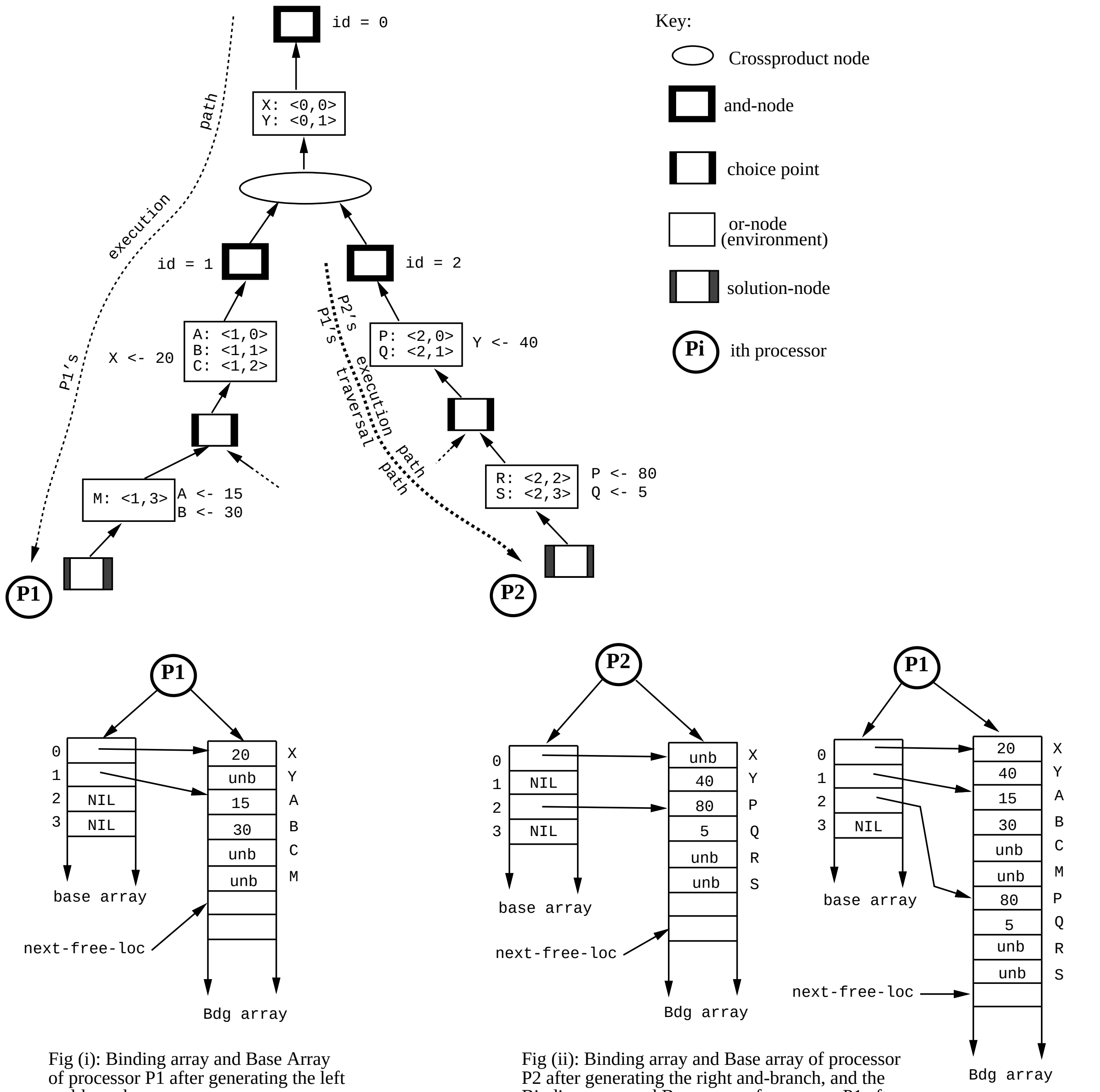


Fig (i): Binding array and Base Array of processor P1 after generating the left and-branch.

Fig (ii): Binding array and Base array of processor P2 after generating the right and-branch, and the Binding array and Base array of processor P1 after loading the bindings from the right and-branch. P1 is now ready to proceed with the sequential goal after the CGE that gave rise to the cross-product node.

Figure 6: Execution in the AO-WAM

CGE ($true => g_1\& \ldots \& g_n$). This team will copy all the stack segments in the branch from the root to the CGE including the *parcall* frame[10]. It will also have to copy the stack segments corresponding to the goals $g_1 \ldots g_{i-1}$ (i.e., goals to the left). The stack segments up to the CGE need to be copied because each different alternative within g_i might produce a different binding for a variable, X, defined in an ancestor goal of the CGE. The stack segments corresponding to goals g_1 through g_{i-1} have to be copied because execution of the goals following the CGE might bind a variable defined in one of the goals $g_1 \ldots g_{i-1}$ differently. The stack segments of the goal g_i from the CGE up to the choicepoint from where the alternative was taken also need to be copied (note that because of this, an alternative can be picked up for or-parallel processing from a choicepoint that is in the scope of the CGE only if goals to the left, i.e., $g_1 \ldots g_{i-1}$, have finished). The execution of the alternative in g_i is begun, and when it finishes, the goals $g_{i+1} \ldots g_n$ are started again so that their solutions can be recomputed. Because of recomputation of independent goals ACE can support sequential Prolog semantics [61, 60, 64].

Paged Binding Array based Model

ACE can be seen as subsuming &-Prolog with MUSE while preserving Prolog semantics. In a similar vain, one can combine &-Prolog with Aurora while preserving Prolog semantics. However, as in the case of AO-WAM, the binding array technique has to be extended to accommodate independent and-parallelism. The Paged Binding Array (PBA) based model does this by dividing the binding array into *pages* and maintaining a *Page Table* with a binding array. Like ACE, available processors are divided into teams, where different teams work in or-parallel with each other, while different processors within a team work in independent and-parallel. Different and-parallel computations within an or-parallel computation share the same binding array (thus the paged binding array and the page table is common to all processors in a team), however, each one of them will use a different page, requesting a new page when it runs out of space in the current one. Like AO-WAM, conditional variables are bound to a pair of numbers where the first element of the pair indicates the page number in the binding array, and the second element indicates the offset within this page.

The PBA based model also employs recomputation of independent goals, and therefore can support Prolog semantics [61, 64]. Thus, when a team steals an alternative from a goal inside a CGE, then it updates its binding array and page table so that the computation state that exists at the corresponding choicepoint is reflected in the stealing team. The team then restarts the execution of that alternative, and of all the goals to the right of the goal in the CGE that led to that alternative. In cases, where the alternative stolen is from a choicepoint outside the scope of any CGE, the operations involved are very similar to those in Aurora.

The Paged Binding Array is a very versatile data-structure and can also be used for implementing other and-or parallel systems [65].

So far we have only considered models that combine or- and independent and-parallelism. There are models that combine independent and-parallelism and dependent and-parallelism such as DDAS [123], described earlier, as well as models that combine or-parallelism and dependent and-parallelism such as Andorra-I [120]. Other combined independent and- and -or parallel models have also been proposed [21]. We next describe a model, called IDIOM [63], that combines or-parallelism, independent and-parallelism and (determinate) dependent and-parallelism.

IDIOM

[10]The parcall frame [79] records the control information for the CGE and its independent and-parallel goals.

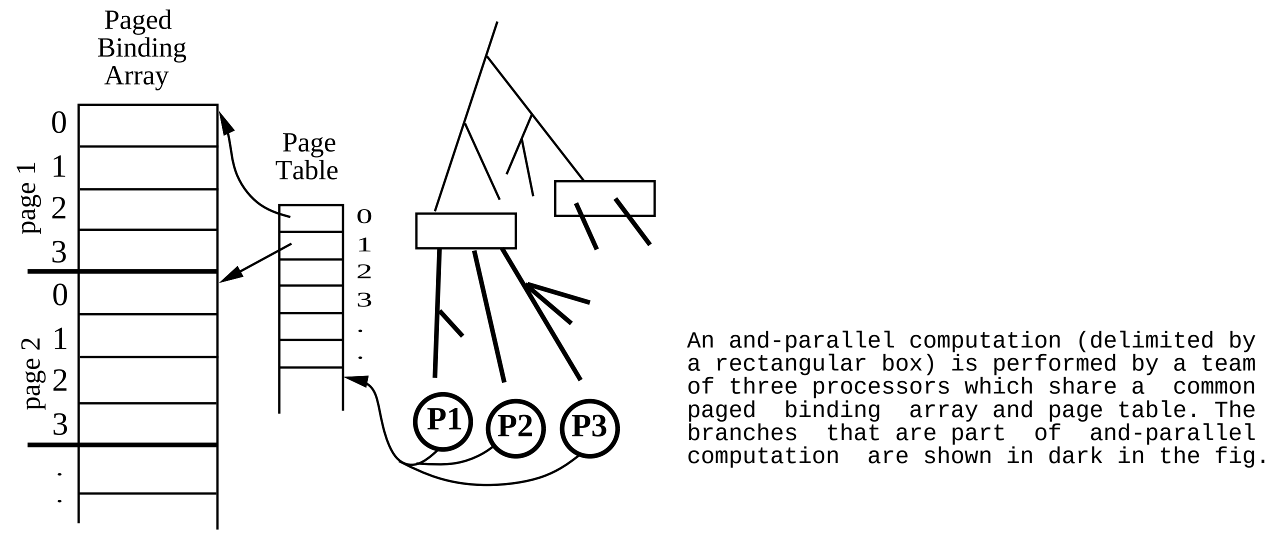


Figure 7: The Paged Binding Array

IDIOM is a model that has been proposed to combine or-parallelism, independent and-parallelism and (determinate) dependent and-parallelism [63]. Essentially, it adds dependent and-parallel execution of determinate goals to AO-WAM. Other models that combine independent and-parallelism and or-parallelism can be extended in a similar way. Like the Andorra-I system, IDIOM performs execution in multiple phases. The different execution phases can be depicted as shown in Figure 8. There are three phases: the Dependent And-parallel Phase (DAP), the Independent And-parallel (IAP) phase, and the Or-parallel (ORP) phase. In the DAP phase all goals that can be determinately reduced are evaluated in parallel (including those inside the CGEs) until none are left. The leftmost goal is then examined to see if it is (i) a simple goal; or (ii) a CGE. In case (i) the ORP phase is entered while in case (ii) the IAP phase is entered. In the ORP phase the first alternative to the goal is selected (other alternatives are made available for or-parallel execution), head unification is performed, and the DAP phase is entered again. In the IAP phase, firstly the condition in the CGE is evaluated. If true, the components of the CGE are made available for independent and-parallel processing. In practice, the leftmost component is selected immediately and the other components are made available for parallel execution. The ORP phase will then be entered to process the leftmost goal of the selected component in or-parallel. If the condition in the CGE evaluates to false, the ORP phase is entered immediately to process the leftmost goal in the CGE in or-parallel. Execution of the continuation of the CGE then continues in the DAP phase.

An implementation for IDIOM has been proposed using binding arrays that essentially combines the implementations of Andorra-I and AO-WAM. One can extend other independent and-or parallel systems with determinate dependent and-parallelism in a similar way [66]. The main advantage of IDIOM is that it permits exploitation of the three major forms of control parallelism using well-understood techniques developed for purely or- and purely and-parallel systems.

6. Implementation and Efficiency Issues in Logic Programming

Engineering an efficient, practical parallel logic programming system is by no means an easy task[11].

[11] For instance, many person-years of efforts have been spent in building some of the existing systems, such as MUSE,

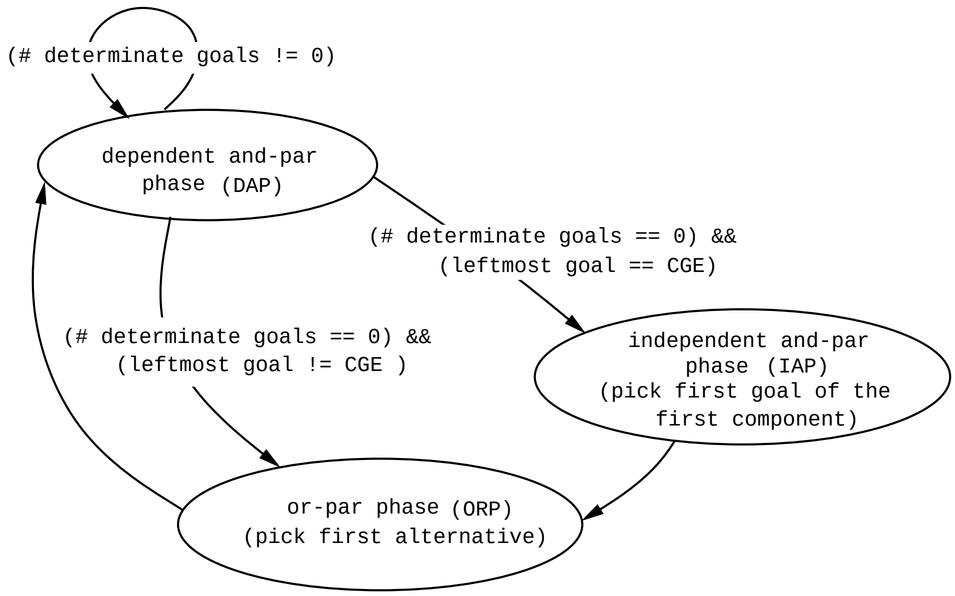


Figure 8: Control in IDIOM

There are numerous issues one has to consider, some of the broad ones are discussed below:

6.1. Process based vs. Processor based

Broadly speaking there have been two approaches that have been taken in implementing parallel logic programming systems which we loosely call the *Process based approach* and the *Processor based approach* respectively.

In the process based approaches, prominent examples of which are Conery's And-Or Process Model [44] and the Reduce-Or Process Model [89], a process is created for every goal encountered during execution. These processes communicate bindings and control information to each other to finally produce a solution to the top-level query. Process based approaches have also been used for implementing committed choice languages [128]. Process based approaches are suited for implementation on non-shared memory MIMD processors[12], at least from a conceptual point of view, since different processes can be mapped to different processors at runtime quite easily.

In processor based approaches, multiple threads are created that are executed in parallel to produce answers to the top level query. Typically, each thread is a WAM-like processor. Examples of processor based systems are Aurora, MUSE, &-Prolog, Andorra-I, PEPSys, AO-WAM, DDAS, ACE, PBA, etc. Processor based systems are more suited for shared memory machines, although techniques like MUSE show a high degree of locality in memory reference behavior and hence are suited for non-shared memory machines as well [3, 7]. As has been shown by the ACE model, MUSE's stack-copying technique can be applied to and-or parallel systems as well, so one can envisage implementing a processor based system on a non-shared memory machine using stack-copying [67]. Alternatively, one could employ novel scalable virtual shared memory architectures that have been proposed and built recently [145].

A parallel logic programming system should possess the following two properties:

Aurora, Andorra-I, and &-Prolog.

[12]Some more proposals for parallel execution of logic programs can be found in [88].

- On a single processor, the performance of the parallel system should be comparable to sequential logic programming implementations (i.e., there should not be any slow down compared to a sequential system). Systems such as MUSE, Aurora and &-Prolog indeed achieve this goal.
- The parallel system should be able to take advantage of the sequential compilation technology [141, 2, 138] that has advanced rapidly in the last decade.

Experience has shown that process based system lose out on both the above counts. The process based approach is good for committed choice languages, where there is no backtracking involved, however, this is not true for non-deterministic parallel logic programming. Presence of backtracking makes the process model too complex for non-deterministic parallel logic programming. Further, the process based approach exploits parallelism at a level that is too fine grained, resulting in high parallel overhead and unpromising absolute performances (but good speed-ups because the large parallel overhead gets evenly distributed!). Current processor based systems are not only highly efficient, they can easily assimilate any future advances that will be made in the sequential compilation technology. However, it must be pointed out that increasing the granularity of processes to achieve better absolute performance was indeed attempted by Kalé et al for ROPM (a process-based model) with good results [90, 117].

6.2. Memory Management

Memory management, or managing the memory space occupied by run-time data-structures such as stacks, heaps, etc., is an issue that needs to be tackled in any parallel system. In parallel logic programming systems memory management is further complicated due to the presence of backtracking that may occur on failure of goals.

In sequential Prolog implementations, memory is efficiently utilized because the search tree is constructed in a depth-first order, so that at any given moment a single branch of the tree resides in the stack. The following two rules always hold in a traditional sequential systems:

1. If a node n_1 in the search tree is in a branch to the right of another branch containing node n_2, then the data-structures corresponding to node n_2 would be reclaimed before those of n_1 are allocated.

2. If a node n_1 is the ancestor of another node n_2 in the search tree, then the data-structures corresponding to n_2 would be reclaimed before those of n_1.

As a result of these two rules, space is always reclaimed from the top of the stacks during backtracking in sequential depth first search based logic programming systems. However, in parallel logic programming systems, these rules may not hold, because two branches may be simultaneously active due to or-parallelism (making rule 1 difficult to enforce), or two conjunctive goals may be simultaneously active due to and-parallelism (making rule 2 difficult to enforce). Of course, in a parallel logic system, usually, each processor has its own set of stacks (the multiple stacks are referred to as a *cactus* stack since each stack corresponds to a part of the branch of the search tree), so it is possible to enforce the two rules in each stack to ensure that space is reclaimed only from the top of individual stacks. If we impose this restriction then while memory management becomes easier, some parallelism may be lost since an idle processor may not be able to pick available work in a node because doing so will violate this restriction. If we do not impose this restriction, then we are forced to deal with the "garbage slot" problem (namely, a data-structure that has been backtracked over is trapped in the stack below a goal that is still in use) and the "trapped goal" problem (namely, an active goal is trapped below another, and there is no space contiguous to this active goal to expand it further) [78], which results in the LIFO nature of stacks being destroyed.

The approach taken by many systems is to allow trapped goals and garbage slots in the stacks. Space needed to expand a trapped goal further is allocated at the top of the stack (resulting in stack-frames corresponding to a given goal becoming non-contiguous). Garbage slots created are marked as such, and are reclaimed when everything above them has also turned into garbage. This technique is employed in the Aurora, &-Prolog, and Andorra-I systems. In Aurora the garbage slot is referred to as a *ghost node*. If garbage slots are allowed, then the system will use up more memory, but work-scheduling becomes simpler and processing resources are utilized more efficiently.

6.3. Work Scheduling

The Work Scheduler, or the software that matches available work with processors, is a very important component of a parallel system. Parallel logic programming systems are no exceptions. If a parallel logic system is to obey Prolog semantics then scheduling becomes even more important, because in such a case, for or-parallelism, the scheduler should prefer goals in the left branches of the search tree to those in the branches to the right, while for and-parallelism prefer goals to the left over those to right. In parallel systems that support cuts, work that is not in the scope of any cut should be preferred over work that is in the scope of a cut, because it is likely that the cut may be executed causing a large part of the work in its scope to go wasted [8, 20, 129].

The scheduler is also influenced by how the system manages its memory. For instance, if the restriction of only reclaiming space from the top of a stack is imposed and garbage slots/trapped goals are not allowed, then the scheduler has to take this into account and at any given moment schedule only those goals that meet these criteria.

Schedulers in systems that combine more than one form of parallelism have to figure out how much of the resources should be committed to exploiting a particular kind of parallelism. For example, in Andorra-I and ACE systems, that divide available processors into teams, the scheduler has to determine the sizes of the teams, and decide when to migrate a processor from a team that has no work left to another that does have work, and so on [54].

The fact that Aurora, quite a successful or-parallel system, has about five schedulers built for it [31, 19, 129, 29], is a testimony to the importance of work-scheduling for parallel logic programming systems. Design of efficient and flexible schedulers is still a topic of research [54].

6.4. Granularity

Granularity of computation, or the average amount of work done between two calls to the scheduler by the processor, is another aspect that is important for parallel system design. We would like the granularity of computation to be large, so that the scheduling overhead is a small fraction of the total work done by a processor. Granularity issues have a direct bearing on the scheduler. It is the scheduler's responsibility to find work for a processor that is of large enough size. The Aurora and MUSE or-parallel systems, keep track of granularity by tracking the *richness* of nodes, i.e., the amount of work (measured in terms of number of untried alternatives in choicepoints) that is available in the subtree rooted at a node. Processors will tend to pick work from nodes that have high richness. Aurora and MUSE systems also make a distinction between *private* and *public* part of the tree to keep granularity high. Essentially, work created by another processor can only be picked up from the public region. In the private region, the processor that owns that region is responsible for all the work generated, thereby keeping the granularity high. In the private region execution is very close to sequential execution, resulting in high efficiency. Only when the public region runs out of work, a part of the private region of some processor is made public.

Granularity issues have also been considered at compile time. For instance, while generating CGEs at compile time [105, 106, 86], those that contain goals of too small a size should not be generated. This of course requires some means of measuring granularity at compile-time itself. Some progress has been made in compile-time granularity estimation but it largely remains an open research problem [153, 48].

6.5. Parallel Execution Visualization

Visualization of execution has been found to be of tremendous help in debugging and fine-tuning general parallel programs. Parallel execution of logic programs is no exception. Many tools have been already been developed to visualize parallel execution of logic programs. They work by logging events during parallel execution, and then using this event log for creating an (animated or static) graphical representation of the execution. The first visualization tool for parallel logic programs was developed for the Argonne Model [51]. This was subsequently adopted by the Aurora System under the name Wamtrace. The MUSE group also developed visualization tools, called MuseTrace, for visualizing or-parallel execution. Visualizers for or-parallel execution show the dynamically growing or-parallel search tree. Visualization tools (known as VisAndOr) have also been developed for independent and-parallelism by the &-Prolog group. These tools were later generalized to graphically represent or-parallel execution as well [33]. Visualization tools have also been developed for dependent and-parallelism in the context of committed choice languages, for example those for visualizing KL1 and GHC execution [134, 1].

The importance of visualization tools in the development of a parallel cannot be stressed enough. They help not only the users in debugging and fine-tuning their programs, but also the system implementors who need to understand execution behavior for fine-tuning their processor scheduling software.

6.6. Compile-time Support

Compile-time support is crucial for efficiency of parallel logic programming systems. Compile-time analysis tools based on Abstract Interpretation techniques [45] have been extensively used in many parallel logic programming systems. For instance, &-Prolog, AO-WAM, ACE, and PBA, all rely on Sharing and Freeness analysis for automatic generation of CGEs at compile-time [105, 106, 86]. The Andorra-I system relies on Determinacy Analysis done at compile-time for detecting determinacy of goals at runtime [121, 47]. Compile-time analysis can hence be used for making many decisions, which would have otherwise been taken at run-time, at compile-time itself, e.g., detection of determinacy, generation of CGEs, etc. Compile-time analysis has also been used for transforming Prolog programs into AKL [70] programs [28], and has also been used for supporting Prolog semantics in parallel systems that contain dependent and-parallelism, e.g., Andorra-I [121]. Compile-time analysis has also been employed to estimate granularity of goals, to help the scheduler in making better decisions as to which goal to pick [153, 48], etc.

Compile-time analysis has a number of potential applications in parallel logic programming, in addition to those already mentioned: for instance, in detecting speculative and non-speculative regions at compile-time, detecting whether a side-effect will be ever executed at run-time or not, detecting producer and consumer instances of variables, etc. Compiler support will play a crucial role in future parallel logic programming systems. However, a great deal of research is still needed in building more powerful compile-time analysis tools that can infer more properties of the program at compile-time itself to make parallel execution of logic program more efficient.

7. Conclusions and Future of Parallel Logic Programming

In this survey article we described different sources of implicit parallelism in logic programming languages and the many problems encountered in exploiting them. Different execution models proposed for exploiting these many kinds of parallelisms were surveyed. We also discussed some efficiency issues that arise in parallel logic programming.

Parallel logic programming is an exciting area of research and will continue to be so, until the dream of *efficiently* exploiting all sources of parallelism present in logic programs in the most cost effective way is realized. The current state-of-the-art is that there are very efficiently engineered systems that exploit only a single form of parallelism, e.g., MUSE, Aurora and Elipsys (or-parallelism), &-Prolog (independent and-parallelism), and many efficient implementations of committed choice languages [128, 84]. There are a few systems that efficiently exploit more than one sources of parallelism, such as Andorra-I [120], and others that are being built [60]. However, there are none that exploit *all* sources of parallelism present in logic programs. Efforts are already under way to remedy this [102, 63, 65], and we believe that that is where bulk of the parallel logic programming research in the future will lie. Research is also needed in other aspects of parallel logic programming; for example, in finding out how best to support sequential Prolog semantics on parallel logic programming systems of the future; building better and smarter schedulers; finding better memory management strategies; building compile-time tools that will take away the overhead at run-time by relegating many of the operations to compile-time; and building tools for visualizing parallel execution.

In the future, systems that exploit parallelism implicitly from logic (Prolog) programs will satisfy the same demand from the domain of symbolic applications, that automatic parallelizing compilers of Fortran are doing for numerical applications today.

Acknowledgements

We are grateful to Saumya Debray, co-editor of the 10 year commemorative issue of the Journal of Logic Programming, for giving us the opportunity to write this article. Thanks are also due to Bharat Jayaraman for helping with an earlier article on which this article is based. Thanks to Enrico Pontelli and Manuel Carro who read drafts of this paper. Our deepest thanks go to anonymous referees whose comments tremendously improved the paper. Gopal Gupta was supported by NSF Grants CCR 92-11732 and HRD 93-53271, Grant AE-1680 from Sandia National Labs, and by NATO Grant CRG 921318 during the preparation of this paper.

References

[1] S. Aikawa et al. ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems. In *International Conference on Fifth Generation Computer Systems,* 1992, Tokyo, pages 286–293.

[2] H. Aït-Kaci. Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press. 1992.

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

[4] Khayri A. M. Ali and Roland Karlsson. The MUSE Approach to Or-Parallel Prolog. *International Journal of Parallel Programming*, 19(2): 129–162, April 1990.

[5] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling Or-Parallelism in MUSE. *International Journal of Parallel Programming*, 19(6): 445–475, December 1990.

[6] Khayri A. M. Ali and Roland Karlsson. OR-parallel Speedups in a Knowledge Based System: on MUSE and Aurora. In *the Proceedings of the International Conference on Fifth Generation Computer Systems*, June 1992.

[7] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. Performance of MUSE on Switch-Based Multiprocessor Machines. *New Generation Computing*, 11(1,4): 81–103, 1992.

[8] Khayri A. M. Ali and Roland Karlsson. Scheduling Speculative Work in MUSE and Performance Results. *International Journal of Parallel Programming*, 21(6): December 1992.

[9] N. J. Aparicio, J. Cunha, L. Monteiro, and L. M. Pareira. Delta-Prolog: a Distributed Backtracking Extension with Events. In *Proc. 3rd International Conference on Logic Programming*, Lecture Notes in Computer Science 225, 1986, Springer Verlag, pages 225–240.

[10] R. Bahgat. Pandora: Non-Deterministic Parallel Logic Programming. PhD Thesis, Department of Computing, Imperial College of Science and Technology, Feb. 1991. Published by World Scientific Publishing Co. 1993.

[11] M. García de la Banda and M. Hermenegildo. Effectiveness of Combined Sharing and Freeness Analysis using Abstract Interpretation. Technical report, U. of Madrid (UPM), Facultad Informática, January 1992.

[12] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *Proceedings of the 1993 International Logic Programming Symposium.* MIT Press, October 1993.

[13] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *Proceedings of the 1993 International Logic Programming Symposium.* MIT Press, October 1993.

[14] J. Barklund. Parallel Unification, Ph.D. Thesis, Uppsala University, 1990.

[15] J. Barklund, H. Millroth. Providing Iteration and Concurrency in Logic Program through Bounded Quantifications. In *Proc. International Conf. on Fifth Generation Computer Systems*, June 1992, pages 817–824.

[16] U. Baron, J. C. de Kergommeaux et al. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation of Results. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1988, pages 841–850.

[17] M. Ratcliffe, J-C Syre. A Parallel Logic Programming Language for PEPSys. In *Proceedings of IJCAI*, 1987, Milan, pages 48–55.

[18] A. J. Beaumont, Scheduling strategies and speculative work, *In Parallel Execution of Logic Programs*, Beaumont & Gupta (Eds), Springer Verlag, LNCS 569, June 1991.

[19] A. J. Beaumont, S. Muthu Raman, Péter Szeredi, and David H. D. Warren. Flexible scheduling or-parallelism in Aurora: the Bristol scheduler. In *PARLE 91, Conference on Parallel Architectures and Languages Europe*, pages 403–420. Springer-Verlag, June 1991. Lecture Notes in Computer Science, Vol 506.

[20] Anthony Beaumont and David H. D. Warren. Scheduling speculative work in or-parallel Prolog systems. In *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, June 1993.

[21] P. Biswas, S-C Su, D.Y.Y. Yun. A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND Parallelism (RAP) in Logic Programs. In *Fifth International Logic Programming Conference,* Seattle, WA., August 1988, pages 1160–1179.

[22] J. Bevemyr, T. Lindgren, H. Millroth. Reform Prolog: The Language and its Implementation. In *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, June 1993, pages 283–298.

[23] P. Borgwardt. Parallel Prolog using Stack Segments on Shared Memory Multiprocessors. In *Proceedings of the 1984 International Symposium on Logic Programming,* Atlantic City, NJ, 1984, pages 2–11.

[24] K. De Bosschere and J-M Jacquet. Multi-Prolog: Definition, Operational Semantics and Implementation. In *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, June 1993.

[25] Per Brand. Wavefront scheduling. Internal Report, Gigalips Project, 1988.

[26] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[27] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. Technical Report TR Number CLIP7/93.0, U. of Madrid (UPM), Facultad Informática UPM, Madrid-Spain, October 1993.

[28] F. Bueno and M. Hermenegildo. An Automatic Translation Scheme from Prolog to the Andorra Kernel Language. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 759–769, 1992.

[29] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1590–1605. MIT Press, August 1988.

[30] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. Technical report CLIP 5/92, Univ. of Madrid (UPM), Facultad Informática, Madrid-Spain, August 1992.

[31] Alan Calderwood and Péter Szeredi. Scheduling or-parallelism in Aurora—the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.

[32] Mats Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[33] M. Carro, L. Gómez, and M. Hermenegildo. Some event-driven paradigms for the visualization of logic programs. In *Proceedings of the Tenth International Conference on Logic Programming*. MIT Press, June 1993.

[34] S-E. Chang and Y.P. Chiang. Restrict And-Parallelism Model with Side Effects. In *Proceedings of North American Conference on Logic Programming*, 1989, MIT Press, pages 350–368.

[35] J-H. Chang, A. M. Despain, and D. DeGroot. And-Parallelism of Logic Programs based on Static Data Dependency Analysis. In *Digest of Papers of COMPCON Spring 1985*, pages 218–225, 1985.

[36] P. Ciancarini. Blackboard Programming in Shared Prolog. In *Languages and Compilers for Parallel Computing*. MIT Press, June 1993.

[37] A. Ciepielewski. Scheduling in Or-parallel Prolog Systems: Survey, and Open Problems. In *International Journal of Parallel Programming*, 1992.

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

[39] K. Clark, S. Gregory. Parlog: Parallel Programming in Logic. In *A.C.M. TOPLAS*, Vol. 8, No. 1, Jan. 1986.

[40] W. F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. In *New Generation Computing*, No. 5, 1988, 361–376.

[41] J. S. Conery, D. F. Kibler. Parallel Interpretation of Logic Programs. In *Proceedings of the Conference on Functional Languages and Computer Architecture*, Oct., 1981. pages 163–170.

[42] J. S. Conery, D. F. Kibler. And Parallelism in Logic Programs. In *Proceedings of the International Joint Conference in AI*, 1983.

[43] J. S. Conery. Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. In *1987 IEEE International Symposium in Logic Programming*, San Francisco, pages 457–467

[44] J. Conery. Parallel Interpretation of Logic Programs. Kluwer Academic Press, 1987.

[45] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Model for Static Analysis of Programs for Construction or Approximation of Fix-points. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pages 238–252, 1977.

[46] J.A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. In *New Generation Computing* 10(4):385–422 August, 1992.

[47] S. K. Debray, D. S. Warren. Functional Computations in Logic Programs. In *ACM Transactions on Programming Languages and Systems*, 11(3):451–481. 1989.

[48] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*. ACM Press, June 1990.

[49] D. DeGroot. Restricted AND-parallelism. In *International Conference on Fifth Generation Computer Systems*, Nov., 1984.

[50] D. DeGroot. Restricted And-Parallelism and Side-effects. In *International Symposium on Logic Programming*, San Francisco, 1987, pages 80–89.

[51] T. Disz, E. Lusk, A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *Proc. IEEE International Symposium on Logic Programming*, pages 46–53, IEEE Computer Society, 1987.

[52] T. Disz, E. Lusk, R. Overbeek. Experiments with OR-parallel Logic Programs. In *IEEE International Symposium in Logic Programming*, San Francisco, 1987.

[53] Terrence Disz and Ewing Lusk. A graphical tool for observing the behavior of parallel logic programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 46–53. IEEE Computer Society Press, 1987.

[54] I. Dutra. A Flexible Scheduler for the Andorra-I System. In *ICLP'91 Workshop on Parallel Execution of Logic Programs*. Springer Verlag, LNCS 569. pages 70–82.

[55] I. Foster and S. Taylor. Strand: New Concepts in Parallel Programming. Prentice Hall, 1990.

[56] I. Futo. Prolog with Communicating Processes: From T-Prolog to CSR-Prolog. In Proc. 10th International Conference on Logic Programming, June, 1993. pages 3–17.

[57] G. Gupta, B. Jayaraman. And-Or Parallelism on Shared Memory Multiprocessors. In *Journal of Logic Programming*, 17(1):59–89.

[58] G. Gupta, B. Jayaraman. Analysis of Or-parallel Execution Models. In *ACM Transactions on Programming Languages*. 15(4):659–680, Sept. 1993.

[59] G. Gupta. Parallel Execution of Logic Programs on Shared Memory Multiprocessors. Ph.D. thesis. University of North Carolina at Chapel Hill. 1991.

[60] G. Gupta, M. Hermenegildo, ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP91 Workshop on Parallel Execution of Logic Programs*. Lecture Notes in Computer Science 569, Springer Verlag, 1991.

[61] G. Gupta, M. Hermenegildo, and V. Santos-Costa. And-Or Parallel Prolog: A Recomputation Based Approach. *New Generation Computing*, 11(3-4):297–323, 1993.

[62] G. Gupta, V. Santos Costa. And-Or Parallelism in Full Prolog based on Paged Binding Arrays. In *Proceedings of Parallel Architectures and Languages Europe (PARLE '92)*, Paris, June 1992.

[63] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[64] G. Gupta and V. Santos Costa. Cut and Side-Effects in And-Or Parallel Prolog. In Proc. 4th IEEE Symposium on Parallel and Distributed Processing, Arlington, 1992.

[65] G. Gupta and V. Santos Costa. Shared Paged Binding Array: A Universal Data-structure for Parallel Logic Programming. Technical Report 92-CS-23, Dept. of Computer Science. New Mexico State University, 1992.

[66] G. Gupta and V. Santos Costa. A Systematic Approach to Exploiting Parallelism in Logic Programs. In *Proceedings of 26th Hawaii International Conference on System Sciences*, IEEE press, Vol II, pages 417–426.

[67] G. Gupta, M. Hermenegildo, V. Santos Costa. Generalized Stack-copying for And-Or Parallel Implementations. In *JICLP'92 Workshop on Distributed and Parallel Implementations of Logic Programming Systems*, November 1992.

[68] G. Gupta, D.H.D. Warren. An Interpreter for the Extended Andorra Model. Technical Report 92-CS-24, Department of Computer Science, New Mexico State University, Department of Computer Science, 1992.

[69] S. Haridi. A Logic Programming Language Based on the Andorra Model. In *New Generation Computing*, Vol. 7, No. 2-3, pages 109–125.

[70] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of International Conference on Logic Prog.*, MIT Press, June, 1990.

[71] S. Haridi, P. Brand. The Andorra Parallel Logic Language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1988.

[72] A. Ciepielewski and S. Haridi. A Formal Model for Or-parallel Execution of Logic Programs. In *IFIP 83, North Holland, P.C. Mason (ed.)*, 1983.

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

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

[75] B. Hausman. Pruning and scheduling speculative work in or-parallel Prolog. In *PARLE 89, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, 1989.

[76] B. Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[77] B. Hausman, A. Ciepielewski, and A. Calderwood, "Cut and Side-Effects in Or-Parallel Prolog," In *International Conference on Fifth Generation Computer Systems,* Tokyo, Nov. 88, pages 831–840.

[78] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, May 1987.

[79] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[80] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[81] M. V. Hermenegildo, R. I. Nasr, Efficient Implementation of backtracking in AND-parallelism. In *3rd International Conference on Logic Programming*, London, 1986. pages 40–54.

[82] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[83] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with UPSHOT. Technical Report ANL-91/15, Argonne National Laboratory, 1991.

[84] K. Hirata et al. Parallel and Distributed Implementation of Logic Programming Language KL1. In *International Conference on Fifth Generation Computer Systems.* Tokyo, 1992. pages 436–459.

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

[86] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[87] S. Janson, J. Montelius. A Sequential Implementation of AKL. In *Proceedings of ILPS'91 Workshop on Parallel Execution of Logic Programs*, Oct. 1991.

[88] P. Kacsuk. Execution Models of Prolog for Parallel Computers, Research Monograph, MIT Press, 1990.

[89] L. V. Kalé, Parallel Architectures for Problem Solving. Ph.D. Thesis, Dept. of Computer Science, SUNY-Stony Brook, 1985.

[90] L. V. Kalé, B. Ramkumar. Machine Independent AND and OR Parallel Execution of Logic Programs: Part I — The Binding Environment. In *IEEE Transactions on Parallel Distributed Systems*, to appear.

[91] Roland Karlsson. *A High Performance OR-parallel Prolog System.* PhD thesis, The Royal Institute of Technology, Stockholm, March 1992.

[92] Feliks Kluźniak. Developing applications for Aurora. Technical Report TR-90-17, University of Bristol, Computer Science Department, August 1990.

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

[94] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[95] Y-J. Lin, V. Kumar. AND-parallel execution of Logic Programs on a Shared Memory Multiprocessor : A Summary of Results. In *Fifth International Logic Programming Conference,* Seattle, WA, 1988.

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

[97] P. López and M. Hermenegildo. Dynamic Term Size Computation in Logic Programs via Program Transformation. Technical report, U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1993.

[98] Ewing Lusk, Shyam Mudambi, Ross Overbeek, and Péter Szeredi. Applications of the Aurora parallel Prolog system to computational molecular biology. In *Proceedings of the ILPS'93*, October, 1993.

[99] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[100] D. Maier and D. S. Warren. Computing with Logic. Benjamins-Cummings Publishing Co., 1988.

[101] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Proceedings of the 20th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1994.

[102] R. Moolenaar and B. Demoen. A Parallel Implementation for AKL. In *Proc. Conference on Programming Language Implementation and Logic Programming* (Bruynooghe and Penjam eds), 5th International Symposium, Tallinn, Estonia, LNCS 714, Springer Verlag, pages 246–261.

[103] T. Moto-oka (ed.). Proceedings of International Conferences on Fifth Generation Computer Systems. Japan Information Processing Development Center (North-Holland), 1982.

[104] Shyam Mudambi. Performance of Aurora on NUMA machines. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 International Logic Programming Symposium*. MIT Press, October 1991.

[105] K. Muthukumar, M. V. Hermenegildo. Determination of Variable Dependence Information through Abstract Interpretation. In *Proc. of N. American Conference on Logic Programming '89*, MIT Press, 1989.

[106] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[107] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[108] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[109] K. Muthukumar and M.V. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[110] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[111] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming*. MIT Press, June 1989.

[112] D. Palmer and L. Naish. NUA Prolog: An Extension of the WAM for Parallel Andorra. In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, 1991, pages 429–442.

[113] J. L. Peterson, A. Silberschatz, Operating Systems Concepts. *2nd Edition*, Addison Wesley Publishing Co., 1986.

[114] G. H. Pollard. Parallel Execution of Horn Clause Programs. Ph.D. Thesis, Dept. of Computing, Imperial College, 1981.

[115] B. Ramkumar and L. V. Kalé. Compiled Execution of the REDUCE-OR Process Model. In *Proceedings of the North American Conference on Logic Programming*, MIT Press, 1989, pages 313–331.

[116] B. Ramkumar and L. V. Kalé. Joining And Parallel Solutions in And/Or Parallel Systems. In *Proc. of N. American Conference on Logic Programming*, MIT Press, 1990, pages 624–641.

[117] B. Ramkumar, L.V. Kalé. Machine Independent AND and OR Parallel Execution of Logic Programs: Part II - Compiled Execution. In *IEEE Transactions on Parallel and Distributed Systems.* To appear.

[118] A. Singhal, Y. Patt. Unification Parallelism: How much can be Exploited? In *Proceedings of North American Conference on Logic Programming.* MIT Press, 1989, pages 1135–1148.

[119] D. A. Smith. MultiLog: Data Or-parallel Logic Programming. In *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, June 1993, pages 314–332.

[120] V. Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM Press, April 1991.

[121] V. Santos Costa, D. H. D. Warren, R. Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, 1991, pages 443–456.

[122] V. Santos Costa, D. H. D. Warren, R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, 1991, pages 825–839.

[123] K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.*, MIT Press, 1992

[124] K. Shen. Studies in And/Or Parallelism in Prolog. U. of Cambridge 1992.

[125] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium.* MIT Press, October 1991.

[126] K. Shen and M. Hermenegildo. A Flexible Scheduling and Memory Management Scheme for Non-Deterministic, And-parallel Execution of Logic Programs. Technical report, U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1993.

[127] E. Shapiro. The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, Vol 21, No. 3, September 1989, pages 413–510.

[128] E. Shapiro. Concurrent Prolog : Collected Papers. MIT Press, 1987.

[129] Raéd Sindaha. The Dharma Scheduler—Definitive Scheduling in Aurora on Multiprocessors Architecture. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 296–303. IEEE Computer Society Press, December 1992. Also available from University of Bristol as Technical Report CSTR-92-23.

[130] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[131] Péter Szeredi. Using dynamic predicates in an or-parallel Prolog system. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 International Logic Programming Symposium*, pages 355–371. MIT Press, October 1991.

[132] Péter Szeredi, Mats Carlsson, and Rong Yang. Interfacing engines and schedulers in or-parallel Prolog systems. In *PARLE 91, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, June 1991. Lecture Notes in Computer Science, Vol 506.

[133] E. Tick. Parallel Logic Programming. MIT Press. 1991.

[134] E. Tick. Visualizing Parallel Logic Programming with VISTA. In *Proc. International Conference on Fifth Generation Computer Systems*. pages 934–942. Tokyo, 1992.

[135] P. Tinker. Performance of an OR-parallel Logic Programming system. In *International Journal of Parallel Programming*, **17**(1), pages 59–92, 1988.

[136] K. Ueda, Guarded Horn Clauses, Ph.D. Thesis, University of Tokyo, 1986.

[137] J. D. Ullman. Principles of Database and Knowledge-base Systems. Computer Science Press, 1989. Volume II.

[138] P. Van Roy. Can Logic Programming Execute as Fast as Imperative Programming?. Ph.D. Thesis, University of California at Berkeley. Nov. 1990.

[139] A. Véron, J. Xu, et al. Virtual Memory Support for Parallel Logic Programming Systems. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, Springer Verlag, June, 1991.

[140] D. S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. In *The 1984 Int. Symp. on Logic Programming*. Atlantic City, pages 198–202.

[141] D. H. D. Warren. An Abstract Instruction Set for Prolog. Tech. Note 309, SRI International, 1983.

[142] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102. IEEE Computer Society Press, 1987.

[143] D. H. D. Warren. Or-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes on Computer Science NCS 250, Springer Verlag, 1987.

[144] D.H.D. Warren. The Andorra Principle. Seminar given at Gigalips Workshop, SICS, Sweden, 1988.

[145] 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 International Conference on Fifth Generation Computer Systems '88*, pages 943–952, 1988.

[146] Patrick Weemeeuw and B. Demoen. Memory compaction for shared memory multiprocessors, design and specification. In *Proceedings of the North American Conference on Logic Programming.* MIT Press, October 1990.

[147] H. Westphal, P. Robert, J. Chassin and J. Syre. The PEPSys Model: Combining Backtracking, AND- and OR-parallelism. In *1987 IEEE International Symposium in Logic Prog.*, San Francisco, pages 436–448.

[148] D. S. Wise. Prolog Multiprocessors. Prentice-Hall, 1986.

[149] N. S. Woo and K.-M. Choe. Selecting the Backtrack Literal in the AND/OR Process Model. In IEEE Symposium on Logic Programming, pages 200–210, 1986.

[150] H. Yasuhara, K. Nitadori. ORBIT: A Parallel Computing Model of Prolog. In *New Generation Computing*, 2:227–228, 1984.

[151] R. Yang. P-Prolog: A Parallel Logic Programming Language. Ph.D. Thesis. Keio University. Published by World Scientific Publishers, 1987.

[152] R. Yang et al. Performance of the Compiler-based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, June 1993.

[153] X. Zhong, E. Tick et al. Towards an Efficient Compile-time Granularity Algorithm. In *International Conference on Fifth Generation Computer Systems.* Tokyo, 1992. pages 809–816.