# Parallel Execution of Prolog Programs: A Survey

**Gopal Gupta**

*Department of Computer Science*
*Box 830688/EC31*
*University of Texas at Dallas*
*Richardson, TX 75083-0688*
gupta@utdallas.edu

**Enrico Pontelli**

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

**Khayri A.M. Ali and Mats Carlsson**

*Swedish Institute of Computer Science, SICS*
*Box 1263, SE-164 29*
*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

Contents

## 1. INTRODUCTION

The technology for sequential implementation of logic programming languages has evolved considerably in the last two decades. In recent years, it has reached a notable state of maturity and efficiency. Today, a wide variety of commercial logic programming systems are available that are being used to develop large, real-life applications. An excellent survey of sequential implementation technology that has been developed for Prolog is presented by Van Roy [Van Roy 1994].

For years logic programming has been considered well suited for execution on multiprocessor architectures. Indeed research in parallel logic programming is vast and dates back to the inception of logic programming itself—one of the earliest published work being Pollard's Ph.D. Thesis [Pollard 1981]. Even Kowalski mentions the possibility of executing logic programs in parallel in his seminal book "Logic for Problem Solving" [Kowalski 1979]. There has been a healthy interest in parallel logic programming ever since, 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 such conferences.

This interest in parallel execution of logic programs arises from two perspectives:

(1) continuous research in simple, efficient, and practical ways to make parallel and distributed architectures easily *programmable* drew the attention to logic programming, since, potentially, parallelism can be *exploited implicitly* from logic programs (i.e., parallelism can be extracted from logic programs automatically without any user intervention);

(2) the everlasting myth that logic programming languages have low execution efficiency: this prompted researchers to look for alternative ways of achieving speed, i.e., through the use of parallelism.

As mentioned, the literature on parallel execution of logic programs is vast and varied. There are two major (and non-independent) schools of thought. The first approach, which is the main focus of this survey, relies on *implicit* exploitation of parallelism from logic programs. This means that the parallelization of the execution can (potentially) occur without any input from the programmer. Note that these model do not prevent programmer intervention, but usually they either make it optional or they keep it at a very high level.

In contrast, a number of approaches have been developed which target the extraction of parallelism through the use of *explicit* constructs introduced in the source language. This can be done by extending a logic programming language with explicit constructs for concurrency or by modifying the semantics of the logic programming language in a suitable way. Approaches to explicit exploitation of parallelism from logic programs can be largely classified into three categories:

(1) Those that add explicit message passing primitives to Prolog, e.g., Delta Prolog [Aparício et al. 1986] and CS-Prolog [Futo 1993]. Multiple Prolog processes are run in parallel and they communicate with each other via explicit message passing or other rendezvous mechanisms.

(2) Those that add *blackboard* primitives to Prolog, e.g., Shared Prolog [Ciancarini 1993]. These primitives are used by multiple Prolog processes running in parallel to communicate with each other via the common blackboard.

Some recent proposals in this category include

- the Jinni system [Tarau 1998] developed by Tarau, a Java-based logic programming system including multi-threading and blackboard-based communication; this work is a continuation of the previous work by Tarau and de Bosschere [de Bosschere and Tarau 1996];
- the CIAO system [Hermenegildo 1994] supports novel Prolog database operations which allow to use the database as a (synchronizing) blackboard [Carro and Hermenegildo 1999].

Blackboard primitives are currently supported by a number of other Prolog systems, including SICStus [Carlsson et al. 1995] and YAP [Santos Costa, Damas, Reis, Azevedo 1999].

(3) Those based on guards, committed choice, and data-flow synchronization, e.g., Parlog, GHC, KL1 (and its portable C-based implementation KLIC [Chikayama et al. 1994]), and Concurrent Prolog [Clark and Gregory 1986; Ueda 1986; Shapiro 1987; 1989].

This class includes the class of concurrent constraint languages (e.g., LIFE [Ait-Kaci 1993] and ccl(fd) [Van Hentenryck, Saraswat, Deville 1998]) and the class of distributed constraint languages such as Oz/Mozart [Haridi, Van Roy, Brand, Schulte 1998; Smolka 1995] and AKL [Haridi and Jason 1990].

Each of the three approaches above has been explored and there is extensive research literature that can be found. They all involve complex issues of language extension and design, as well as of implementation. However, in order to keep the paper focused we will consider these approaches only marginally or in those cases where they introduce execution mechanisms which are applicable also in the case of implicit exploitation of parallelism (e.g., committed choice languages).

In the rest of this work we will focus primarily on the parallel execution of Prolog programs, although occasional generalizations to logic languages with a different operational semantics will be considered (e.g., we briefly discuss parallelization in constraint logic programming languages). This choice is dictated by the wider use of Prolog w.r.t. other logic languages, and a consequent wider applicability of the results accomplished. Observe also that parallelization of Prolog raises issues that are absent from the parallelization of other logic languages (e.g., due to the presence of extra-logical predicates). Throughout this work we will often use the terms "logic programs" and "Prolog programs" interchangeably, thus assuming sequential Prolog semantics as the target operational behavior (a discussion of the differences between general logic programming and Prolog is presented in Section 2). Parallel execution of other logic-based languages, such as committed-choice languages, raises issues similar to those discussed in this paper.

The objective of this paper is to provide a uniform view of the research in parallel logic programming. Due to the extensive body of research in this field, we will not be able to cover every single aspect and model which have been presented in the literature. Thus, our focus will lie on highlighting the fundamental problems and the key solutions that have been proposed. This survey expands on the work done by other researchers in the past in proposing an organized overview of parallel logic programming. In particular, this work expands on the remarkable survey on parallel logic programming systems by Chassin de Kergommeaux and Codognet [Chassin de Kergommeaux and Codognet 1994] by covering the research performed in the last 8 years and by provide a more in-depth analysis of various areas. Other surveys have also appeared in the literature, mostly covering more limited areas of parallel logic programming [Gupta and Jayaraman 1993b; Kacsuk 1990; Takeuchi 1992; Delgado-Rannauro 1992a; 1992b].

The paper is organized as follows. The next section provides a brief introduction to logic programming and parallel logic programming, focusing on the distinction between the different forms of parallelism exploited in logic programming. Section 3 illustrates the issues involved in *or-parallel* execution of Prolog programs. Section 4 describes *independent and-parallelism* and discusses the solutions adopted in the literature to handle this form of parallelism. Section 5 introduces the notion of *dependent and-parallelism* and describes different techniques adopted to support it in different systems. The issues arising from the concurrent exploitation of and- and or-parallelism are presented in Section 6, along with the most relevant proposals to tackle such issues. Section 7 describes the techniques adopted in the literature to exploit data parallelism from logic programs. Section 9 covers a variety of issues related to implementation and efficiency of parallel logic programming (e.g., optimizations, static analysis, support tools). Section 10 gives a brief overview of the types of applications to which parallel logic programming has been successfully applied. Finally, Section 11 draws some conclusions and gives some insights on current and future research directions in the field.

In the rest of this work we assume the reader to have familiarity with the basic terminology of logic programming and Prolog [Lloyd 1987; Sterling and Shapiro 1994].

## 2. LOGIC PROGRAMMING AND PARALLELISM

In this section we present a brief introduction to logic programming and Prolog. A more detailed presentation of these topics can be found elsewhere [Lloyd 1987; Ait-Kaci 1992; Sterling and Shapiro 1994].

### 2.1 Logic Programs and Prolog

A logic program is composed by a set of *Horn clauses*. Using Prolog's notation, each clause is a formula of the form

$$Head :- B_1, B_2, \ldots, B_n$$

where *Head*, $B_1, \ldots, B_n$ are atomic formulae and $n \geq 0$.[1] Each clause represents a logical implication of the form

$$\forall(B_1 \wedge \ldots \wedge B_n \rightarrow \mathit{Head})$$

A separate type of clauses are those where *Head* is the atom `false`, which are simply written as

$$:-B_1, \ldots, B_n$$

These type of clauses are called *goals* (or *queries*). Each atom in a goal is called a *subgoal*.

Each atomic formula is composed by a predicate applied to a number of arguments (terms), and this will be denoted as $p(t_1, \ldots, t_n)$—where $p$ is the predicate name, and $t_1, \ldots, t_n$ are the terms used as arguments. Each term can be either a constant ($c$), a variable ($X$), or a complex term ($f(s_1, \ldots, s_m)$, where $s_1, \ldots, s_m$ are themselves terms and $f$ is the *functor* of the term).

Execution of a logic program typically involves a program $P$ and a goal $:-G_1, \ldots, G_n$, and the objective is to verify whether there exists an assignment $\sigma$ of terms to the variables in the goal such that $(G_1 \wedge \ldots \wedge G_n)\sigma$ is a logical consequence of $P$.[2] $\sigma$ is called a *substitution*: a substitution is an assignments of terms to a set of variables (the *domain* of the substitution). If a variable $X$ is assigned a term $t$ by a substitution, then $X$ is said to be *bound* and $t$ is the (run-time) binding for the variable $X$. The process of assigning values to the variables in $t$ according to a substitution $\sigma$ is called *binding application*.

Prolog, as well as many other logic programming systems, make use of *SLD-resolution* to carry out program's execution. Execution of a program $P$ w.r.t. a goal $G$ proceeds by transforming a *resolvent* using a sequence of *resolution steps*. Each resolvent represents a conjunction of subgoals. The initial resolvent corresponds to the goal $G$. Each resolution step proceeds as follows:

- Let us assume that $:-A_1, \ldots, A_k$ is the current resolvent. An element $A_i$ of the resolvent is selected (*selected subgoal*) according to a predefined *computation rule*. In the case of Prolog, the computation rule selects the leftmost element of the resolvent.
- If $A_i$ is the selected subgoal, then the program is searched for a clause $\mathit{Head} :- B_1, \ldots, B_h$ whose head successfully unifies with $A_i$. Unification is the process which determines the existence of a substitution $\sigma$ such that $\mathit{Head}\sigma = A_i\sigma$. If there are rules satisfying this property then one is selected (according to a *selection rule*) and a new resolvent is computed by replacing $A_i$ with the body of the rule and properly instantiating the variables in the resolvent:

  $$:-(A_1, \ldots, A_{i-1}, B_1, \ldots, B_h, A_{i+1}, \ldots, A_k)\sigma$$

  In the case of Prolog, the clause selected is the first one in the program whose head unifies with the selected subgoal.
- If no clause satisfies the above property, then a failure occurs. Failures are cured using *backtracking*. Backtracking explores alternative execution paths by reducing one of the preceding resolvents with a different clause.
- The computation stops either when a solution is determined (i.e., the resolvent contains zero subgoals) or when all alternatives have been explored without any success.

An intuitive procedural description of this process is represented in Figure 2. The operational semantics of a logic based language is determined by the choice of computation rule (selection of the subgoal in the resolvent—called $\texttt{select}_{\texttt{literal}}$ in Figure 2) and the choice of selection rule (selection of the clause to compute the new resolvent—called $\texttt{select}_{\texttt{rule}}$). In the case of Prolog, the computation rule selects the leftmost subgoal in the resolvent, while the selection rule selects the first clause in the program which successfully unifies with the selected subgoal.

Many logic languages (e.g., Prolog) introduce a number of *extra-logical predicates*, used to perform tasks such as:

[1] If $n = 0$ then the formula is simply written as *Head* and called *fact*.

[2] Following standard practice, the notation $e\sigma$ denotes the application of the substitution $\sigma$ to the expression $e$—i.e., each variable $X$ in $e$ will be replaced by $\sigma(X)$.

(1) perform input/output (e.g., read and write files);
(2) add a limited form of control to the execution (e.g., the cut (`!`) operator, used to remove some unexplored alternatives from the computation);
(3) perform meta-programming operations; these are used to modify the structure of the program (e.g., `assert` and `retract`, add or remove clauses from the program), or query the status of the execution (e.g., `var` and `nonvar`, used to test the binding status of a variable).

An important aspect of many of these extra-logical predicates is that their behavior is *order-sensitive*, meaning that they can produce a different outcome depending on when they are executed. In particular, this means that they can potentially produce a different result if a different selection rule or a different computation rule is adopted.

In the rest of this work we will focus on execution of Prolog programs (unless explicitly stated otherwise); this means that we will assume that programs are executed according to the computation and selection rule of Prolog. We will also frequently use the term *observable semantics* to indicate the overall observable behavior of an execution—i.e., the order in which all visible activities of a program execution take place (order of input/output, order in which solutions are obtained, etc.). If a computation respects the observable Prolog semantics, then this means that the user does not see any difference between such computation and a sequential Prolog execution of the same program.

### 2.2 The Warren Abstract Machine

The *Warren Abstract Machine (WAM)* [Warren 1983; Ait-Kaci 1992] has become a de-facto standard for the sequential implementations of Prolog and Logic Programming languages. The WAM defines an abstract architecture whose instruction set is designed to

(1) allow an easy mapping from Prolog source code to WAM instructions;
(2) be sufficiently low-level to allow an efficient emulation and/or translation to native machine code.

Most implementations of Prolog currently rely either directly on the WAM, or on a sufficiently similar architecture.

The WAM is a stack-based architecture, sharing some similarities with imperative languages implementation schemes (e.g., use of `call/return` instructions, use of frames for maintaining procedure's local environment), but extended in order to support the features peculiar to Logic Programming, namely *unification* and *backtracking* (and some other variations, like the need of supporting dynamic type checking).

The WAM is a stack-based architecture. At any instance, the state of the machine is defined by the content of its memory areas (illustrated in figure 1). The state can be subdivided into *internal* and *external* state.

- *Internal State:* it is described by the content of the machine registers. The purpose of most of the registers is described in figure 1.
- *External State:* it is described by the content of the logical data areas of the machine:
  - *Heap:* data areas in which complex data structures (lists and Prolog's compound terms) are allocated.
  - *Local Stack:* (also known as *Control Stack*) it serves the same purpose as the control stack in the implementation of imperative languages—it contains control frames, called *environments* (akin to the activation records used in implementation of imperative languages), which are created upon entering a new clause (i.e., a new "procedure") and are used to store the local variables of the clause and the control information required for "returning" from the call.
  - *Choice Point Stack:* choice points encapsulate the execution state for backtracking purposes. A choice point is created whenever a call having multiple possible solution paths (i.e., more than one clause successfully match the call) is encountered. Each choice point should contain sufficient information to restore the status of the execution at the time of creation of the choice point, and should keep track of the remaining unexplored alternatives.
  - *Trail Stack:* during an execution variables can be instantiated (they can receive bindings). Nevertheless, during backtracking these bindings need to be undone (to restore the previous state of execution). In order to make this possible, bindings that can be subject to this operation are registered in the trail stack. Each choice point records the point of the trail where the undoing activity needs to stop.

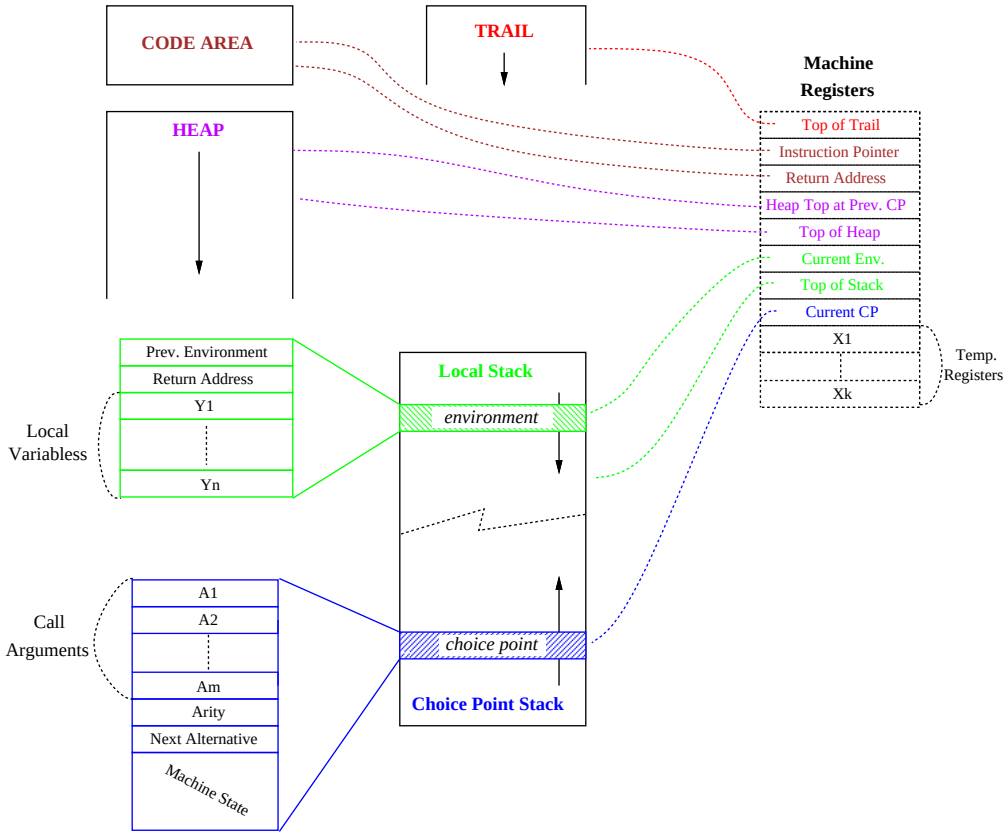



Fig. 1. Organization of the WAM

Prolog is a dynamically typed language; hence it requires type information to be associated with each data object. In the WAM, Prolog terms are represented as *tagged words*: each word contains:

(1) a *tag* describing the type of the term (atom, number, list, compound structure, unbound variable);
(2) a *value* whose interpretation depends on the tag of the word; e.g., if the tag indicates that the word represents a list, then the value field will be a pointer to the first node of the list[3].

Prolog programs are compiled into a series of abstract instructions operating on the previously described memory areas. In a typical execution, whenever a new subgoal is selected (i.e., a new "procedure call" is performed), the following steps are taken:

- the arguments of the call are prepared and loaded into the temporary registers $X_1, \ldots, X_n$—the instruction set contains a family of instructions, the "`put`" instructions, for this purpose.
- the clauses matching the subgoal are detected and, if more than one is available, a choice point is allocated (using the "`try`" instructions);
- the first clause is started: this requires *head unification*—i.e., unification between the head of the clause and the subgoal to be solved—to be performed (using "`get/unify`" instructions). If head unification is successful (and assuming that the rule contains some user-defined subgoals), then an environment for the clause is created ("`allocate`") and the body of the clause is executed, otherwise backtracking to the last choice point created takes place.
- backtracking involves extracting a new alternative from the topmost choice point ("`retry`" will extract the next alternative, assuming this is not the last one, while "`trust`" will extract the last alternative and remove

[3]Lists in Prolog, as in Lisp, are composed of *nodes*, where each node contains a pointer to an element of the list (the *head*) and a pointer to the rest of the list (the *tail*).

the exhausted choice point), restoring the state of execution associated with such choice point (in particular, the content of the topmost part of the trail stack is used to remove bindings performed after the creation of the choice point), and restarting the execution of the new alternative.

The WAM has been designed in order to optimize the use of resources during execution, improving speed and memory consumption. Optimizations which are worth mentioning are:

- *Last Call Optimization:* [Warren 1980] represents an instance of the well-known *Tail-recursion optimization* commonly used in the implementation of many programming languages. Last call optimization allows to reuse the environment of a clause for the execution of the last subgoal of the clause itself;
- *Environment Trimming:* [Ait-Kaci 1992] allows a progressive reduction of the size of the environment of a clause during the execution of the clause itself, by removing the local variables that are not needed in the rest of the computation.
- *Shallow Backtracking:* [Carlsson 1989] the principle of *procrastination* [Gupta and Pontelli 1997] has been applied to the allocation of choice points in the WAM: the allocation of a choice point is delayed until a successful head unification has been detected. This allows in many occasions to avoid the allocation of the choice point at all—if no head unification succeed, or if the successful one is the last clause defining such predicate.
- *Indexing:* this technique is used to guide the analysis of the possible clauses that can be used to solve the current subgoal. The values of the arguments can be used to prune the search space at runtime. The original WAM supplies some instructions ("`switch`" instructions) to analyze the functor of the first argument and select different clusters of clauses depending on its value. Since many programs cannot profit from first-argument selection, more powerful indexing techniques have been proposed, taking into account more arguments and generating more complex decision trees [Hickey and Mudambi 1989; Van Roy and Despain 1992; Taylor 1991].

### 2.3 Logic Programming and Parallelism

Parallelization of logic programs can be seen as a direct consequence of Kowalski's principle [Kowalski 1979]

$$Programs = Logic + Control$$

Program development separates the control component from the logical specification of the problem, thus making the control of execution an orthogonal feature, independent from the logical specification of the problem. The lack (or, at least, the limited presence) of knowledge about control in the program allows the run-time systems to adopt different execution strategies without affecting the declarative meaning of the program (i.e., the set of logical consequences of the program). Not only does this allow cleaner (declarative) semantics for logic programs, and hence a better understanding of them by their users, it 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*.

Apart from the separation between logic and control, from a programming languages perspective, logic programming offers two key features which make exploitation of parallelism more practical than in traditional imperative languages:

(1) From an operational perspective, logic programming languages are *single-assignment* languages; variables are mathematical entities which can be assigned a value at most once during each derivation—this relieves a parallel system from having to keep track of complex flow dependencies, as in the parallelization of traditional programming languages [Zima and Chapman 1991].

(2) The operational semantics of logic programming, contrary to imperative languages, makes substantial use of *non-determinism*—which in turn can be easily converted into parallelism without radical modifications of the overall operational semantics. Furthermore, control in most logic programming languages is largely implicit, thus limiting programmers' influence on the development of the flow of execution.

The second point is of particular importance: the ability to convert existing non-determinism into parallelism leads to the possibility of extracting parallelism directly from the execution model without any modification to the language (*implicit parallelization*).

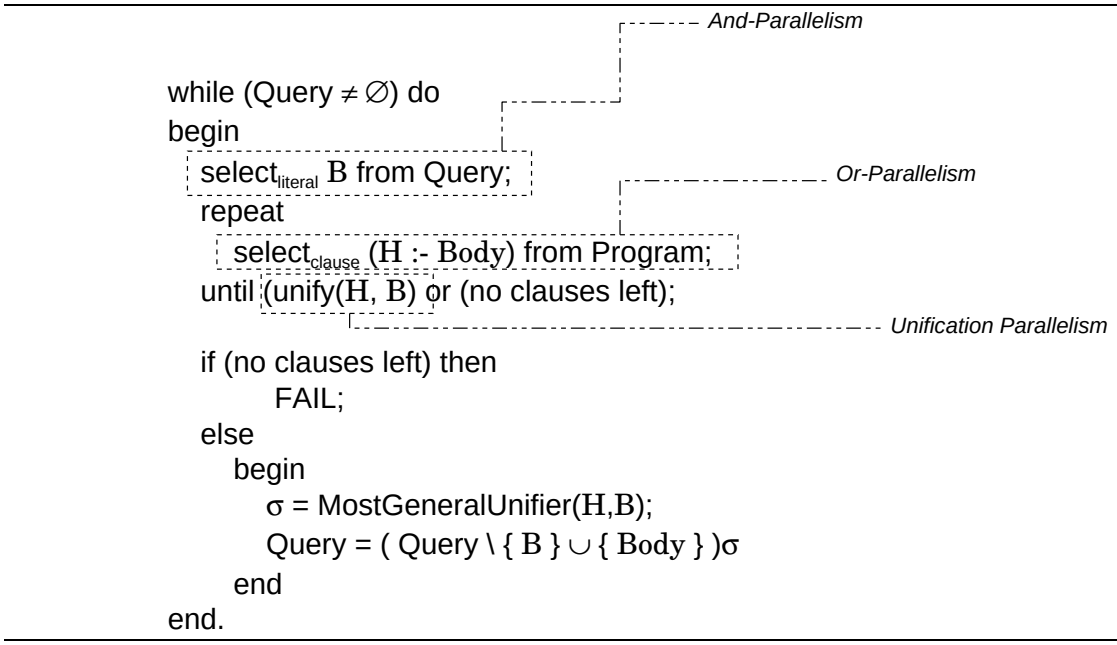



Fig. 2. Operational Semantics and Non-determinism

The typical strategy adopted in the development of parallel logic programming systems has been based on the translation of one (or more) of the non-deterministic choices present in the operational semantics (see Figure 2) into parallel computations. This leads to three main forms of parallelism:

- *And-Parallelism*, which originates from parallelizing the selection of the next literal to be solved—thus allowing multiple literals to be solved concurrently.
- *Or-Parallelism*, which originates from parallelizing the selection of the clause to be used in the computation of the resolvent—thus allowing multiple clauses to be tried in parallel.
- *Unification Parallelism*, which arises from the parallelization of the unification process.

The next three subsections elaborates on these three forms of parallelism.

### 2.3.1 *Unification Parallelism*

*Unification parallelism* arises during the unification of the arguments of a goal with the arguments 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 [Barklund 1990]. This can be easily illustrated as follows: a standard unification (*à la Robinson*) is approximatively structured as

```
unify(Arg1, Arg2):
   if (Arg1 is a complex term f(t1,...,tn) and Arg2 is a complex term g(s1,...,sm)) then
       if (f is equal to g and n is equal to m) then
              unify(t1,s1), unify(t2,s2), ..., unify(tn,sn)
       else
           fail
   else
      ....
```

Thus, unification of two complex terms is broken down in pairwise unification of the different arguments. For example, the process of unifying two terms

```
person(birth(day(12),month(1),year(99)),address(street(foothills),number(2),city(cruces)))
             person(birth(day(X),month(1),Y), address(Z,W,city(cruces)))
```

requires the separate unification between the arguments

```
birth(day(12),month(1),year(99))                    =    birth(day(X),month(1),Y)
address(street(foothills),number(2),city(cruces))   =   address(Z,W,city(cruces))
```

Unification parallelism takes advantage of the sequence of unifications between the arguments of complex structures, by performing them concurrently:

```
doall
   r1 = unify(t1,s1);
   ...
   rn = unify(tn,sn);
endall
return (r1 and ... and rn);
```

where `doall` indicates the parallel execution of all the statements between `doall` and `endall`.

Unification parallelism is very fine-grained and is best exploited by building specialized CPUs with multiple unification units [Singhal and Patt 1989]. Parallel unification also needs to deal with complex dependency issues [Singhal and Patt 1989; Barklund 1990]. Unification parallelism has not been the major focus of research in parallel logic programming.

### 2.3.2 *Or-Parallelism*

*Or-Parallelism* originates from the parallelization of the $\texttt{select}_{\texttt{clause}}$ phase in Figure 2. Thus, or-parallelism arises when more than one rule defines a relation and a subgoal 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 in parallel the search space generated by the presence of multiple clauses applicable at each resolution step. Observe that each parallel computation is potentially computing an alternative solution to the original goal.

Note that or-parallelism encompasses not only the actual concurrent execution of different alternatives, but also the concurrent *search* for the different alternatives which are applicable to the selected subgoal; some researchers have proposed techniques to explicitly parallelize this search process, leading to the so called *search parallelism* [Bansal and Potter 1992; Kasif, Kohli, Minker 1983].

Or-parallelism frequently arises in applications that explore a large search space via backtracking. This is the typical case in application areas such as expert systems, optimization and relaxation problems, parsing, natural language processing, and scheduling. Or-parallelism also arises in the context of parallel execution of deductive database systems [Ganguly et al. 1990; Wolfson and Silberschatz 1988].

### 2.3.3 *And-Parallelism*

*And-Parallelism* arises from the parallelization of the $\texttt{select}_{\texttt{literal}}$ phase in Figure 2. Thus, and-parallelism arises when more than one goal is present in the resolvent, and (some of) these goals are executed in parallel. And-parallelism thus permits exploitation of parallelism within the computation of a single solution to the original goal.

And-parallelism arises in most applications, but is particularly present in divide&conquer applications, list processing applications, various constraint solving problems and system applications.

In the literature it is common to distinguish two forms of and-parallelism:

- *Independent and-parallelism (IAP)* arises when, given two or more subgoals, the runtime bindings for the variables in these goals prior to their execution are such that each goal has no influence on the outcome of the other goals. Such goals are said to be *independent* and their parallel execution gives rise to *independent and-parallelism*. The typical example of independent goals is represented by goals which, at run-time, do not share any unbound variable—i.e., the intersection of the sets of variables accessible by each goal is empty. More refined notions of independence, e.g., non-strict independence, have also been proposed [Cabeza and Hermenegildo 1994].
- *Dependent and-parallelism* arises when, at runtime, two or more goals in the body of a procedure have a common variable and are executed in parallel, possibly "competing" in the creation of bindings for the

common variable (or "cooperating", if the goals share the task of creating the binding for the common variable). Dependent and-parallelism can be exploited in varying degrees, ranging from models which faithfully reproduce Prolog's observable semantics to models which use specialized forms of dependent and-parallelism (e.g., *stream parallelism*) to support coroutining and other alternative semantics—as in the various committed choice languages [Shapiro 1987; Tick 1995].

The distinction between independent and-parallelism and dependent and-parallelism is based on the granularity of computation considered. Parallelism is always obtained by executing two (or more) operations in parallel if those two operations do not influence each other in any way (i.e., they are independent); otherwise, parallel execution would not be able to guarantee correctness. For independent and-parallelism entire goals have to be independent of each other to be executed in parallel. On the other hand, in dependent and-parallelism the steps inside execution of each goal are examined, and steps in each goal that do not interfere with each other are executed in parallel. Thus, independent and-parallelism could be considered macro level and-parallelism, while dependent and-parallelism could be considered as micro level and-parallelism. As is perhaps now obvious to the reader, dependent and-parallelism is harder to exploit for Prolog (unless adequate changes to the operational semantics are introduced, as in the case of concurrent logic languages [Shapiro 1987]).

### 2.4 Discussion

Or-parallelism and and-parallelism identify opportunities for transforming certain sequential components of the operational semantics of logic programming into concurrent operations. In the case of or-parallelism, the exploration of the different alternatives in a choice-point is parallelized, while in the case of and-parallelism the resolution of distinct subgoals is parallelized. In both cases, we expect the system to provide a number of computing resources which are capable of carrying out the execution of the different instances of parallel work (i.e., clauses from a choice-point or subgoals from a resolvent). These computing resources can be seen as different Prolog engines which are cooperating in the parallel execution of the program. We will often refer to these computing entities as *workers* or *agents*. The term *process* has also been frequently used in the literature to indicate these computing resources—as workers are typically implemented as separate processes. The complexity and capabilities of each agent vary across the different models proposed. Certain models view agents as *processes* which are created for the specific execution of an instance of parallel work (e.g., an agent is created to specifically executed a particular subgoal), while other models view agents as representing individual *processors*, which have to be repeatedly scheduled to execute different instances of parallel work during the execution of the program. We will return to this distinction later on in Section 9.1.

Intuitively, or-parallelism and and-parallelism are largely orthogonal to each other, as they parallelize independent points of non-determinism in the operational semantics of the language. Thus, one would expect that the exploitation of one form form of parallelism does not affect the exploitation of the other, and it should be feasible to exploit both of them simultaneously. However, practical experience has demonstrated that this orthogonality does not easily translate at the implementation level. For various reasons (e.g., conflicting memory management requirements) combined and/or-parallel systems turned out to be extremely complicated, and so far no *efficient* parallel system has been built that achieves this ideal goal. At the implementation level, there is considerable interaction between and- and or-parallelism and most proposed systems have been forced into restrictions on both forms of parallelism (these issues are discussed at length in Section 6).

On the other hand, one of the ultimate aims of researchers in parallel logic programming has been to extract the best execution performance from a given logic program. Reaching this goal of maximum performance entails exploiting multiple forms of parallelism to achieve best performance on arbitrary applications. Indeed, various experimental studies (e.g., [Shen and Hermenegildo 1991; Pontelli, Gupta, Wiebe, Farwell 1998]) seem to suggest that there are large classes of applications which are rich in either one of the two forms of parallelism, while others offer modest quantities of both. In these situations, the ability to concurrently exploit multiple forms of parallelism becomes essential.

It is important to underline that the overall goal of research in parallel logic programming is the achievement of higher performance through parallelism. Accomplishing good *speedups* may not necessarily translate to an actual improvement in performance with respect to state-of-the-art sequential systems—e.g., the cost

of managing the exploitation of parallelism can make the performance of the system on a single processor considerably slower than a standard sequential system.

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 of these can be found in [Ait-Kaci 1992]. A general familiarity with various concepts in parallelism is also assumed. An excellent exposition of the needed concepts can be found in [Gottlieb and Almasi 1994; Zima and Chapman 1991].

The largest part of the body of research in the field of parallel logic programming focused on the development of systems on *Shared Memory* architectures—and indeed many of the techniques presented are specifically designed to take advantage of a single shared storage. Research on execution of logic programs on *Distributed Memory* architectures (e.g., [Benjumea and Troya 1993; Kacsuk and Wise 1992]) has been more sparse and less incisive. The current trend of research indicates an increasing emphasis towards distributed memory architectures [Araujo and Ruz 1998; Castro et al. 1998; Gupta and Pontelli 1999a], thanks to their increased availability at affordable prices and their scalability. Nevertheless, the focus of this survey is on describing execution models for shared memory architectures.

## 3. OR-PARALLELISM

Or-parallelism arises when a subgoal can unify with the 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. For example, consider the following simple logic program:

```
f :- t(X, three), p(Y), q(Y).
p(L) :- s(L, M), t(M, L).
p(K) :- r(K).
q(one).
q(two).
r(one).
r(three).
s(two, three).
s(four, five).
t(three, three).
t(three, two).
```

and the query `?- f.` The calls to `t`, `p`, and `q` are non-deterministic and lead to the creation of choice-points. In turn, the execution of `p` leads to the call to the subgoal `s(L,M)`, which leads to the creation of another choice-point. The multiple alternatives in these choice-points can be executed in parallel.

A convenient 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 $L_1, L_2, \ldots, L_m$ and $\theta = mgu(H, L_1)$.

Figure 3 shows the or-parallel tree for the simple program presented above. Note that, since we are considering execution of Prolog programs, the construction of the or-parallel tree will follow the operational semantics of Prolog—at each node we will consider clauses applicable to the first subgoal, and the children of a node will be considered ordered from left to right according to the order of the corresponding clauses in the program.

Note that each node of the or-parallel tree in Figure 3 contains the variables found in its corresponding clause, i.e., it holds that clause's *environment*. During sequential execution this or-parallel tree is searched in a depth-first manner. However, if multiple agents are available, then multiple branches of the tree can be searched simultaneously giving rise to or-parallelism. If the different branches are searched in or-parallel, then note that one will be confronted with the following problem: the variable `Y` receives different bindings in different branches of the tree all of which will be active at the same time. Storing and later accessing these bindings efficiently is a problem. In sequential execution the binding of a variable is stored in the memory location allotted to that variable. Since branches are explored one at a time, and bindings are untrailed during backtracking, no problems arise. In parallel execution, multiple bindings exist at the same time, hence they cannot be stored in a single memory location allotted to the variable. This problem, known as the *multiple environment representation problem*, is a major problem in implementing or-parallelism and is discussed in the next section.

Or-parallelism manifests itself in a number of applications [Kluźniak 1990]. 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 tree 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, and in applications of generate-and-test nature—the various alternatives can be generated and tested in or-parallel. This can be seen for example in the following simple program to solve the 8-queen 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).
```

The call to `delete/3` in the second clause of `queens/3` acts as a generator of bindings for the variable `X` and creates a number of choice-points. The predicate `delete/3` will be called again in the recursive invocations of `queens/3`, creating yet more choice-points and yet more untried alternatives that can be picked up by agents for or-parallel processing.

### 3.1 Challenges in the 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 agents. 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 the or-tree, all nodes above (and including) the least common ancestor node of these two nodes are shared between the two branches. A variable created in one of these ancestor

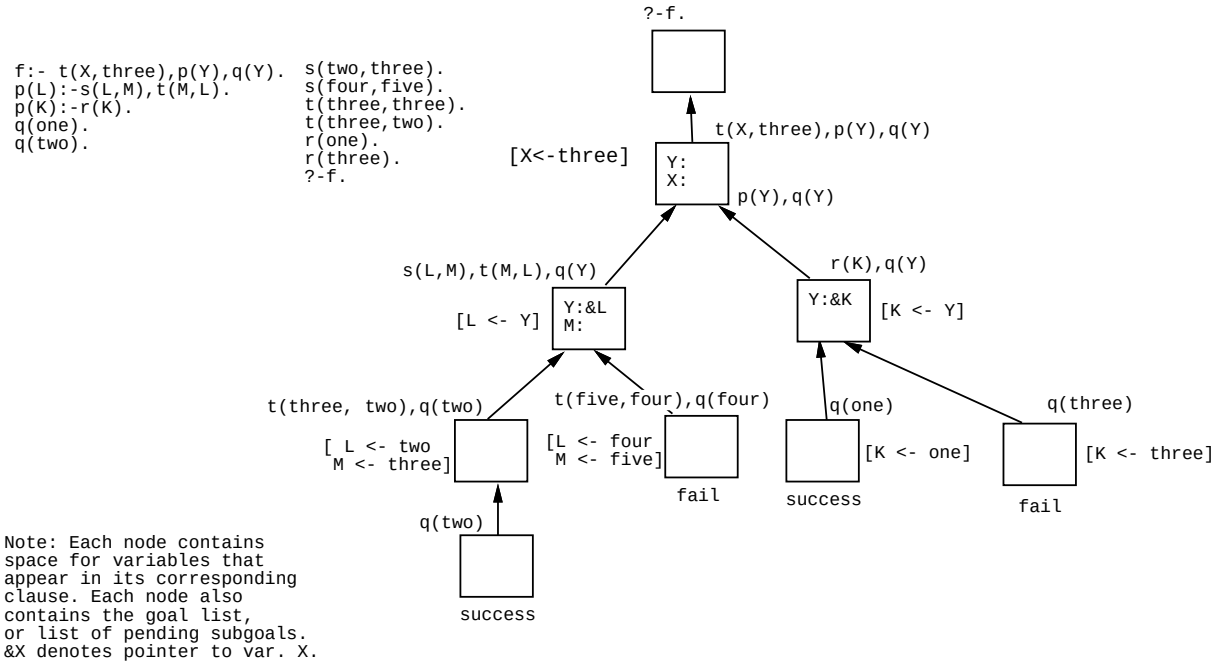



Fig. 3. An Or-parallel Tree

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.

Consider a variable `V` in node $n_1$, whose binding `b` has been created in node $n_2$. If there are no branch points between $n_1$ and $n_2$, then the variable `V` will have the binding `b` in every branch that is created below $n_2$. Such a binding can be stored in-place in `V`—i.e., it can be directly stored in the memory location allocated to `V` in $n_1$. However, if there are branch points between $n_1$ and $n_2$, then the binding `b` cannot be stored in-place, since other branches created between nodes $n_1$ and $n_2$ may impart different bindings to `V`. The binding `b` is applicable to only those nodes that are below $n_2$. Such a binding is known as a *conditional binding* and such a variable as a *conditional variable*. For example, variable `Y` in Figure 3 is a conditional variable. A binding that is not conditional, i.e., one that has no intervening branch points (or choice points) between the node where this binding was generated and the node containing the corresponding variable, is termed *unconditional*. The corresponding variable is called an *unconditional variable* (for example, variable `X` in Figure 3).

The main problem in implementing or-parallelism is the efficient representation of the multiple environments that co-exist simultaneously in the or-parallel tree corresponding to a program's execution. 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 [Warren 1987b; Gupta and Jayaraman 1993b]. 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 or recreated in 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 worker begins execution of a new branch. In [Gupta and Jayaraman 1993b] 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*[4] should be constant-time.

It has been shown that it is impossible to satisfy these three criteria simultaneously [Gupta and Jayaraman 1993b]. 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 workers (where *work* in an or-parallel setting will mean an unexplored branch of the or-parallel tree represented as an untried alternative in a choice-point). The design of the scheduler is very important in an or-parallel system, and is discussed in the context of the various execution models proposed (Section 3.5).

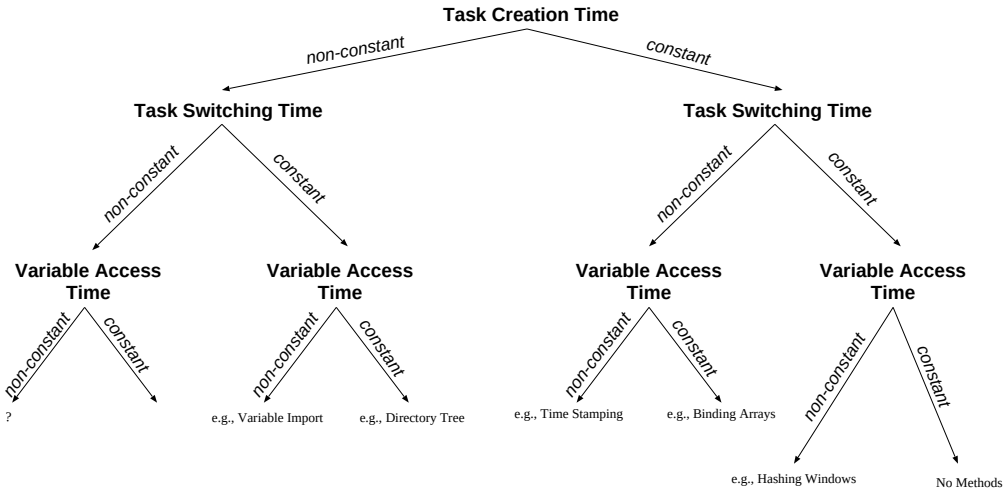
### 3.2 Or-parallel Execution Models

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 [Gupta and Jayaraman 1993b]). These models differ in the technique they employ for solving the problem of environment representation. The three criteria mentioned in the previous section allows us to draw a clean classification of the different models proposed—the models are classified depending on which criteria they meet. This is illustrated in Figure 4; the different models will be associated to one of the leaves of the tree, depending on which criteria they meet and which criteria they violate. Observe that the rightmost leave in the tree is necessarily empty, since no model can meet all the three criteria (this is discussed more formally in Section 3.4). The classification of the models presented in this section is summarized in the table in Figure 4.

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 [Hausman, Ciepielewski, Haridi 1987], Binding Arrays Scheme [Warren 1984; 1987a], Argonne-SRI Model [Warren 1987b], Manchester-Argonne Model [Warren 1987b], Delphi Model [Clocksin and Alshawi 1988], Randomized Method [Janakiram, Agarwal, Malhotra 1988], BC-Machine [Ali 1987], MUSE [Ali and Karlsson 1990a] (and its variations, such as stack splitting [Gupta and Pontelli 1999a], SBA [Correia et al. 1997], PBA [Gupta and Santos Costa 1992c; Gupta, Santos Costa, Pontelli 1994]), Virtual Memory Binding Arrays model [Véron et al. 1993] and, Kabu-Wake Model [Masuzawa et al. 1986]; 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 [Ciepielewski and Haridi 1983], and Environment Closing Method [Conery 1987b]; 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 [Borgwardt 1984], Favored-Bindings Model [Disz et al. 1987], and Virtual Memory Hashing Windows model [Véron et al. 1993]. Likewise, example of a model that only satisfies criterion 1 (constant time task-creation) is the Time-Stamping Model [Tinker 1988], while the example of a model that only satisfies criterion 3 (constant-time task switching) is the Variable Import Scheme [Lindstrom 1984]. We describe some of these execution models for or-parallelism in greater detail below. A detailed study and derivation of some of the or-parallel models has also been done in [Warren 1987b]. Some alternative models for or-parallelism, such as Sparse Binding Array and Paged Binding Arrays, are separately described in Section 6.3, since their design is mostly motivated by the desire of integrate exploitation of or- and and-parallelism.

As noted in Figure 4, we are also imposing an additional classification level, which separates the models proposed into two classes. The first class contains all those models in which the different workers explore a unique representation of the computation tree—which is shared between workers. The second class contains

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




| | MODEL | Constant-time Task Creation | Constant-time Task Switching | Constant-time Variable Access |
|---|---|---|---|---|
| Shared Tree | Directory Tree | | X | X |
| | Hashing Windows | X | X | |
| | Favored Bindings | X | X | |
| | Time Stamping | X | | |
| | Environment Closing | | X | X |
| | Binding Arrays | X | | X |
| | Version Vectors | X | | X |
| Non-Shared Tree | Kabu-Wake | X | | X |
| | MUSE | X | | X |
| | Delphi | X | | X |

Fig. 4. Classification of Or-parallel Models

those models in which every worker maintains a separate data structure representing (part of) the computation tree.

### 3.2.1 *Shared Representation of Computation Tree*

*A. Directory Tree Method*

In the directory tree method [Ciepielewski and Haridi 1983], developed by Ciepielewski and Haridi in the early 80s for their or-parallel Token Machine [Ciepielewski and Hausman 1986], each branch of the or-tree has an associated process. A process is created each time a new node in the tree is created, and the process expires once the creation of the children processes is completed. 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. The complete binding environment of a process is described by a *directory*—thus, a directory is essentially a "summary" of a branch up to the node representing the process. 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 $i$th location in the directory contains a pointer to the $i$th context for that process.

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 (i.e., associating one of the processes to an actual processor) does not involve any state change.

A prototypical implementation of this scheme was developed and some results concerning memory performance are reported in [Ciepielewski and Hausman 1986]. The cost of directories creation is potentially very high and the method leads to large memory consumption and poor locality [Crammond 1985].

*B. Hashing Windows Method*

The hashing windows scheme, proposed by Borgwardt [Borgwardt 1984], 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.

The hashing windows scheme has found implementation in the Argonne National Laboratory's Parallel Prolog [Butler et al. 1986] and in the *PEPSys* system [Westphal, Robert, Chassin, Syre 1987; Chassin de Kergommeaux and Robert 1990]. The goal of the PEPSys (Parallel ECRC Prolog System) project was to develop technology for the concurrent exploitation of and-parallelism and or-parallelism (details on how and-parallelism and or-parallelism are combined are discussed in Section 6.3.1). The implementation of hashing windows in PEPSys is optimized w.r.t. what mentioned earlier. Bindings are separated into two classes [Chassin de Kergommeaux and Robert 1990]:

- *Shallow Bindings:* these are bindings which are performed by the same process which created the variables; such bindings are stored in-place (in the environment). A stamp (called *Or-Branch-Level (OBL)*) is also stored with the binding. The OBL keeps track of the number of choice points present in the stack at each point in time.
- *Deep Bindings:* these are binding performed to variables which lay outside of the local computation. Access to such bindings is performed using hashing windows.

Variable look-up makes use of the OBL to determine whether the in-place binding is valid or not—by comparing the OBL of the binding with the OBL existing at the choice point which originated the current process. Details of these mechanisms are presented in [Westphal, Robert, Chassin, Syre 1987]. A detailed study of the performance of PEPSys has been provided in [Chassin de Kergommeaux 1989].

*C. Favored-Bindings Method*

The favored binding method [Disz et al. 1987] 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.

*D. Time Stamping Method*

The time-stamping method, developed by Tinker and Lindstrom [Tinker 1988], uses time stamps to distinguish the correct bindings for an environment. All bindings for a variable are visible to all the workers (which are distinct processes created when needed). 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 process 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.

This scheme was provided as part of the design of the BOPLOG system—an or-parallel Prolog system for BBN's Butterfly architectures (a distributed memory machine with global addressing capabilities). The method suggests a potential for lack of locality of reference, as the global address space is extensively searched in accessing bindings.

*E. Environment Closing Method*

The environment closing method was proposed by Conery [Conery 1987b] and is primarily designed for distributed memory 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 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.

This method trades synchronization time required to exchange variable bindings during parallel computations, with the extra time required to close the environment. The foundation of this method can be traced back to the *Variable Import* method [Lindstrom 1984], where *forward unification* is used to close the environment of a new clause and *backward unification* is used to communicate the results at the end of a clause. The scheme presented by Conery has also been adopted in the ROPM system [Kalé, Ramkumar, Shu 1988].

*F. Binding Arrays Method*

In the binding arrays method [Warren 1984; 1987a] each worker has an auxiliary data structure called the *binding array*.[5] Each conditional variable along a branch is numbered sequentially outward from the root.

To perform 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.[6] The counter is then incremented. When a conditional variable gets bound, the binding is stored in the binding array of the worker 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. Note that bindings of all variables, whether conditional or unconditional, are accessible in constant time. This is illustrated in Figure 5. Worker P1 is exploring the leftmost branch (with terminal success node labeled `n1`). The conditional variables `X` and `M` have been allocated offsets `0` and `1` respectively. Thus, the bindings for `X` and `M` are stored in the locations `0` and `1` of the binding array. The entries stored in the trail in nodes are shown in square brackets in the figure. Suppose the value of variables `M` is needed in node `n1`; `M`'s offset stored in the memory location allocated to it is then obtained. This offset is 1, and is used by worker P1 to index into the binding array, and obtain `M`'s binding. Observe that the variable `L` is *unconditionally* aliased to `X`, and for this reason `L` is made point to `X`. The unconditional nature of the binding does not require allocation of an entry in the binding array for `L`.

To ensure consistency, when a worker switches from one branch (say $b_i$) of the or-tree to another (say $b_j$), it has to update its binding array by de-installing bindings from the trail of the nodes that are in $b_i$ and installing the correct bindings from the trail of the nodes that are in $b_j$. For example, suppose worker P1 finishes work along the current branch and decides to migrate to node `n2` to finish work that remains there. To be able to do so, it will have to update its binding array so that the state that exists along the branch from root node to node `n2` is reflected in its environment. This is accomplished by making P1 to travel up along the branch from node `n1` towards the least common ancestor node of `n1` and `n2`, and removing those conditional bindings from its binding array that it made on the way down. The variables whose bindings need to be removed are found in the trail entries of intervening nodes. Once the least common ancestor node is reached, P1 will move towards node `n2`, this time installing conditional bindings found in the trail entries of nodes passed along the way. This can be seen in Figure 5. In the example, while moving up worker P1 untrails the bindings for `X` and `M`, since the trail contains references to these two variables. When moving down to node `n2`, worker P1 will retrieve the new bindings for `X` and `M` from the trail and install them in the binding array.
The binding arrays method has been used in the Aurora or-parallel system, which is described in more detail in Section 3.5. Other systems have also adopted the binding arrays method (e.g., the Andorra-I system [Santos Costa, Warren, Yang 1991a]). Furthermore, a number of variations on the idea of binding arrays have been proposed—e.g., Paged Binding Arrays, Sparse Binding Arrays—mostly aimed to provide better support for combined exploitation of and-parallelism and or-parallelism. These are discussed in Sections 6.3.6 and 6.3.7.

### *G. Versions Vectors Method*

The versions vectors method [Hausman, Ciepielewski, Haridi 1987] 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 worker with processor-id $i$ (processor ids are numbered from 1 to $n$, where $n$ is the total number of workers) is stored at offset $i$ in the vector. The binding is also recorded in the trail, as in the binding arrays method. Like in the binding arrays method, on switching to another branch a worker with pid $j$ has to update the $j$th slots of versions vectors of all conditional variables that lie in the intervening nodes to reflect the correct bindings corresponding to the new site.

[5] Note that the description that follows is largely based on [Warren 1987a] rather than on [Warren 1984]. The binding arrays technique in [Warren 1984] is not primarily concerned with or-parallelism but rather with (primarily sequential) non-depth-first search.

[6] Most systems, e.g., Aurora, initially treat all the variables as conditional, thus placing them in the binding array.

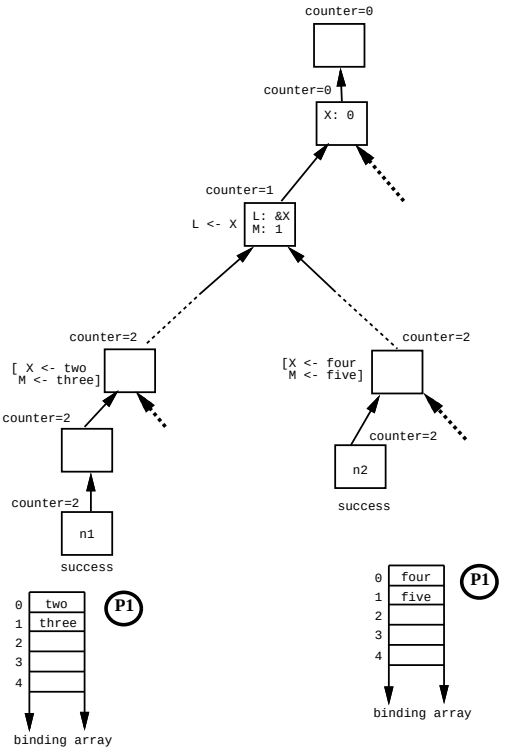



Fig. 5. The Binding Arrays Method

To our knowledge the method has never been integrated in an actual prototype. Nevertheless, the model has the potential to provide good performance, including the ability to support the orthogonality principle required by combined exploitation of and-parallelism and or-parallelism (see Section 6.3.7).

### 3.2.2 *Non-Shared Representation of Computation Tree*

*A. Stack-copying Method*

In the Stack-copying method [Ali and Karlsson 1990a; 1990a] a separate environment is maintained by each worker in which it can write without causing any binding conflicts. In Stack-copying even unconditional bindings are not shared, as they are in the other methods described above. When an idle worker P2 picks an untried alternative from a choice-point created by another worker P1, it copies all the stacks of P1. As a result of copying, each worker can carry out execution exactly like a sequential system, requiring very little synchronization with other workers.

In order to avoid duplication of work, part of each choice point (specifically the set of unexplored alternatives) is moved to a frame created in an area easily accessible by each worker. This allows the system to maintain a *single* list of unexplored alternatives for each choice point, which is accessed in mutual exclusion by the different workers. A frame is created for each shared choice point and is used to maintain various scheduling information (e.g., bitmaps keeping track of workers working below each choice point). This is illustrated in Figure 6. Each choice point shared by multiple workers has a corresponding frame in the separate Shared Space. Access to the unexplored alternatives (which are now located in these frames) will be performed in mutual exclusion, thus guaranteeing that each alternative is executed by exactly one worker.

The copying of stacks can be made more efficient through the technique of *incremental copying*. The idea of *incremental copying* is based on the fact that the idle worker 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. In Figure 7 this is illustrated in an example. In Figure 7(i) we have two workers immediately after a sharing operations which has transferred three choice points from worker `P1` to `P2`. In

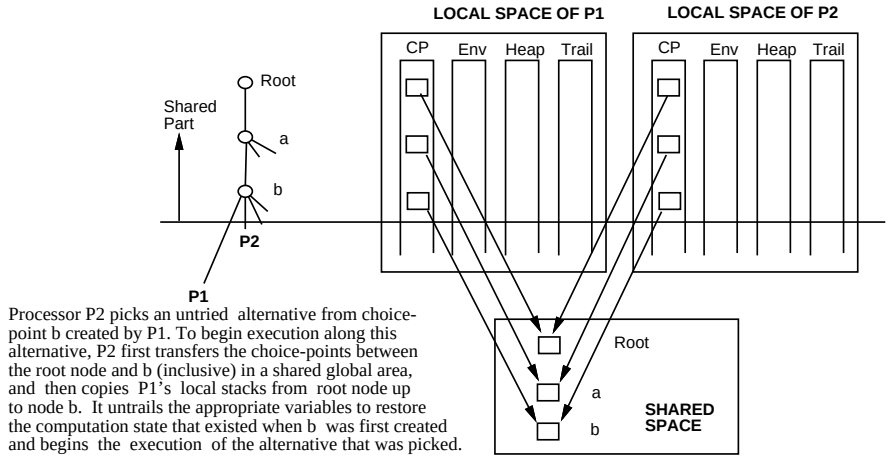



Fig. 6. Stack Copying and Choice Points

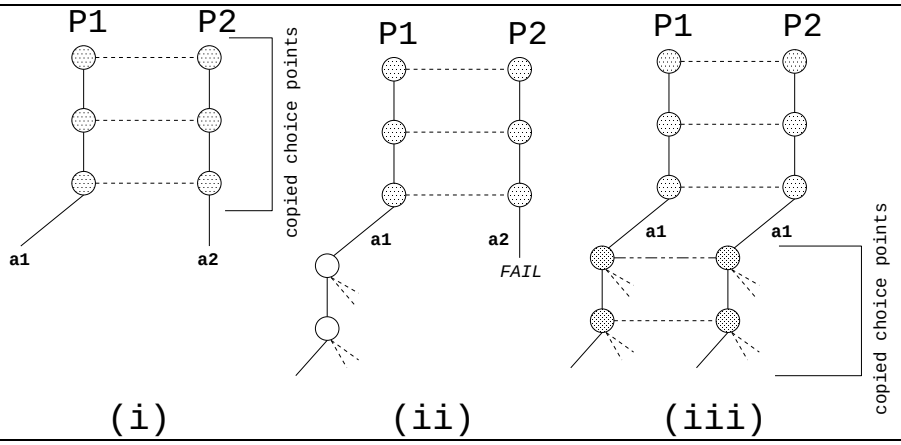



Fig. 7. Incremental Stack Copying

Figure 7(ii) worker `P1` has generated two new (private) choice points while `P2` has failed in its alternative. Figure 7(iii) shows the resulting situation after another sharing between the two workers; incremental copying has been applied, thus leading to the copy of only the two new choice points.

Recently, incremental copying have been proved to have some drawbacks with respect to management of combined and-parallelism and or-parallelism as well as management of special types of variables (e.g., attributed variables). Recent schemes, such as the COWL models (described in Section 6.3.5) overcome many of these problems.

This model is an evolution of the work on BC-machine by Ali [Ali 1987]—a model where different workers concurrently start the computation of the query and automatically select different alternatives when choice points are created. It has also found a different instantiation in the Kabu Wake model [Masuzawa et al. 1986]. In this method, idle workers request work from busy ones, and work is transmitted by copying environments between workers. The main difference w.r.t. the previously described approach is that the source worker (i.e., the busy worker from where work is taken) is required to "temporarily" backtrack to the choice point to be split in order to undo bindings before copying take place.

Stack copying has found efficient implementation in a variety of systems, such as MUSE [Ali and Karlsson 1990a] (discussed in more detail in Section 3.5.2), Eclipse [Wallace, Novello, Schimpf 1997], and YAP [Santos Costa, Damas, Reis, Azevedo 1999]. Stack copying has also been adopted in a number of distributed memory implementations of Prolog, such as OPERA [Briat et al. 1992] and PALS [Villaverde, Guo, Pontelli, Gupta

2000].

*B. Stack Splitting*

In the stack-copying technique, each choice-point has to be *"shared"*—i.e., transfered to a common shared area accessible by all the workers—to make sure that the selection of its untried alternatives by various concurrent workers is serialized, so that no two workers can pick the same alternative. The shared choice point is locked while the alternative is selected to achieve this effect. As discussed in [Gupta and Pontelli 1999a] this method allows the use of very efficient scheduling mechanisms—such as the scheduling on bottom-most choice point used by Aurora and MUSE—but may cause excessive lock contention, or excessive network traffic if realized on a distributed memory system. However, there are other simple ways of ensuring that no alternative is simultaneously selected by multiple workers: *the untried alternatives of a choice-point can be split between the two copies of the choice-point stack*. This operation is called *Choice-point Stack-Splitting*, or simply *Stack-splitting*. This will ensure that no two workers pick the same alternative.

Different schemes for splitting the set of alternatives between the two (or more) choice-points can be envisioned—e.g., each choice-point receives half of the alternatives, or the partitioning can be guided by additional information regarding the unexplored computation, such as granularity and likelihood of failure. In addition, the need for a shared frame, as a critical section to protect the alternatives from multiple executions, has disappeared, as each stack copy has a choice-point, though their contents differ in terms of which unexplored alternatives they contain. All the choice-points can be evenly split in this way during the copying operation. The choice-point stack-splitting operation is illustrated in figure 8.

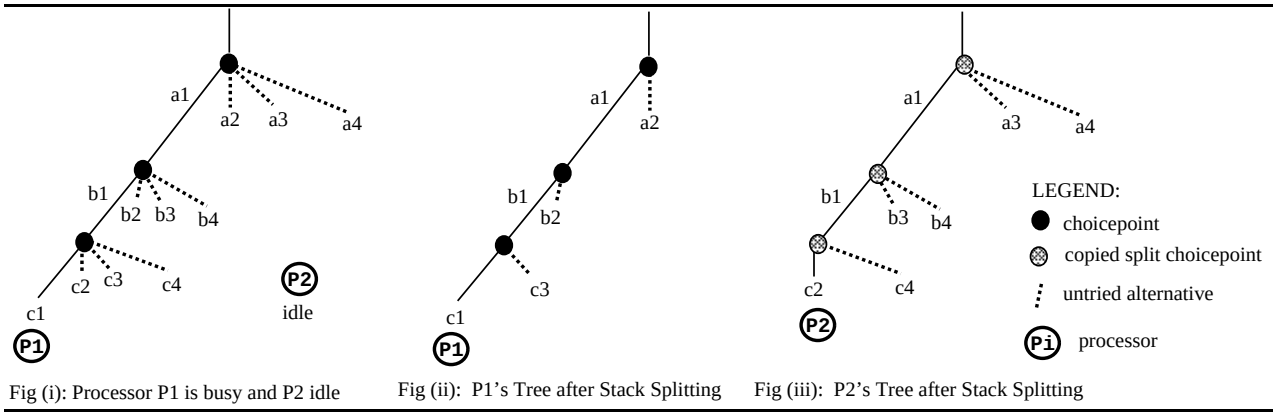



Fig. 8. Stack-splitting based or-parallelism

The major advantage of stack-splitting is that scheduling on bottom-most can still be used without incurring huge communication overheads. Essentially, after splitting the different or-parallel threads become fairly independent of each other, and hence communication is minimized during execution. This makes the stack-splitting technique highly suitable for distributed memory machines. The possibility of parameterizing the splitting of the alternatives based on additional semantic information (granularity, non-failure, user annotations) can further reduce the likelihood of additional communications due to scheduling.

In [Gupta and Pontelli 1999a] results have been reported indicating that for various benchmarks, stack splitting obtains better speedups than MUSE on shared memory architectures—thanks to a better locality of computation and reduced interaction between workers. Preliminary work on implementing stack-splitting on distributed memory machine has also provided positive results in terms of speedups and efficiency [Villaverde, Guo, Pontelli, Gupta 2000].

*C. Recomputation-based Models*

In the stack-copying schemes, an idle worker acquires work by copying the data structures associated to a given segment of the computation, in order to recreate the state of the computation from where the new alternative will start. An alternative approach to copying is to have idle workers recreate such data-structures

by repeating the computation from the root of the or-tree all the way to the choice-point from where a new alternative will be taken. Thus, the content of the stacks of the abstract machine is reconstructed, rather than copied. This approach is at the base of the Delphi system [Clocksin and Alshawi 1988] and of the Randomized Parallal Backtracking method [Janakiram, Agarwal, Malhotra 1988].

These recomputation-based methods have the clear advantage of reducing the interactions between workers during the sharing operations. In Delphi, the exchange of work between workers boils down to the transfer of an *oracle* from the busy worker to the idle one. An oracle contains identifiers which describe the the path in the or-tree that the worker needs to follow to reach the unexplored alternative. A centralized controller is in charge of allocating oracles to idle agents. The method has attracted considerable attention, but has provided relatively modest parallel performances on arbitrary Prolog programs. Variations of this method have been effectively used to parallelize specialized types of logic programming computations (e.g., in the parallelization of Stable Logic Programming computations [Pontelli and El-Khatib 2001]).

### 3.3 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 extra-logical, meta-logical and input/output predicates have been incorporated in Prolog. Some researchers have taken the view that a parallel logic programming system should transparently execute Prolog programs in parallel [Lusk et al. 1990; Hausman, Ciepielewski, Calderwood 1988]. That is, the same effect should be seen by a user during parallel execution of a Prolog programs, as far as input/output etc. are concerned (including printing of the final solutions), as in its sequential execution with Prolog computation and selection rules. Such a system is said to support *(observable) 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 extra-logical predicates only when the branch containing it becomes the leftmost in the search tree. Different techniques have been proposed to detect when a branch of the or-parallel tree becomes the leftmost active branch in the tree [Ali and Karlsson 1990a; Kalé, Padua, Sehr 1988; Sindaha 1993]. Arguably, the techniques used in Aurora have been the most well researched and successful [Hausman, Ciepielewski, Calderwood 1988; Hausman 1990]. In this approach, the system maintains for each node $n$ in the search tree a pointer to one of its ancestor nodes $m$, called the *sub-root node*, which represents the highest ancestor (i.e., closer to the root) such that $n$ lies in the leftmost branch of the tree rooted at $m$. If $m$ is equal to the root of the tree, then the node $n$ is leftmost branch of the search tree.

In addition to this, various or-parallel Prolog systems (e.g., Aurora and MUSE) provide variants of the different order-sensitive predicates which can be executed without requiring any form of synchronization—these are typically called *cavalier* predicates. The use of cavalier extra-logical predicates leads to an operational behavior different from that of Prolog—e.g., a cavalier write operation is going to be executed immediately irrespectively of the execution of the other extra-logical predicates in the search tree.

An issue that arises in the presence of pruning operators such as cuts and commits during or-parallel execution is that of *speculative work* [Hausman 1989; 1990; Ali and Karlsson 1992b; Beaumont and Warren 1993; Sindaha 1992]. 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 sequential Prolog semantics entail that if `q(X)` succeeds then the second clause for `p` shall 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 supports sequential Prolog semantics to take speculative work into account. Essentially, such a scheduler should bias all the workers 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 [Ali and Karlsson 1992b; Beaumont 1991; Beaumont and Warren 1993; Sindaha 1992].

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 [Ciepielewski 1992]. 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 in Section 3.5, where a significant amount of effort has been invested in designing and fine-tuning the or-parallel system and its schedulers.

## 3.4 Problem Abstraction and Complexity

### 3.4.1 *Abstraction of the Problems*

In this section we provide a brief overview of the theoretical abstraction of the problems arising in or-parallel execution of Prolog programs. Complete details regarding this study can be found elsewhere [Ranjan, Pontelli, Gupta 1999]. Execution of a program can be abstracted as building a (rooted, labeled) tree. For the sake of simplicity, we will assume that the trees are binary; this assumption does not lead to any loss of generality because, for a given program, the number of branches at any given node is bounded by some constant. The process of building the tree can be abstracted through the following three operations:

(1) create_tree($\gamma$) which creates a tree containing only the root, with label $\gamma$;

(2) expand($u, \gamma_1, \gamma_2$) which, given one leaf $u$ and two labels $\gamma_1$ and $\gamma_2$, creates two new nodes (one for each label) and adds them as children of $u$ ($\gamma_1$ as left child and $\gamma_2$ as right child);

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

These three operations are assumed to be the only ones available to modify the "physical structure" of the tree.

The abstraction of an or-parallel execution should account for the various issues present in or-parallelism—e.g., management of variables and of their bindings, creation of tasks etc. Variables that arise during execution, whose multiple bindings have to be correctly maintained, can be modeled as attributes of the nodes in the tree. $\Gamma$ denotes a set of $M$ variables. If the computation tree has size $N$, then it is possible to assume $M = O(N)$. At each node $u$, three operations are possible:

- assign a variable $X$ to a node $u$.
- dereference a variable $X$ at node $u$—that is, identify the ancestor $v$ of $u$ (if any) which has been assigned $X$.
- alias two variables $X_1$ and $X_2$ at node $u$; this means that for every node $v$ ancestor of $u$, every reference to $X_1$ in $v$ will produce the same result as $X_2$ and vice-versa.

The previous abstraction assumed the presence of one variable binding per node. This restriction can be made without loss of generality—it is always possible to assume that the number of bindings in the node is bound by a program dependent constant. The problem of supporting these dynamic tree operations has been referred to as the $\mathcal{OP}$ problem [Ranjan, Pontelli, Gupta 1999].

### 3.4.2 *Complexity on Pointer Machines*

In this section we summarize the complexity results that have been developed for the abstraction of or-parallelism described in the previous section. The complexity of the problem has been studied on *pointer machines* [Ben-Amram 1995; Schönhage 1980]. Pointer machine is a formal model for describing algorithms, which relies on an elementary machine whose memory is composed only by records connected via pointers. The interesting aspect of this model is that it allows a more refined characterization of complexity than the more traditional RAM model.

**Lower Bound for $\mathcal{OP}$:** As mentioned earlier, the only previous work that deals with the complexity of the mechanisms for or-parallelism is [Gupta 1994; Gupta and Jayaraman 1993b]. This previous work provides an informal argument to show that a generic $\mathcal{OP}$ problem with $N$ variables and $M$ operations has a lower bound

which is strictly worse than $\Omega(N+M)$. Intuitively, this means that no matter how good is an implementation model for or-parallelism, it will incur some costs during the execution which are dependent on the size of the computation (e.g., the number of choice points created). This intuitive result has been formally proved to hold in [Ranjan, Pontelli, Gupta 1999], and can be summarized by the following theorem:

THEOREM 3.1. *On pointer machines, the worst case time complexity of $\mathcal{OP}$ is $\Omega(\lg N)$ per operation even without aliasing.*

The basic idea of the proof is that since there is no direct addressing in the pointer machines starting from a particular node only a "small" number of nodes can be accessed in a small number of steps. Thus, if we need to relate variables and choice points in a very large tree, we need to incur a cost which is dependent on the size of the tree. Thus, at least one of the operations involved in the $\mathcal{OP}$ problem will take in the worst case an amount of time which is at least as large as $\lg N$ (where $N$ is the number of choice points in the computation tree).

It is also interesting to point out that the result does not depend on the presence of the alias operation; this means that the presence of aliasing between unbound conditional variables during an or-parallel execution does not create any serious concern (note that this is not the case for other forms of parallelism, where aliasing is a major source of complexity).

The result essentially states that, no matter how smart is the implementation scheme selected, there will be cases which will lead to a non-constant time cost. This proof confirms the result conjectured in [Gupta and Jayaraman 1993b]. This non-constant time nature is also evident in all the implementation schemes presented in the literature—e.g., the creation of the shared frames and the copying of the choice points in MUSE [Ali and Karlsson 1990a], the installation of the bindings in Aurora [Lusk et al. 1990], management of time-stamps in various other models [Gupta 1994].

**Upper Bound for $\mathcal{OP}$:** The relevant research on complexity of the $\mathcal{OP}$ problem has been limited to showing that a constant time cost per operation cannot be achieved in any implementation scheme. Limited effort has been placed to supply a tight upper bound to this problem. Most of the implementation schemes proposed in the literature can be shown to have a worst case complexity of $O(N)$ per operation. Currently, the best result achieved is the following:

THEOREM 3.2. *The $\mathcal{OP}$ problem with no aliasing can be solved on a pointer machine with a single operation worst–case time complexity of $O(\sqrt[3]{N}(\lg N)^k)$ for a small $k$.*

| *Method* | *Complexity* |
|---|---|
| **Known Upper Bound** | $\tilde{O}(K \times N^{\frac{1}{3}})$ |
| Stack Copying [Ali and Karlsson 1990a] | $\tilde{O}(K \times N)$ |
| Directory Tree Method [Gupta 1994] | $\tilde{O}(K \times N \lg N)$ |
| Binding Arrays [Lusk et al. 1990] | $\tilde{O}(K \times N)$ |
| Environment Closing [Gupta 1994] | $\tilde{O}(K \times N)$ |

Table I. Worst-case Complexity of Some Or-parallel Schemes ($K$ operations)

The lower bound produced, $O(\lg N)$ per operation, is a confirmation and refinement of the results proposed by Gupta and Jayaraman [Gupta and Jayaraman 1993b], and a further proof that an ideal or-parallel system (where all the basic operations are realized with constant-time overhead) cannot be realized. The upper bound, $\tilde{O}(\sqrt[3]{N})$, even if far from the lower bound, is of great importance, as it indicates that (at least *theoretically*) there are implementation schemes which have a worst case time complexity better than that of the existing models. Table I compares the *worst case* time complexity of performing a sequence of $K$ operations, on an

$N$ node tree, for some of the most well known schemes for or-parallelism [Gupta 1994]. The proof of theorem 3.2 indeed provides one of such models—although it is still an open issue whether the theoretical superiority of such model can be translated into a practical implementation scheme.

## 3.5 Experimental Systems

In this section we illustrate in more detail two of the most efficient or-parallel systems implemented.

### 3.5.1 *The Aurora Or-parallel Prolog System*

Aurora is a prototype or-parallel implementation of the full Prolog language developed for UMA (Uniform Memory Access) shared-memory multiprocessors such as the Sequent Symmetry and subsequently ported [Mudambi 1991] to NUMA (Non-Uniform Memory Access) architectures such as the BBN TC-2000 (a scalable architecture with Motorola 88000 processors[7]). Let us remind that UMA architectures are characterized by the fact that each processor in the system guarantee the same average access time to any memory location, while NUMA architectures (e.g., clusters of shared memory machines) may lead to different access time depending on the memory location considered.

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 (initially 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 [Warren 1987a] and refined in [Lusk et al. 1990]. The SRI-model employs binding arrays 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 [Szeredi, Carlsson, Yang 1991; Carlsson 1990] 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 [Santos Costa, Warren, Yang 1991a] to support both and- and or-parallelism. The Aurora engine and compiler [Carlsson 1990] were constructed by adapting SICStus Prolog 0.6 [Carlsson et al. 1995]. Garbage collection for Aurora has been investigated by Weemeeuw [Weemeeuw and Demoen 1990].

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). However, during 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. This overhead incurred, 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 the private region is exactly like sequential Prolog execution. Nodes are transferred from the private region of a worker $P$ to the public region by the scheduler, which 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 an order-sensitive predicate, the given branch is suspended until it becomes leftmost [Hausman 1990]. This technique ensures that the order-sensitive predicates are executed in the same left-to-right order as in a sequential implementation, thus preserving compatibility with these implementations.

[7] Although the porting did not involve modifications of the system structure to take full advantage of the architecture's structure.

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 order-sensitive 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.) [Szeredi 1991].

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 [Hausman 1989; 1990] 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 [Disz and Lusk 1987; Herrarte and Lusk 1991; Carro et al. 1993].

Several or-parallel applications for Aurora were studied in [Kluźniak 1990] and [Lusk, Mudambi, Overbeek, Szeredi 1993]. The non-synchronized dynamic database features have been exploited in the implementation of a general algorithm for solving optimization problems [Szeredi 1991].

Three schedulers are currently operational. Two older schedulers were written [Butler et al. 1988; Brand 1988], but have not been updated to comply with the scheduler-engine interface:

- **The Manchester Scheduler.** The Manchester scheduler [Calderwood and Szeredi 1989] 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 [Szeredi 1989].
- **The Bristol Scheduler.** The Bristol scheduler 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. This idea was originally explored in the context of the MUSE system, and successively integrated in a preliminary version of the Bristol Scheduler [Beaumont et al. 1991]. The present version of the scheduler [Beaumont and Warren 1993] addresses the problem of efficiently scheduling speculative work. It actively seeks the least speculative, selecting a leftmost branch if the work is speculative and a 'richest' branch (i.e., branch with most work) if the work is non-speculative.
- **The Dharma Scheduler.** The Dharma scheduler [Sindaha 1993; 1992] 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.

The speed-ups obtained by all schedulers of Aurora for a diverse set of benchmark programs have been very encouraging. Some of the benchmark programs contain significant amount of speculative work, in which speed-ups are measured for finding the first (leftmost) solution. The degree of speedup obtained for such benchmark programs depends on where in the Prolog search tree the first solution is, and on the frequency of workers moving from right to left towards less speculative work. There are other benchmark programs that have little or no speculative work because they produce all solutions. The degree of speedup for such benchmark programs depends on the amount of parallelism present and on the granularity of parallelism.
More on the Aurora system, and a detailed discussion of its performance results, can be found in [Calderwood and Szeredi 1989; Szeredi 1989; Beaumont et al. 1991; Beaumont and Warren 1993; Sindaha 1992]. Recently, Aurora was also ported on distributed memory architectures [Silva and Watson 2000].

### 3.5.2 *The MUSE Or-parallel Prolog System*

The MUSE or-parallel Prolog system has been designed and implemented on a number of UMA and NUMA computers (Sequent Symmetry, Sun Galaxy, BBN Butterfly II, etc.) [Ali and Karlsson 1990a; 1990a; 1992a; Ali et al. 1992; Ali and Karlsson 1992b; Karlsson 1992]. 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 (SICStus WAM [Carlsson et al. 1995]).

The MUSE model assumes a number of extended WAMs (called *workers*, as in Aurora), 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. To allow copying of memory between workers without the need of any pointer relocation operation, MUSE makes use of a sophisticated memory mapping scheme. The memory is partitioned between the different workers; each worker is implemented as a separate process, and each process maps its own local partition to the same range of memory addresses—which allows for copying without pointer relocations. The partitions belonging to other processes are instead locally mapped to different address ranges. This is illustrated in Figure 9. The partition of worker 1 is mapped at different address ranges in different workers; the local partition reside at the same address range in each worker.

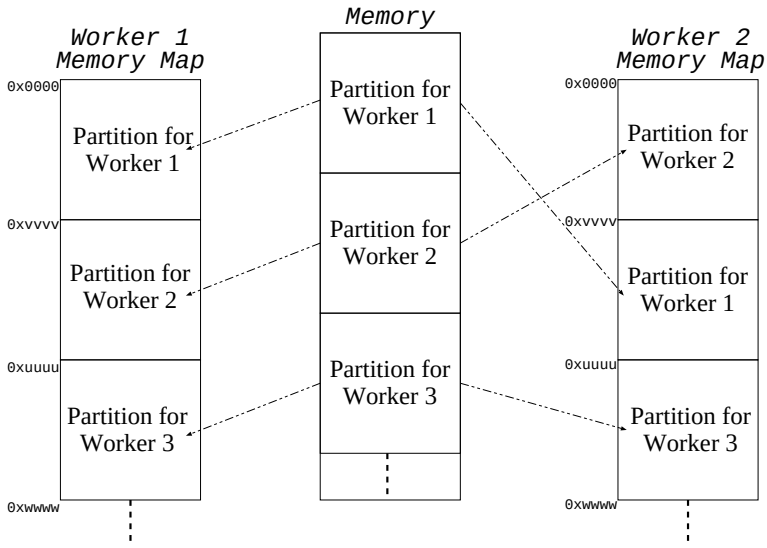



Fig. 9. Memory Organization in MUSE

Workers make a number of choice-points sharable, and they get work from those shared choice-points (nodes) by the normal backtracking of Prolog. Like Aurora, the Muse system has two components: the engine and the scheduler. The engine performs the actual Prolog work, while 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 [Carlsson et al. 1995]. 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 and non-speculative work [Ali and Karlsson 1992b]. For purposes of scheduling, the Prolog tree is divided into two sections: the right section contains voluntarily suspended work and the left section contains active work. Voluntarily suspended work refers to the work that was suspended because the worker doing it found other work to the left of the current branch that is less speculative (recall that following sequential Prolog semantics, the more a branch is to the right in the or-parallel tree the more speculative it is because its chances of being pruned away by a cut are higher). Active work is work that is non-speculative and is actively pursued by workers. The available workers concentrate on the available non-speculative work in the left section. When the amount of work in the left section is not enough for the workers, some of the leftmost part of the voluntarily suspended section (i.e., speculative work) will be resumed. A worker doing speculative work will always suspend its current work and migrate to another node to its left if that node has less speculative work.

The scheduling strategy for non-speculative work, in general, 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 non-speculative 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 non-speculative work in the system, it will resume the leftmost part of the voluntarily 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 up to 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 [Ali and Karlsson 1992a; Ali et al. 1992; Ali and Karlsson 1992b; Karlsson 1992].

MUSE can be considered one of the first commercial parallel logic programming systems to ever be developed—MUSE has been included for a number of years as part of the standard distribution of SICS-tus Prolog [Carlsson et al. 1995][8].

## 4. 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 (e.g., occurrence of the same variable in the argument terms of two subgoals at runtime) 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 in the next subsection.

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.
```

Assuming the execution of this program by supplying the first argument as input, the two lists 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.

Similarly to the case of or-parallelism, development of an and-parallel computation can be depicted using a tree structure (*and-tree*). In this case, each node in the tree is labeled by a conjunction of subgoals and it contains as many children as subgoals in the conjunction. Figure 10 illustrates a simple and-tree for the execution of `fib(2,X)` w.r.t. the above program. The dashed line in Figure 10 is used to denote the fact that it is irrelevant whether the subgoal `X is N1+N2` is a child of either of the two nodes above.

Independent 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.).

[8] MUSE is not supported anymore by SICS.

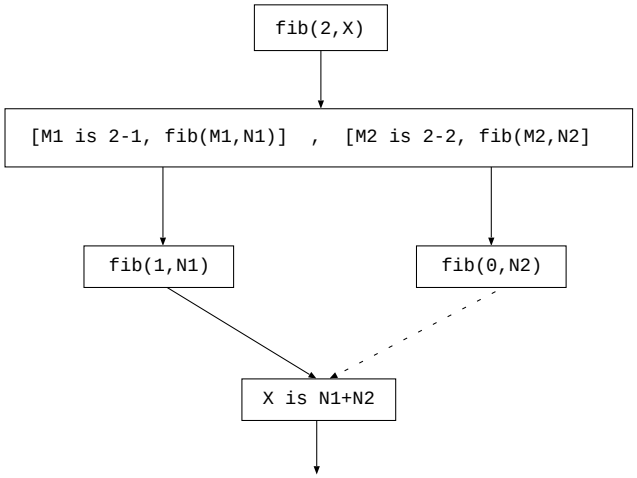



Fig. 10. An And-tree for And-parallelism

### 4.1 Problems in Implementing Independent And-parallelism

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

An independent and-parallel execution can be divided into three phases [Conery and Kibler 1983]:

i. Ordering Phase: deals with detection of dependencies among goals.
ii. Forward Execution Phase: deals with the steps needed to select the next subgoal for execution and initiate its execution.
iii. Backward Execution Phase: deals with steps to be taken when a goal fails, i.e., the operation of backtracking.

#### 4.1.1 *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 an and-parallel execution is initiated without caring for data dependencies, then it may lead to wasteful computation. Consider the following rules:

```
solves(X) :- producer(X), consumer(X).
```

and the goal

```
?- solves(Z).
```

Suppose `X` is an "output argument" for `producer` and "input argument" for `consumer`, i.e., there is a data dependency between the two subgoals. Suppose we initiate their execution in parallel and assume that `producer` produces the binding `X = a` while `consumer` concurrently searches for values of `X`. The goal `consumer` 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, `consumer` computes bindings for `X` which will eventually be discarded, hence these computations are wasted. If `consumer` had known that the only permissible value of `X` is `a`, its search space would have been narrowed. Secondly, every time `consumer` produces a binding for `X` the binding value has to be unified with the binding value produced by `producer` to determine that they are identical. This unification, termed *back-unification* [Wise 1986], can introduce some extra overhead.

Data dependencies cannot always be detected at compile time, because in many cases they arise only during program execution. Consider the clause:

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

It may appear that the 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 independent 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 may miss many instances of independent and-parallelism but incur very little run-time overhead, while purely run time techniques may capture maximal independent and-parallelism at the expense of costly overhead. Data dependencies cannot always be detected entirely at compile time, although compile-time analysis tools can uncover a significant number of them. 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 are known to become bound before the subgoal starts and output variables are variables that will be bound by the subgoal during its execution.
Modes have also been introduced in the committed choice languages [Tick 1995; Shapiro 1987] to actually control the and-parallel execution (but leading to an operational semantics different from Prolog's one).

ii. Static Data Dependency Analysis: In this technique the goal and the program clauses are globally analyzed at compile time, assuming a worst cases for subgoal dependencies [Chang, Despain, DeGroot 1985]. No checks are done at runtime. Since the analysis is done at compile-time, assuming a 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, e.g., by Conery in his AND/OR model [Conery and Kibler 1981; 1983; Conery 1987a]. 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 independent and-parallelism could be potentially exploited (but after paying a significant cost at runtime). A simplified version of this idea has also been used in the APEX system [Lin and Kumar 1988]. In this model a token-passing scheme is adopted: a token exists for each variable and is made available to the leftmost subgoal accessing the variable. A subgoal is executable as soon as it owns the tokens for each variable in its binding environment.

iv. A fourth approach, which is midway between (ii) and (iii), encapsulates the dependency information in the code generated by the compiler—in the form of source code annotations—along with the addition of some extra conditions (tests) on 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) [DeGroot 1984], and was formalized and enhanced by Hermenegildo and Nasr [Hermenegildo and Nasr 1986]. Although it does not capture all the instances of independent and-parallelism present in the program, it does manage to exploit a substantial part of it.
The typical format used to describe the annotations produced to identify instances of independent and-parallelism is the following:

$$\texttt{( conditions} \Rightarrow \texttt{goal}_1 \texttt{ \& ... \& goal}_n \texttt{ )}$$

where '&' indicates a *parallel conjunction*—i.e., subgoals that can be solved concurrently (while the "," is maintained to represent *sequential conjunction*, i.e., to indicate that the subgoals should be solved sequentially). This form of annotation is discussed in detail in Section 4.3.

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 can be generated via compile-time analysis [Muthukumar and Hermenegildo 1989a; 1991; Jacobs and Langen 1989; Hermenegildo and Green 1991] based on abstract interpretation [Cousot and Cousot 1977; 1992].

#### 4.1.2 *Forward Execution Phase*

The forward execution phase follows the ordering phase. It selects independent goals that can be executed in independent and-parallel, and initiates their execution. The execution continues like normal sequential Prolog 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 the efficient determination of the goals that are ready for independent and-parallel execution. Different models have adopted different approaches to tackle this issue, and they are described in the successive subsections.

Various works have pointed out the importance of good scheduling strategies. Work by Hermenegildo et al. [Hermenegildo 1987] provided ideas on using more sophisticated scheduling techniques aimed at guaranteeing a correct match between the logical organization of the computation and its physical distribution on the stacks—with the aim of simplifying backtracking. Related research on scheduling for independent and-parallel systems has been proposed by Dutra [Dutra 1994]. In [Pontelli and Gupta 1995a] a methodology is described which adapts scheduling mechanisms developed for or-parallel systems to the case of independent and-parallel system. In the same way in which or-parallel system tries to schedule first work that is more likely to succeed, and-parallel systems will gain from scheduling first work that is more likely to fail. The advantage of doing this comes from the fact that most IAP systems supports intelligent forms of backtracking over and-parallel calls, which allow to quickly propagate failure of a subgoal to the whole parallel call. Thus, if a parallel call does not have solutions, the sooner we find a failing subgoal, the sooner backtracking can be started. Some experimental results have been provided in [Pontelli and Gupta 1995a] to support this perspective. This notion is also close to the *first-fail principle* widely used in constraint logic programming [Van Hentenryck 1989b].

#### 4.1.3 *Backward Execution Phase*

The need for a backward execution phase arises from the non-deterministic nature of logic programming—a program's execution involves choosing at each resolution step one of multiple candidate clauses, and this choice may potentially lead to distinct solutions.

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.

In presence of IAP, backtracking becomes considerably more complex, especially if the system strives to explore the search space in the same order as in a sequential Prolog execution; in particular

- IAP leads to the loss of correspondence between logical organization of the computation and its physical layout; this means that *logically* contiguous subgoals (i.e., subgoals which are one after the other in the resolvent) may be *physically* located in non-contiguous parts of the stack, or in stacks of different workers. In addition, the order of subgoals in the stacks may not correspond to their backtracking order.
  This is illustrated in the example in Figure 11. Worker 1 starts with the first parallel call, making `b` and `c` available for remote execution and locally starting the execution of `a`. Worker 2 immediately starts and completes the execution of `b`. In the meantime, Worker 1 opens a new parallel call, locally executing `d` and making `e` available to other workers. At this point, Worker 2 may choose to execute `e`, and then `c`. The final placement of subgoals in the stacks of the two workers is illustrated on the right of Figure 11. As we can see, the physical order of the subgoals in the stack of Worker 2 does not match the logical order. This will clearly create an hazard during backtracking, since Prolog semantics require to explore first the alternatives of `b` before those of `e`, while the computation of `b` is trapped on the stack below that of `e`.
- backtracking may need to continue to the (logically) preceding subgoal, which may still be executing at the time backtracking takes place.

These problems are complicated by the fact that independent and-parallel subgoals may have nested independent and-parallel subgoals currently executing which have to be terminated or backtracked over.

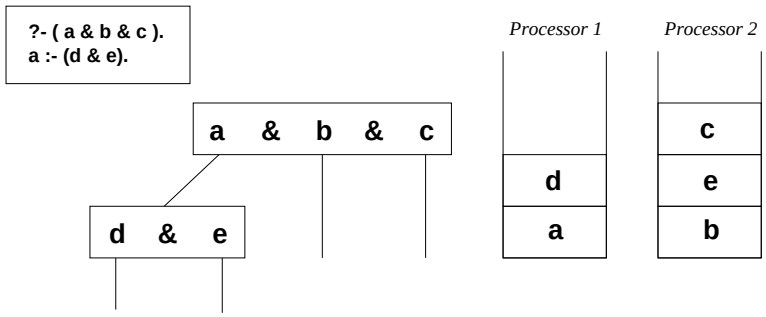



Fig. 11. Lack of Correspondence between Physical and Logical Computation

Considerably different approaches have been adopted in the literature to handle the backward execution phase. The simplest approach, as adopted in models like Epilog, ROPM, AO-WAM [Wise 1986; Ramkumar and Kalé 1989], is based on removing the need for actual backtracking over and-parallel goals through the use of parallelism and solutions reuse. E.g., as shown in Figure 12, two threads of execution are assigned to the distinct subgoals, and they will be used to generate (via local standard backtracking) all solutions to `a` and `b`. The backward execution phase is then replaced by a relatively simpler cross product operation. Although intuitively simple, this approach suffers from major drawbacks, including the extreme complexity of recreating Prolog semantics—i.e., the correct order of execution of order-sensitive predicates as well as the correct repetition of side-effect predicates as imposed in the recomputation-oriented Prolog semantics. In this context, by recomputation-oriented semantics we indicate the fact that a subgoal is completely recomputed for each alternative of the subgoals on its left; e.g., in a goal such as `?- p,q`, the goal `q` is completely recomputed for each solution of `p`.

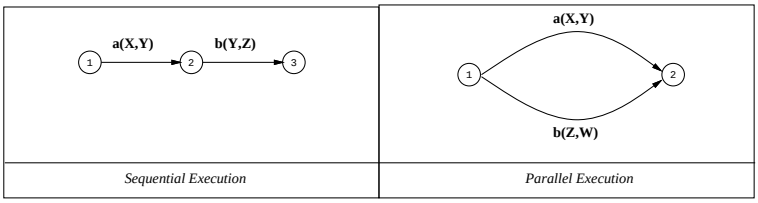



Fig. 12. Solution Reuse

In the context of independent and-parallel systems based on recomputation (such as those proposed by DeGroot [DeGroot 1987], Hermenegildo [Hermenegildo 1986a], Kumar and Lin [Lin and Kumar 1988], and Pontelli and Gupta [Pontelli, Gupta, Tang, Carro, Hermenegildo 1996]), different backtracking algorithms have been proposed. In the past, backtracking algorithms have been proposed which later turned out to be incomplete [Woo and Choe 1986].

The most popular correct backtracking algorithm for IAP has been presented by Hermenegildo and Nasr [Hermenegildo and Nasr 1986] and efficiently developed in &-Prolog [Hermenegildo and Green 1991] and &ACE/ACE [Pontelli and Gupta 1998]. A relatively similar algorithm has also been used in APEX [Lin and Kumar 1988] and the algorithm has been extended to handle dependent and-parallelism as well [Shen 1992a]. Let us consider the following query:

$$\texttt{?- } b_1\texttt{, } b_2\texttt{, (}q_1\texttt{ \& }q_2\texttt{ \& }q_3\texttt{), } a_1\texttt{, } a_2$$

and let us consider the possible cases that can arise whenever one of the subgoals in the query fails.

(1) if either $a_2$ or $b_2$ fails, then standard backtracking is used and backtracking is continued, respectively, in $a_1$ or $b_1$ (see Case 1 in Figure 13);

(2) if $a_1$ fails (*outside backtracking*) then backtracking should continue inside the parallel call, in the subgoal $q_3$ (see Case 2 in Figure 13). The fact that $a_1$ was executing implies that the whole parallel call (and in

particular $q_3$) was completed. In this case the major concern is to identify the location of the computation $q_3$, which may lie in a different part of the stack (not necessarily immediately below $a_1$) or in the stack of a different worker. If $q_3$ does not offer alternative solutions, then, as in standard Prolog, backtracking should propagate to $q_2$ and eventually to $q_1$. Each one of these subgoals may lie in a different part of the stack or in the stack of a different worker. If none of the subgoals returns any alternative solution, then ultimately backtracking should be continued in the sequential part of the computation which precedes the parallel call ($b_2$). If $q_i$ succeeds and produces a new solution, then some parallelism can be recovered by allowing parallel recomputation of the subgoals $q_j$ for $j > i$.

(3) if $q_i$ ($i \in \{1, 2, 3\}$) fails (*inside backtracking*) during its execution, then
- the subgoals $q_j$ ($j > i$) should be removed;
- as soon as the computation of $q_{i-1}$ is completed, backtracking should move to it and search for new alternatives.

This is illustrated in Case 3 of Figure 13. In practice all these steps can be avoided relying on the fact that the parallel subgoals are independent—thus failure of one of the subgoals cannot be cured by backtracking on any of the other parallel subgoals. Hermenegildo suggested a form of semi-intelligent backtracking, in which the failure of either one of the $q_i$ causes the failure of the whole parallel conjunction and backtracking to $b_2$.

To see why independent and-parallel systems should support this form of semi-intelligent backtracking consider the goal:

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

Suppose `b` and `c` are independent subgoals and can be executed in independent 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* [Cox 1984]. As should be obvious, knowledge about data dependencies is needed for both intelligent backtracking as well as independent and-parallel execution. Thus, if an independent and-parallel system performs data dependency analysis for parallel execution, it should take further advantage of it for intelligently backtracking as well. Note that the 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 [Hermenegildo and Nasr 1986; Pontelli and Gupta 1998] as well as those that employ fully intelligent backtracking [Lin 1988; Codognet and Codognet 1989; Winsborough 1987] have been proposed and implemented. In particular, the work Codognet and Codognet [Codognet and Codognet 1989] shows how to use a *Dynamic Conflict Graph* (a unification graph recording for each binding the literal responsible for it), designed to support sequential intelligent backtracking [Codognet, Codognet, Filé 1988] to support both forward and backward and-parallel execution.

A further distinction has been made in the literature [Pontelli, Gupta, Tang, Carro, Hermenegildo 1996], regarding how outside backtracking is carried out:

- *private backtracking:* each worker is allowed to backtrack only on the computations lying in their own stacks. Thus, if backtracking has to be propagated to a subgoal lying in the stack of another worker $P$, then a specific message has be sent to $P$, and $P$ will (typically asynchronously) carry out the backtracking activity;
- *public backtracking:* each worker is allowed to backtrack on any computation, independently from where it resides—it can also backtrack on computations lying on the stack of a different workers.

Private backtracking has been adopted in various systems [Hermenegildo and Green 1991; Shen 1992a]. It has the advantage of allowing each worker to have complete control of the parts of computation which have been locally executed; in particular, it facilitates the task of performing garbage collection as well as local optimizations. On the other hand, backtracking becomes an asynchronous activity, since a worker may not be ready to immediately serve a backtracking request coming from another worker. A proper management of this

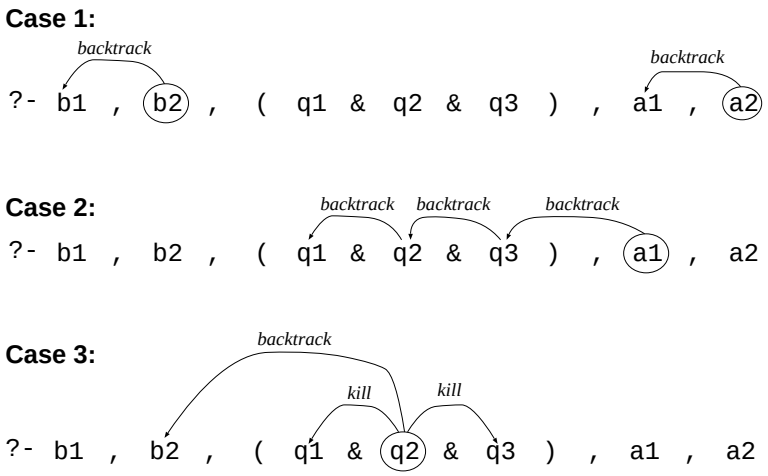



Fig. 13. Backtracking on And-parallel Calls

message passing activities (e.g., to avoid the risk of deadlocks) makes the implementation very complex [Shen 1992b; Pontelli, Gupta, Tang, Carro, Hermenegildo 1996]. Furthermore, experiments performed in the &ACE system [Pontelli and Gupta 1998] demonstrated that public backtracking is considerably more efficient than private backtracking—by allowing synchronous backtracking, without delays in the propagation of failures. At the implementation level, public backtracking is also simpler—just requiring mutual exclusion in the access of certain memory areas. The disadvantage of public backtracking is the occasional inability of immediately recovering memory during backtracking—since in general we cannot allow one worker to recover memory belonging to a different worker.

### 4.2 Support for Full Prolog

Like in the case of or-parallel systems, some researchers have favored supporting Prolog's sequential semantics in independent and-parallel systems [Muthukumar and Hermenegildo 1989b; DeGroot 1987; Chang and Chiang 1989]. 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 an order-sensitive predicate, then the order-sensitive predicate in the right goal can only be performed after the last order-sensitive predicate in the goal to the left has been executed. Given that this property is undecidable in general, it is typically approximated by suspending the side effect until the branch in which it appears is the leftmost in the computation tree—i.e., all the branches on the left have completed. 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 extra-logical predicate (e.g., input/output operations) that would be executed had we backtracked into `b`. Limited intelligent backtracking can be maintained and applied to the subgoals lying on the right of the failing one.

The issue of speculative computation also arises in independent 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 (workers) 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.

### 4.3 Independent And-parallel Execution Models

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

(1) Conery's abstract parallel implementation [Conery and Kibler 1981; 1983];
(2) And-Parallel Execution (APEX) Model of Lin and Kumar [Lin and Kumar 1988]; and,

(3) Restricted And-parallel (RAP) model, introduced by DeGroot [DeGroot 1984], and extended by Hermenegildo and Nasr [Hermenegildo and Nasr 1986; Hermenegildo 1986a] and by Gupta and Pontelli [Pontelli, Gupta, Hermenegildo 1995; Pontelli, Gupta, Tang, Carro, Hermenegildo 1996].

*Conery's Model*

In this method [Conery and Kibler 1983] a data-flow graph is constructed during 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, the forward execution phase ensues. In this phase independent 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 incrementally 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 [Lin and Kumar 1988]. 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 the *B-lists* are created, merged and manipulated at runtime. APEX has been implemented on shared memory multiprocessors for pure logic programs [Lin and Kumar 1988].

*RAP Model*

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

$$(\mathtt{condition} \Rightarrow \mathtt{goal}_1 \;\&\; \mathtt{goal}_2 \;\&\; \ldots \;\&\; \mathtt{goal}_n),$$

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* is a conjunction of constraints of the type: $ground(v_1, \ldots, v_n)$, 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 unconditionally executed in parallel. The groundness and independence conditions are evaluated at runtime. A simple technique which keeps track of groundness and independence properties of variables through tags associated to the heap locations is presented in [DeGroot 1984]. 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 instances of independent 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 [Peterson and Silberschatz 1986]). Experimental evidence has demonstrated that among all the models the RAP model comes closest to realizing the criteria mentioned in the previous section. This model has been formalized and extended by Hermenegildo and Nasr, and has been efficiently implemented using WAM-like instructions [Hermenegildo 1986a; Pontelli, Gupta, Hermenegildo 1995] as the &-Prolog/CIAO system [Hermenegildo and Green 1991], as the &ACE/ACE system [Pontelli, Gupta, Hermenegildo 1995; Pontelli, Gupta, Tang, Carro, Hermenegildo 1996], and as the dependent and-parallel DASWAM system [Shen 1992b; 1992a]. The CGEs are generated at compile time [Muthukumar and Hermenegildo 1989a; 1991; Jacobs and Langen 1989] using the technique of abstract interpretation [Cousot and Cousot 1977; 1992]. CGEs generated through this analysis at compile-time manage to capture a substantial amount of independent and-parallelism [Muthukumar and Hermenegildo 1990].

## 4.4 Experimental Systems

### 4.4.1 *The* &-Prolog *AND-Parallel Prolog System*

&-Prolog is a prototype Prolog implementation, relying on SICStus Prolog, 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, and more complex forms of and-parallelism (i.e., not just independent and-parallelism) can also be expressed. 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 shared memory 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 [Carro et al. 1993]. 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).

**&-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 [Muthukumar and Hermenegildo 1989b]. 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 [Debray et al. 1990]. This information is used in an additional pass aimed at introducing granularity control, implemented using dynamic term size computation techniques [Hermenegildo and Lopez-Garcia 1995]. 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 [Muthukumar and Hermenegildo 1990; Codish et al. 1995; Cabeza and Hermenegildo 1994]. A final pass (an extension of the SICStus compiler) produces code for a specialized WAM engine (called *PWAM* and described below) from an already parallelized &-Prolog program.

The global analysis mentioned above is performed by using the technique of "abstract interpretation" [Cousot and Cousot 1992] 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$^3$" [Hermenegildo, Warren, Debray 1992] 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 [Bruynooghe 1991] and the specialized fixpoint algorithms described in [Muthukumar and Hermenegildo 1989a; Muthukuar et al. 1999; Muthukumar and Hermenegildo 1992]. 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 [Muthukumar and Hermengildo 1991; Hermenegildo, Warren, Debray 1992; Codish et al. 1995].

**&-Prolog Run-Time System:** The &-Prolog run-time system is based on the Parallel WAM (PWAM) model [Hermenegildo and Green 1991], an evolution of RAP-WAM [Hermenegildo 1986b; 1986a; Tick 1991], itself an extension of the Warren Abstract Machine (WAM) [Warren 1983]. 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 (the behavior of some WAM instructions has to be modified to meet the needs of the PWAM—e.g., the instructions associated to the management of choice points) 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*. While the PWAM uses conventional *environment sharing* for sequential goals—i.e., an environment is created for each clause executed, which maintains the data local to the clause—it uses a combination of *goal stacking* and environment sharing for parallel goals: for each parallel goal, a goal descriptor is created and stored in the goal stack, but their associated storage is in shared environments in the stack. The goal descriptor contains a pointer to the environment for the goal, a pointer to the code of the subgoal, and additional control information. 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. A parcall frame is created as soon as a CGE (with a satisfiable condition part) is encountered. The CGE contains, between the other things, a *slot* for each subgoal present in the parallel call; these slots will be used to keep track of the status of the execution of the corresponding parallel subgoal.

*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 [Hermenegildo 1987; Shen and Hermenegildo 1993]. As illustrated in Figure 14, whenever a PWAM selects a parallel subgoal for execution, it creates an *input marker* in its control stack. The marker denotes the beginning of a new subgoal. Similarly, as soon as the execution of a parallel subgoal is completed, an *end marker* is created on the stack. As shown in the figure, the input marker of a subgoal contains a pointer to the end marker of the subgoal on its left; this is needed to allow backtracking to propagate from parallel subgoal to parallel subgoal in the correct (i.e., Prolog) order.

Figure 14 illustrates the different phases in the forward execution of a CGE. As soon as the CGE is encountered, a parcall frame is created by Worker 1. Since the parallel call contains three subgoals, Worker 1 will keep one for local execution (`p1`) while the others will be made available to the other workers. This is accomplished by creating two new entries (one for `p2` and one for `p3`) in the goal stack. Idle workers will detect the presence of new work and will extract subgoals from remote goal stacks. In the example, Worker 2

takes p2 while Worker 3 takes p3. Each idle worker will start the new execution by creating an input marker to denote the beginning of a new subgoal. Upon completion of each subgoal, the workers will create end markers. The last worker completing a subgoal (in the figure we have identified Worker 2 as the last one to complete), will create the appropriate links between markers and proceed with the (sequential) execution of the continuation (p4).

In practice, the stack is divided into a separate *control* stack (for choice point and markers) and a separate *local* stack (for environments, including parcall frames), for reasons of locality and locking. A goal stack is maintained by each worker and contains the subgoals which are available for remote execution.

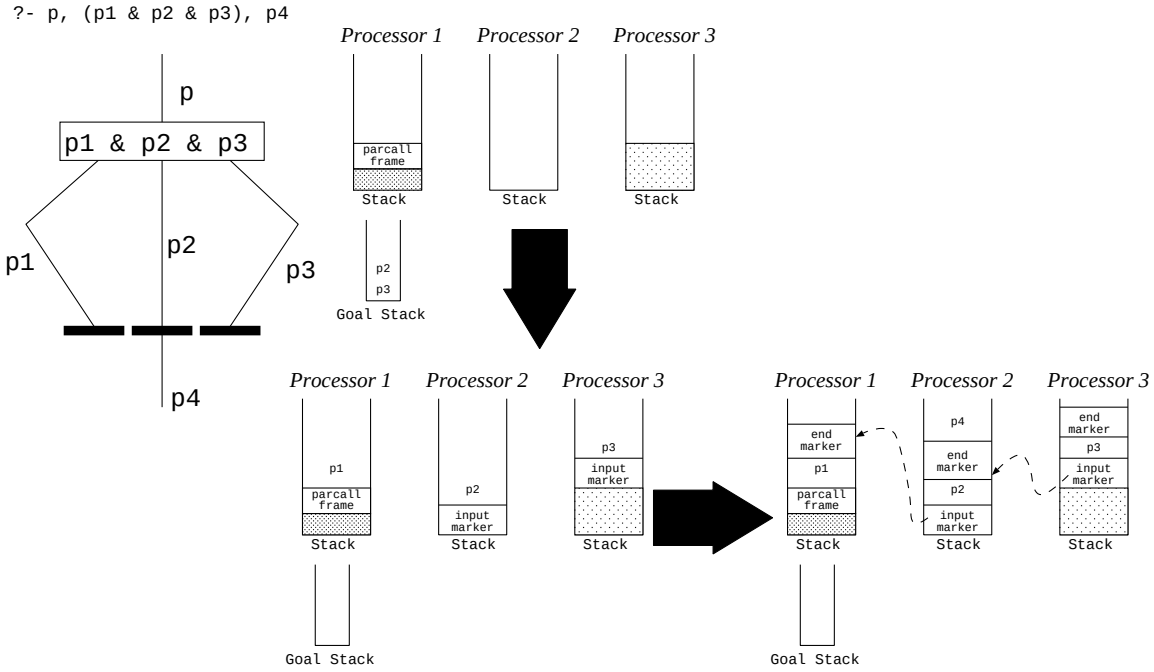



Fig. 14. Computation's Organization in PWAM

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 [Hermenegildo 1987; Hermenegildo and Green 1991; Shen and Hermenegildo 1993].

**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 [Hermenegildo, Warren, Debray 1992]. MA$^3$ is a first generation and-parallel analyzer, based on abstract compilation and efficient implementation techniques (e.g., extension tables), which has been used to extract term groundness and term independence information at the different program points. These information are in turn used to generate conditional graph expressions, and eventually simplify their condition part. The number of CGEs generated, the 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 incurred in performing global analysis is fairly reasonable while the figures obtained close to what is possible manually.

Early experimental results regarding the performance of the second generation parallelizing compiler in terms of attainable program speedups were reported in [Codish et al. 1995] both without global analysis and also with sharing and sharing+freeness analysis running in the PLAI framework [Jacobs and Langen 1989; Muthukumar and Hermenegildo 1989a; Muthukuar et al. 1999; Muthukumar and Hermengildo 1991]. Speedups were obtained using the IDRA system [Fernandez, Carro, Hermenegildo 1996], which collects traces from sequential executions and uses them to simulate an ideal parallel execution of the same program..[9] 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 [Muthukumar and Hermenegildo 1990] (UDG/MEL/CDG), can be found in [Bueno, Garcia de la Banda, Hermenegildo 1999; Garcia de la Banda, Hermenegildo, Marriott 1996]. Although work still remains to be done, specially in the area of detecting non-strict independence[10], results compared encouragingly well with those obtained from studies of theoretical ideal speedups for optimal parallelizations, such as those given in [Shen and Hermenegildo 1991].

Finally, experimental results regarding the run-time system can be found in [Hermenegildo and Green 1991]. Actual speedups obtained on the Sequent Balance and Symmetry systems were reported for the parallelized programs for different numbers of workers. Various benchmarks have been tested, ranging from simple problems (e.g., matrix multiplication) to very large applications (e.g., parts of the abstract interpreter). Particularly good results have been achieved on divide and conquer programs with sufficiently large granularity. 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 workers. 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.[11]

#### 4.4.2 *The* &ACE *System*

The &ACE [Pontelli, Gupta, Hermenegildo 1995; Pontelli, Gupta, Tang, Carro, Hermenegildo 1996] system is an independent and-parallel Prolog system developed at New Mexico State University as part of the ACE project. &ACE has been designed as a next-generation independent and-parallel system and is an evolution of the PWAM design (used in &-Prolog). Like &-Prolog, &ACE relies on the execution of Prolog programs annotated with Conditional Graph Expressions.

The forward execution phase is articulated in the following steps. As soon as a parallel conjunction is reached, a *parcall frame* is allocated in a separate stack—differently from &-Prolog, which allocates parcall frames on the environment stack; this allows for easier memory management[12] (e.g., does not prevent the use of last-call optimization) and for application of various determinacy-driven optimizations [Pontelli, Gupta, Tang 1995] and alternative scheduling mechanisms [Pontelli, Gupta, Tang, Carro, Hermenegildo 1996]. Slots describing the parallel subgoals are allocated in the Heap and organized in a (dynamic) linked list, thus allowing their dynamic manipulation at run-time. Subgoals in the goal stack (as in the PWAM model) are replaced by a simple frame placed in the goal stack and pointing to the goal frame—this has been demonstrated [Pontelli, Gupta, Hermenegildo 1995] to be more effective and flexible than actual goal stacking. These data structures are described in Figure 15.

The use of markers to identify segments of the computation has been removed in &ACE and replaced by a novel technique called *stack linearization* which allows to link choice points lying in different stacks in the correct logical order; this allows to limit to the minimum the changes to the backtracking algorithm, thus making backtracking over and-parallel goals very efficient. The only marker needed is the one which indicates the beginning of the continuation of the parallel call. Novel uses of the trail stack (by trailing status flags in

[9] 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.
[10] The notion of non-strict independence is described in Section 5.3.3.
[11] Performance of such systems ranges from about the same as SICStus to to about twice the speed, depending on the program.
[12] &ACE is built on top of the SICStus WAM, which frequently performs on-the-fly computation of the environment registers. The presence of parcall frames on the same stack creates enormous complications to the correct management of such registers.

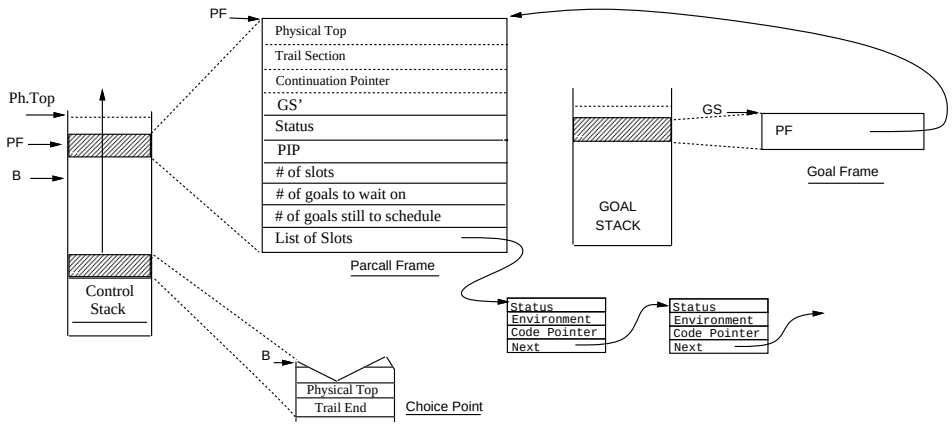



Fig. 15. Parcall Frames and Goals in &ACE

the subgoals slots) allows to integrate outside backtracking without any explicit change in the backtracking procedure.

Backward execution represents another novelty in &ACE. Although it relies on the same general backtracking scheme developed in PWAM (the point backtracking scheme described in Section 4.1.3), it introduces the additional concept of *backtracking independence* which allows to take full advantage of the semi-intelligent backtracking phase during inside backtracking. Given a subgoal of the form:

$$? - b, (g_1 \& g_2), a$$

backtracking independence requires that the bindings to the variables present in $g_1, g_2$ are posted either before the beginning of the parallel call or at its end. This allows to kill subgoals and backtrack without having to worry about untrailing external variables. Backtracking independence is realized through compile-time analysis and through the use of special run-time representation of global variables in parallel calls [Pontelli and Gupta 1998].

&ACE has been developed my modifying the SICStus WAM and currently runs on Sequent Symmetry and Sun Sparc multiprocessors (Solaris). The use of the new memory management scheme, combined with a plethora of optimizations [Gupta and Pontelli 1997; Pontelli, Gupta, Tang 1995; Pontelli, Gupta, Tang, Carro, Hermenegildo 1996], allows &ACE to be very effective in exploiting parallelism, even from rather fine grained applications [Pontelli, Gupta, Hermenegildo 1995]. The performance of the system is on average within 5% from the performance of the original sequential engine, thus denoting a very limited amount of overhead. The presence of an effective management of backtracking has also lead to various cases of super-linear speedups [Pontelli and Gupta 1998].

## 5. DEPENDENT AND-PARALLELISM

*Dependent And-Parallelism (DAP)* generalizes independent and-parallelism by allowing the concurrent execution of subgoals accessing intersecting sets of variables. The "classical" example of DAP is represented by a goal of the form `?- p(X) & q(X)`[13] where the two subgoals may potentially compete in the creation of a binding for the unbound variable `X`.

Unrestricted parallel execution of the above query (in Prolog) is likely to produce non-deterministic behavior: the outcome will depend on the order in which the two subgoals access $X$. Thus, the first aim of any system exploiting dependent and-parallelism is to ensure that the operational behavior of dependent and-parallel execution is consistent with the intended semantics—(sequential) Prolog semantics in this case. This amounts to

- making sure that all the parallel subgoals agree on the values given to the shared variables;

[13] As for independent and-parallelism, we will use "&" to denote parallel conjunction, while "," will be kept to indicate sequential conjunctions.

- guaranteeing that the order in which the bindings are performed does not lead to any violation of the observable behavior of the program (Prolog semantics).

It is possible to show that the problem of determining the correct moment in time when a binding can be performed without violating Prolog semantics is in general undecidable. The different models designed to support DAP differ in the approach taken to solve this problem, i.e., they differ in how they conservatively approximate such undecidable property.

The question then arises whether dependent and-parallelism is fruitful at all. Because typically in a query such as above, $p$ will produce a binding for $X$ while $q$ will process (or consume) it. If this order between production of binding and its consumption is to be preserved, $q$ will be suspended until execution of $p$ is over. However, this is not always the case, and execution of $p$ and $q$ can be overlapped in certain situations:

(1) $q$ may first perform significant amount of computation before it needs the binding of $X$; this computation can be overlapped with computation of $p$, because it doesn't depend on $X$;

(2) $p$ may first partially instantiate $X$. In such a case $q$ can start working with the partially instantiated value, while $p$ is busy computing the rest of the binding for $X$.

In the rest of this section we will use the following terminology. Unbound variables which are accessible by different parallel subgoals are called *shared* (or *dependent*) variables. The SLD computation tree generated by Prolog enforces an ordering between the subgoals which appear in the tree. We will say that a subgoal $A$ is on the left of $B$ if the subgoal $A$ appears on the left of $B$ in the SLD tree generated by Prolog.

The scope for exploitation of dependent and-parallelism strongly depends on the semantics of the logic language considered. E.g., DAP execution of pure Prolog—where no order-sensitive predicates appear—makes implementation simple and creates the potential for high speedups. Similarly, the semantics of languages like Parlog and other committed-choice languages is designed to provide a relatively convenient management of specialized forms of DAP (stream parallelism), simplifying the detection of dependencies. In the context of this paper we will focus on the DAP execution of Prolog programs—thus, the ultimate goal of the DAP execution models, as far as this paper is concerned, is to speedup execution of the programs through parallelism reproducing the same observable behavior as in a sequential Prolog execution.

### 5.1 Issues

Supporting DAP requires tackling a number of issues. These include:

(1) *detection of parallelism:* determination of which subgoals should be considered for DAP execution.

(2) *management of DAP goals:* activation and management of parallel subgoals;

(3) *management of shared variables:* validation and control of shared variables to guarantee Prolog semantics;

(4) *backtracking:* management of non-determinism in presence of DAP executions.

In the rest of this section we will deal with all these issues except for issue 2: management of subgoals does not present any new challenge w.r.t. the management of parallel subgoals in the context of independent and-parallelism.

### 5.2 Detection of Parallelism

Annotating a program for fruitful DAP execution resembles in some aspects automatic parallelization for IAP [Cabeza and Hermenegildo 1994; Muthukuar et al. 1999]. This should come as no surprise: DAP is nothing else than a finer grain instance of the general principle of independence, applied to the level of variable bindings. Relatively little work is present in the literature for detecting and analyzing fruitful DAP. The first work on this specific problem is that by Giacobazzi [Giacobazzi and Ricci 1990], which attempts a bottom-up abstract interpretation to identify pipelined computations. Some similarities are also shared with the various studies on *partitioning techniques* for declarative concurrent languages [Traub 1989], that aim to identify partitioning of the program components into sequential threads, and the work on management of parallel tasks in committed-choice languages [Ueda and Morita 1993].

Automatic and semi-automatic detection of potential valid sources of DAP in logic programs has been directly tackled in [Pontelli, Gupta, Pulvirenti, Ferro 1997]. This proposal generates code annotations which are

extensions of the CGE format (similar to those originally introduced by Shen [Shen 1992a])—they additionally identify and make explicit the variables that are shared between the goals in the parallel conjunction. Given the goals $\ldots G_1, \ldots, G_n \ldots$, in which the subgoals $G_1, \ldots, G_n$ are to be executed in DAP, the general structure of an extended CGE is the following:

$$\ldots, \$mark([X_1, \ldots, X_m]),$$

$$(\langle Cond \rangle \Longrightarrow \$and\_goal(\theta_1, G_1^{\theta_1}) \ \& \ \ldots \ \& \ \$and\_goal(\theta_n, G_n^{\theta_n})), \ldots$$

where:

- $X_1, \ldots, X_m$ are the *shared* variables for subgoals $G_1, \ldots, G_n$, i.e., all those variables for which different subgoals may attempt conflicting bindings;
- if $X_1^j, \ldots, X_{k_j}^j \subseteq \{X_1, \ldots, X_m\}$ are the shared variables present in the subgoal $G_j$, then $\theta_j$ is a renaming substitution for the variables $X_i^j (1 \leq i \leq k_j)$—i.e., a substitution which replace each $X_i^j$ with a brand new variable. This allows each subgoal in the conjunction to have a fresh and independent access to each shared variable.
  In this framework the mapping is described as a sequence of pairs $[X_i^j, X_i^{new(j)}]$, where $X_i^{new(j)}$ is the new variable introduced to replace variable $X_i^j$.
- *Cond* is a condition, that will be evaluated at runtime (e.g., for checking groundness, independence, comparing dynamically computed grain-sizes to thresholds).

A DAP annotated version of the recursive clause in the program for *naive reverse* will look as follows:

```
nrev([X|Xs],  Y) :- $mark([Z]),
         ( $and_goal([[Z,Z1]],nrev(Xs, Z1)) &
           $and_goal([[Z,Z2]],append(Z2, [X], Y)) ).
```

The `$mark/1` is a simple directive to the compiler to identify shared variables. The shared variables are given different names in each of the parallel goals. The shared variable `Z` is accessed through the variable `Z1` in `nrev` and through the variable `Z2` in the `append` subgoal. The use of new names for the shared variables allows the creation of separate access paths to the shared variables, which in turn facilitates more advanced run-time schemes to guarantee the correct semantics (such as the Filtered Binding Model presented later in this Section).

The process of annotating a program for exploitation of dependent and-parallelism described in [Pontelli, Gupta, Pulvirenti, Ferro 1997] operates through successive refinements:

(1) identification of clauses having a structure compatible with the exploitation of DAP—i.e., they contain at least one group of consecutive non-builtin predicates. Each maximal group of contiguous and non-builtin goals is called a *partition*.

(2) use of sharing and freeness [Cabeza and Hermenegildo 1994; Muthukuar et al. 1999] information (determined via abstract interpretation) to identify the set of shared variables for each partition;

(3) refinement of the partition to improve DAP behavior through the following transformations:
   - collapsing of consecutive subgoals;
   - splitting of partitions in subpartitions;
   - removal of subgoals lying at the beginning or end of a partition.

   The transformations are driven by the following principles:
   - parallel subgoals should display a sufficiently large grain size to overcome the parallelization overhead;
   - dependent subgoals within a partition should demonstrate a good degree of overlapping in their executions.

   The first aspect can be dealt with through the use of cost analysis [Debray et al. 1997b; Tick and Zhong 1993], while the second one is dealt with in [Pontelli, Gupta, Pulvirenti, Ferro 1997] through the use of *instantiation analysis*, based on the estimation of the size of the computation which precedes the binding of shared variables.

Further improvements have been devised in [Pontelli, Gupta, Pulvirenti, Ferro 1997] through the use of sharing and freeness to detect at compile-time subgoals that will definitely bind dependent variables—i.e., automatic detection of definite producers.

## 5.3 Management of Variables

### 5.3.1 *Introduction*

The management of shared variables in a dependent and-parallel execution requires solving two key issues. The first issue is related to the need of guaranteeing mutual exclusion during the creation of a binding for a shared variable. The second, and more important, issue is concerned with the process of *binding validation*, i.e., guaranteeing that the outcome of the computation respects sequential observable Prolog semantics. These two issues are discussed in the next two subsections.

### 5.3.2 *Mutual Exclusion*

The majority of the schemes proposed to handle DAP rely on a single representation of each shared variable—i.e., all the threads of computation access the same memory area which represents the shared variable. Considering that we are working in a Prolog-like model, at any time at most one of these threads will be allowed to actually bind the variable. Nevertheless, the construction of a binding for a variable is *not* an atomic operation—unless the value assigned to the variable is atomic. Furthermore, in the usual WAM, the assignment of a value can be realized through the use of `get` instructions, which are characterized by the fact that they proceed *top-down* in the construction of the term. This means that first the unbound variable is assigned a template of the term to be constructed—e.g., through a `get_structure` instruction—and successively the subterms of the binding are constructed. This makes the binding of the variable a non-atomic operation. For example, if the two subgoals executing in parallel are `p(X)` and `q(X)`, which are respectively defined by the following clauses:

```
p(X) :- X = f(b,c), .....
q(X) :- X = f(Y,Z), (var(Y) -> ... ; ...).
```

The WAM code for the clause for `p` will contain a sequence of instructions of the type

```
get_structure f, A1
unify_constant b
unify_constant c
...
```

An arbitrary interleaving between the computations (at the level of WAM instructions) can lead `q` to access the binding for `X` immediately after the `get_structure` but before the successive `unify_constant`—leading `q` to wrongfully succeed in the `var(Y)` test. Clearly, as long as we allow consumers to have continuous access to the bindings produced by the producer, we need to introduce some mechanisms capable of guaranteeing atomicity of *any* binding to shared variables.

The problem has been discussed in various works. In the context of the JAM implementation of Parlog [Crammond 1992] the idea is to have the compiler generate a different order of instructions for what concerns the construction of complex terms: the pointer to a structure is not written until the whole structure has been completely constructed. This approach requires a radical change in the compiler. Furthermore, the use of this approach requires a special action at the end of the unification, in order to make the structure "public"—and this overhead will be encountered in general for *every* structure built, independently from whether this will be assigned to a dependent variable or not.

Another solution has been proposed in *Andorra-I* [Santos Costa, Warren, Yang 1996]; in this system, terms which need to be matched with a compound term (i.e., using the `get_structure` instruction in the WAM) are locked—i.e., a mutual exclusion mechanism is associated to it—and a special instruction (`last`) is added by the compiler at the end of the term construction to release the lock—i.e., terminate the critical section.

Another approach, adopted in the DASWAM system [Shen 1992b], consists of modifying the `unify` and *get* instructions in such a way that they always overwrite the successive location on the heap with a special value. Every access to term will inspect such successive location to verify whether the binding has been completed or not. No explicit locks or other mutual exclusion mechanisms are required. On the other hand:

- while reading the binding for a dependent variable, every location accessed needs to be checked for validity;
- an additional operation (pushing an invalid status on the successive free location) is performed during each operation involved in the construction of a dependent binding.
- a check needs to be performed during each operation which constructs a term, in order to understand whether the term has been assigned to a dependent variable or not—or, alternatively, the operation of pushing the invalid status is performed indiscriminately during the construction of *any* term, even if it will not be assigned to a dependent variable.

A novel solution [Pontelli 1997], which does not suffer from most of the drawbacks previously described, is to have the compiler generate a different sequence of instructions to face this kind of situations. The *get_structure* and *get_list* instructions are modified, by adding a third argument:

`get_structure` ⟨`functor`⟩ ⟨`register`⟩ ⟨`jump label`⟩

where the ⟨`jump label`⟩ is simply an address in the program code. Whenever the dereferencing of the ⟨`register`⟩ leads to an unbound shared variable, instead of entering write mode (as in standard WAM behavior), the abstract machine performs a jump to the indicated address (⟨`jump label`⟩). The address contains a sequence of instructions which performs the construction of the binding in a *bottom-up* fashion—which allows for the correct atomic execution.

#### 5.3.3 *Binding Validation*

A large number of schemes have been proposed to handle bindings to dependent variables such that Prolog semantics is respected. We can classify the different approaches according to two orthogonal criteria [Pontelli and Gupta 1997b; 1997a]:

- *Validation time:* the existing proposals either
  - remove inconsistencies on binding shared variables only once a conflict appears and threatens Prolog semantics (*curative schemes*)
  - prevent inconsistencies by appropriately delaying and ordering shared variable bindings (*preventive schemes*)
- *Validation resolution:* the existing proposals either
  - perform the validation activity at the level of the parallel subgoals (*goal-level validation*)
  - perform the validation activity at the level of the individual shared variable (*binding-level validation*)

**Curative Approaches:** Curative approaches rely on validation of the bindings to shared variables *after* they are performed.

Performed at the goal level (see Figure 16), implies that each and-parallel subgoal develops its computation on local copies of the environments, introducing an additional "merging" step at the end of the parallel call—to verify consistency of the values produced by the different computations for the shared variables. This approach, adopted mainly in some of the older process-based models, like Epilog [Wise 1986] and ROPM [Ramkumar and Kalé 1992], has the advantage of being extremely simple, but it suffers some serious drawbacks:

(1) it produces highly speculative computations (due to the lack of communication between parallel subgoals);

(2) it may produce parallel computations that terminates in a time longer than the corresponding sequential ones;

(3) it makes extremely difficult to enforce Prolog semantics.

Performed at the binding level (see Figure 17), validation does not preempt bindings from taking place (i.e., any goal can bind a shared variable), but special rollback actions are needed whenever a violation of program semantics is detected. The two most significant proposals where this strategy is adopted are those made by Tebra [Tebra 1987] and by Drakos [Drakos 1989]. They can be both identified as instances of a general scheme, named *Optimistic Parallelism*. In optimistic parallelism, validation of bindings is performed not at binding time (i.e., the time when the shared variable is bound to a value), but only when a conflict occurs (i.e., when a producer attempts to bind a shared variable that had already been bound earlier by a consumer goal.) In case of a conflict, the lower priority binding (made by the consumer), has to be undone, and the consumer goal rolled back to the point where it first accessed the shared variable. These model have various

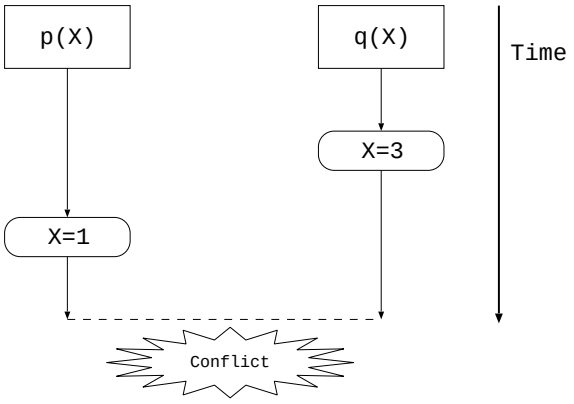



Fig. 16. Goal Level Curative Approach

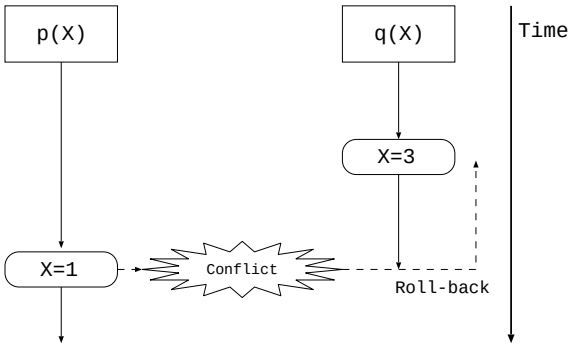



Fig. 17. Binding Level Curative Approach

drawbacks, ranging from their highly speculative nature to the limitations of some of the mechanisms adopted (e.g., labeling schemes to record binding priorities), and to the high costs of rolling back computations.

**Preventive Approaches:** Preventive approaches are characterized by the fact that bindings to shared variables are prevented unless they are guaranteed to not threaten Prolog semantics.

Performed at the goal level, preventive schemes delay the execution of the whole subgoal until its execution will not affect Prolog semantics. Various models have embraced this solution:

- *Non-Strict Independent And-Parallelism (NSI):* [Cabeza and Hermenegildo 1994] allows to extend the scope of independent and-parallelism to subgoals that have variables in common, as long as at most one subgoal can bind each shared variable, and the binding will not affect the computation of the remaining subgoals.
- The *Basic Andorra Model* [Haridi 1990; Warren 1988; Santos Costa, Warren, Yang 1991a], Parallel NU-Prolog [Naish 1988], Pandora [Bahgat 1993], and P-Prolog [Yang 1987] are all characterized by the fact that parallel execution is allowed between dependent subgoals only if there is guarantee that there exists at most one single matching clause. 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 goal 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. A similar approach has been adopted in Pandora [Bahgat 1993], which represents a combination of the Basic Andorra Model and the Parlog committed-choice approach to execution [Clark and Gregory 1986]; Pandora introduces 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).

Performed at the binding level, preventive schemes allow a greater degree of parallelism to be exploited. The large majority of such schemes rely on enforcing a stronger notion of semantics (*Strong Prolog Semantics*):

bindings to shared variables are performed *in the same order* as in a sequential Prolog execution. The most relevant schemes are:

- *Committed-Choice languages:* we will only deal briefly with the notion of committed-choice languages in this paper, since they implement a semantics which is radically different from Prolog. Committed-choice languages [Tick 1995] disallow (to a large extent) non-determinism by requiring the computation to commit to the clause selected for resolution. Committed-choice languages support dependent and-parallel execution and handle shared variables via a preventive scheme based on the notion of producer and consumers. Producer and consumers are either explicitly identified at the source level (e.g., via mode declarations) or implicitly through strict rules on binding of variables that are external to a clause.
- *DDAS-based schemes:* these schemes offer a direct implementation of strong Prolog semantics through the notion of *producer and consumer* of shared variables. At each point of the execution only one subgoal is allowed to bind each shared variable (*producer*), and this corresponds to the leftmost active subgoal which has access to such variable. All remaining subgoals are restricted to read-only accesses to the shared variable (*consumers*); each attempt by a consumer of binding an unbound shared variable will lead to the suspension of the subgoal. Each suspended consumer will be resumed as soon as the shared variable is instantiated. Consumers may also become producers if they become the leftmost active computations. This can happen if the designated producer terminates without binding the shared variable.
  Detecting producer and consumer status is a complex task. Different techniques have been described in the literature to handle this process. Two major implementation models have been proposed to handle producer/consumer detection, *DASWAM* and the *Filtered-Binding Model*, which are described at the end of this section. An alternative implementation model based on *Attributed Variables* [Le Huitouze 1990] has been proposed in [Hermenegildo, Cabeza, Carro 1995]: each dependent variable $X$ is split into multiple instances, one for each subgoal belonging to the parallel call. Explicit procedures are introduced to handle unification and transfer bindings to the different instances of each shared variable. The idea behind this model is attractive, and it shares some commonalities with the Filtered Binding model presented in Section 5.5.3.

**Classification:** As done for or-parallelism in Section 3.4, it is possible to propose a classification of the different models for DAP based on the complexity of the basic operations. The basic operations required to handle forward execution in DAP are:

- *task creation:* creation of a parallel conjunction
- *task switching:* scheduling and execution of a new subgoal
- *variable access/binding:* access and/or binding of a variable

It is possible to prove, by properly abstracting these operations as operations on dynamic tree structures, that at least one of them requires a time complexity which is strictly worse than $\Omega(1)$ [Pontelli, Ranjan, Gupta 1997; Ranjan,Pontelli,Gupta,Longpre 2000]. Interestingly enough, this result ceases to hold if we disallow aliasing of shared variables during the parallel computation—intuitively, aliasing of shared unbound variables may create long chains of shared variables bound to each other, and the chain has to be maintained (and traversed) to determine exactly whether a binding for the variable is allowed or not. A similar restriction is actually present in the DASWAM system, to simplify the implementation of the variables management scheme. Nevertheless, the Filtered-binding Model is the only model proposed that succeeds in achieving constant time complexity in all the key operations in absence of shared variables aliasing.

The classification of the different models according to the complexity of the three key operations is illustrated in Figure 18. Unrestricted DAP means DAP with possible aliasing of unbound shared variables.

### 5.4 Backtracking

Maintaining Prolog semantics during parallel execution also means supporting non-deterministic computations, i.e., computations that can potentially produce multiple solutions. In many approaches DAP has been restricted to only those cases where $p$ and $q$ are deterministic [Bevemyr et al. 1993; Shapiro 1987; Santos Costa, Warren, Yang 1991a]. This is largely due to the complexity of dealing with distributed backtracking.

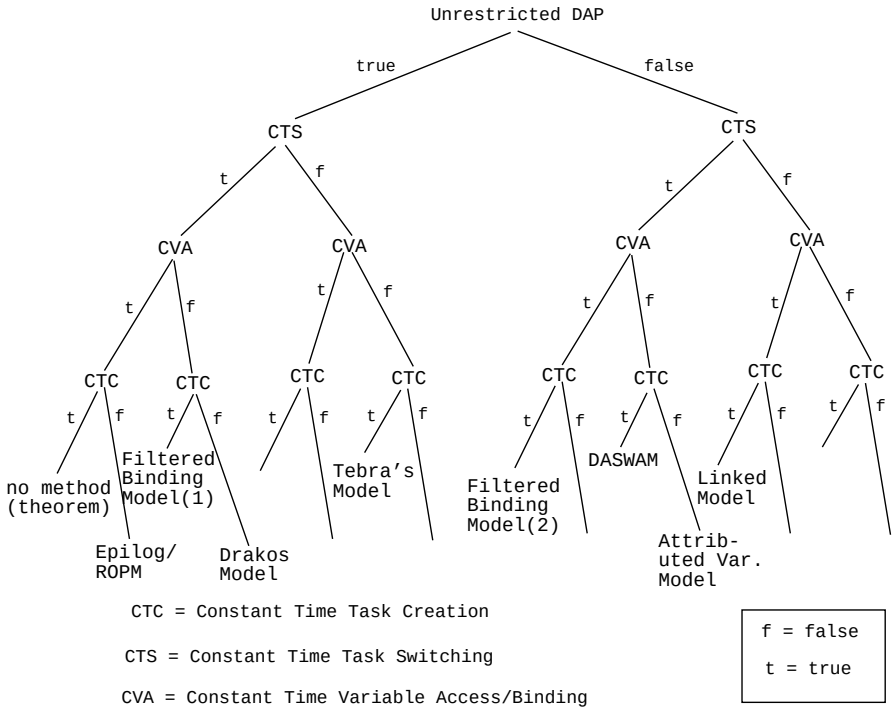



Fig. 18. Classification of DAP models

Nevertheless, it has been shown [Shen 1992b] that imposing this kind of restriction on DAP execution may severely limit the amount of parallelism exploited. The goal is to exploit DAP even in non-deterministic goals.

Backtracking in the context of DAP is more complex than in the case of independent and-parallelism. While outside backtracking remains unchanged, inside backtracking—i.e., backtracking within subgoals which are part of a parallel call—loses its "independent" nature, which guaranteed the semi-intelligent backtracking described earlier. Two major issues emerge. First of all, failure of a subgoal within a parallel conjunction does not lead to the failure of the whole conjunction, but requires killing the subgoals on the right and backtracking to be propagated to the subgoal immediately to the left—an asynchronous activity, since the subgoal on the left may be still running;

In addition, backtracking *within* a parallel subgoal may also affect the execution of other parallel subgoals. In a parallel conjuction like `p(X) & q(X)`, backtracking within `p(X)` which leads to a modification of the value of `X` will require rolling back the execution of `q(X)` as well, since `q(X)` may have consumed the value of `X` which has just been untrailed.

Implementations of this scheme have been proposed in [Shen 1992b; 1992a; Pontelli and Gupta 1997a]; optimizations of this scheme have also been described in [Shen 1994].

## 5.5 Experimental Systems

In this section we present some representative systems which support dependent and-parallelism. Some other systems which use dependent and-parallelism in conjunction with other forms of parallelism (e.g., ROPM) will be described in Section 6. In this section we do not discuss Committed-choice language—their sequential and parallel execution model have been described in detail in other surveys (e.g., [Tick 1995]).

### 5.5.1 *Andorra-I*

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 [Warren 1984; 1987a]. 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 [Yang et al. 1993] and is a descendent of the earlier interpreted version [Santos Costa, Warren, Yang 1991a].

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. As mentioned earlier, this is because execution in the Basic Andorra Model is divided into two phases—determinate and non-determinate—execution of non-determinate phase is begun only after all "forced choices"—i.e., choices for which only one alternative is left—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 [Santos Costa, Warren, Yang 1996; 1991b].

The Andorra-I system supports full Prolog, in that execution can be performed in such a way that sequential Prolog semantics is preserved [Santos Costa, Warren, Yang 1996; 1991b]. 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.[14]

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 [Santos Costa, Warren, Yang 1991c; Yang et al. 1993] combines the implementation techniques used in implementing Parlog, namely the JAM system [Crammond 1992], and the Aurora system [Lusk et al. 1990]. 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 was not a problem in JAM. The Andorra-I system was the first one to employ the notion of teams of workers, where available workers 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 workers within a team, i.e., workers 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 (*reconfigurer*) [Dutra 1994; 1996]. The notion of teams of workers was also adopted by the ACE [Gupta, Pontelli, Hermenegildo, Santos Costa 1994] and the PBA [Gupta and Santos Costa 1992c; Gupta, Santos Costa, Pontelli 1994; Gupta, Hermenegildo, and Santos Costa 1993] 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 [Palmer and Naish 1991].

### 5.5.2 *DASWAM*

DASWAM [Shen 1992a; 1992b] is an implementation model for the DDAS execution scheme described in Section 5.3.3. DASWAM has been designed as an extension of the PWAM model used for independent and-parallelism. Memory management is analogous to PWAM—and relies on the use of parcall frames to represent parallel conjunctions, and on the use of markers to delimit segments of stacks associated with the execution of a given subgoal.

Shared variables are represented as a new type of tagged cell and each shared variable is uniquely represented—thus all workers access the same representation of the shared variable. Producer and consumer status is deter-

[14] In spite of this, there are cases where Andorra-I and Prolog leads to different behavior; in particular, there are non-terminating Prolog programs which will terminate in Andorra-I and vice versa.

mined via a search operation, performed at the time of variable binding. Each dependent variable identifies the parcall frame which introduced the variable (*home parcall*); a traversal of the chain of nested parallel calls is needed to determine whether the binding attempt lies in the leftmost active subgoal. The knowledge of the subgoal is also needed to create the necessary suspension record—where information regardinga suspended consumer is recorded. The process is illustrated in Figure 19. Each dependent cell maintains pointers to the parcall frame which introduced that dependent variable. Additionally, the parcall frames are linked to each other to recreate the nesting relation of the parallel conjunctions. This arrangement implies a complexity which is linear in the size of the computation tree in order to determine producer/consumer status and subgoals on which to suspend [Shen 1992b; 1992a].

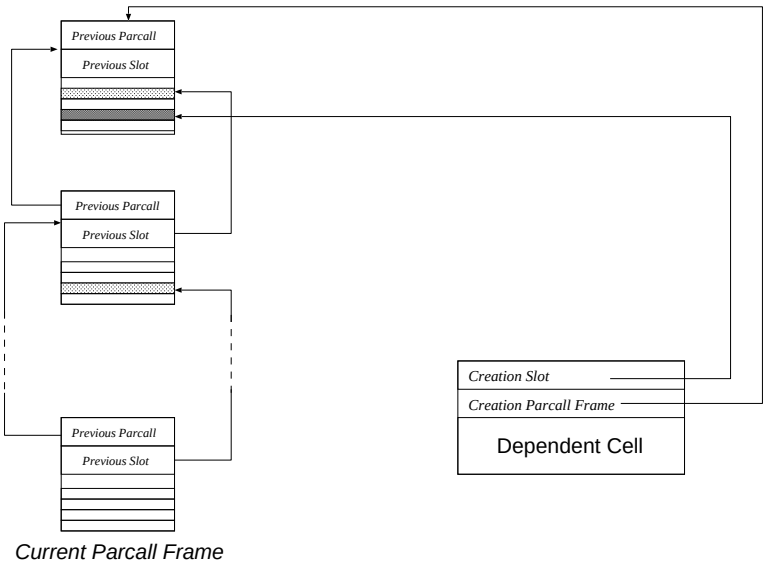



Fig. 19. DASWAM Implementation

### 5.5.3 *ACE*

The Filtered Binding Model is an instance of the class of models which use *binding-level* validation and are *preventive*. The specific approach assumes a program statically annotated to identify the promising sources of parallelism. Each subgoal maintains an independent access path to the shared variable. The idea of the Filtered Binding model is to directly encode in the access path itself the information (the *filter* or *view*) that allows a subgoal to discriminate between producer and consumer accesses. The different access paths are created via specialized WAM instructions, which are introduced via the `$mark` predicate introduced by the parallelizing compiler (see Section 5.2).

Figure 20 presents an intuitive schema of this idea. Each subgoal has a local path to access the shared object (in this case a heap location allocated to hold the value of the shared variable) and the path contains a filter. In the figure the filter is linked to information stored in the subgoal descriptor—this common information will be used to verify when the subgoal is a viable producer (i.e., it is the leftmost active subgoal in the parallel call).

Every access to a shared variable by a subgoal will go through the filter corresponding to that subgoal, which will allow it to determine the "type" of the access (producer or consumer).

By properly organizing the unification process, as long as there is guarantee that no aliasing between shared variables occurs (unless they are both producer accesses), it can be proved that at any time a variable access will require traversal of at most one filter—which means constant-time validation of any access. The setup of a parallel call and the detection of the continuation also do not require any non constant-time operation (the cost is always bounded by the number of dependent variables detected by the compiler in that parallel

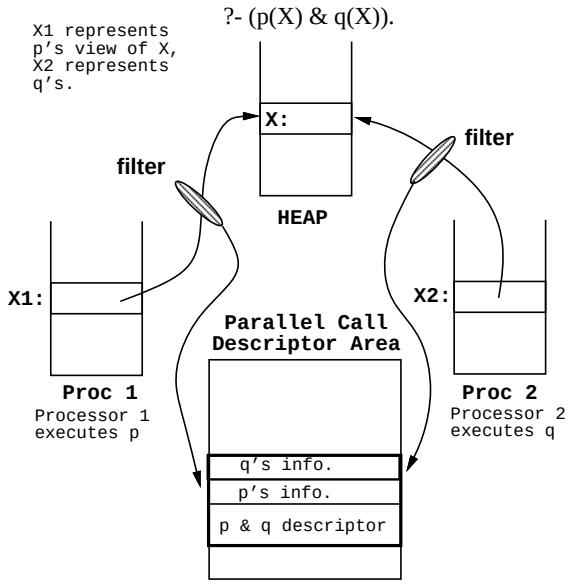



Fig. 20. The Filtered Binding Model

call[15]). An additional step is required when a subgoal terminates: if it is a producer goal, then on termination it should transfer the producer status to the next active subgoal in the parallel call by changing its filter. This is also a constant-time operation, as the next goal to the right can be found by looking at the descriptor of the parallel call.

Thus, the Filtered Binding model is a model that exploits restricted DAP and *performs all operations in constant-time*. The restriction is that unbound shared variables are not allowed to be bound to each other (unless the goal doing the aliasing is a producer for both). If this restriction is relaxed then a non-constant overhead will be produced in the variable access operation—in such a case a non-constant time overhead is unavoidable. The current implementation, realized in the ACE system [Gupta, Pontelli, Hermenegildo, Santos Costa 1994; Pontelli, Gupta, Hermenegildo 1995], represents filters as a word in the subgoal descriptor, and paths as a pair of words, one pointing to the actual variable and one pointing to the filter. Local paths related to shared variables introduced in the same parallel call share the same filter. Consumer accesses suspend in presence of unbound variables. Variable suspensions have been implemented using the traditional *suspension lists* [Crammond 1992].

The implementation of the Filtered Binding Model in the ACE system [Pontelli and Gupta 1997a] supports both busy-waiting and goal suspension (e.g. release of suspended computation). The two methods are alternated during execution depending on the granularity of the computation and on the amount of time the goal has been suspended.

## 6. COMBINING OR-PARALLELISM AND AND-PARALLELISM

### 6.1 Issues

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. A number of prototypes have been implemented and successfully applied to the development and parallelization of very large real-life applications (see also Section 10). Public domain parallel logic programming systems are available (e.g., CIAO [Hermenegildo 1994], which includes &-Prolog, YapOr [Santos Costa, Damas, Reis, Azevedo 1999], KLIC [Chikayama et al. 1994]). For

[15]We are also working under the assumption that the compiler marks goals for DAP execution conservatively, i.e., during execution if a shared variable `X` is bound to a structure containing an unbound variable `Y` before the parallel conjunction corresponding to `X` is reached then both `X` and `Y` are marked as shared. Otherwise, for correctness, the structure `X` is bound to will have to be traversed to find all unbound variables occurring in it and mark them as shared.

some time, a number of commercial strength parallel Prolog systems have also appeared on the market, including SICStus Prolog, which includes the or-parallel MUSE system, and ECLiPSe, which includes an or-parallel version of ElipSys. In spite of the fact that these commercial strength Prolog systems have progressively dropped their support for parallelism (this is mostly due to commercial reasons—the high cost of maintaining the parallel execution mechanisms), these systems demonstrate that we possess the technology for developing effective and efficient Prolog systems exploiting a single form of parallelism.

Although, very general models for parallel execution of logic programs have been proposed, e.g., the *Extended Andorra Model (EAM)* (described later in this section), 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 source of parallelism in logic programs[16]. The implementation of the Basic Andorra Model [Haridi 1990; Warren 1988], namely, Andorra-I [Santos Costa, Warren, Yang 1991c] can be viewed in that way since it combines (determinate) dependent and-parallelism, implemented using techniques from JAM [Crammond 1992], with or-parallelism, implemented using Binding Arrays technique [Lusk et al. 1990; Warren 1987a]. Likewise, the PEPSys model [Westphal, Robert, Chassin, Syre 1987; Baron et al. 1988], the AO-WAM [Gupta and Jayaraman 1993a], ROPM [Kalé 1985; Ramkumar and Kalé 1989; Kalé and Ramkumar 1992], ACE [Gupta, Pontelli, Hermenegildo, Santos Costa 1994; Gupta, Hermenegildo, and Santos Costa 1993], the PBA models [Gupta and Santos Costa 1992c; Gupta, Santos Costa, Pontelli 1994; Gupta, Hermenegildo, and Santos Costa 1993], SBA [Correia et al. 1997], FIRE [Shen 1997], and the COWL models [Santos Costa 1999] 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 and in the flavor of and-parallelism they support. One should also note that, in fact, Conery's model described earlier is an and-or parallel model [Conery 1987a] 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 [Gupta, Santos Costa, Yang, Hermenegildo 1991]. 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 source of parallelism.

The problems faced in implementing combined and- and or-parallel system are unfortunately not only the sum of problems faced in implementing and-parallelism and or-parallelism individually. 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. This should come as no surprise. The issues which are involved in handling and-parallelism and or-parallelism impose requirements that are antithetical to each other on the resulting execution model. For example, or-parallelism focuses on improving the separation between the parallel computations, by assigning separate environments to the individual computing agents; and-parallelism relies on the ability of different computing agents to cooperate and share environments to construct a single solution to the problem.

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 [Gupta and Jayaraman 1993a] and ROPM [Kalé 1985]; or, (ii) devise a new language that will allow extra-logical features but in a controlled way, e.g., PEPSys [Ratcliffe and Syre 1987; Westphal, Robert, Chassin, Syre 1987; Chassin de Kergommeaux and Robert 1990]. The disadvantage with both these approaches is that existing Prolog programs cannot be immediately parallelized. Various approaches have been proposed that allow support for Prolog's sequential semantics even during parallel execution [Santos Costa 1999; Correia et al. 1997; Castro et al. 1998; Ranjan, Pontelli, Gupta 2000; Gupta, Pontelli, Hermenegildo, Santos Costa 1994; Gupta, Santos Costa, Pontelli 1994; Santos Costa, Warren, Yang 1991b].

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

[16]Simulations have shown that indeed better speed-ups will be achieved if more than one source of parallelism are exploited [Shen 1992b; Shen and Hermenegildo 1991].

the same generation" taken from [Ullman 1989]:

```
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 independent and-parallel or a sequential implementation, the goal `parent(john, Yp)` will be recomputed for every solution to `parent(fred, Xp)`[17]. This is clearly redundant since the two parent goals are independent of each other. Theoretically, 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[18]. However, for independent goals with very small granularity, the gain from solution sharing may be overshadowed by the cost of computing the crossproduct 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, 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 [Gupta, Pontelli, Hermenegildo, Santos Costa 1994] and the PBA model [Gupta, Hermenegildo, and Santos Costa 1993], 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 [Gupta, Santos Costa, Pontelli 1994; Gupta, Hermenegildo, and Santos Costa 1993].

### 6.2 Scheduling in And/Or-Parallel Systems

The combination of and- and or-parallelism offers additional challenges. During and-parallel execution, the scheduler is in charge of assigning subgoals to the workers. In presence of or-parallelism, the scheduler is in charge of assigning alternatives to the different workers. When allowing both kinds of parallelism to be exploited at the same time, the system needs to deal with an additional level of scheduling, i.e., determining whether an idle worker should perform or-parallel work or and-parallel work. The problem has been studied in depth by Dutra [Dutra 1994; 1996]. The solution, which has been integrated in the Andorra-I system [Santos Costa, Warren, Yang 1991a], relies on organizing workers into teams, where each team exploits or-parallelism while each worker within a team exploits and-parallelism. The top-level scheduler dynamically manages the structure of the teams, allowing migration of workers from one team to the other—used to perform load-balancing at the level of and-parallelism—as well as allowing the dynamic creation of new teams—used to load-balance or-parallelism. Different strategies have been compared to decide how to reconfigure the teams. For example, in [Dutra 1994] two strategies are compared:

- *work-based strategy:* in which task sizes are estimated at run-time and used to decide workers' allocation;
- *efficiency-based strategy:* in which allocation of workers is based on their current efficiency—i.e., the percentage of time they spent doing useful computation.

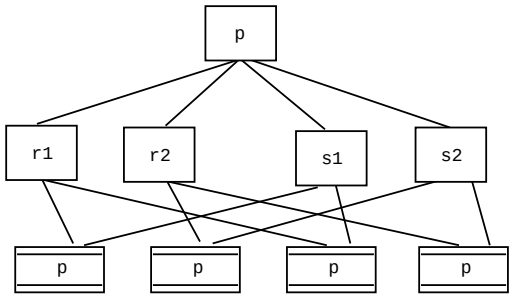
The two strategies have been compared in Andorra-I and the results have been reported in [Dutra 1994; 1996]. The comparison suggests that work-based strategies works well when the estimate of the task size is sufficiently precise; furthermore, if the grain size is small the reconfigurer tends to be called too frequently and/or the scheduler causes excessive task switches. The efficiency-based strategies seems to scale up better with increasing number of workers, reducing idle time and number of reconfigurations.

### 6.3 Models for And/Or-Parallelism

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

[17] Respecting Prolog semantics, a purely independent and-parallel system can avoid recomputation of independent goals but most existing ones do not.

[18] This, as practice suggests, may not always be the case.




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 least-common-hash-window.

Fig. 21. Join Cells

### 6.3.1 *The PEPSys Model*

The PEPSys model [Westphal, Robert, Chassin, Syre 1987; Baron et al. 1988; Chassin de Kergommeaux and Robert 1990] combines and- and or-parallelism using a combination of techniques of time-stamping and *hashing windows* for maintaining multiple environments. In PEPSys (as already discussed in Section 3.2), 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. A process is created for each combination of solutions in the cross-product set. 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 21). 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 least-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 worker does not 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, the actual implementation of PEPSys limited the exploitation of and-parallelism to the case of deterministic goals [Chassin de Kergommeaux 1989]. PEPSys was later modified and evolved into the ElipSys System [Véron et al. 1993]: the hashed windows have been replaced with Binding Arrays and it has also been extended to handle constraints. In turn, ElipSys evolved into the parallel support for the ECLiPSe constraint logic programming system—where or-parallelism only is exploited, using a combination of copying and recomputation [Herold 1995].

#### 6.3.2 *The ROPM Model*

ROPM (Reduce-Or Parallel Model) [Kalé 1991] was devised by Kalé in his Ph.D. Thesis [Kalé 1985]. 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 [Kalé 1991]:

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:

- 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.
- 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.
- 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.

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 22).

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 [Ramkumar and Kalé 1990]. 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

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

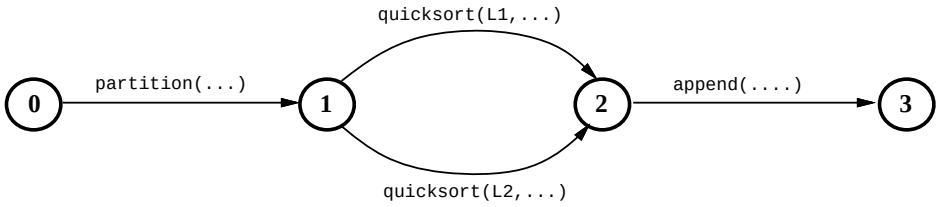



Fig. 22. An Example Data Join Graph

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 [Westphal, Robert, Chassin, Syre 1987] and AO-WAM [Gupta and Jayaraman 1993a]), 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 [Ramkumar and Kalé 1990]. However, efficiency of the computation of the join has been made more 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. ROLOG is a complete implementation, which includes support for side effects [Kalé, Padua, Sehr 1988]. 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 [Ramkumar and Kalé 1989; Conery 1987b] for multiple environment representation.

ROLOG is probably the most advanced process-based model proposed to handle concurrent exploitation of and-parallelism and or-parallelism. Other systems based on similar models have also been proposed in the literature, e.g., OPAL [Conery 1992]—where execution is governed by a set of And and Or processes: And processes solve the set of goals in the body of a rule, and Or processes coordinate the solution of a single goal with multiple matching clauses. And and Or processes communicate solely via messages.

### 6.3.3 *The AO-WAM model*

The AO-WAM model [Gupta and Jayaraman 1993a; Gupta 1994] combines or-parallelism and independent and-parallelism. Independent and-parallelism is exploited in the same way as in &-Prolog and &ACE, and solutions to independent goals are reused (and not recomputed). To represent multiple or-parallel environments in the presence of independent and-parallelism, the AO-WAM extends the binding arrays technique [Warren 1984; 1987a].

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 goal 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 worker has its own binding array. In addition, each worker 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 worker 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 23). Also, on switching nodes, a worker must update its binding array and base array with the help of the trail, like in Aurora.

#### 6.3.4 *The ACE Model*

ACE (And/Or-parallel Copying-based Execution of logic programs) [Gupta, Pontelli, Hermenegildo, Santos Costa 1994; Pontelli and Gupta 1997b] is another model that has been proposed for exploiting or- and independent and-parallelism simultaneously. ACE[19] 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/&ACE and MUSE. The implementation can be envisaged as multiple copies of &-Prolog [Hermenegildo and Green 1991] 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 workers are divided into teams like Andorra-I, where different teams execute in or-parallel with each other while different workers 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 CGE ($true \Rightarrow g_1 \& \ldots \& g_n$). This team will copy all the stack segments in the branch from the root to the CGE including the *parcall* frame[20]. 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 [Gupta, Hermenegildo, and Santos Costa 1993; Gupta, Pontelli, Hermenegildo, Santos Costa 1994; Gupta and Santos Costa 1992a].

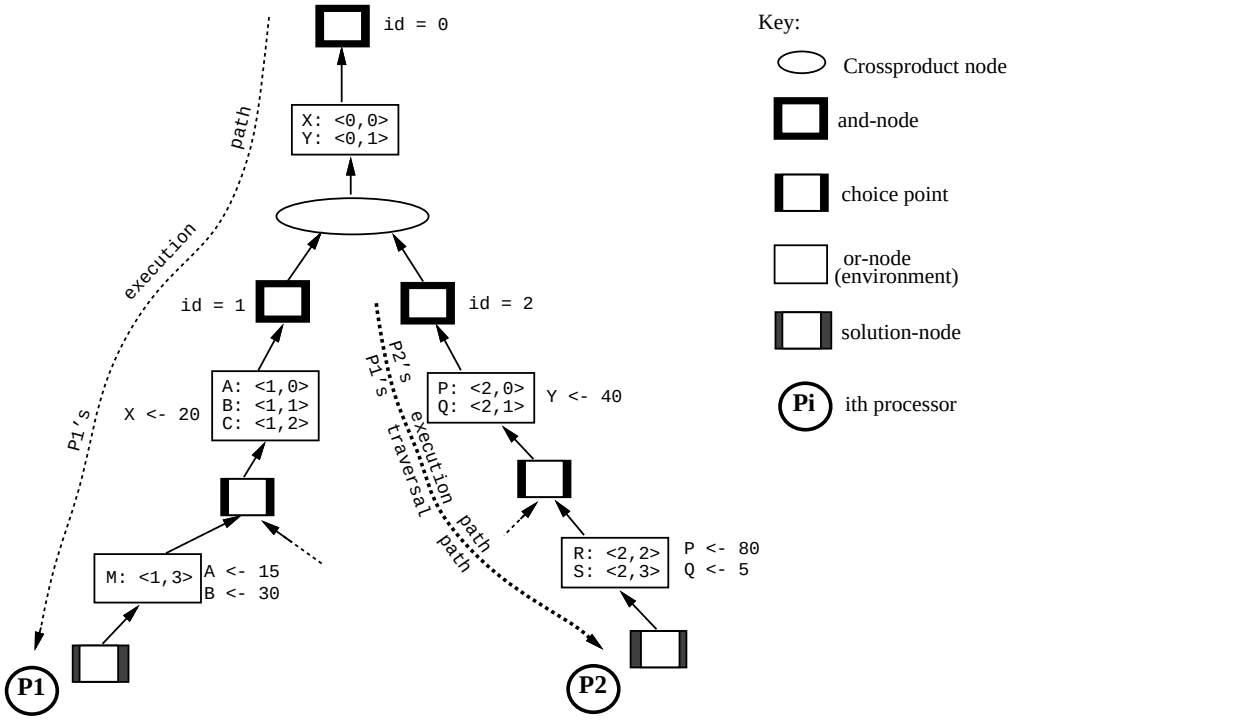
This is also illustrated in Figure 24. The four frames represent four teams working on the computation. The second team recomputes the goal `b`, while the third and fourth teams takes the second alternative of `b` respectively from the first and second team.

#### 6.3.5 *The COWL Models*

The actual development of an or-parallel system based on stack-copying requires a very careful design of the memory management mechanisms. As mentioned in Section 3.5.2 whenever a copy operation takes place, we would like to transfer data structures between agents without the need to perform any pointer-relocation

[19] Note that the ACE platform has been used to experiment with both combined and/or-parallelism as well as dependent and-parallelism, as illustrated in Section 5.5.3.

[20] The parcall frame [Hermenegildo 1986a] records the control information for the CGE and its independent and-parallel goals.



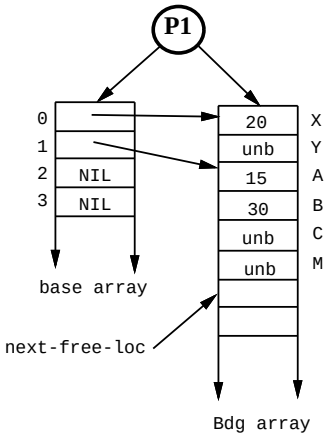




Fig (i): Binding array and Base Array of processor P1 after generating the left and-branch.

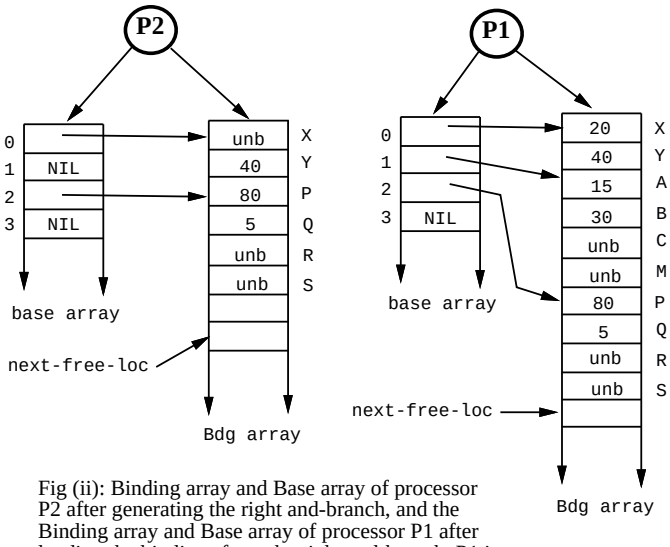



Fig (ii): Binding array and Base array of processor P2 after generating the right and-branch, and the Binding array and Base array of processor P1 after loading the bindings from the right and-branch. P1 is now ready to proceed with the sequential goal after the CGE that gave rise to the cross-product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

Fig. 23. Execution in the AO-WAM

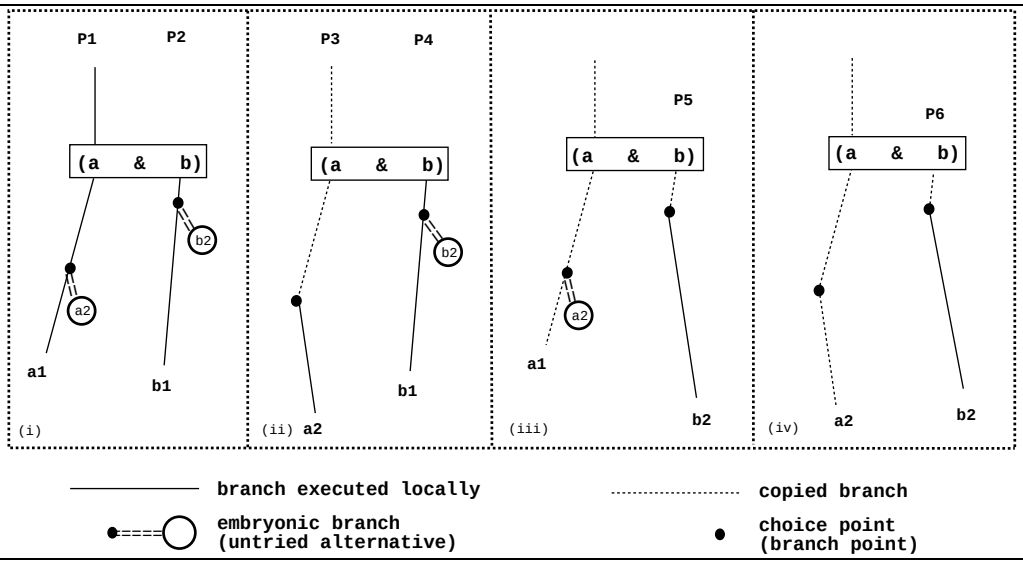


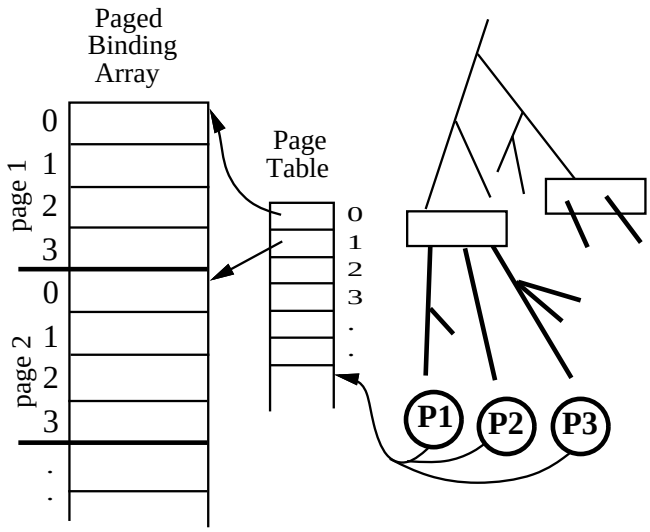

Fig. 24. Execution in ACE

operation. In systems like MUSE and ACE, this has been achieved by using memory mapping techniques which allow the different workers to map their stacks at the same virtual addresses. This technique works well for purely or-parallel systems, but tends to break down when or-parallelism is paired with concurrent exploitation of independent and-parallelism. Stack-copying takes advantage of the fact that the data to be transferred are occupying contiguous memory locations. In a team-based system organization, we need to transfer data structures which have been created by different team members; such data structures are likely to be not contiguous in memory, thus requiring a complex search process to determine the relevant areas to be copied. Furthermore, possible conflicts may arise during copying if parts of the address space of a team have been used for different purposes in different teams.

A simple solution to these issues have been recently proposed by V. Santos Costa in the *Copy-On-Write for Logic Programs (COWL)* methods [Santos Costa 1999]. In COWL, each team occupy a different segment of the overall address space (thus, avoiding conflicts between members of different teams during copying), called *team workspace*. Whenever copying is required, one team simply copies the other team's space into its own. Copying is performed using operating system support for *copy-on-write*—two workers share the same data until one of the tries to write on them; at that point a copy of the data is made and the two workers go their separate ways with private copies of such data. Copying only at "write" time makes copies of data areas (particularly read-only copies) very inexpensive. Thus, in COWL, when copying is required, the destination team releases its own memory mapping and maps (as copy-on-write) the source team's space. Thus, actual data are not copied immediately, but they are automatically transferred by the operating system whenever they are needed. The basic COWL scheme (also known as $\alpha COWL$) has been also extended to optimize the copying by avoiding wasting computation locally performed in the team and reusable after the copying operation (i.e., avoid one team to copy data structures from its own workspace), leading to a second model, called $\beta COWL$.

### 6.3.6 *Paged Binding Array based Model*

ACE can be seen as combining &-Prolog with MUSE, while preserving Prolog semantics. In a similar vein, 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 workers are divided into teams, where different teams work in or-parallel with each other, while different workers within a team work in independent and-parallel. Different and-parallel computations within an or-parallel computation share the same binding array (thus the




An and-parallel computation (delimited by a rectangular box) is performed by a team of three processors which share a common paged binding array and page table. The branches that are part of and-parallel computation are shown in dark in the fig.

Fig. 25. The Paged Binding Array

paged binding array and the page table is common to all workers 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 [Gupta, Hermenegildo, and Santos Costa 1993; Gupta and Santos Costa 1992a]. 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 forms of and-or parallelism [Gupta, Santos Costa, Pontelli 1994].

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 [Shen 1992a], described earlier, as well as models that combine or-parallelism and dependent and-parallelism such as Andorra-I [Santos Costa, Warren, Yang 1991a]. Other combined independent and- and -or parallel models have also been proposed [Biswas et al. 1988; Gupta, Santos Costa, Yang, Hermenegildo 1991].

### 6.3.7 *The Principle of Orthogonality*

One of the overall goals that has been largely ignored in the design of and-or parallel logic programming systems is the principle of *orthogonality* [Correia et al. 1997]. In an orthogonal design, or-parallel execution should be unaware of and-parallel execution and vice-versa. Thus, orthogonality allows the separate design of the data structures and execution mechanisms for or-parallelism and and-parallelism. Achieving this goal is a very ambitious: orthogonality implies that

- each worker should be able to backtrack to a shared choice point and be aware only of or-parallelism;
- whenever a worker enters the public part of the or-tree, the other workers working in the team should be able to continue unaffected their and-parallel computations.

Most existing proposals for combined and/or-parallelism do not meet the principle of orthogonality. Let us consider for example the PBA model and let us consider the computation as shown in Figure 26.

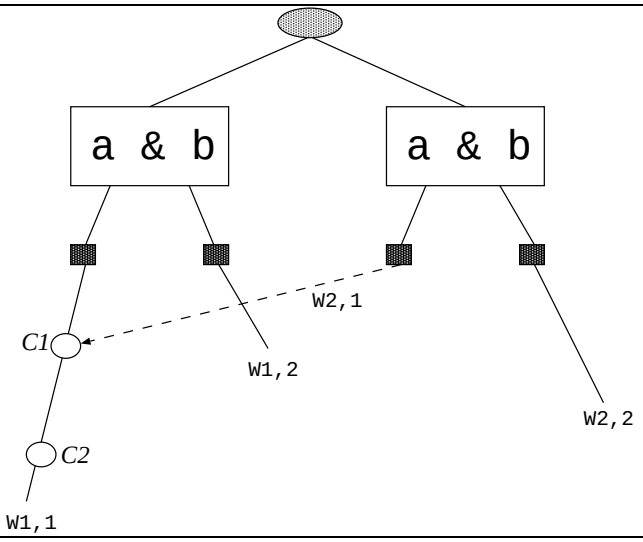



Fig. 26. Lack of Orthogonality in PBA

Let us assume the following configuration:

- workers *W1,1* and *W1,2* compose the first team which is operating on the parallel call on the left; worker *W1,1* makes use of pages 1 and 3—page 1 used before choice point *C1* while page 3 is used after that choice point—while worker *W1,2* makes use of page 2.
- worker *W2,1* and *W2,2* compose team number 2 which is working on the copy of the parallel call (on the right). The computation originates from stealing one alternative from choice point *C1*. In this case, worker *W2,2* makes use of both pages 2 and 3.

If worker *W2,1* backtracks and asks for a new alternative from the first team (one of the alternatives of *C2*), then it will need to use page 3 for installing the bindings created by the team 1 after the choice point *C1*. But for team 2 the page 3 is not available (being used by *W2,2*). Thus worker *W2,2* will be "affected" by backtracking of *W2,1* on a shared choice point.

Various solutions are currently under exploration to support orthogonality. Between the schemes proposed:

- the *Shared Paged Binding Array (SPBA)* [Gupta, Santos Costa, Pontelli 1994] extends the PBA scheme by requiring the use of a global and shared paged binding array;
- the *Sparse Binding Array* [Correia et al. 1997] each conditional variable is guaranteed to have a binding array index which is unique in the whole computation tree and relying on operating system techniques to maintain the large address space that each worker needs to create (each worker needs virtual access to the address space of each worker in the system);
- the COWL methods presented in Section 6.3.5.

A comparison between these three schemes has been presented in [Santos Costa, Rocha, Silva 2000].

### 6.3.8 *The Extended Andorra Model*

The Extended Andorra Model [Warren 1988; Haridi and Jason 1990; Gupta and Warren 1992] and the Andorra Kernel Language (AKL) (later renamed Agent Kernel Language) [Haridi and Jason 1990] combine exploitation of or-parallelism and dependent and-parallelism. Intuitively, both models rely on the creation of copies 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. 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 have been different interpretations 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 27). 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 27.

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 [Warren 1988; Haridi and Jason 1990; Gupta and Warren 1992]. 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 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—i.e., general Prolog.

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 [Janson and Montelius 1991]. A parallel implementation has also been undertaken by Moolenaar and Demoen [Moolenar and Demoen 1993]. A very efficient parallel implementation of AKL has been proposed by Montelius in the Penny system [Montelius 1997;

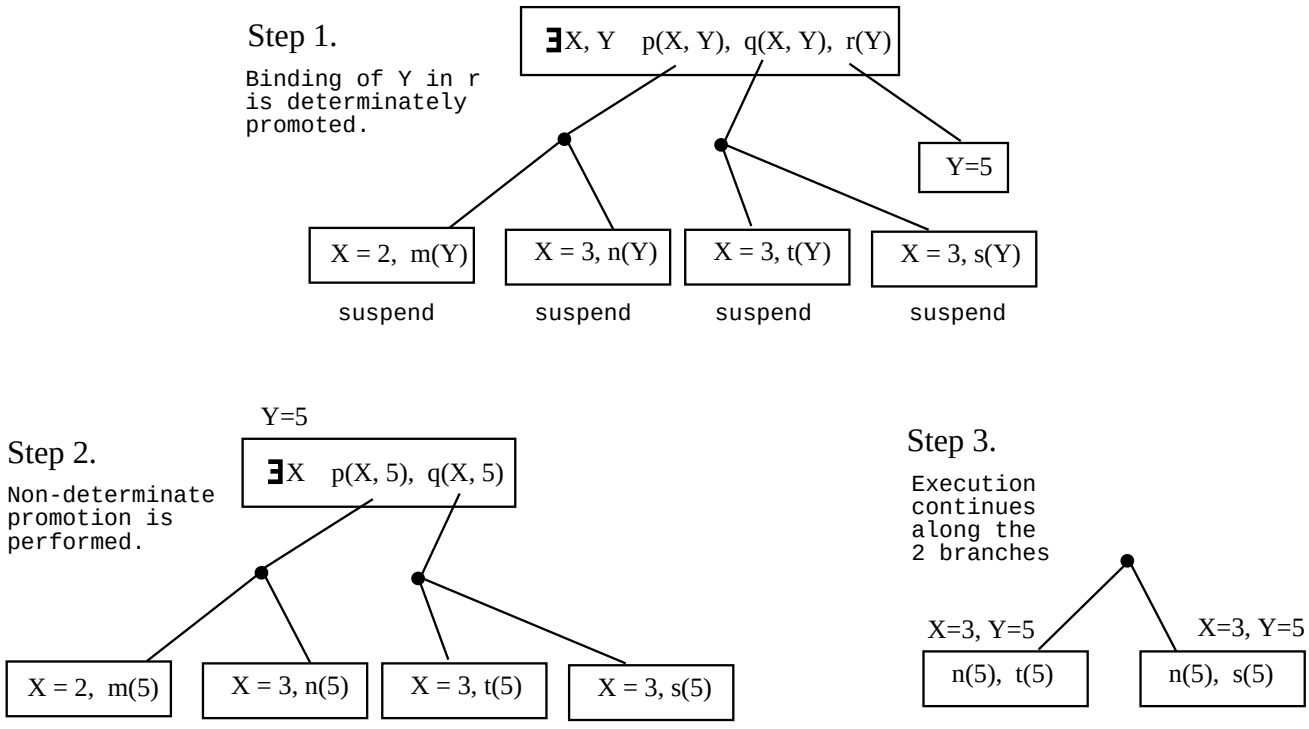



Fig. 27. Execution in EAM

Montelius and Ali 1996]. This implementation combines techniques from or-parallelism and committed-choice languages. 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 [Bueno and Hermengildo 1992]. The development of AKL has been discontinued, although many of the ideas explored in the AKL project have been reused in the development of the concurrent constraint language Oz [Haridi, Van Roy, Brand, Schulte 1998; Popov 1997].

More faithful models to support the execution of the EAM have been recently described and are currently under implementation—e.g., the BEAM model [Lopes and Santos Costa 1999]. The literature also contains proposals of extensions of Prolog that tries to more naturally integrate EAM-style of computations. One example is represented by the *Extended Dynamic Dependent* scheme [Gupta and Pontelli 1999c]. This model has been developed as an extension of the Filtered-Binding model used in the ACE system to support dependent and-parallelism. The model extends Prolog-like dependent and-parallelism by allowing the deterministic promotion step of EAM. This typically allows improved termination properties, reduced number of suspensions during parallel execution, and simple forms of coroutining. These results can be achieved reusing most of the existing (and efficient) technology developed for pure dependent and-parallelism, thus avoiding dramatic changes in the language semantics and novel and complex implementation mechanisms.

## 7. DATA PARALLELISM VS. CONTROL PARALLELISM

Most of the research has focused on exploiting parallelism only on MIMD architectures, viewing or-parallelism and and-parallelism as forms of *control-parallelism*. Intuitively, this means that parallelism is exploited by creating multiple threads of control, which are concurrently performing different operations. An alternative view has been to treat specialized forms of or- and and-parallelism as *data parallelism*. Data parallelism relies on the idea of maintaining a single thread of control, which concurrently operates on multiple data instances. Similarly to what we have considered so far, we can talk about *data or-parallelism* and *data and-parallelism*.

In both cases, the focus is on the parallelization of repetitive operations which are simultaneously applied to a large set of data. This pattern of execution is very frequent in logic programs, as exemplified by frequently

used predicates such as `map`:

```
map([],[]).
map([X|Y],[X1|Y1]) :-
                process(X,X1),
                map(Y,Y1).
```

where the computation indicated by `process` is repeated for each element of the input list. In this context, data parallelism implies that exploitation of parallelism is driven by the computation data-flow, in contrast with standard and- and or-parallelism, which relies on the parallelization of the control structure of the computation (i.e., the construction of the derivation tree).

Exploitation of data parallelism has been shown to lead to good performance on both SIMD and MIMD architectures; the relatively regular format of the parallelism exploited allows simpler and more efficient mechanisms, thus leading to reduced overhead and improved efficiency even on MIMD architectures.

### 7.1 Data Or-Parallelism

In a data or-parallel system, exemplified by the MultiLog system [Smith 1996], or-parallelism of a highly regular nature is exploited on a SIMD architecture. There is one control thread but multiple environments. Data or-parallelism as exploited in MultiLog is useful in applications of generate-and-test nature, 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).
```

In a standard Prolog execution the solutions to `member/2` are enumerated one by one via backtracking, and each solution is separately processed by `process`. 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 variable `Z`. The `process` goal is then executed in data parallel for each binding received by `Z`. Note that the executions of the various `process` goals differ only in the value of the variable `Z`. Therefore, only one control thread is needed which operates on data that is different on different workers, with unification being the only data parallel operation. It is also important to observe that `process/1` is executed once, rather than once per solution of the `member/2` predicate.

Multilog provides a single syntactic extension w.r.t. Prolog: the `disj` annotation allows the compiler to identify the generator predicate. Thus, for a goal of the form `?- disj generate(X)` Multilog will produce a complete description of the set of solutions (as a disjunction of bindings for `X`) before proceeding with the rest of the execution.

For a (restricted) set of applications—e.g., generate and test programs—a data or-parallel system such as MultiLog has been shown to produce good speed-ups.

Techniques, such as the *Last Alternative Optimization* [Gupta and Pontelli 1999b], have been developed to allow traditional or-parallel systems to perform more efficiently in presence of certain instances of data or-parallelism.

### 7.2 Data And-Parallelism

The idea of data parallel execution can also be also naturally applied to and-parallel goals: clauses that contain recursive calls can be unfolded and the resulting goals executed in data parallel. This approach, also known as *recursion parallelism*, has been successfully exploited through the notion of *Reform Compilation* [Millroth 1990]. Consider the following program:

```
map([],[]).
map([X|Y],[X1|Y1]) :- proc(X,X1), map(Y,Y1).
?- map([1, 2, 3], Z).
```

Unfolding this goal we obtain:

```
Z = [X1,X2,X3|Y], proc(1,X1), proc(2,X2), proc(3,X3),map([],Y).
```

Note that the three `proc` 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.

In more general terms, given a recursively defined predicate `p`

$$\begin{aligned} \mathtt{p}(\bar{X}) &:- \ \Delta. \\ \mathtt{p}(\bar{X}) &:- \ \Phi, \mathtt{p}(\bar{X}'), \Psi. \end{aligned}$$

if a goal $\mathtt{p}(\bar{a})$ is determined to perform at least $n$ recursive calls to `p`, then the second clause can be unfolded as:

$$\mathtt{p}(\bar{X}) :- \underbrace{\Phi_1, \ldots, \Phi_n}_{(1)}, \underbrace{\mathtt{p}(\bar{b})}_{(2)}, \underbrace{\Psi_n, \ldots, \Psi_1}_{(3)}.$$

where $\Phi_i$ and $\Psi_i$ are the instances of goals $\Phi$ and $\Psi$ obtained at the $i^{th}$ level of recursion. This clause can be executed by first running, in parallel, the goals $\Phi_1, \ldots, \Phi_n$, then executing $\mathtt{p}(\bar{b})$ (typically the base case of the recursion), and finally running the goals $\Psi_n, \ldots, \Psi_1$ in parallel as well. In practice the unfolded clause is not actually constructed, instead the head unification for the $n$ levels of recursion is performed at the same time as the size of the recursion is determined, and the body of the unfolded clause is compiled into parallel code.

*Reform Prolog* [Bevemyr et al. 1993] is an implementation of a restricted version of the reform compilation approach. In particular only predicates performing integer-recursion or list-recursion and for which the size of the recursion is known at the time of the first call are considered for parallel execution.

To achieve efficient execution, Reform Prolog requires the generation of deterministic bindings to the external variables, thus relieving the system from the need to perform complex backtracking on parallel calls. Sophisticated compile-time analysis tools have been developed to guarantee the conditions necessary for the parallel execution and to optimize execution [Lindgren 1993]. Reform Prolog has been ported on different MIMD architectures, such as Sequent [Bevemyr et al. 1993] and KSR-1 [Lindgren, Bevemyr, Millroth 1995].

Exploitation of data and-parallelism explicitly through *bounded quantification* has also been proposed [Barklund and Millroth 1992]. In this case, the language is extended with constructs used to express bounded forms of universal quantification (e.g., $\forall(X \in S)\varphi$). Parallelism is exploited by concurrently executing the body of the quantified formula (e.g., $\varphi$) for the different values in the domain of the quantifiers (e.g., the different values in the set $S$).

Recently, various works have also attempted to create a bridge between and-parallelism and data and-parallelism. The objective of these works is to allow the identification of instances of data and-parallelism in generic and-parallel programs, and the use of specialized and more efficient execution mechanisms to these cases [Pontelli and Gupta 1995b; Hermenegildo and Carro 1995].

## 8. PARALLEL CONSTRAINT LOGIC PROGRAMMING

Although the main focus of this survey is parallel execution of Prolog programs, we would like to briefly overview in this section the most relevant efforts which have been made towards parallel execution of *Constraint Logic Programming (CLP)*. This is of interest since many of the techniques adopted for parallel execution of CLP are directly derived from those used in the parallelization of Prolog computations.

The notion of data parallelism has been also adapted to execute Finite Domain Constraint Logic Programming. The two most representative examples are the parallel implementation of Chip [Van Hentenryck 1989a] and the Firebird system [Tong and Leung 1993].

The parallel implementation of Chip [Van Hentenryck 1989a] has been realized using the PEPSys or-parallel system. In this implementation, parallelism is exploited from the choice-points generated by the labeling phase introduced during resolution of finite domain constraints. The results reported in [Van Hentenryck 1989a] are encouraging, and prove that or-parallel technique are quite suitable to support also CLP executions.

Firebird [Tong and Leung 1993] is a data parallel extension of flat GHC (a committed-choice language) with finite domain constraints, relying on the data or-parallel execution obtained from the parallelization of the labeling phase of CLP. Execution includes *non-deterministic* steps, leading to the creation of parallel choice-points, and *indeterministic* steps, based on the usual committed-choice execution behavior. Arguments

of the predicates executed during an indeterministic step can possibly be vector of values—representing the possible values of a variable—and are explored in data parallel. The overall design of Firebird resembles the model described earlier for Multilog.

The implementation of Firebird has been developed on a DECmpp SIMD parallel architecture, and has shown considerable speedups for selected benchmarks (e.g., about two orders of magnitude of speedup for the *n-queens* benchmark using 8,192 processors) [Tong and Leung 1995].

A number of other proposals have appeared in the literature which instead provide parallelization of constraint logic programs based on the ideas of control parallelism (instead of data parallelism). One of the first works in this field is [Gregory and Yang 1992], in which finite domain constraint solving operations are mapped to the parallel execution mechanisms of Andorra-I.

Another proposal is represented by GDCC [Terasaki et al. 1992], an extension of KL1 (running on the PSI architecture) with constraint solving capabilities—constructed following the cc model proposed by Saraswat [Saraswat 1989]. GDCC provides two levels in the exploitation of parallelism: *(i)* the gdcc language is an extension of the concurrent KL1 language, which includes *ask* and *tell* of constraints; this language can be executed in parallel using the parallel support provided by KL1; *(ii)* gdcc has been interfaced to a number of constraint solvers (e.g., algebraic solvers for non-linear equations), which are themselves capable of solving constraint in parallel.

Recently, the focus have shifted on the direct parallelization of the sources of non-determinism inherent in the operational semantics of CLP. The work in [Pontelli and El-Khatib 2001] presents a methodology for exploring in parallel the alternative elements of a constraint domain, while [Ruiz-Andino, Araujo, Saenz, Ruz 1999] revisits the techniques used to parallelize arc-consistency algorithms (e.g., parallel AC3 [Samal and Henderson 1987] and AC4 [Nguyen and Deville 1998]) and applies them to the specific case of indexical constraints in CLP over finite domains.

## 9. IMPLEMENTATION AND EFFICIENCY ISSUES IN PARALLEL LP

Engineering an efficient, practical parallel logic programming system is by no means an easy task[21]. There are numerous issues one has to consider, some of the broad ones are discussed below:

### 9.1 Process-based vs. Processor-based

Broadly speaking there are 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 [Conery 1987a] and the Reduce-Or Process Model [Kalé 1985], 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 [Shapiro 1987]. Process-based approaches are suited for implementation on non-shared memory MIMD processors[22], 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 stack-copying and stack-splitting show a high degree of locality in memory reference behavior and hence are suited for non-shared memory machines as well [Ali 1987; Ali et al. 1992]. 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 [Gupta, Hermenegildo, Santos Costa 1992; Gupta and Pontelli 1999a]. Alternatively, one could employ scalable virtual shared memory

[21] For instance, many person-years of efforts have been spent in building some of the existing systems, such as MUSE, Aurora, Andorra-I, ACE, and &-Prolog.

[22] Some more proposals for distributed execution of logic programs can be found in [Kacsuk 1990].

architectures that have been proposed [Warren and Haridi 1988] and built (e.g., KSR, SGI Origin, IBM NUMA-Q).
A parallel logic programming system should possess the following two properties:

- On a single processor, the performance of the parallel system should be comparable to sequential logic programming implementations (i.e., there should not be limited slow down compared to a sequential system). Systems such as MUSE, ACE, Aurora and &-Prolog indeed get very close to achieving this goal.
- The parallel system should be able to take advantage of the sequential compilation technology [Warren 1983; Ait-Kaci 1992; Van Roy 1990] that has advanced rapidly in the last two decades.

Experience has shown that process-based system lose out on both the above counts. Similar accounts have been reported also in the context of committed-choice languages (where the notion of process-based matches well with the view of each subgoal as an individual process which is enforced by the concurrent semantics of the language)—indeed the fastest parallel implementation of committed-choice languages (e.g., [Crammond 1992; Rokusawa, Nakase, Chikayama 1996]) rely on a processor-based implementation. In the context of Prolog, the 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 has been attempted for process-based models with good results [Kalé and Ramkumar 1992; Ramkumar and Kalé 1992].

### 9.2 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 $\mathtt{n}_1$ in the search tree is in a branch to the right of another branch containing node $\mathtt{n}_2$, then the data structures corresponding to node $\mathtt{n}_2$ would be reclaimed before those of $\mathtt{n}_1$ are allocated.

(2) If a node $\mathtt{n}_1$ is the ancestor of another node $\mathtt{n}_2$ in the search tree, then the data structures corresponding to $\mathtt{n}_2$ would be reclaimed before those of $\mathtt{n}_1$.

As a result of these two rules, space is always reclaimed from the top of the stacks during backtracking in logic programming systems which perform a depth-first search of the computation tree, as Prolog does. 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 worker 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 this restriction is imposed, then while memory management becomes easier, some parallelism may be lost since an idle worker may not be able to pick available work in a node because doing so will violate this restriction. If this restriction is not imposed, then it becomes necessary 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 [Hermenegildo 1987], which results in the LIFO nature of stacks being destroyed.

The approach taken by many parallel systems (e.g., the ACE and DASWAM and-parallel systems and the Aurora or-parallel system) 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"—such as choice-points and goals descriptors—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.

While considerable effort has been invested in the design of garbage collection schemes for sequential Prolog implementations (e.g., [Pittomvils, Bruynooghe, Willems 1985; Appleby, Carlsson, Haridi, Sahlin 1988; Older and Rummell 1992; Bekkers, Ridoux, Ungaro 1992]), considerably more limited effort has been placed on adapting these mechanisms to the case of parallel logic programming systems. Garbage collection is indeed a serious concern, since parallel logic programming systems tend to consume more memory than sequential ones (e.g., use of additional data structures, such as parcall frames, to manage parallel executions). For example, results reported for the Reform Prolog system indicates that on average 15% of the execution time is spent in garbage collection. Some early work on parallelization of the garbage collection process (applied mostly to basic copying garbage collection methods) can be found in the context of parallel execution of functional languages (e.g., [Halstead 1984]). In the context of parallel logic programming, two relevant efforts are:

- the proposal by Ali [Ali 1995], which provides a parallel version of a copying garbage collector, refined to guarantee avoidance of unnecessary copying (e.g., copy the same data twice) and load balancing between workers during garbage collection;
- the proposal by Bevemyr [Bevemyr 1995], which extends the work by Ali into a generational copying garbage collector (objecs are divided in generations, where newer generations contains objects more recently created; the new generation is garbage collected more often then the old one).

Generational garbage collection algorithms have also been proposed in the context of parallel implementation of committed-choice languages (on PIM architectures) [Ozawa, Hosoi, Hattori 1990; Xu, Koike, Tanaka 1989].

### 9.3 Optimizations

A system that builds an and-or tree to solve a problem with non-determinism may look trivial to implement at first, but experience shows that it is quite a difficult task. A naive parallel implementation may lead to a slow down, or, may incur a severe overhead compared to a corresponding sequential system. The parallelism present in these frameworks is typically very irregular and unpredictable; for this reason, parallel implementations of non-deterministic languages typically rely on *dynamic scheduling*. Thus, most of the work for partitioning and managing parallel tasks is performed during run-time. These duties are absent from a sequential execution and represent *parallel overhead*. Excessive parallel overhead may cause a naive parallel system to run many times slower on one processor compared to a similar sequential system.

A large number of *optimizations* have been proposed in the literature to improve the performance of individual parallel logic programming systems (e.g., [Ramkumar and Kalé 1989; Shen 1994; Pontelli, Gupta, Tang, Carro, Hermenegildo 1996]). Nevertheless, limited effort has been placed in determining overall principles which can be used to design over-the-border optimization schemes for entire classes of systems. A proposal in this direction has been put forward by Gupta & Pontelli [Gupta and Pontelli 1997; Pontelli, Gupta, Tang 1995]. The proposal presents a number of general optimization principles that can be employed by implementors of parallel non-deterministic systems to keep the overhead incurred for exploiting parallelism low. These principles have been used to design a number of optimization schemes—such as the *Last Parallel Call Optimization* [Pontelli, Gupta, Tang 1995] (used for independent and-parallel systems) and the *Last Alternative Optimization* [Gupta and Pontelli 1999b] (used for or-parallel systems).

Parallel execution of a logic programming system can be viewed as the *parallel traversal/construction* of an *and-or tree*. Given the and-or tree for a program, its sequential execution amounts to traversing the and-or tree in a pre-determined order. Parallel execution is realized by having different workers concurrently traversing different parts of the and-or tree in a way consistent with the operational semantics of the programming language. By operational semantics we mean that data-flow (e.g., variables bindings) and control-flow (e.g.,

input/output operations) dependencies are respected during parallel execution (similar to loop parallelization of Fortran programs, where flow dependencies have to be preserved). Parallelism allows overlapping of exploration of different parts of the and-or tree. Nevertheless, as mentioned earlier, this does not always translate to an improvement in performance. This happens mainly because of the following reasons:

- the tree structure developed during the parallel computation needs to be explicitly maintained, in order to allow for proper management of non-determinism and backtracking—this requires the use of additional data structures, not needed in sequential execution. Allocation and management of these data structures represent overhead during parallel computation with respect to sequential execution;
- the tree structure of the computation needs to be repeatedly traversed in order to search for multiple alternatives and/or cure eventual failure of goals, and such traversal often requires synchronization between the workers. The tree structure may be traversed more than once because of backtracking, and because idle workers may have to find nodes that have work after a failure takes place or a solution is reported (dynamic scheduling). This traversal is much simpler in a sequential computation, where the management of non-determinism is reduced to a linear and fast scan of the branches in a predetermined order.

Based on this it is possible to identify ways of reducing these overheads.

*Traversal of Tree Structure:* there are various ways in which the process of traversing the complex structure of a parallel computation can be made more efficient:

(1) simplification of the computation's structure: by reducing the complexity of the structure to be traversed it should be possible to achieve improvement in performance. This principle has been reified in the already mentioned Last Parallel Call Optimization and the Last Alternative Optimization, used to flatten the computation tree by collapsing contiguous nodes lying on the same branch if some simple conditions hold.

(2) use of the knowledge about the computation (e.g., determinacy) in order to guide the traversal of the computation tree: information collected from the computation may suggest the possibility of avoiding traversing certain parts of the computation tree.
This has been reified in various optimizations, including the *Determinate Processor Optimization* [Pontelli, Gupta, Tang 1995].

*Data Structure Management:* since allocating data structures is generally an expensive operation, the aim should be to reduce the number of new data structures created. This can be achieved by:

(1) reusing existing data structures whenever possible (as long as this does preserve the desired execution behavior).
This principle has been implemented, for example, in the *Backtracking Families Optimization* [Pontelli, Gupta, Tang, Carro, Hermenegildo 1996].

(2) avoiding allocation of unnecessary structures: most of the new data structures introduced in a parallel computation serve two purposes: *(i)* support the management of the parallel parts of the computation; *(ii)* support the management of non-determinism.
This principle has been implemented in various optimizations, including the *shallow backtracking optimization* [Carlsson 1989] and the *Shallow Parallelism Optimization* [Pontelli, Gupta, Tang 1995].

This suggests possible conditions under which one can avoid creation of additional data structures: *(i)* no additional data structures are required for parts of the computation tree which are *potentially* parallel but are actually explored by the same computing agent (i.e., potentially parallel but practically sequential); *(ii)* no additional data structures are required for parts of the computation that will not contribute to the non-deterministic nature of the computation (e.g., deterministic parts of the computation).

### 9.4 Work Scheduling

The Work Scheduler, or the software that matches available work with workers, 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—including supporting execution of pruning and other order-sensitive operations—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 [Ali and Karlsson 1992b; Beaumont and Warren 1993; Sindaha 1992; Beaumont 1991].

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 workers into teams, the scheduler has to determine the sizes of the teams, and decide when to migrate a worker from a team that has no work left to another that does have work, and so on [Dutra 1994; 1995].

The fact that Aurora, quite a successful or-parallel system, has about five schedulers built for it [Calderwood and Szeredi 1989; Beaumont et al. 1991; Sindaha 1992; Butler et al. 1988], 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 [Dutra 1991; Ueda and Montelius 1996].

### 9.5 Granularity

Granularity of computation, or the average amount of work done between two calls to the scheduler by the worker, is another aspect that is important for parallel system design. It is desirable to have a large granularity of computation, so that the scheduling overhead is a small fraction of the total work done by a worker. The general idea is that if the gain obtained by executing a task in parallel is less than the overheads required to support the parallel execution, then the task is better executed sequentially. Granularity issues have a direct bearing on the scheduler. It is the scheduler's responsibility to find work for a worker 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 choice-points—that is available in the subtree rooted at a node. Workers 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 worker can only be picked up from the public region. In the private region, the worker 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 worker is made public. In these systems, granularity control is completely performed at run-time.

Modern systems implement granularity control using a two-phase process [Shen, Santos Costa, King 1998; Tick and Zhong 1993; Debray et al. 1990]:

(1) at *compile-time* a global analysis tool performs an activity typically called *cost estimation*. Cost estimates are parametric formulae expressing lower or upper bounds to the time complexity of the different (potentially) parallel tasks.

(2) at *run-time* the cost estimates are instantiated, before task's execution, and compared with predetermined thresholds; parallel execution of the task is allowed only if the cost estimate is above the threshold.

For instance, the two-phase granularity control has been integrated in the generation of CGEs for and-parallelism [Muthukumar and Hermenegildo 1989a; Jacobs and Langen 1989], by adding the threshold test in the condition part of the graph expression:

$$( \mathit{cost\_estimate}(n_1, \ldots, n_k) > \tau \Rightarrow \mathtt{goal}_1 \ \mathtt{\&} \ \ldots \ \mathtt{\&} \ \mathtt{goal}_m )$$

the $m$ subgoals will be allowed in a parallel execution only if the result of the *cost_estimate* is above the threshold $\tau$. The parameters of *cost_estimate* are those goals input arguments which directly determine the

time-complexity of the parallel subgoals—as identified by the global analysis phase. For example, a modified annotation for the recursive clause of Fibonacci may look as follows:

```
fib(N,Res) :-
      N1 is N-1, N2 is N-2,
      ( N > 5 => fib(N1,R1) & fib(N2,R2) ;
                 fib(N1,R1), fib(N2,R2)
      ),
      R is R1 + R2.
```

(under the simplistic assumption that for values of N larger than 5 it is deemed worthwhile to exploit parallelism). The key problem in this two-phase approach is the automatic derivation of those functions which bound the time-complexity of given tasks.

The first proposals in this regard are those made by Tick & Zhong [Tick and Zhong 1993] and by Lin, Debray, and Hermenegildo [Debray et al. 1990]. Both the schemes are capable of deriving cost estimation which represent upper bounds for the time-complexity of the selected tasks.

The use of upper-bounds is sub-optimal in the context of granularity control—the fact that the upper bound is above a threshold does not guarantee that the actual time-complexity of the task is going to be above the threshold. For this reason more recent efforts have focused on the derivation of lower-bound estimates [King, Shen, Benoy 1997; Debray et al. 1997b]. A very effective implementation of some of these techniques have been realized in the CASLOG system [Debray et al. 1990] and integrated in the CIAO logic programming system [Hermenegildo 1994]. Lower bound analysis is considerably more complex than upper-bound analysis. First of all, it requires the ability of determining properties of tasks with respect to failure [Debray et al. 1997a]. If we focus on the computation of a single solution, then for a clause $C : H : -B_1, \ldots, B_k$ one can make use of the relation

$$Cost_C(n) \geq \sum_{i=1}^{r} Cost_{B_i}(\phi_i(n)) + h(n)$$

where

- $n$ is the representation of the size of the input arguments to the clause $C$
- $\phi_i(n)$ is the (lower bound) of the relative size of the input arguments to $B_i$
- $B_r$ is the rightmost literal in $C$ which is guaranteed to not fail
- $h(n)$ is the lower bound of the cost of head unification and tests for the clause $C$

The lower bound $Cost_p$ for a predicate $p$ is obtained by taking the *minimum* of the lower bounds for the clauses defining $p$.

For the more general case of estimation of lower bound for the computation of all the solutions, it becomes necessary to estimate the lower bound to the number of solutions that each literal in the clause will return. In [Debray et al. 1997b] the problem is reduced to the computation of the chromatic polynomial of a graph.

In [King, Shen, Benoy 1997] bottom-up abstract interpretation techniques are used to evaluate lower-bound inequalities (i.e., inequalities of the type $d_{min} \leq t_{min}(l)$, where $d_{min}$ represents the threshold to allow spawning of parallel computations, while $t_{min}(l)$ represents the lower bound to the computation time for input of size $l$) for large classes of programs.

Metrics different from task complexity have been proposed to support granularity control. A related effort is the one by Shen, Costa, and King [Shen, Santos Costa, King 1998], which makes use of the amount of work performed between major sources of overheads—called *distance metric*—to measure granularity.

### 9.6 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. In fact, in spite of the emphasis on implicit exploitation of parallelism, speedups and execution times can be affected by the user through the use of user annotations (e.g., CGEs) and/or simple program transformations—such as folding/unfolding of subgoals or modification of the order of subgoals and program clauses.

The goal of a visualization tool is to produce a visual representation of certain *observable* characteristics of the parallel execution. Each observable characteristic is denoted by an *event*; the parallel execution is thus represented by a collection of time-annotated events, typically called a *trace*. Many tools have already been developed to visualize parallel execution of logic programs. The large majority of the tools developed so far are *post-mortem* visualization tools: they work by logging events during parallel execution, and then using this trace for creating a graphical representation of the execution.

Different design choices have been considered in the development of the different tools [Carro et al. 1993; Vaupel, Pontelli, Gupta 1997]. The existing systems can be distinguished according to the following criteria:

- *Static vs. Dynamic:* static visualization tools produce a static representation of the observable characteristics of the parallel computation; on the other hand, dynamic visualization tools produce an animated representation, synchronizing the development of the representation with the time-stamps of the events in the trace.
- *Global vs. Local:* global visualization tools provide a single representation which captures all the different observable characteristics of the parallel execution; local visualization tools instead allow the user to focus on specific observable characteristics.

The first visualization tools for parallel logic programs were developed for the Argonne Model [Lusk and Disz 1987] and for the ElipSys system [Dorochevsky and Xu 1991]. The former was subsequently adopted by the Aurora System under the name *Aurora Trace*. The MUSE group also developed visualization tools, called *Must*, for visualizing or-parallel execution—which is itself based on the Aurora Trace design. All these visualizers for or-parallel execution are *dynamic* and show the dynamically growing or-parallel search tree. Figure 28 shows a snapshot of Must—circles denote choice points and the numbers denote the position of the workers in the computation tree.

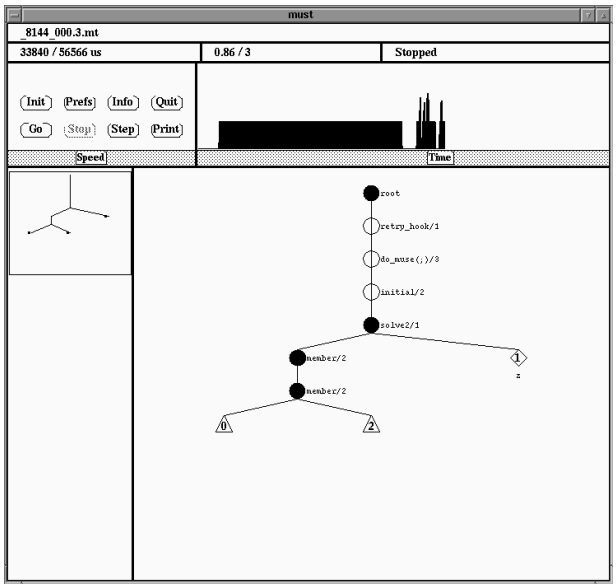



Fig. 28. Snapshot of Must

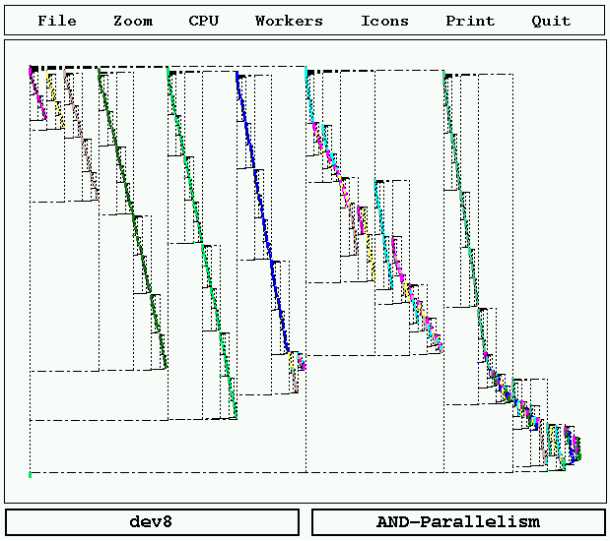



Fig. 29. Snapshot of VisAndOr

Static representation tools have been developed for both or- and and-parallelism. Notable efforts are represented by *VisAndOr* [Carro et al. 1993] and *ParSee* [Kusalik and Prestwich 1996]. Both the tools are capable of representing either or- or and-parallelism—although neither of them can visualize the concurrent exploitation of the two forms of parallelism—and they are aimed at producing a static representation of the distribution of work between the available workers. Figure 29 shows a snapshot of VisAndOr's execution. VisAndOr's effort is particularly relevant, since it is one of the first tools with such characteristics to be

developed, and because it defined a standard in the design of trace format—adopted by various other systems [Vaupel, Pontelli, Gupta 1997; Kusalik and Prestwich 1996; Fonseca et al. 1998]. Must and VisAndOr have been integrated in the ViMust system; a time-line moves on the VisAndOr representation synchronized with the development of the computation tree in Must [Carro et al. 1993].

Other 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 [Tick 1992; Aikawa et al. 1992].

Tools have also been developed for visualizing combined and/or-parallelism, as well as to provide a better balance between dynamic and static representations—e.g., VACE [Vaupel, Pontelli, Gupta 1997], based on the notion of C-trees [Gupta, Pontelli, Hermenegildo, Santos Costa 1994], and VisAll [Fonseca et al. 1998]. Figure 30 shows a snapshot of VACE.

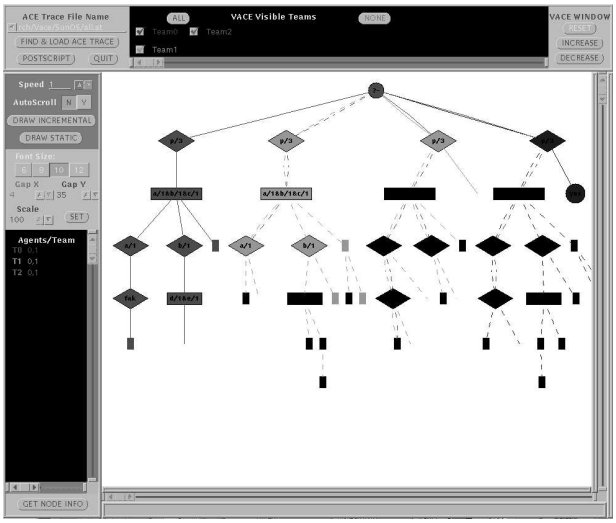



Fig. 30. Snapshot of VACE

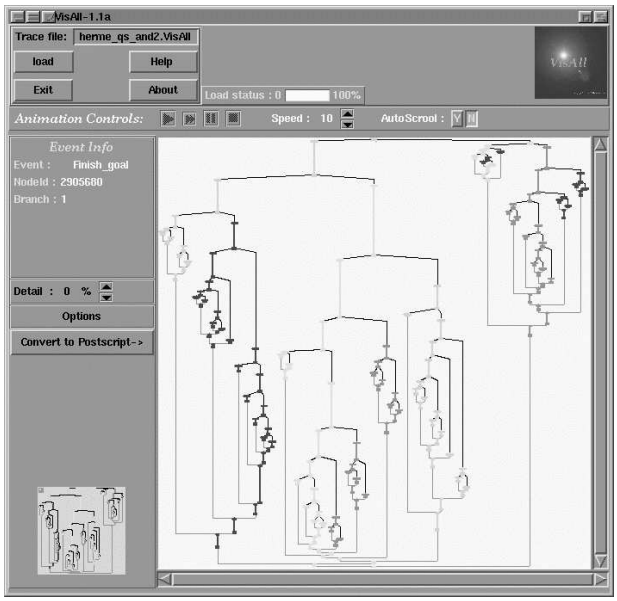



Fig. 31. Snapshot of VisAll

A final note is for the *VisAll* system [Fonseca et al. 1998]. VisAll provides a universal visualization tool which subsumes the features offered by most of the existing ones—including the ability to visualize combined and/or-parallel executions. VisAll receives as input a trace together with the description of the trace format—thus allowing it to process different trace formats. Figure 31 shows a snapshot of VisAll representing an and-parallel computation.

The importance of visualization tools in the development of a parallel logic programming system 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 agent scheduling software.

### 9.7 Compile-time Support

Compile-time support is crucial for efficiency of parallel logic programming systems. Compile-time analysis tools based on Abstract Interpretation techniques [Cousot and Cousot 1992] 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 [Muthukumar and Hermenegildo 1989a; 1991; Jacobs and Langen 1989]. ACE makes use of abstract interpretation techniques to build extended CGEs for dependent and-parallelism [Pontelli, Gupta, Pulvirenti, Ferro 1997]. The Andorra-I system relies on determinacy analysis done at compile-time for detecting determinacy of goals at runtime [Santos

Costa, Warren, Yang 1991b; Debray and Warren 1989]. 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 [Haridi and Jason 1990] programs [Bueno and Hermengildo 1992], and has also been used for supporting Prolog semantics in parallel systems that contain dependent and-parallelism, e.g., Andorra-I [Santos Costa, Warren, Yang 1991b]. 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 [Zhong et al. 1992; Debray et al. 1990], to improve independence in and-parallel computations [Pontelli and Gupta 1998], 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, detecting whether a variable is conditional or not, 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.

### 9.8 Architectural Influence

As for any parallel system, also in the case of parallel logic programming the characteristics of the underlying architecture have profound impact on the performance of the system.

A number of experimental works have been conducted to estimate the influence of different architectural parameters on individual parallel systems. Relevant work has been proposed by

- Hermenegildo and Tick [Hermenegildo and Tick 1989; Tick 1987] proposed various studies estimating the performance of and-parallel systems on shared memory machines;
- Montelius and Haridi [Montelius and Haridi 1997; Montelius 1997] have proposed detailed performance analysis of the Penny system, mostly using the SIMICS Sparc processor simulator;
- Gupta and Pontelli [Gupta and Pontelli 1999a] have used simulation studies (based on the use of the SIMICS simulator) to validate the claim that stack-splitting improves the locality of an or-parallel computation based on stack copying;
- Bianchini, Costa, and Dutra [Santos Costa, Bianchini, Dutra 1997] have also analyzed the performance of parallel logic programming systems (specifically Aurora and Andorra-I) using processor simulators (specifically a simulator of the MIPS processor). Their extensive work has been aimed at determining the behavior of parallel logic programming systems on parallel architectures (with a particular focus on highly scalable architectures, e.g., distributed shared memory machines). In [Santos Costa, Bianchini, Dutra 1997] the simulation framework adopted is presented, along with the development of a methodology for understanding cache performance. The results obtained have been used to provide concrete improvements to the implementation of the Andorra-I system [Santos Costa, Bianchini, Dutra 2000]. The impact of cache coherence protocols on the performance of parallel Prolog systems is studied in more detail in [Dutra, Santos Costa, Bianchini 2000; Silva, Dutra, Bianchini, Santos Costa 1999; Calegario and Dutra 1999].

These works tend to agree on the importance of considering architectural parameters in the design of a parallel logic programming systems. For example, the results achieved by Costa et al. for the Andorra-I systems indicate that:

- or-parallel prolog systems provide a very good locality of computation, thus the system does not seem to require very large cache sizes;
- small cache blocks appear to provide better behavior, especially in presence of or-parallelism—the experimental work by [Dutra, Santos Costa, Bianchini 2000] indicates a high-risk of false-sharing in presence of blocks larger than 64 bytes;
- in [Dutra, Santos Costa, Bianchini 2000] compares the effect of Write Invalidate vs. Write Update as cache coherence protocols. The study underlines the superiority of a particular version of the Write update algorithm (an hybrid method where each node independently decides upon receiving an update request whether to update the local copy of data or simply invalidate it).

Similar results have been reported in [Montelius and Haridi 1997], which underlines the vital importance of good cache behavior and avoidance of false sharing for exploitation of fine-grain parallelism in Penny.

## 10. APPLICATIONS AND APPLICABILITY

One can conclude from the discussion in the previous sections, a large body of research has been developed in the design of parallel execution models for Prolog programs. Unfortunately, relatively modest emphasis has been placed on the study of the *applicability* of these techniques to real-life problems.

A relevant study in this direction has been proposed in [Shen and Hermenegildo 1991; Shen 1992b]. This work considered a large pool of application and studied their behavior with respect to the exploitation of or-parallelism, independent and-parallelism and dependent and-parallelism. The pool of applications considered includes traditional toy benchmark programs (e.g., n-queens, matrix multiplication) as well as larger Prolog applications (e.g., Warren's WARPLAN planner, Boyer-Moore's Theorem Prover, the Chat NLP application). The results can be summarized as follows:

- Depending on their structure, there are applications that are very rich in either forms of parallelism—i.e., either they offer considerable or-parallelism and almost no and-parallelism or vice-versa.
- Neither of the two forms of parallelism is predominant over the other.
- There are a large number of applications which offer moderate quantities of both forms of parallelism. In particular, the real-life applications considered offered limited amounts of both forms of parallelism. In these cases, experimental results showed that concurrent exploitation of both forms of parallelism will benefit over exploitation of a single form of parallelism.

The various implementations of parallel logic programming systems developed have been effectively applied to speedup execution of various large real-life applications. These include:

- independent and dependent and-parallelism has been successfully extracted from Prolog-to-WAM compilers (e.g., the PLM compiler) [Pontelli, Gupta, Tang, Carro, Hermenegildo 1996];
- Static Analyzers for Prolog programs [Pontelli, Gupta, Tang, Carro, Hermenegildo 1996; Hermenegildo and Green 1991]
- Natural Language Processing applications have been very successfully parallelized extracting both or- and and-parallelism—e.g., the Chat system [Santos Costa, Warren, Yang 1991a; Shen 1992b], the automatic translator Ultra [Pontelli, Gupta, Wiebe, Farwell 1998], the word-disambiguation application Artwork [Pontelli, Gupta, Wiebe, Farwell 1998].
- Computational Biology applications—e.g., the Aurora or-parallel system has been used to parallelize Prolog applications for DNA sequencing [Lusk, Mudambi, Overbeek, Szeredi 1993].
- both Aurora and ACE have been applied to provide parallel and concurrent backbones for Internet-related applications [Szeredi, Molnár, Scott 1996; Pontelli 2000].
- Andorra-I has been used in the development of advanced traffic management systems [Hasenberger 1995], used by British Telecom to control traffic flow on their telephony network.
- Telecommunication applications [Crabtree 1991; Santos Costa, Warren, Yang 1991c].
- Aurora has been used to develop a number of concrete applications. Particularly important are those developed in the context of the Cubiq project:
  - the EMRM system, a medical record management system, which supports collection of medical information following the SOAP medical knowledge model [Szeredi and Farkas 1996].
  - The CONSULT credit rating system, which makes use of rule-based specification of credit assessment procedures [IQSoft 1992].

This body of experimental work indicates that the existing technology for parallel execution of logic programs is effective when applied to large and complex real-life Prolog applications. Further push for application of parallelism comes from the realm of constraint logic programming. Preliminary work on the Chip and ECLiPSe systems has demonstrated that the techniques described in this paper can be easily applied to parallelization of the relevant phases of constraint handling. Considering that most constraint logic programming applications are extremely computation-intensive, the advantages of parallel execution are evident.

## 11. CONCLUSIONS AND FUTURE OF PARALLEL LOGIC PROGRAMMING

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

Parallel logic programming is a challenging 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, Yap, and Elipsys (or-parallelism), &-Prolog and &ACE (independent and-parallelism), DASWAM and ACE (dependent and-parallelism), and many efficient implementations of committed choice languages [Shapiro 1987; Hirata et al. 1992]. There are a fewer systems that efficiently exploit more than one source of parallelism, such as Andorra-I [Santos Costa, Warren, Yang 1991a], and others that are being built [Gupta, Pontelli, Hermenegildo, Santos Costa 1994; Correia et al. 1997; Santos Costa 1999]. However, there are none that exploit *all* sources of parallelism present in logic programs. Efforts are already under way to remedy this [Montelius 1997; Santos Costa 1999; Gupta, Santos Costa, Pontelli 1994; Pontelli and Gupta 1997b; Correia et al. 1997; Castro et al. 1998], and we believe that that is where the bulk of the parallel logic programming research in the future will lie.

The orthogonality principle [Correia et al. 1997] and the duality principle [Pontelli and Gupta 1995a] dictate that the ideal parallel logic programming should be a "plug-and-play" system, where a basic Prolog kernel engine can be incrementally extended with different modules implementing different parallelization strategies, scheduling strategies, etc., depending on the needs of the user (Figure 32). We hope that with enough research effort this ideal will be achieved.

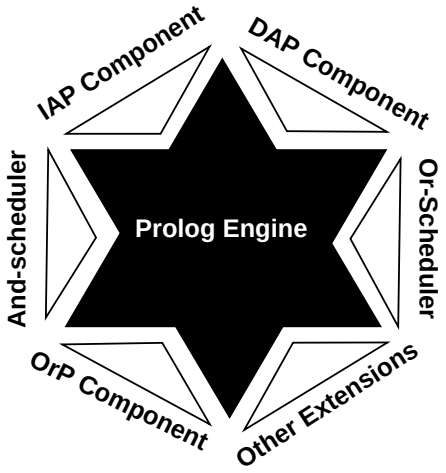



Fig. 32. Plug-and-Play Parallel Prolog Systems

Further Research is still 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 reduce the overhead at run-time by relegating many of the operations to compile-time; and building tools for visualizing parallel execution. It should be noted that while most of these problems arise in any parallel system, in the case of parallel logic programming systems they are harder to solve due to the presence of non-determinism and backtracking.

The current evolutionary trend in the design of parallel computer systems is towards building heterogeneous architectures that consist of a large number of relatively small-sized shared memory machines connected through fast interconnection networks. Taking full advantage of the computational power of such architectures is known to be a very difficult problem [Bader and JaJa 1997]. Parallel Logic programming systems constitute

a viable solution to this problem; however, considerable research on design and implementation of parallel logic programming systems on distributed memory multiprocessors is needed before parallel logic programming can be indeed regarded as a viable solution. Distributed implementation of parallel logic programming systems is one direction where we feel future research effort should be invested.

Finally, technology developed for parallel execution of Prolog programs has progressively expanded and found application in the parallelization of other logic-based paradigms and/or in the parallelization of alternative strategies for execution of Prolog programs. This includes:

- combination of parallelism and *tabled* execution of Prolog programs [Guo and Gupta 2000; Guo 2000; Freire, Hu, Swift, Warren 1995; Rocha, Silva, Santos Costa 1999], which opens the doors to parallelization of applications in a number of interesting application areas, such as model checking and database cleaning.
- parallelization of models computation in the context of *non-monotonic reasoning* [Pontelli and El-Khatib 2001; Finkel, Marek, Moore, Truszcynski 2001].
- use of parallelism in the execution of *inductive logic programs* [Page 2000; Ohwada, Nishiyama, Mizoguchi 2000].


## Acknowledgments

Thanks are due to Bharat Jayaraman for helping with an earlier article on which this article is based. Thanks to Manuel Carro who read drafts of this paper. Our deepest thanks to the anonymous referees whose comments tremendously improved the paper. Gopal Gupta and Enrico Pontelli are partially supported by NSF Grants CCR 98-75279, CCR 98-20852, CCR 99-00320, CDA 97-29848, EIA 98-10732, CCR 96-25358, HRD 99-06130, and by a grant from the US-Spain Research Commission.


## REFERENCES


Aikawa, S., et al. 1992. ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems. In ICOT Staff (eds.), *International Conference on Fifth Generation Computer Systems,* Tokyo, pages 286–293.

Aït-Kaci, H. 1992. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press. `www.isg.sfu.ca/~hak/documents/wam.html`.

Aït-Kaci, H. 1993. An introduction to LIFE: Programming with Logic, Inheritance, Functions, and Equations. In D. Miller (ed.) *International Logic Programming Symposium*, MIT Press, pages 52–68.

Ali, K.A.M. 1988. Or-parallel execution of Prolog on BC-machine. In R. Kowalski and K. Bowen (eds.) *Proceedings of 5th International Conference and Symposium on Logic Prog. Seattle,* MIT Press, pp. 1531–1545.

Ali, K.A.M. 1995. A Parallel Copying Garbage Collection Scheme for Shared-memory Multiprocessors. In E. Tick and T. Chikayama (eds.) *Proceedigns of the ICOT/NSF Workshop on Parallel Logic Programming and its Programming Environments*, Technical Report CSI-TR-94-04, University of Oregon, pp. 93–96.

Ali, K.A.M. and Karlsson, R. 1990a. The MUSE Approach to Or-Parallel Prolog. In *International Journal of Parallel Programming*, 19(2): 129–162.

Ali, K.A.M. and Karlsson, R., 1990b. Full Prolog and Scheduling Or-Parallelism in MUSE. In *International Journal of Parallel Programming*, 19(6): 445–475.

Ali, K.A.M. and Karlsson, R. 1992a. OR-parallel Speedups in a Knowledge Based System: on MUSE and Aurora. In ICOT Staff (eds.) *the Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 739–745.

Ali, K.A.M. and Karlsson, R., 1992b. Scheduling Speculative Work in MUSE and Performance Results. In *International Journal of Parallel Programming*, 21(6).

Ali, K.A.M., Karlsson, R., and Mudambi, S. 1992. Performance of MUSE on Switch-Based Multiprocessor Machines. In *New Generation Computing*, 11(1,4): 81–103.

Aparício, N.J., Cunha, J., Monteiro, L., and Pereira, L.M., 1986. Delta-Prolog: a Distributed Backtracking Extension with Events. In E. Shapiro (ed.) *Proc. 3rd International Conference on Logic Programming*, Lecture Notes in Computer Science 225, Springer Verlag, pages 225–240.

Appleby, K., Carlsson, M., Haridi, S., and Sahlin, D. 1988. Garbage Collection for Prolog based on WAM. In *Communications of the ACM*, 31(6):719–741.

Araujo, L. and Ruz, J., 1998. A Parallel Prolog System for Distributed Memory. In *Journal of Logic Programming*, Vol. 33, No. 1, pages 49–79.

Bader, D.A. and JaJa, J., 1997. SIMPLE: a Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors. Technical Report, University of Maryland.

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

Bansal, A.K. and Potter, J. 1992. An Associative Model for Minimizing Matching and Backtracking Overhead in Logic Programs with Large Knowledge Bases. In *Engineering Applications of Artificial Intelligence*, Volume 5, Number 3, pp. 247-262.

Barklund, J., 1990. *Parallel Unification.* Ph.D. Thesis, Uppsala Theses in Computing Science 9, Uppsala University.

Barklund, J. and Millroth, H., 1992. Providing Iteration and Concurrency in Logic Program through Bounded Quantifications. In ICOT Staff (eds.) *Proc. International Conf. on Fifth Generation Computer Systems*, pages 817–824.

Baron, U., Chassin de Kerommeaux, J., Hailperin, M., Ratcliffe, M., Robert, P., Syre, J-C., and Westphal, H., 1988. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation of Results. In ICOT Staff (eds.) *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, pages 841–850.

Beaumont, A.J. 1991. Scheduling Strategies and Speculative Work. In *Parallel Execution of Logic Programs*, Beaumont & Gupta (Eds), Springer Verlag, Lecture Notes in Computer Science 569, pages 120–131.

Beaumont, A.J., Muthu Raman, S., Szeredi, P., and Warren, D.H.D., 1991. Flexible scheduling or-parallelism in Aurora: the Bristol scheduler. In E. Aarts, J. van Leeuwen, M. Rem (eds.) *PARLE 91, Conference on Parallel Architectures and Languages Europe*, Springer-Verlag, Lecture Notes in Computer Science 506, pages 421–438.

Beaumont, A.J. and Warren, D.H.D., 1993. Scheduling speculative work in or-parallel Prolog systems. In D.S. Warren (ed.) *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, pages 135–149.

Bekkers, Y., Ridoux, O., and Ungaro, L. 1992. Dynamic Memory Management for Sequential Logic Programming Languages. In Y. Bekkers, J. Cohen (eds.) *Proceedings of the International Workshop on Memory Management*, Springer Verlag, pp. 82–102.

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

Benjumea, V. and Troya, J.M. 1993. An OR Parallel Prolog Model for Distributed Memory Systems. In M. Bruynooghe and J. Penjam (eds.) *Symposium on Programming Languages Implementation and Logic Programming*, Springer Verlag, pages 291–301.

Bevemyr, J. 1995. A Generational Parallel Copying Garbage Collection for Shared Memory Prolog. In *Workshop on Parallel Logic Programming Systems*, Portland, Oregon, 1995.

Bevemyr, J., Lindgren, T., and Millroth, H. 1993. Reform Prolog: The Language and its Implementation. In D.S. Warren (ed.) *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, pages 283–298.

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

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

Brand, P. 1988. Wavefront scheduling. Internal Report, Gigalips Project.

Briat, J., Favre, M., Geyer, C., and Chassin de Kergommeaux, J. 1992. OPERA: Or-Parallel Prolog System on Supernode. In Kacsuk and Wise eds., *Implementations of Distributed Prolog*, J. Wiley & Sons, pp. 45–64.

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

Bueno, F., García de la Banda, M., and Hermenegildo, M.V. 1999. Effectiveness of Abstract Interpretation in Automatica Parallelization: a Case Study in Logic Programming. In *Transaction on Programming Languages and Systems*, ACM, 21(2):189–239.

Bueno, F. and Hermenegildo, M.V. 1992. An Automatic Translation Scheme from Prolog to the Andorra Kernel Language. In ICOT Staff (eds.) *Proc. International Conference on Fifth Generation Computer Systems*, pages 759–769.

Butler, R., Lusk, E., McCune, W., and Overbeek, R. 1986. Paralle Logic Programming for Numerical Applications. In E. Shapiro (ed.) *Proceedings of the Third International Conference on Logic Programming*, Springer Verlag, pp. 357–388.

Butler, R., Disz, T., Lusk, E., Olson, R., Overbeek, R., and Stevens, R. 1988. Scheduling OR-parallelism: an Argonne perspective. In R. Kowalski and K. Bowen (eds.) *Proceedings of the Fifth International Conference on Logic Programming*, MIT Press, pages 1590–1605.

Cabeza, D. and Hermenegildo, M.V. 1994. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In B. Le Charlier (ed.) *International Static Analysis Symposium*, Springer Verlag, pages 297–313.

Calderwood, A. and Szeredi, P. 1989. Scheduling or-parallelism in Aurora—the Manchester scheduler. In G. Levi and M. Martelli (eds.) *Proceedings of the Sixth International Conference on Logic Programming*, MIT Press, pages 419–435.

Calegario, V.M. and Dutra, I. 1999. In P. Amestoy et al. (eds.) *Proceedings of EuroPar*, Springer Verlag, pp. 1484–1491.

Carlsson, M. 1989. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In G. Levi and M. Martelli (eds.) *International Conference on Logic Programming*, MIT Press, pages 3–16.

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

Carlsson, M. et al. 1995. SICStus Prolog User's Manual. Technical Report, Swedish Institute of Computer Science, ISBN 91-630-3648-7.

Carro, M., Gómez, L., and Hermenegildo, M.V. 1993. Some event-driven paradigms for the visualization of logic programs. In D.S. Warren (ed.) *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, pages 184–200.

Carro, M. and Hermenegildo, M.V. 1999. Concurrency in Prolog Using Threads and a Shared Database. In D. De Schreye (ed.) *International Conference on Logic Programming*, MIT Press, pages 320–334.

Castro, L.F., Santos Costa, V., Geyer, C.F.R., Silva, F., Vargas, P.K., and Correia, M.E. 1999. DAOS: Scalable And-Or Parallelism. In D. Pritchard and J. Reeve (eds.) *Proceedings of EuroPar*, Springer Verlag, pp. 899–908.

Chang, S-E. and Chiang, Y.P. 1989. Restrict And-Parallelism Model with Side Effects. In E. Lusk and R. Overbeek (eds.) *Proceedings of North American Conference on Logic Programming*, MIT Press, pages 350–368.

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

Chassin de Kergommeaux, J. 1989. Measures of the PEPSys Implementation on the MX500. Technical Report, CA-44, ECRC.

Chassin de Kergommeaux, J. and Codognet, P. 1994. Parallel Logic Programming Systems. In *ACM Computing Surveys*, 26(3):295–336.

Chassin de Kergommeaux, J. and Robert, P. 1990. An Abstract Machine to Implement Or-And Parallel Prolog Efficiently. In *Journal of Logic Programming*, 8(3):249–264.

Chikayama, T., Fujise, T., and Sekit, D. 1994. A Portable and Efficient Implementation of KL1. In M. Hermenegildo and J. Penjam (eds.) *Proceedings of the Sixth International Symposium on Programming Languages Implementation and Logic Programming*, Springer Verlag, pp. 25–39.

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

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

Ciepielewski, A. and Hausman, B. 1986. Performance Evaluation of a Storage Model for OR-parallel Execution of Logic Programs. In *Symposium on Logic Prog.*, IEEE Computer Society, pages 246–257.

Ciepielewski, A. and Haridi, S. 1983. A Formal Model for Or-parallel Execution of Logic Programs. In R. Mason (ed.) *IFIP 83, North Holland, P.C. Mason (ed.)*, pages 299–305.

Clark, K. and Gregory, S. 1986. Parlog: Parallel Programming in Logic. In *Transaction on Programming Languages and Systems*, ACM, Vol. 8, No. 1, pages 1–49.

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

Codish, M., Mulkers, A., Bruynooghe, M., García de la Banda, M.J., and Hermenegildo, M.V. 1995. Improving Abstract Interpretations by Combining Domains. In *Transactions on Programming Languages and Systems*, ACM, 17(1):28–44.

Codognet, C. and Codognet, P. 1989. Non-Deterministic Stream And-Parallelism based on Intelligent Backtracking. In G. Levi and M. Martelli (eds.) *Proceedings of the International Conference on Logic Programming*, MIT Press, pp. 63–79.

Codognet, C., Codognet, P., and Filé, G. 1988. Yet Another Intelligent Backtracking Method. In R. Kowalski and K. Bowen (eds.) *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, MIT Press, pp. 447-465.

Conery, J.S. 1987a. *Parallel Interpretation of Logic Programs.* Kluwer Academic Press.

Conery, J.S. 1992. The OPAL Machine. In Kacsuk and Wise (eds.) *Implementations of Distributed Prolog*, J. Wiley & Sons.

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

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

Conery, J.S. and Kibler, D.F. 1983. And Parallelism in Logic Programs. In A. Bundy (ed.) *Proceedings of the International Joint Conference in AI*, William Kaufmann, pages 539–543.

Correia, E., Silva, F., and Santos Costa, V. 1997. The SBA: Exploiting Orthogonality in And-Or Parallel System. In J. Maluszynski (ed.) *International Logic Programming Symposium*, MIT Press, pages 117–131.

Cousot, P. and Cousot, R. 1977. 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.

Cousot, P. and Cousot, R. 1992. Abstract Interpretation and Application to Logic Programs. In *Journal of Logic Programming*, 13(1/4):103–179.

Cox, P.T. 1984. Finding Backtrack Points for Intelligent Backtracking. In J. Campbell (ed.) *Implementations of Prolog*, Ellis Horwood/Halsted Press/Wiley.

Crabtree, B. 1991. A Clustering System to Network Control. Technical Report, British Telecom.

Crammond, J. 1985. A Comparative Stufy of Unification Algorithms for Or-Parallel Execution of Logic Languages. In *IEEE Transaction on Computers*, 34(10):911–971.

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

de Bosschere, K. and Tarau, P. 1996. Blackboard-based Extensions in Prolog. In *Software Practice and Experience*, 26(1):49–69.

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

Debray, S.K., Lin, N-W., and Hermenegildo, M.V. 1990. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*. ACM Press, pages 174–188.

Debray, S.K., López-García, P., and Hermenegildo, M.V. 1997a. Non-failure Analysis of Logic Programs. In L. Naish (ed.) *Proc. International Conference on Logic Programming*, MIT Press, pages 48–62.

Debray, S.K., López-García, P., Hermenegildo, M.V., and Lin, N-W. 1997b. Lower Bound Cost Estimation for Logic Programs. In J. Maluszynski (ed.) *Proc. of International Logic Programming Symposium*, MIT Press, pages 291–305.

DeGroot, D. 1984. Restricted AND-parallelism. In ICOT Staff (eds.) *International Conference on Fifth Generation Computer Systems*, Ohmsha Ltd., pages 471–478.

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

Delgado-Rannauro, S.A. 1992a. Or-Parallel Logic Computational Models. In P. Kacsuk and M. Wise (eds.) *Implementations of Distributed Prolog*, J. Wiley & Sons, pp. 3–26.

Delgado-Rannauro, S.A. 1992b. Restricted And- and And/Or-Parallel Logic Computational Models. In P. Kacsuk and M. Wise (eds.) *Implementations of Distributed Prolog*, J. Wiley & Sons, pp. 121–141.

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

Disz, T., Lusk, E., and Overbeek, R. 1987. Experiments with OR-parallel Logic Programs. In *International Symposium in Logic Programming*, IEEE Computer Society, San Francisco, pages 46–53.

Dorochevsky, M. and Xu, J. 1991. Parallel Execution Tracer. Internal Report, ECRC.

Drakos, N. 1989. Unrestricted And-Parallel Execution of Logic Programs with Dependency Directed Backtracking. In N. Sridharan (ed.) *International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pages 157–162.

Dutra, I. 1991. A Flexible Scheduler for the Andorra-I System. In G. Gupta and A. Beaumont (eds.) *ICLP'91 Workshop on Parallel Execution of Logic Programs*. Springer Verlag, LNCS 569. pages 70–82.

Dutra, I. 1994. Strategies for Scheduling And- and Or-Parallel Work in Parallel Logic Programming Systems. In M. Bruynooghe (ed.) *International Logic Programming Symposium*, MIT Press, pp. 289-304.

Dutra, I. 1995. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System*. Ph.D. Dissertation, University of Bristol, 1995.

Dutra, I. 1996. Distributing And-Work and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, IEEE Computer Society, pp. 646–655.

Dutra, I., Santos Costa, V., and Bianchini, R. 2000. The Impact of Cache Coherence Protocols on Parallel Logic Programming Systems. In J. Lloyd et al. (eds.) *Proceedings of the International Conference on Computational Logic*, Springer Verlag, pp. 1285–1299.

Fernández, M.J., Carro, M., and Hermenegildo, M.V. 1996. IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In L. Bouge et al. (eds.) *Proceedings of Euro-Par*, Springer Verlag, pp. 724–733.

Finkel, R., Marek, V., Moore, N., Truszcynski, M. 2001. Computing Stable Models in Parallel. In A. Provetti and S. Tran (eds.) *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, AAAI.

Fonseca, N., Santos Costa, V., and Dutra, I. 1998. VisAll: a Universal Tool to Visualize the Parallel Execution of Logic Programs. In J. Jaffar (ed.) *Proc. Joint International Conference and Symposium on Logic Programming*, MIT Press, pages 100–114.

Freire, J., Hu, R., Swift, T., and Warren, D.S. 1995. Exploiting Parallelism in Tabled Evaluations. In M. Hermenegildo and S. Swierstra (eds.) *Proceedings of the Symposium on Programming Languages Implementations and Logic Programming*, Springer Verlag, pp. 115–132.

Futó, I. 1993. Prolog with Communicating Processes: From T-Prolog to CSR-Prolog. In D.S. Warren (ed.) *Proceedings of the Tenth International Conference on Logic Programming*, MIT Press, pages 3–17.

Ganguly, S., Silberschatz, A., and Tsur, S. 1990. A Framework for the Parallel Processing of Datalog Queries. In H. Garcia-Molina and H. Jagadish (eds.) *Proceedings of the SIGMOD Conference on Management of Data*, ACM, pages 143–152.

García de la Banda, M.J., Hermenegildo, M.V., and Marriott, K. 1996. Independence in Dynamically Scheduled Logic Languages. In M. Hanus and M. Rodriguez-Artalejo (eds.) *Proceedings of the International Conference on Algebraic and Logic Programming*, Springer Verlag, pp. 47–61.

Giacobazzi, R. and Ricci, L. 1990. Pipeline Optimizations in And-parallelism by Abstract Interpretation. In D.H.D. Warren and P. Szeredi (eds.) International Conference on Logic Programming, MIT Press, pages 291–305.

Gottlieb, G. and Almasi, G. 1994. *Highly Parallel Computing. 2nd Edition.* Benjamin Cummings Publishing Company.

Gregory, S. and Yang, R. 1992. Parallel Constraint Solving in Andorra-I. In ICOT Staff (eds.) *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, pp. 843–850.

Guo, H-F. 2000. High Performance Logic Programming. Ph.D. Thesis, New Mexico State University.

Guo, H-F. and Gupta, G. 2000. A simple scheme for implementing tabled LP systems based on dynamic reordering of alternatives. In *Proceedings of the Workshop on Tabling in Parsing and Deduction*, Vigo, Spain.

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

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

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

Gupta, G. and Jayaraman, B. 1993a. And-Or Parallelism on Shared Memory Multiprocessors. In *Journal of Logic Programming*, 17(1):59–89.

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

Gupta, G. and Pontelli, E. 1997. Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems. In *International Parallel Processing Symposium*, IEEE Computer Society.

Gupta, G. and Pontelli, E. 1999a. Stack-splitting: A Simple Technique for Implementing Or-Parallelism and And-Parallelism on Distributed Machines. In D. De Schreye (ed.) *Proc. International Conference on Logic Programming*, MIT Press, pages 290–304.

Gupta, G. and Pontelli, E. 1999b. Last Alternative Optimization for Or-parallel Logic Programming Systems. In *Parallelism and Implementation Technology for Constraint Logic Programming*, Nova Science Eds., pages 107–132.

Gupta, G. and Pontelli, E. 1999c. Extended Dynamic Dependent And-Parallelism in ACE. In *Journal of Functional and Logic Programming*, MIT Press, Vol. 99, Special Issue #1.

Gupta, G., Pontelli, E., Hermenegildo, M.V, and Santos Costa, V. 1994. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In P. van Hentenryck (ed.) *Proceedings of the International Conference on Logic Programming*, MIT Press, pages 93–109.

Gupta, G. and Santos Costa, V. 1992b. 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.

Gupta, G. and Santos Costa, V. 1992c. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *Proceedings of Parallel Architectures and Languages Europe (PARLE)*, Springer Verlag, pp. 617–632.

Gupta, G. and Santos Costa, V. 1992a. Cut and Side-Effects in And-Or Parallel Prolog. In Journal of Logic Programming, 27(1):45–71.

Gupta, G., Santos Costa, V., and Pontelli, E. 1994. Shared Paged Binding Arrays: A Universal Data-structure for Parallel Logic Programming. In E. Tick (ed.) *Proc. NSF/ICOT Workshop on Parallel Logic Programming and its Environments*, University of Oregon.

Gupta, G., Santos-Costa, V., Yang, R., and Hermenegildo, M.V. 1991. IDIOM: Integrating Dependent and-, Independent and-, and Or-parallelism. In V. Saraswat and K. Ueda (eds.) *International Logic Programming Symposium*, MIT Press, pages 152–166.

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

Halstead, R.H. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the Symposium on LISP and Functional Programming*, ACM, pp. 9–17.

Haridi, S. 1990. A Logic Programming Language Based on the Andorra Model. In *New Generation Computing*, Vol. 7, No. 2-3, pages 109–125.

Haridi, S. and Janson, S. 1990. Kernel Andorra Prolog and its Computation Model. In D.H.D. Warren and P. Szeredi (eds.) *Proceedings of International Conference on Logic Prog.*, MIT Press, pages 31–46.

Haridi, S., Van Roy, P., Brand, P., and Schulte, C. 1998. Programming Languages for Distributed Applications. In *New Generation Computing*, 16(3):223–261.

Hasenberger, J. 1995. Modelling and Redesign the Advanced Traffic Management System in Andorra-I. In V. Santos Costa (ed.) *Proceedings of the Workshop on Parallel Logic Programming Systems*, Portland, Oregon.

Hausman, B. 1989. Pruning and scheduling speculative work in or-parallel Prolog. In E. Odijk, M. Rem, J-C. Syre (eds.) *PARLE, Conference on Parallel Architectures and Languages Europe.* Springer-Verlag, pp. 133–150.

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

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

Hausman, B., Ciepielewski, A., and Haridi, S. 1987. Or-Parallel Prolog Made Efficient on Shared Memory Multiprocessors. In *Proceedings of the International Symposium on Logic Programming*, IEEE Computer Society, San Francisco, CA, pages 69–79.

Herold, A. 1995 The Handbook of Parallel Constraint Logic Programming Applications. Technical Report, ECRC.

Hermenegildo, M.V. 1986a. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In E. Shapiro (ed.) *Third International Conference on Logic Programming*, Lecture Notes in Computer Science 225, Springer-Verlag, pages 25–40.

Hermenegildo, M.V. 1986b. *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.

Hermenegildo, M.V. 1987. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In J-L. Lassez (ed.) *Fourth International Conference on Logic Programming*, MIT Press, pages 556–575.

Hermenegildo, M.V. 1994. Towards Efficient Parallel Implementation of Concurrent Constraint Logic Programming. In E. Tick (ed.) *Proc. ICOT/NSF Workshop on Parallel Logic Programming and its Programming Environments*, University of Oregon.

Hermenegildo, M.V. 1994. Some Methodological Issues in the Design of CIAO, a Generic, Parallel Concurrent Constraint Logic Programming System. In A. Borning (ed.) *Proceedings of the Conference on Principles and Practices of Constraint Programming*, Springer Verlag, pp. 123-133.

Hermenegildo, M.V., Cabeza, D., and Carro, M. 1995. Using Attributed Variables in the Implementation of Parallel and Concurrent Logic Programming Systems. In L. Sterling (ed.) *International Conference on Logic Programming*, MIT Press, pages 631–645.

Hermenegildo, M.V. and Carro, M. 1995. Relating Data Parallelism and And-Parallelism in Logic Programs. In S. Haridi and P. Magnusson (eds.) *Proceedings of EuroPar*, Springer Verlag, pp. 27–41.

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

Hermenegildo, M.V. and López-García, P. 1995. Efficient Term Size Computation for Granularity Control. In L. Sterling (ed.) *Proceedings of the International Conference on Logic Programming*, MIT Press, pp. 647–661.

Hermenegildo, M.V. and Nasr, R.I. 1986. Efficient Implementation of backtracking in AND-parallelism. In E. Shapiro (ed.) *3rd International Conference on Logic Programming*, Lecture Notes in Computer Science 225, Springer Verlag, London, pages 40–54.

Hermenegildo, M.V. and Tick, E. 1989. Memory Performance of AND-Parallel Prolog on Shared-Memory Architectures. In *New Generation Computing*, Vol. 7, No. 1.

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

Herrarte, V. and Lusk, E. 1991. Studying parallel program behavior with UPSHOT. Technical Report ANL-91/15, Argonne National Laboratory.

Hickey, T. and Mudambi, S. 1989. Global Compilation of Prolog. In *Journal of Logic Programming*, 7(3):193–230.

Hirata, K. et al. 1992. Parallel and Distributed Implementation of Logic Programming Language KL1. In ICOT Staff (eds.) *International Conference on Fifth Generation Computer Systems*, Ohmsha Ltd., Tokyo, pages 436–459.

Le Huitouze, S. 1990. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Maluszynski (eds.) *Symposium on Programming Languages Implementation and Logic Programming*, Springer Verlag, pages 136–150.

IQSoft, 1992. CUBIQ - Development and Application of Logic Programming Tools for Knowledge Based Systems. `http://www.iqsoft.hu/projects/cubiq/cubiq.html`.

Janakiram, V., Agarwal, D., and Malhotra, R. 1988. A Randomized Parallel Backtracking Algorithm. In *IEEE Transactions on Computers,* 37(12), pages 1665–1676.

Jacobs, D. and Langen, A. 1992. Static Analysis of Logic Programs for Independent And-Parallelism. In *Journal of Logic Programming*, 13(1/4):291–314.

Janson, S. and Montelius, J. 1991. A Sequential Implementation of AKL. In T. Beaumont and G. Gupta (eds.) *Proceedings of ILPS'91 Workshop on Parallel Execution of Logic Programs.*

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

Kacsuk, P. and Wise, M. 1992. *Implementation of Distributed Prolog.* J. Wiley & Sons.

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

Kalé, L.V. 1991. The REDUCE OR Process MOdel for Parallel Execution of Logic Programming. In *Journal of Logic Programming*, 11(1):55–84.

Kalé, L.V., Padua, D.A., and Sehr, D.C. 1988. Or-Parallel Execution of Prolog with Side Effects. In *Journal of Supercomputing.*

Kalé, L.V. and Ramkumar, B. 1992. Machine Independent AND and OR Parallel Execution of Logic Programs: Part I — The Binding Environment. In *IEEE Transactions on Parallel Distributed Systems.* 5(2):170-192.

Kalé, L.V., Ramkumar, B., and Shu, W.W. 1988. A Memory Organization Independent Binding Environment for AND and OR Parallel Execution of Logic Programs. In R. Kowalski and K. Bowen (eds.) *Proceedings of the Fifth International Conference and Symposium on Logic Programs*, MIT Press, pp. 1223–1240.

Karlsson, R. 1992. *A High Performance OR-parallel Prolog System.* PhD thesis, The Royal Institute of Technology, Stockholm.

Kasif, S., Kohli, M., and Minker, J. 1983. PRISM: A Parallel Inference System for Problem Solving. In A. Bundy (ed.) *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 544-546.

King, A., Shen, K., and Benoy, F. 1997. Lower-bound Time-complexity Analysis of Logic Programs. In J. Maluszynski (ed.) *Proc. International Logic Programming Symposium*, MIT Press, pages 261–276.

Kluźniak, F. 1990. Developing applications for Aurora. Technical Report TR-90-17, University of Bristol, Computer Science Department.

Kowalski, R. 1979. *Logic for Problem Solving.* North Holland.

Masuzawa, H., Kumon, K., Itashiki, A., Satoh, K., and Sohma, Y. 1986. Kabu-Wake Parallel Inference Method and its Evaluation. In *Proceedings of the Fall Joint Computer Conference*, IEEE Computer Society, pp. 955–962.

Kusalik, A.J. and Prestwich, S. 1996. Visualizing Parallel Logic Program Execution for Performance Tuning. In M. Maher (ed.) *Proc. Joint International Conference and Symposium on Logic Programming*, MIT Press, pages 498–512.

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

Lin, Y-J. and Kumar, V. 1988. AND-parallel execution of Logic Programs on a Shared Memory Multiprocessor : A Summary of Results. In R. Kowalski and K. Bowen (eds.) *Fifth International Logic Programming Conference,* MIT Press, Seattle, WA, pages 1123–1141.

Lindgren, T. 1993. The Compilation and Execution of Recursion Parallel Logic Programs for Shared Memory Multiprocessors. Ph.D. Dissertation, Uppsala University.

Lindgren, T., Bevemyr, J., and Millroth, H. 1995. Compiler Optimizations in Reform Prolog: Eperiments on the KSR-1 Multiprocessor. In S. Haridi and P. Magnusson (eds.) *Proc. of EURO-PAR Conference*, Springer Verlag.

Lindstrom, G. 1984. Or-Parallelism on Applicative Architectures. In *International Logic Programming Conference*, Uppsala, Sweden, pages 159–170.

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

Lopes, R.S. and Santos Costa, V. 1999. The BEAM: Towards a First EAM Implementation. In *Parallelism and Implementation of Logic and Constraint Programming*, Nova Science.

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

Lusk, E., Mudambi, S., Overbeek, R., and Szeredi, P. 1993. Applications of the Aurora parallel Prolog system to computational molecular biology. In D. Miller (ed.) *International Logic Programming Symposium*, MIT Press, pages 353–369.

Lusk, E., Warren, D.H.D., Haridi, S., et al. 1990. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271.

McBryan, O. 1994. The KSR1 Computer. Tech. Report, University of Colorado, `http://wwwmcb.cs.colorado.edu/home/capp/ksr.html`.

Millroth, H. 1990. *Reforming Compilation of Logic Programs.* Ph.D. Dissertation, Uppsala Theses in Computing Science 10, Uppsala University.

Montelius, J. 1997. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages.* Ph.D. Dissertation, Uppsala Theses in Computing Science 28, Uppsala University.

Montelius, J. and Ali, K.M. 1996. A Parallel Implementation of AKL. In *New Generation Computing*, Vol. 14, No. 1, pages 31–52.

Montelius, J. and Haridi, S. 1997. An Evaluation of Penny: A System for Fine Grain Implicit Parallelism. In *International Symposium on Parallel Symbolic Computation*, ACM Press.

Moolenaar, R. and Demoen, B. 1993. A Parallel Implementation for AKL. In M. Bruynooghe and J. Penjam (eds.) *Proc. Conference on Programming Language Implementation and Logic Programming*, 5th International Symposium, Tallinn, Estonia, LNCS 714, Springer Verlag, pages 246–261.

Mudambi, S. 1991. Performance of Aurora on NUMA machines. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the International Logic Programming Symposium*, MIT Press, pages 793–806.

Muthukumar, K. and Hermenegildo, M.V. 1989a. Determination of Variable Dependence Information through Abstract Interpretation. In E. Lusk and R. Overbeek (eds.) *Proceedings of the North American Conference on Logic Programming*, MIT Press, pages 166–185.

Muthukumar, K. and Hermenegildo, M.V. 1989b. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In G. Levi and M. Martelli (eds.) *International Conference on Logic Programming.* MIT Press, pages 80–97.

Muthukumar, K. and Hermenegildo, M.V. 1990. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In D.H.D. Warren and P. Szeredi (eds.) *Proceedings of the International Conference on Logic Programming*, MIT Press, pages 221–237.

Muthukumar, K. and Hermenegildo, M.V. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In K. Furukawa (ed.) *Proceedings of the International Conference on Logic Programming*, MIT Press, pages 49–63.

Muthukumar, K. and Hermenegildo, M.V. 1992. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. In *Journal of Logic Programming*, 13(2-3):315–347.

Muthukumar, K., Bueno, F., García de la Banda, M.J., and Hermenegildo, M.V. 1999. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. In *Journal of Logic Programming*, 38(2):165–218.

Naish, L. 1988. Parallelizing NU-Prolog. In R. Kowalski and K. Bowen (eds.) *Proceedings of the International Conference on Logic Programming*, MIT Press, pp. 1546–1564.

Nguyen, T. and Deville, Y. 1998. A Distributed Arc-Consistency Algorithm. In *Science of Computer Programming*, 30:227–250.

Ohwada, H., Nishiyama, H., and Mizoguchi, F. 2000. Concurrent Execution of Optimal Hypothesis Search for Inverse Entailment. In J. Cussens and A. Frisch (eds.) *Proceedings of the Inductive Logic Programming Conference*, Springer Verlag, pp. 165–173.

Older, W.J. and Rummell, J.A. 1992. An Incremental Garbage Collector for WAM-Based Prolog. In K. Apt (ed.) *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, pp. 369–383.

Ozawa, T., Hosoi, A., and Hattori, A. 1990. Generation Type Gargabe Collection for Parallel Logic Languages. In S. Debray and M. Hermenegildo (eds.) *Proceedings of the North American Conference on Logic Programming*, MIT Press, pp.291–305.

Page, D. 2000. ILP: Just Do It. In J. Cussens and A. Frisch (eds.) *Proceedings of the Inductive Logic Programming Conference*, Springer Verlag, pp. 21–39.

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

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

Pittomvils, E., Bruynooghe, M., and Willems, Y.D. 1985. Towards a Real-Time Garbage Collector for Prolog. In *Proceedings of the Symposium on Logic Programming*, IEEE Computer Society, pp. 185–198.

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

Pontelli, E. 1997. *High-Performance Parallel Logic Programming.* Ph.D. Thesis, New Mexico State University.

Pontelli, E. 2000. Concurrent Web Programming in CLP(WEB). In *Hawaian International Conference of Computers and Systems Science*, IEEE Computer Society.

Pontelli, E. and El-Khatib, O. 2001. Experiments in Parallel Execution of Answer Set Programs. In A. Provetti and S. Tran (eds.) *AAAI Spring Symposium on Answer Set Programming*, AAAI.

Pontelli, E. and Gupta, G. 1995a. On the Duality Between And-parallelism and Or-parallelism. In S. Haridi and P. Magnusson (eds.) *Euro-Par*, Springer Verlag, pages 43–54.

Pontelli, E. and Gupta, G. 1995b. Data And-Parallel Logic Programming in &ACE. In *Proceedings of the Symposium on Parallel and Distributed Processing*, IEEE Computer Society.

Pontelli, E. and Gupta, G. 1997a. Implementation Mechanisms for Dependent And-Parallelism. In L. Naish (ed.) *International Conference on Logic Programming*, MIT Press, pages 123–137.

Pontelli, E. and Gupta, G. 1997b. Parallel Symbolic Computation with ACE. In *Annals of AI and Mathematics*, Volume 21, Number 2–4, pages 359–395.

Pontelli, E. and Gupta, G. 1998. Efficient Backtracking in And-Parallel Implementations of Non-deterministic Languages. In T. Lai (ed.) *International Conference on Parallel Processing*, IEEE Computer Society, pp. 338–345 .

Pontelli, E., Gupta, G., and Hermenegildo, M.V. 1995. A High-Performance Parallel Prolog Systems. In *International Parallel Processing Symposium*, IEEE Computer Society, pp. 564–571.

Pontelli, E., Gupta, G., Pulvirenti, F., and Ferro, A. 1997. Automatic Compile-time Parallelization of Prolog Programs for Dependent And-Parallelism. In L. Naish (ed.) *International Conference on Logic Programming*, MIT Press, pages 108–122.

Pontelli, E., Gupta, G., and Tang, D. 1995. Determinacy Driven Optimizations of Parallel Prolog. In L. Sterling (ed.) *International Conference on Logic Programming*, MIT Press, pages 615–629.

Pontelli, E., Gupta, G., Tang, D., Carro, M., and Hermenegildo, M.V. 1996. Improving the Efficiency of Nondeterministic Independent And-Parallel Systems. *Computer Languages*, Vol. 22, No. 2/3, pages 115–142.

Pontelli, E., Gupta, G., Wiebe, J., and Farwell, D. 1998. Natural Language Multiprocessing: a case study. In *15th National Conference on Artificial Intelligence*, AAAI, pages 76–82.

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

Popov, K. 1997. A Parallel Abstract Machine for the Thread-Based Concurrent Language Oz. In E. Pontelli and V. Santos Costa (eds.) *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, Port Jefferson, NY.

Ramkumar, B. and Kalé, L.V. 1989. Compiled Execution of the REDUCE-OR Process Model. In E. Lusk and R. Overbeek (eds.) *Proceedings of the North American Conference on Logic Programming*, MIT Press, pages 313–331.

Ramkumar, B. and Kalé, L.V. 1990. Joining And Parallel Solutions in And/Or Parallel Systems. In E. Lusk and R. Overbeek (eds.) *Proc. of N. American Conference on Logic Programming*, MIT Press, pages 624–641.

Ramkumar, B. and Kalé, L.V. 1992. Machine Independent AND and OR Parallel Execution of Logic Programs: Part I and II. In *IEEE Transactions on Parallel and Distributed Systems*, Volume 2, Number 5.

Ranjan, D., Pontelli, E., and Gupta, G. 1999. The Complexity of Or-Parallelism. In *New Generation Computing*, Vol. 17, No. 3, pp. 285-308.

Ranjan, D., Pontelli, E., and Gupta, G. 2000. Data Structures for Order-Sensitive Predicates in Parallel Nondeterministic Systems. In *Acta Informatica*, 37(1):21–43.

Ranjan, D., Pontelli, E., Gupta, G., and Longpre, L. 2000. The Temporal Precedence Problem. In *Algorithmica*, Vol. 28, pp. 288–306.

Ratcliffe, M. and Syre, J.C. 1987. A Parallel Logic Programming Language for PEPSys. In J. McDermott (ed.) *Proceedings of IJCAI*, Milan, pages 48–55.

Rocha, R., Silva, F., and Santos Costa, V. 1999. Or-Parallelism within Tabling. In G. Gupta (ed.) *Proceedings of the Workshop on Practical Aspects of Declarative Languages*, Springer Verlag, pp. 137–151.

Rokusawa, K., Nakase, A., and Chikayama, T. 1996. Distributed Memory Implementation of KLIC. In *New Generation Computing*, 14(3):261–280.

Ruiz-Andino, A., Araujo, L., Sáenz, F., and Ruz, J.J. 1999. Parallel Execution Models for Constraint Programming over Finite Domains. In G. Nadathur (ed.) *Proceedings of the Conference on Principles and Practice of Declarative Programming*, Springer Verlag, pp. 134–151.

Samal, A. and Henderson, T. 1987. In *International Journal of Parallel Programming*, 16(5):341–364.

Santos Costa, V. 1999 COWL: Copy-On-Write for Logic Programs. In *Proceedings of IPPS/SPDP*, IEEE Computer Society, pp. 720–727.

Santos Costa, V., Bianchini, R., and Dutra, I. 1997. Parallel Logic Programming Systems on Scalable Multiprocessors. In *Proceedings of the International Symposium on Parallel Symbolic Computation*, pp. 58–67.

Santos Costa, V., Bianchini, R., and Dutra, I. 2000. In *Journal of Parallel and Distributed Computing*, 60(7):835–852.

Santos Costa, V., Damas, L., Reis, R., and Azevedo, R. 1999. *YAP User's Manual.* `www.ncc.up.pt/~vsc/Yap`.

Santos Costa, V., Rocah, R., and Silva, F. 2000. Novel Models for Or-Parallel Logic Programs: A Performance Analysis. In A. Bode et al. (eds.) *Proceedings of Euro-Par*, Springer Verlag, pp. 744–753.

Santos Costa, V., Warren, D.H.D., and Yang, R. 1991a. 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, pp. 83–93.

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

Santos Costa, V., Warren, D.H.D., and Yang, R. 1991c. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In K. Furukawa (ed.) *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, pages 825–839.

Santos Costa, V., Warren, D.H.D., and Yang, R. 1996. Andorra-I Compilation. In *New Generation Computing*, 14(1):3–30.

Saraswat, V. 1989. *Concurrent Constraint Programming Languages.* Ph.D. Dissertation, Carnegie-Mellon University.

Schönhage, A. 1980. Storage Modification Machines. In *SIAM Journal of Computing*, Vol. 9, No. 3, pages 490–508.

Shapiro, E. 1987. *Concurrent Prolog : Collected Papers.* MIT Press.

Shapiro, E. 1989. The Family of Concurrent Logic Programming Languages. In *ACM Computing Surveys*, Vol 21, No. 3, pages 413–510.

Shen, K. 1992a. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In K. Apt (ed.) *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, pages 717–731.

Shen, K. 1992b. *Studies in And/Or Parallelism in Prolog.* Ph.D. Dissertation, University of Cambridge.

Shen, K. 1994. Improving the Execution of the Dependent And-parallel Prolog DDAS. In C. Halatsis et al. (eds.) *PARLE: Parallel Architectures and Languages Europe*, Springer Verlag, pages 438–452.

Shen, K. 1997. A New Implementation Scheme for Combining And/Or Parallelism. In E. Pontelli and Santos Costa (eds.) *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, Port Jefferson, NY.

Shen, K. and Hermenegildo, M.V. 1991. A Simulation Study of Or- and Independent And-parallelism. In V. Saraswat and K. Ueda (eds.) *Proceedings of the International Logic Programming Symposium.* MIT Press, pages 135–151.

Shen, K. and Hermenegildo, M.V. 1993. 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.

Shen, K., Santos Costa, V., and King, A. 1998. Distance: a New Metric for Controlling Granularity for Parallel Execution. In J. Jaffar (ed.) *Proc. Joint International Conference and Symposium on Logic Programming*, MIT Press, pages 85–99.

Silva, M.G., Dutra, I., Bianchini, R., and Santos Costa, V. 1999. The Influence of Computer Architectural Parameters on Parallel Logic Programming Systems. In G. Gupta (ed.) *Proceedings of the Workshop on Practical Aspects of Declarative Languages*, Springer Verlag, pp. 122–136.

Silva, F. and Watson, P. 2000. Or-Parallel Prolog on a Distributed Memory Architecture. *Journal of Logic Programming*, North-Holland, Vol. 43, No. 2, pp. 173–186.

Sindaha, R. 1992. 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.

Sindaha, R. 1993. Branch-Level Scheduling in Aurora: The Dharma Scheduler. In D. Miller (ed.) *Proceedings of International Logic Programming Symposium*, MIT Press, pp. 403–419.

Singhal, A. and Patt, Y. 1989. Unification Parallelism: How much can be Exploited? In E. Lusk and R. Overbeek (eds.) *Proceedings of North American Conference on Logic Programming.* MIT Press, pages 1135–1148.

Smith, D.A. 1996. MultiLog and Data Or-Parallelism. In *Journal of Logic Programming*, 29(1-3):195-244.

Smolka, G. 1995. The Oz Programming Model. In *Computer Science Today*, Springer Verlag, pages 324–343.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog.* MIT Press.

Szeredi, P. 1989. Performance analysis of the Aurora or-parallel Prolog system. In E. Lusk and R. Overbeek (eds.) *Proceedings of the North American Conference on Logic Programming*, MIT Press, pages 713–732.

Szeredi, P. 1991. Using dynamic predicates in an or-parallel Prolog system. In V. Saraswat and K. Ueda (eds.) *Proceedings of the International Logic Programming Symposium*, MIT Press, pages 355–371.

Szeredi, P., Carlsson, M., and Yang, R. 1991. Interfacing engines and schedulers in or-parallel Prolog systems. In E. Aarts et al. (eds.) *PARLE 91, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, Lecture Notes in Computer Science 506, pages 439–453.

Szeredi, P. and Farkas, Z. 1996. Handling Large Knowledge Bases in Parallel Prolog. In *Proceedings of the Workshop on High Performance Logic Programming Systems*, ESSLLI, Prague.

Szeredi, P., Molnár, K., and Scott, R. 1996. Serving Multiple HTML Clients from a Prolog Application. In *Proceedings of the JICSLP'96 Post-Conference Workshop on Logic Programming Tools for Internet Applications*, Bonn.

Takeuchi, A. 1992. *Parallel Logic Programming* 1992. Kluwer Academic Press.

Tarau, P. 1998. Inference and Computation Mobility with Jinni. Technical Report, University of North Texas.

Taylor, A. 1991. High-Performance Prolog Implementation. Ph.D. Thesis, University of Sydney.

Tebra, H. 1987. Optimistic And-Parallelism in Prolog. In J. de Bakker, A. Nijman, P. Treleaven (eds.) *Parallel Architectures and Languages Europe*, Springer Verlag.

Terasaki, S., Hawley, D.J., Sawada, H., Satoh, K., Menju, S., Kawagishi, T., Iwayama, N., and Aiba, A. 1992. Parallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers. In ICOT Staff (eds.) *Proceedings of the Conference on Fifth Generation Computing Systems*, ICOT, pp. 330–346.

Tick, E. 1987. *Memory Performance of Prolog Architectures.* Kluwer Academic Press.

Tick, E. 1991. *Parallel Logic Programming.* MIT Press.

Tick, E. 1992. Visualizing Parallel Logic Programming with VISTA. In ICOT Staff (eds.) *Proc. International Conference on Fifth Generation Computer Systems*, Ohmsha Ltd., Tokyo, pages 934–942.

Tick, E. 1995. The Deevolution of Concurrent Logic Programming Languages. In *Journal of Logic Programming*, 23(2):89–123.

Tick, E. and Zhong, X. 1993. A Compile-time Granularity Analysis Algorithm and its Performance Evaluation. In *New Generation Computing*, 11(3):271–295.

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

Tong, B. and Leung, H. 1993. Concurrent Constraint Logic Programming on Massively Parallel SIMD Computers. In D. Miller (ed.) *Proc. International Logic Programming Symposium*, MIT Press, pages 388–402.

Tong, B. and Leung, H. 1995. Performance of a Data Parallel Concurrent Constraint Programming System. In K. Kanchanasut and J-J. Levy (eds.) *Asian Computing Science Conference*, Springer Verlag, pages 319–334.

Traub, K.R. 1989. Compilation as Partitioning: a New Approach to Compiling Non-strict Functional Languages. In *Conf. on Functional Programming Languages and Computer Architecture*, ACM Press, pages 75–88.

Ueda, H. and Montelius, J. 1996. Dynamic Scheduling in an Implicit Parallel System. In *Proceedings of the ISCA 9th International Conference on Parallel and Distributed Computing Systems*, ISCA.

Ueda, K. 1986. *Guarded Horn Clauses.* Ph.D. Thesis, University of Tokyo.

Ueda, K. and Morita, M. 1993. Moded Flat GHC and its Message-oriented Implementation Technique. In *New Generation Computing*, Volume 11, Number 3/4, pages 323–341.

Ullman, J.D. 1989. *Principles of Database and Knowledge-base Systems.* Computer Science Press.

Van Hentenryck, P. 1989a. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In G. Levi and M. Martelli (eds.) *Proc. of the Sixth International Conference on Logic Programming*, MIT Press, pages 165–180.

Van Hentenryck, P. 1989b. *Constraint Satisfaction in Logic Programming.* MIT Press.

Van Hentenryck, P., Saraswat, V., and Deville, Y. 1998. Design, Implementation, and Evaluation of the Constraint Language cc(FD). In *Journal of Logic Programming*, 37(1-3), pages 139–164.

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

Van Roy, P. 1994. 1983-1993: The Wonder Years of Sequential Prolog Implementation. In *Journal of Logic Programming*, Vol. 19/20, pages 385–441.

Van Roy, P. and Despain, A.M. 1992. High-Performance Logic Programming with the Aquarius Prolog Compiler. In *IEEE Computer*, 25(1):54–68.

Vaupel, R., Pontelli, E., and Gupta, G. 1997. Visualization of And/Or-Parallel Execution of Logic Programs. In L. Naish (ed.) *Proc. International Conference on Logic Programming*, MIT press, pages 271–285.

Véron, A., Schuerman, K., Reeve, M., Li, L-L. 1993. Whya dn How in the ElipSys Or-Parallel CLP System. In A. Bode, M. Reeve, G. Wolf (eds.) *PARLE93: Conference on Parallel Architectures and Languages Europe*, Springer Verlag, pp. 291–303.

Villaverde, K., Guo, H-F., Pontelli, E., and Gupta, G. 2000. Incremental Stack Splitting. In I. Dutra (ed.) *Proceedings of the Workshop On Parallelism and Implementation Technology for Constraint Logic Programming*, London.

Wallace, M., Novello, S., and Schimpf, J. 1997. ECLiPSe: A Platform for Constraint Logic Programming. Technical Report, IC-Parc, Imperial College, London.

Warren, D.S. 1984. Efficient Prolog Memory Management for Flexible Control Strategies. In *Int. Symp. on Logic Programming*, IEEE Computer Society, Atlantic City, pages 198–202.

Warren, D.H.D. 1980. An Improved Prolog Implementation which Optimizes Tail Recursion. Technical Report 156, University of Edinburgh.

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

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

Warren, D.H.D. 1987b. Or-Parallel Execution Models of Prolog. In H. Ehrig et al. (eds.) *Proceedings of TAPSOFT '87*, Lecture Notes on Computer Science 250, Springer Verlag, pages 243–259.

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

Warren, D.H.D. and Haridi, S. 1988. The Data Diffusion Machine – a Shared Virtual Memory Architecture for Parallel Execution of Logic Programs. In ICOT Staff (eds.) *Proceedings of International Conference on Fifth Generation Computer Systems*, Springer Verlag, pages 943–952.

Weemeeuw, P. and Demoen, B. 1990. Memory compaction for shared memory multiprocessors, design and specification. In S. Debray and M. Hermenegildo (eds.) *Proceedings of the North American Conference on Logic Programming*. MIT Press, pages 306–320.

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

Winsborough, W. 1987. Semantically Transparent Reset for And Parallel Interpreters based on the Origin of Failure. In *Proceedings of the Fourth Symposium on Logic Programming*, IEEE Computer Society, pp. 134–152.

Wise, D.S. 1986. *Prolog Multiprocessors.* Prentice-Hall.

Wolfson, O. and Silberschatz, A. 1988. Distributed Processing of Logic Programs. In H. Boral and P. Larson (eds.) *Proceedings SIGMOD Conference on Management of Data*, ACM, pages 329–336.

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

Xu, L., Koike, H., and Tanaka, H. 1989. Distributed Garbage Collection for the Parallel Inference Engine PIE64. In E. Lusk and R. Overbeek (eds.) *Proceedings of the North American Conference on Logic Programming*, MIT Press, pp. 922–943.

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

Yang, R., Beaumont, T., Dutra, I., Santos Costa, V., and Warren, D.H.D. 1993. Performance of the Compiler-based Andorra-I System. In D.S. Warren (ed.) *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, pages 150–166.

Zhong, X., Tick, E., et al. 1992. Towards an Efficient Compile-time Granularity Algorithm. In ICOT Staff (eds.) *International Conference on Fifth Generation Computer Systems*, Ohmsha Ltd., Tokyo, pages 809–816.

Zima, H. and Chapman, B. 1991. *Supercompilers for Parallel and Vector Computers.* ACM Press.