Parallel Execution of Prolog Programs: A Survey

GOPAL GUPTA
University of Texas at Dallas
ENRICO PONTELLI
New Mexico State University
KHAYRI A.M. ALI and MATS CARLSSON
Swedish Institute of Computer Science
and
MANUEL V. HERMENEGILDO
Technical University of Madrid (UPM)

Since the early days of logic programming, researchers in the field realized the potential for exploitation of parallelism present in the execution of logic programs. Their high-level nature, the presence of nondeterminism, and their referential transparency, among other characteristics, make logic programs interesting candidates for obtaining speedups through parallel execution. At the same time, the fact that the typical applications of logic programming frequently involve irregular computations, make heavy use of dynamic data structures with logical variables, and involve search and speculation, makes the techniques used in the corresponding parallelizing compilers and run-time systems potentially interesting even outside the field. The objective of this article is to provide a comprehensive survey of the issues arising in parallel execution of logic programming languages along with the most relevant approaches explored to date in the field. Focus is mostly given to the challenges emerging from the parallel execution of Prolog programs. The article describes the major techniques used for shared memory implementation of Or-parallelism, And-parallelism, and combinations of the two. We also explore some related issues, such as memory management, compile-time analysis, and execution visualization.

The work of G. Gupta and E. Pontelli is partially supported by NSF Grants CCR 98-75279, CCR 98-20852, CCR 99-00320, CDA 97-29848, EIA 98-10732, CCR 96-25358, and HRD 99-06130.
M. Hermenegildo is partially funded by Spanish Ministry of Science and Technology Grant TIC99-1151 "EDIPIA" and EU RTD 25562 "Radio Web."
G. Gupta, E. Pontelli, and M. Hermenegildo are all partially funded by the US—Spain Research Commission McyT/Fulbright grant 98059 ECCOSIC.
Authors' addresses: G. Gupta, Department of Computer Science, University of Texas at Dallas, Box 830688/EC31, Richardson, TX 75083-0688, e-mail: gupta@utdallas.edu; E. Pontelli, Department of Computer Science, New Mexico State University, Box 30001/CS, Las Cruces, NM 88003, e-mail: epontell@cs.nmsu.edu; K. A. M. Ali and M. Carlsson, Swedish Institute of Computer Science, Box 1263, SE-164 29, Kista, Sweden, e-mail: {khayri, matsc}@sics.se; M. V. Hermenegildo, Facultad de Informática, Univsersidad Politécnica de Madrid, 28660-Boadilla del Monte, Madrid, Spain; e-mail: herme@fi.upm.es.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2001 ACM 0164-0925/01/0700–0472 $5.00

ACM Transactions on Programming Languages and Systems, Vol. 23, No. 4, July 2001, Pages 472–602.

Categories and Subject Descriptors: A.1 [**Introductory and Survey**]; D.3.4 [**Programming Languages**]: Processors; D.3.2 [**Programming Languages**]: Language Classification—*constraint and logic languages*

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Automatic parallelization, constraint programming, logic programming, parallelism, prolog

Dedicated to the memory of
Andrzej Ciepielewski

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 and excellent public-domain implementations are available that are being used to develop large real-life applications. An excellent survey of the sequential implementation technology that has been developed for Prolog is presented by 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 being Pollard's [1981] Ph.D. dissertation. Kowalski [1979] already mentions the possibility of executing logic programs in parallel in his seminal book *Logic for Problem Solving*. 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 proceedings and journals devoted to logic programming and parallel processing, and the number of advanced tutorials and workshops organized on this topic in various conferences.

This interest in parallel execution of logic programs arises from these 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, at least in principle, parallelism can be *exploited implicitly* from logic programs (i.e., parallelism can be extracted from logic programs automatically without any user intervention). Logic languages allow the programmer to express the desired algorithm in a way that reflects the structure of the problem more directly (i.e., staying closer to the specifications). This makes the parallelism available in the problem more accessible to the compiler and run-time system. The relatively clean semantics of these languages also makes it comparatively easy to use formal methods and prove the transformations performed by the parallelizing compiler or run-time system both correct (in terms of computed outputs) and efficient (in terms of computational cost).[1] At the same time, parallelizing

[1]Functional programming is another paradigm that also facilitates exploitation of parallelism.

logic programs implies having to deal with challenges such as highly irregular computations and dynamic control flow (due to the symbolic nature of many of their applications), the presence of dynamically allocated, complex data structures containing logical variables, and having to deal with speculation, all of which lead to nontrivial notions of independence and interesting scheduling and memory management solutions. However, the high-level nature of the paradigm also implies that the study of parallelization issues happens in a better-behaved environment. For example, logical variables are in fact a very "well-behaved" version of pointers.

(2) There is an everlasting myth that logic programming languages have low execution efficiency. While it is now clear that modern compilers for logic programs produce executables with very competitive time and memory performance, this early belief also prompted researchers to use parallelism as an alternative way of achieving speed. As we show, some of the results obtained fortunately combine well with sequential compilation techniques resulting in real speedups over even the most competitive sequential systems.

As mentioned, the literature on parallel execution of logic programs is vast and varied. There are two major (and nonindependent) 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 models 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 that 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.

Finally, a hybrid solution is used in some "implicit" systems, which arguably offers the advantages of both approaches: a user-accessible concurrent language exists (which is typically an extension of Prolog) and allows quite detailed manual parallelization. This language is also used by the parallelizing compiler in order to present to the programmer the transformations it performs on the program during automatic parallelization. This hybrid approach is exemplified by the &-Prolog system's [Hermenegildo and Greene 1991] "CGE" language and other systems that extend &-Prolog, such as ACE [Pontelli et al. 1995, 1996], DASWAM [Shen 1992a], and so on.

Approaches that *require* explicit specification of parallelism from logic programs can be largely classified into these categories:

(1) Those that add explicit message passing primitives to Prolog, for example, Delta Prolog [Pereira et al. 1986] and CS-Prolog [Futó 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, for example, Shared Prolog [Ciancarini 1990]. These primitives are used by multiple Prolog processes

running in parallel to communicate with each other via the common blackboard.

Some notable recent proposals in this category include:

(a) The Jinni system developed by Tarau [1998], a Java-based logic programming system including multithreading and blackboard-based communication; this work is a continuation of the previous work by De Bosschere and Tarau [1996].

(b) The Ciao system [Hermenegildo et al. 1999a; Bueno et al. 1997] supports multithreading and novel Prolog database operations which allows the programmer 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 et al. 1999].

(3) Those based on guards, committed choice, and dataflow synchronization, for example, 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 [Aït-Kaci 1993] and cc(fd) [Van Hentenryck et al. 1998]) and the class of distributed constraint languages such as Oz/Mozart [Haridi et al. 1998; Smolka 1996] and AKL [Haridi and Janson 1990], as well as some extensions of more traditional logic programming systems for distributed execution (e.g., the &-Prolog/Ciao system [Hermenegildo 1994; Cabeza and Hermenegildo 1996; Hermenegildo et al. 1999a] and ACE [Gupta and Pontelli 1999a]).

Each of the above approaches 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 this survey focused, we consider these approaches only marginally or in those cases where they introduce execution mechanisms that are applicable also in the case of implicit exploitation of parallelism (e.g., committed choice languages).

In the rest of this work we focus primarily on the parallel execution of Prolog programs, although occasional generalizations to logic languages with a different operational semantics are considered (e.g., we briefly discuss parallelization in constraint logic programming languages). This choice is dictated by the wider use of Prolog with respect to other logic languages, and a consequent wider applicability of the accomplished results. 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 extralogical predicates). Throughout this work we 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 article, although, interestingly, in some cases of a "dual" nature [Hermenegildo and CLIP Group 1994].

The objective of this article is to provide a uniform view of the research in parallel logic programming. Due to the extensive body of research in this field, we are not able to cover every single aspect and model that has been presented in the literature. Thus, our focus lies on highlighting the fundamental problems and the key solutions that have been proposed. This survey expands the work done by other researchers in the past in proposing an organized overview of parallel logic programming. In particular, this work expands the earlier survey on parallel logic programming systems by Chassin de Kergommeaux and Codognet [1994], by covering the research performed in the last eight years and by providing a more indepth analysis of various areas. Other surveys have also appeared in the literature, mostly covering more limited areas of parallel logic programming or providing a different focus [Santos Costa 2000; Gupta and Jayaraman 1993a; Kacsuk 1990; Takeuchi 1992; Delgado-Rannauro 1992a,b; Hermenegildo 2000].

The article 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 8 presents a brief overview of parallel constraint logic programming. 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 survey we assume the reader to be familiar 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 in the papers mentioned above.

2.1 Logic Programs and Prolog

A logic program is composed of 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 (*atoms*) and $n \geq 0$.[2] Each clause represents a logical implication of the form:

$$\forall v_i (B_1 \wedge \cdots \wedge B_n \rightarrow Head),$$

where v_i are all the variables that appear in the clause. A separate type of clause is where *Head* is the atom `false`, which is simply written as

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

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

Each atomic formula is composed of a predicate applied to a number of arguments (terms), and this is denoted $p(t_1, \ldots, t_n)$–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 in logic programming typically involves a logic program P and a goal $:- G_1, \ldots, G_n$, and the objective is to verify whether there exists an assignment σ of terms to the variables in the goal such that $(G_1 \wedge \cdots \wedge G_n)\sigma$ is a logical consequence of P.[3] σ is called a *substitution*: a substitution is an assignment 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 σ is called *binding application*.

Prolog, as well as many other logic programming systems, makes use of *SLD-resolution* to carry out the execution of a program. The theoretical view of the execution of a program P with respect to a goal G is a series of transformations of a *resolvent* using a sequence of *resolution steps*.[4] 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 renamed clause (i.e., with "fresh variables")

$$Head :- B_1, \ldots, B_h$$

whose head successfully unifies with A_i. Unification is the process that determines the existence of a substitution σ such that $Head\sigma = A_i\sigma$. If there

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

[3] Following standard practice, the notation $e\sigma$ denotes the application of the substitution σ to the expression e–that is, each variable X in e is replaced by $\sigma(X)$.

[4] In fact, the actual execution, as we show later, is very similar to that of standard procedural languages, involving a sequence of procedure calls, returns, etc., and stack-based memory management.

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 cause *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–that is, 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 $\texttt{select}_{\texttt{clause}}$). 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 that successfully unifies with the selected subgoal.

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

(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 metaprogramming 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 extralogical 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 focus on execution of Prolog programs (unless explicitly stated otherwise); this means that we assume that programs are executed according to the computation and selection rule of Prolog. We also frequently use the term *observable semantics* to indicate the overall observable behavior of an execution, that is, 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.

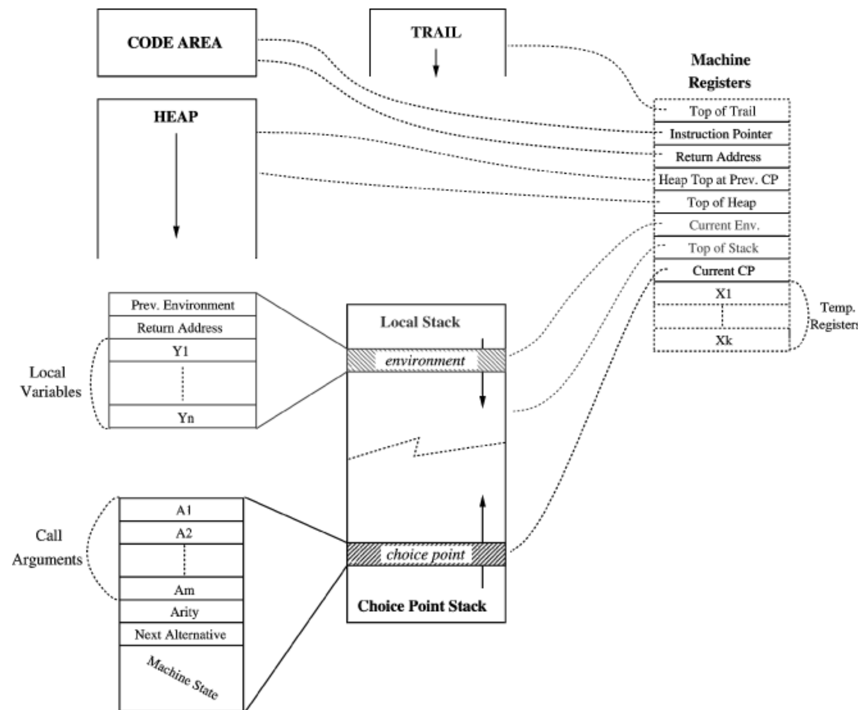


Fig. 1. Organization of the WAM.

2.2 The Warren Abstract Machine

The *Warren Abstract Machine (WAM)* [Warren 1983; Aït-Kaci 1991] has become a de facto standard for 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; and
(2) be sufficiently lowlevel to allow an efficient emulation and/or translation to native machine code.

Most (sequential and parallel) 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 language implementation schemes (e.g., use of `call/return` instructions, use of frames for maintaining a procedure's local environment), but extended in order to support the features peculiar to logic programming, namely, *unification* and *backtracking* (and some other variations, such as the need to support dynamic type checking). 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.

(1) *Internal State:* It is described by the content of the machine registers. The purpose of most of the registers is described in Figure 1.

(2) *External State:* It is described by the content of the logical data areas of the machine:
   (a) *Heap:* Data areas in which complex data structures (lists and Prolog's compound terms) are allocated.
   (b) *Local Stack:* (also known as *Control Stack*). 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 the 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 clause.
   (c) *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 matches 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.
   (d) *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 affected by this operation are registered in the trail stack. Each choice point records the point of the trail where the undoing activity needs to stop.

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); and
(2) a *value* whose interpretation depends on the tag of the word; for example, 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.[5]

Prolog programs are compiled in the WAM 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);

[5] 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 first clause is started: after creating (if needed) the environment for the clause (“`allocate`”), the execution 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 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 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 with the new alternative.

The WAM has been designed in order to optimize the use of resources during execution, improving speed and memory consumption. Optimizations that 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 reuse of the environment of a clause for the execution of the last subgoal of the clause itself;

—*Environment Trimming* [*Warren 1983; Aït-Kaci 1991*]: 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* [*Carrlsson 1989*]: The principle of *procrastination* [Gupta and Pontelli 1997]–postponing work until it is strictly required by the computation–is 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. On many occasions this allows avoiding the allocation of the choice point if head unification fails, 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 run-time. 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; Ramesh et al. 1990].

2.3 Logic Programming and Parallelism

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

$$Programs = Logic + Control.$$

This principle separates the control component from the logical specification of the problem, thus making the control of execution an orthogonal feature, independent of the specification of the problem. The lack of knowledge about control in the program implied by the theoretical view of logic programs 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, at least in theory, different operations in a logic program can be executed in any order without affecting the meaning of the program. In particular, these operations can theoretically be performed by the evaluator *in parallel*.

Apart from the separation between logic and control, from a programming languages perspective, logic programming offers three key features which make exploitation of parallelism more practical than in traditional imperative languages (see Hermenegildo [2000] for some comparisons of the techniques used in parallelizing compilers for logic programs and more traditional programming paradigms):

(1) From an operational perspective, and similar to functional languages, logic programming languages are *single assignment* languages: variables are mathematical entities that can be assigned a value at most once during each derivation. This relieves a parallel system from having to keep track of certain types of flow dependencies, and offers a situation similar to having applied already the "single assignment transformation" often used in the parallelization of traditional programming languages [Zima and Chapman 1991].

(2) In addition, and also similarly to functional languages, logic languages allow coding in a way that expresses the desired algorithm reflecting more directly the structure of the problem (i.e., staying closer to the specifications) and less the control aspects. This makes the parallelism available in the problem more easily accessible to the compiler.

(3) Finally, the operational semantics of logic programming, in contrast to imperative and functional languages, includes a certain degree of *non-determinism*, which can be easily converted into parallelism without radical modifications of the overall operational semantics. This leads to the possibility of extracting parallelism directly from the execution model without any modification to the source program (*implicit parallelization*).

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 the three "classical" forms of parallelism [Conery and Kibler 1981]:

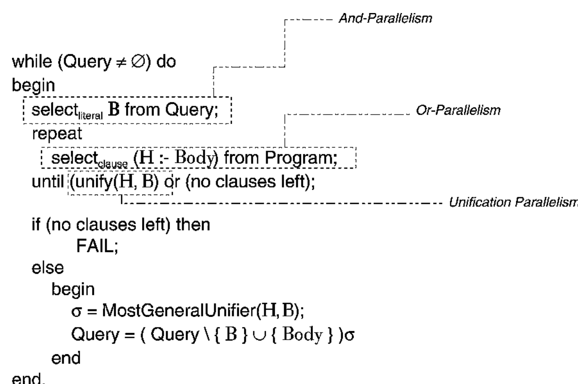


Fig. 2. Operational semantics and nondeterminism.

—*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; and
—*Unification Parallelism*, which arises from the parallelization of the unification process.

The next three sections elaborate 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 approximately 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(hills),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(hills),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 typically very fine-grained, which has prompted the design of 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], which have been shown to be very similar to those used in the and-parallelism [Hermenegildo and Carro 1996; Pontelli and Gupta 1995a; Debray and Jain 1994] (and indeed unification parallelism can be seen as a form of and-parallelism). 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 rule bodies can then be executed in parallel with each other, giving rise to or-parallelism. Or-parallelism is thus a way of searching for solutions to the query faster, 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 et al. 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, certain types of 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 subgoal 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 relevant in divide-and-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 (the descriptions of these types of parallelism are clarified later in the article).

—*Independent and-parallelism (IAP)* arises when, given two or more subgoals, the run-time 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 that, at run-time, do not share any unbound variable; that is, the intersection of the sets of variables accessible by each goal is empty. More refined notions of independence, for example, nonstrict independence, have also been proposed [Hermenegildo and Rossi 1995] where the goals may share a variable but "cooperate" in creating the binding for the common variable.

—*Dependent and-parallelism* arises when, at run-time, two or more goals in the body of a clause have a common variable and are executed in parallel, "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 that faithfully reproduce Prolog's observable semantics to models that 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].

It has been noted that independent and dependent and-parallelism are simply the application of the same principle, *independence*, at different levels of granularity in the computation model. In fact, 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 and/or efficiency. For independent and-parallelism, entire subgoals 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 as *macro level* and-parallelism, while dependent and-parallelism could be considered as *micro level* and-parallelism. Dependent and-parallelism is typically harder to exploit for Prolog, unless adequate changes to the operational semantics are introduced, as in the case of committed choice 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 that 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 that are cooperating in the parallel execution of the program. We often refer to these computing entities as *workers* [Lusk et al. 1990] or *agents* [Hermenegildo and Greene 1991]. 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* that are created for the specific execution of an instance of parallel work (e.g., an agent is created to specifically execute 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 return to this distinction in Section 9.1.

Intuitively, or- and and-parallelism are largely orthogonal to each other, as they parallelize independent points of nondeterminism in the operational semantics of the language. Thus, one would expect that the exploitation of one 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 have 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, 1996b] and Pontelli et al. [1998]) seem to suggest that there are large classes of applications that 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 in a general-purpose system 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 *speed-ups* may not necessarily translate to an actual improvement in performance with respect to state of the art sequential systems; for example, 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. While many early parallel logic programming systems proposed achieved speedups, only a few (e.g., &-Prolog, Aurora, MUSE, ACE, DASWAM) have been shown capable of achieving consistently faster executions than state of the art sequential systems.

In the rest of the article 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 descriptions 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 the foundations of parallel processing; an excellent exposition of the needed concepts can be found in Almasi and Gottlieb [1994] and 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] and Kacsuk and Wise [1992]) has been more sparse and perhaps less incisive. Currently, there is renewed interest in distributed memory architectures [Silva and Watson 2000; Araujo and Ruz 1998; Castro et al. 1999; Gupta and Pontelli 1999c; Hermenegildo 1994; Cabeza and Hermenegildo 1996], 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).
```

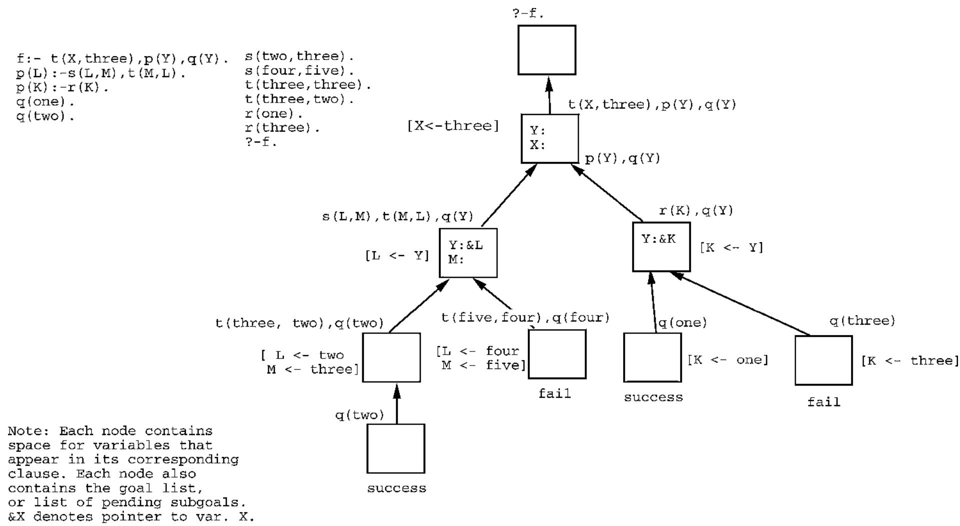


Fig. 3. An Or-parallel tree.

and the query `?- f.` The calls to `t`, `p`, and `q` are nondeterministic 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 nonroot 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 follows the operational semantics of Prolog: at each node we consider clauses applicable to the first subgoal, and the children of a node are considered ordered from left to right according to the order of the corresponding clauses in the program. That is, during sequential execution the or-parallel tree of Figure 3 is searched in a depth-first manner. However, if multiple agents are available, then multiple branches of the tree can be searched simultaneously.

Or-parallelism manifests itself in a number of applications [Kluźniak 1990; Shen 1992b; Shen and Hermenegildo 1996b]. It arises while exercising rules of

an expert system 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.

From the theoretical point of view, or-parallelism poses few problems since the various branches of the or-parallel tree are independent of each other, thus requiring little communication between agents. This has been shown in the literature in a number of related theoretical results which state that, for given sets of conditions (the simplest example being pure programs for which all solutions are requested and no run-time parallelism-related overheads), or-parallel execution of a logic program meets the "no-slowdown" condition: that is, parallel execution will run no slower (and, logically, often much faster) than its sequential counterpart [Hermenegildo and Rossi 1995].

3.1 Challenges in the Implementation of Or-Parallelism

Despite the theoretical simplicity and results, in practice *implementation* of or-parallelism is difficult because keeping the run-time parallelism-related

overheads small (and, therefore, preserving the "no-slowdown" results) is non-trivial due to the practical complications that emerge from 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 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.

To understand this problem, consider Figure 3 where each node of the or-parallel tree contains the variables found in its corresponding clause; that is, it holds that clause's *environment*. If the different branches are searched in or-parallel, then 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.

More generally, 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`; that is, 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, that is, 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* (e.g., variable `X` in Figure 3).

The main problem in implementing or-parallelism is the efficient representation of the multiple environments that coexist 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). 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 accomplishing this effect [Warren 1987b; Gupta and Jayaraman 1993a], 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; and

—recording conditional bindings in a global data structure and attaching a unique identifier to each binding that identifies the branch to which the binding belongs.

Each approach has its associated cost. This cost is nonconstant time and is incurred at the time of variable access, at the time of node creation, or at the time a worker begins execution of a new branch. In Gupta and Jayaraman [1993a] several criteria were derived for an ideal or-parallel system, namely,

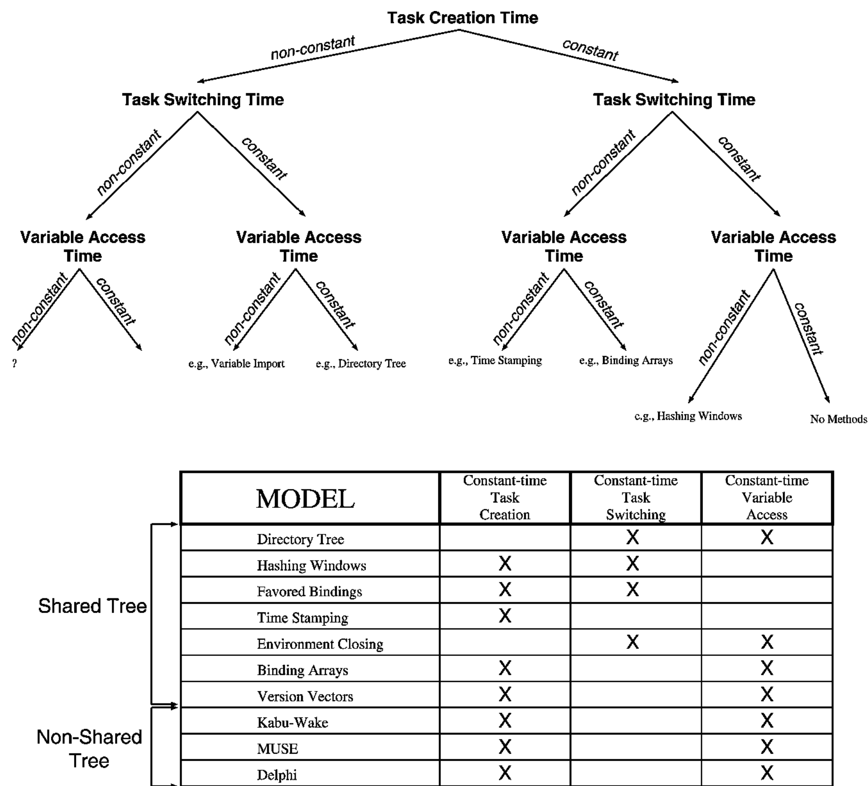
(1) the cost of *environment creation* should be constant-time;
(2) the cost of *variable access and binding* should be constant-time; and
(3) the cost of *task switching*[6] should be constant-time.

It has been shown that it is impossible to satisfy these three criteria simultaneously [Gupta and Jayaraman 1993a; Ranjan et al. 1999]. In other words, the nonconstant time costs in managing multiple or-parallel environments cannot be avoided. Although this nonconstant 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 means 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, in order to avoid excessive (expensive) task switches and to properly handle speculative computations. This 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 [1993a]). These models differ in the techniques they employ for solving the problem of environment representation. The three criteria mentioned in the previous section allow 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 are associated with one of the leaves of the tree, depending on which criteria they meet and which criteria they violate. Observe that the rightmost leaf in the tree is necessarily empty, since no model can meet all 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.

[6]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
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
Kabu-Wake	X		X
MUSE	X		X
Delphi	X		X

Fig. 4. Classification of Or-parallel models.

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 et al. 1987], Binding Arrays Scheme [Warren 1984, 1987c], Argonne–SRI Model [Warren 1987b], Manchester–Argonne Model [Warren 1987b], Delphi Model [Clocksin and Alshawi 1988], Randomized Method [Janakiram et al. 1988], BC-Machine [Ali 1988], MUSE [Ali and Karlsson 1990b] (and its variations, such as stack splitting [Gupta and Pontelli 1999c]), SBA [Correia et al. 1997], PBA [Gupta et al. 1993; Gupta et al. 1994], Virtual Memory Binding Arrays model [Véron et al. 1993], and Kabu Wake Model [Masuzawa et al. 1986]; while the following models employ 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 1987a]; and the following models employ 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, an example of a model that satisfies only criterion 1 (constant time task-creation) is the Time Stamping Model [Tinker 1988], while the example of a model that satisfies only 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 by Warren [1987b]. Some alternative models for or-parallelism, such as Sparse Binding Array and Paged Binding Arrays, are described separately in Section 6.3, since their design is mostly motivated by the desire to integrate exploitation of or-and and-parallelism.

As noted in Figure 4, we are also imposing an additional classification level, which separates the proposed models into 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 those models in which every worker maintains a separate data structure representing (part of) the computation tree.

3.2.1 *Shared Representation of the Computation Tree*

3.2.1.1 *Directory Tree Method.* In the directory tree method [Ciepielewski and Haridi 1983], developed in the early 1980s for the 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 ith location in the directory contains a pointer to the ith context for that process.

When branching occurs, a new directory is created for each child process. For every context in the parent process that 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 with 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 directory creation is potentially very high and the method leads to large memory consumption and poor locality [Crammond 1985].

3.2.1.2 *Hashing Windows Method.* The hashing windows scheme, proposed by 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 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 recursively searched 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 et al. 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 for efficient variable binding lookup. Bindings are separated into two classes [Chassin de Kergommeaux and Robert 1990]:

—*Shallow Bindings:* These are bindings that 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 bindings performed on variables that lie outside the local computation. Access to such bindings is performed using hashing windows.

Variable lookup makes use of the OBL to determine whether the in-place binding is valid, by comparing the OBL of the binding with the OBL existing at the choice point that originated the current process. Details of these mechanisms are presented in Westphal et al. [1987]. A detailed study of the performance of PEPSys has been provided by Chassin de Kergommeaux [1989].

3.2.1.3 *Favored-Bindings Method.* The favored bindings 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 sections 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 that 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 nonfavored 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.

3.2.1.4 *Timestamping Method.* The timestamping method, developed by Tinker [1988], uses timestamps 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 that 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, that created it and the timestamp are examined. The timestamp is then checked to see if it falls in the timespan of process P in any 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 unbound.

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 has a potential for lack of locality of reference, as the global address space is extensively searched in accessing bindings.

3.2.1.5 *Environment Closing Method.* The environment closing method was proposed by Conery [1987a] 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` that 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 et al. 1988a].

3.2.1.6 *Binding Arrays Method.* In the binding arrays method [Warren 1984, 1987c] each worker has an auxiliary data structure called the *binding array*.[7] 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.[8] 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

[7]Note that the description that follows is largely based on Warren [1987c] 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.

[8]Most systems, for example, Aurora, initially treat all the variables as conditional, thus placing them in the binding array.

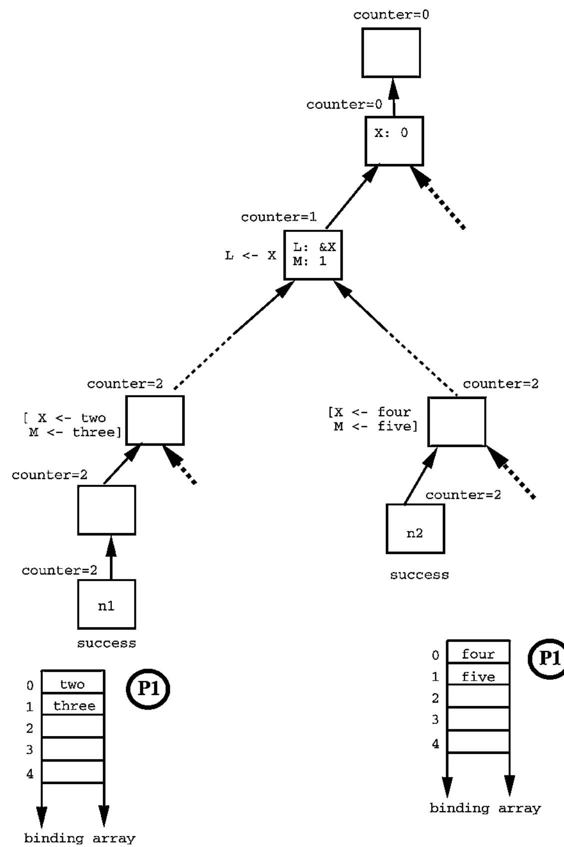


Fig. 5. The Binding Arrays Method.

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 to point to X. The unconditional nature of the binding does not require allocation of an entry in the binding array for L.[9]

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 deinstalling bindings from the trail of the nodes that are in b_i and installing the correct bindings from the trail of the nodes 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 which exists along the branch from the root node to node n2 is reflected in its environment. This is accomplished by

[9]Aurora allocates an entry in the array for each variable, but stores unconditional bindings directly in the stacks.

making P1 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 et al. 1991a]). Furthermore, a number of variations on the idea of binding arrays have been proposed–for example, Paged Binding Arrays and Sparse Binding Arrays–mostly aimed at providing better support for combined exploitation of and-parallelism and or-parallelism. These are discussed in Sections 6.3.6 and 6.3.7.

3.2.1.7 *Versions Vectors Method.* The versions vectors method [Hausman et al. 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. Also as in the binding arrays method, on switching to another branch a worker with pid j has to update the jth slots of versions vectors of all conditional variables that lie in the intervening nodes to reflect the correct bindings corresponding to the new site.

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 *Nonshared Representation of the Computation Tree*

3.2.2.1 *Stack-Copying Method.* In the stack-copying method [Ali and Karlsson 1990a; 1990b], 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 as in a sequential system, requiring very little synchronization with other workers.

In order to avoid duplication of work, part of each choice point (e.g., the pointer to the first unexplored alternative) is moved to a frame created in an

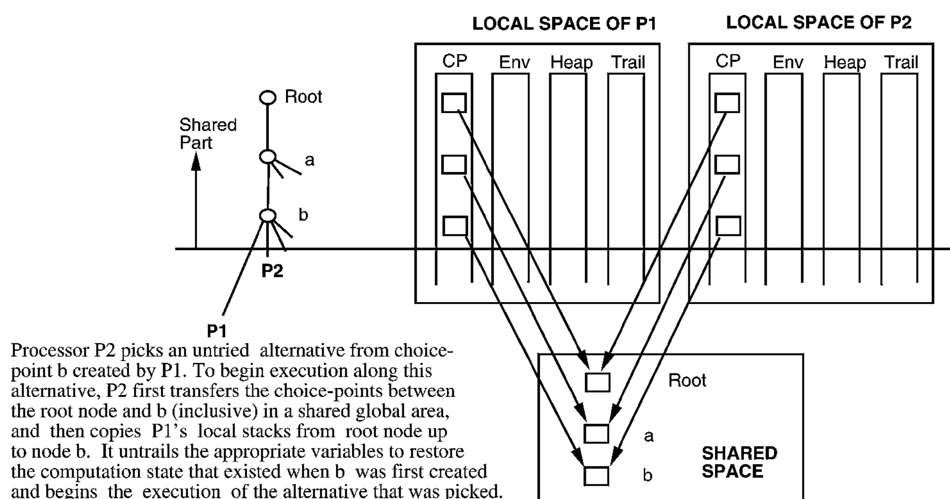


Fig. 6. Stack copying and choice points.

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. This is illustrated in an example in Figure 7. In Figure 7(i) we have two workers immediately after a sharing operation that has transferred three choice points from worker `P1` to `P2`. In 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, leading to the copy of only the two new choice points.

Incremental copying has 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 the BC-machine by Ali [1988], a model where different workers concurrently start the computation of the query and automatically select different alternatives when choice points are created. The idea was already present in the Kabu Wake model [Masuzawa

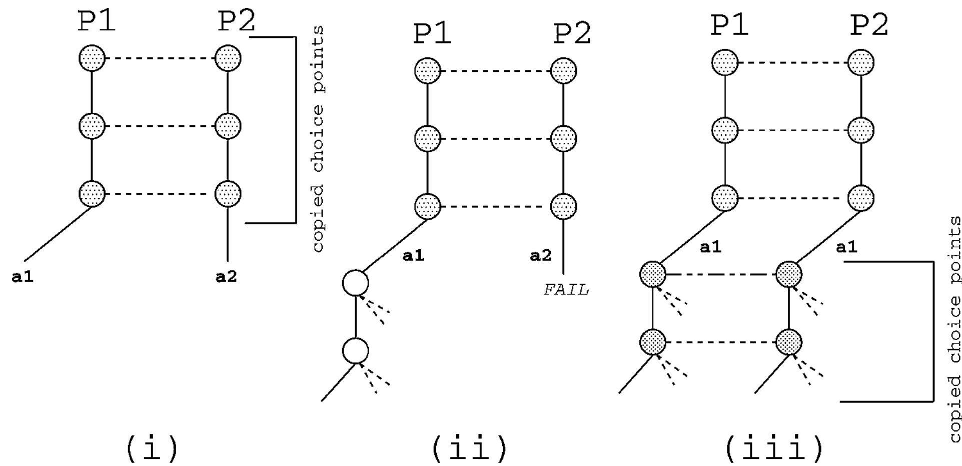


Fig. 7. Incremental stack copying.

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 with respect to 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 takes place.

Stack copying has found efficient implementation in a variety of systems, such as MUSE [Ali and Karlsson 1990b] (discussed in more detail in Section 3.5.2), ECLiPSe [Wallace et al. 1997], and YAP [Rocha et al. 1999b]. 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 et al. 2000].

3.2.2.2 *Stack Splitting.* In the stack-copying technique, each choice point has to be *"shared"* (i.e., transferred 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 [1999c], this method allows the use of very efficient scheduling mechanisms such as the scheduling on the 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

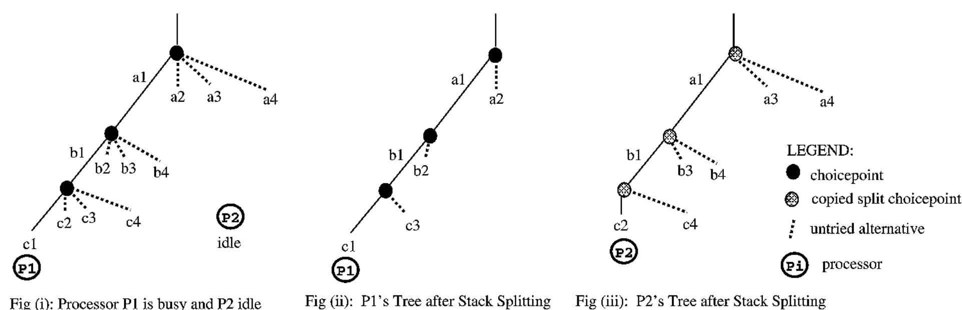


Fig. 8. Stack-splitting based or-parallelism.

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, although 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.

The major advantage of stack-splitting is that scheduling on bottommost 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, nonfailure, user annotations) can further reduce the likelihood of additional communications due to scheduling.

In Gupta and Pontelli [1999c], 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 machines has also provided positive results in terms of speedups and efficiency [Villaverde et al. 2000].

3.2.2.3 *Recomputation-Based Models.* In the stack-copying schemes, idle workers acquire work by copying the data structures associated with a given segment of computation, in order to recreate the state of the computation from where the new alternative will start. An alternative approach 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 an 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 1998] and of the Randomized Parallel Backtracking method [Janakiram et al. 1998].

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 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-Kathib 2001]). The recomputation method has also found applications in the parallelization of constraint logic programming [Mudambi and Schimpf 1994].

3.3 Support for Full Prolog in Or-Parallelism

Most of the or-parallel models described above consider only pure logic programs (pure Prolog) for parallel execution. However, to make logic programming practical many extralogical, metalogical, and input/output predicates have been incorporated in Prolog. Some researchers have taken the view that an or-parallel logic programming system should transparently execute Prolog programs in parallel [Lusk et al. 1990; Hausman et al. 1988].[10] That is, the same effect should be seen by a user during parallel execution of a Prolog program, as far as input/output and the like 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 extralogical predicate 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 et al. 1988b; Sindaha 1993]. Arguably, the techniques used in Aurora have been the most well researched and successful [Hausman et al. 1988; Hausman 1989]. 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 *subroot 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 the 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 that can be executed without requiring any form of synchronization; these are typically called *cavalier* predicates. The use of cavalier extralogical predicates leads to an operational behavior *different* from that of Prolog: for example, a cavalier write operation is going to be executed immediately irrespectively of the execution of the other extralogical predicates in the search tree.

[10]This view has also been taken in and-parallel systems, as we show later [Muthukumar and Hermenegildo 1989; DeGroot 1987b; Chang and Chiang 1989].

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 article, 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 et al. 1999]. Execution of a program can be abstracted as building a (rooted, labeled) tree. For the sake of simplicity, we 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 operations:

(1) create_tree(γ) which creates a tree containing only the root, with label γ;
(2) expand(u, γ_1, γ_2) which, given one leaf u and two labels γ_1 and γ_2, creates two new nodes (one for each label) and adds them as children of u (γ_1 as left child and γ_2 as right child); and
(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 this abstract 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. Γ 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) that has been assigned X; and
- *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 et al. 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]. A pointer machine is a formal model for describing algorithms, which relies on an elementary machine whose memory is composed only of 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] and Gupta and Jayaraman [1993a]. 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 that is strictly worse than $\Omega(N + M)$. Intuitively, this means that no matter how good an implementation model for or-parallelism is, it will incur some costs during the execution that 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 by Ranjan et al. [1999], showing that 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 that 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 the implementation scheme selected is, there will be cases that will lead to a nonconstant time cost. This proof confirms the result put forward in Gupta and Jayaraman [1993a]. This nonconstant time nature is also evident in all the implementation schemes presented in the literature, for example, the creation of the shared frames and the copying of the choice points in MUSE [Ali and Karlsson 1990b], the installation of the bindings in Aurora [Lusk et al. 1990], and the management of timestamps 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 that 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.

The lower bound produced, $O(\lg N)$ per operation, is a confirmation and refinement of the results proposed by Gupta and Jayaraman [1993a], 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, $\tilde{O}(\sqrt[3]{N})$,[11] 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 the upper bound result indeed provides one such model, 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

[11]The notation $\tilde{O}(\sqrt[3]{N})$ indicates that the complexity is within $\lg^k N$ from $\sqrt[3]{N}$, for some small value of k.

Table I. Worst Case Complexity of Some Or-Parallel Schemes (K Operations)

Method	*Complexity*
Known Upper Bound	$\tilde{O}(K \times N^{1/3})$
Stack Copying [Ali and Karlsson 1990a]	$\tilde{O}(K \times N)$
Directory Tree Method [Ciepielewski and Haridi 1983]	$\tilde{O}(K \times N \lg N)$
Binding Arrays [Lusk et al. 1990]	$\tilde{O}(K \times N)$
Environment Closing [Conery 1987a]	$\tilde{O}(K \times N)$

(uniform memory access) shared-memory multiprocessors such as the Sequent Symmetry and subsequently ported [Mudambi 1991] to NUMA (nonuniform memory access) architectures such as the BBN TC-2000 (a scalable architecture with Motorola 88000 processors[12]). Recall that UMA architectures are characterized by the fact that each processor in the system guarantees 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 [1987c] and refined in Lusk et al. [1990]. The SRI-model employs binding arrays for representing multiple environments. In the SRI model, a group of processing agents called *workers* cooperates 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 et al. 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 et al. 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 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

[12]The porting, however, did not involve modifications of the system structure to take full advantage of the architecture's structure.

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 incurred overhead, called *migration cost* (or task-switching cost), is proportional to the number of bindings that are deinstalled and installed. Aurora divides the or-parallel search tree into a *public* region and a *private* region. The public region consists of those nodes from which other workers can pick up untried alternatives. The private region consists of nodes private to a worker that cannot be accessed by other workers. Execution within 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 nonleftmost 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.

It is often the case that this strict form of synchronization is unnecessary, and slows down parallel execution. Aurora therefore provides nonsynchronized variants for most order-sensitive predicates that come in two flavors: the *asynchronous* form respecting the cut pruning operator, and the completely relaxed *cavalier* form. Notably, nonsynchronized 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 [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 et al. [1993]. The nonsynchronized dynamic database features have been exploited in the implementation of a general algorithm for solving optimization problems [Szeredi 1991, 1992].

Three schedulers are currently operational. Two older schedulers were written [Butler et al. 1998; Brand 1998], but have not been updated to comply with the scheduler–engine interface:

(1) *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 by Szeredi [1989].

(2) *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 nonspeculative.

(3) *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 speedups obtained by all schedulers of Aurora for a diverse set of benchmark programs have been very encouraging. Some of the benchmark programs contain a significant amount of speculative work, in which speedups 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], and Sindaha [1992]. The binding array model has also been adapted for distributed shared memory architectures and implemented in the Dorpp system [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 1990b, 1992a,b; Ali et al. 1992; 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

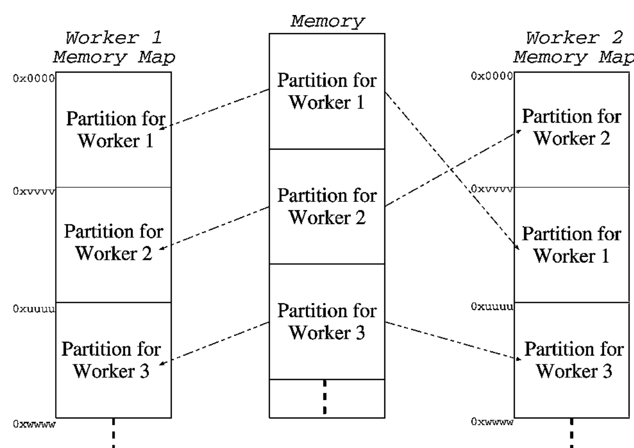


Fig. 9. Memory organization in MUSE.

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 among 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 resides at the same address range in each worker.

Workers make a number of choice points sharable, and they get work from those shared choice points (nodes) by the normal backtracking of Prolog. As in 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 and nonspeculative 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 was less speculative. Active work is work that is nonspeculative and is actively pursued by workers. The available workers concentrate on the available nonspeculative 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 nonspeculative 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 nonspeculative 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 nonspeculative 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 in 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 speedups 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 speedups 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 speedups 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, 1992b; Ali et al. 1992; Karlsson 1992].

MUSE can be considered one of the first *commercial* parallel logic programming systems ever to be developed; MUSE was included for a number of years as part of the standard distribution of SICStus Prolog [Carlsson et al. 1995].[13]

4. INDEPENDENT AND-PARALLELISM

Independent and-parallelism refers to the parallel execution of goals that have no “data dependencies” and thus do not affect each other. To take a simple example, consider the naïve *Fibonacci* program shown below.

[13]However, MUSE is no longer supported by SICS.

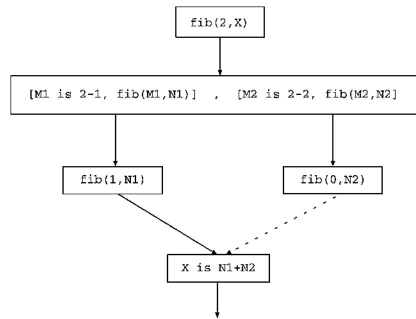


Fig. 10. An and-tree for and-parallelism.

```
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)` with respect to 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 applications, those in which a given problem can be divided into a number of independent subproblems. For example, it appears in divide-and-conquer algorithms, where the independent recursive calls can be executed in parallel (e.g., matrix multiplication, quicksort).

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]:

(1) *Ordering Phase*: Deals with detection of dependencies among goals;
(2) *Forward Execution Phase*: Deals with the steps needed to select the next subgoal for execution and initiate its execution; and

(3) *Backward Execution Phase*: Deals with steps to be taken when a goal fails, that is, the operation of backtracking.

4.1.1 *Ordering Phase: Notions of Independence.* The ordering phase in an 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. An interesting issue that has received much attention in the literature is determining precisely when a data dependency exists. The issues involved in answering this question are, as we show, rather interesting and unique in the case of logic programming [Hermenegildo 2000].

The objective of the process is to uncover as much as possible of the available parallelism, while guaranteeing that the correct results are computed (*correctness*) and that other observable characteristics of the program, such as execution time, are improved (*speedup*) or, at the minimum, preserved (*no-slowdown*)—*efficiency*. A central issue is, then, under which conditions two goals ("statements") in a logic program can be correctly and efficiently parallelized.

For comparison, consider the following segments of programs in (a) a traditional imperative language, (b) a (strict) functional language, and (d) a logic language (we consider case (c) later). We assume that the values of `W` and `Z` are initialized to some value before execution of these statements:

s_1	`Y := W+2;`	`(+ (+ W 2)`	`Y = W+2,`
s_2	`X := Y+Z;`	`Z)`	`X = Y+Z,`
	(a)	*(b)*	*(c)*

(d)
```
main:-
  s1   p(X),
  s2   q(X),
       ...
```
```
p(X) :- X=a.

q(X) :- X=b, large computation.
q(X) :- X=a.
```

For simplicity, we reason about the correctness and efficiency of parallelism using the instrumental technique of considering reorderings (interleavings). Statements s_1 and s_2 in (a) are generally considered to be *dependent* because reversing their order would yield an *incorrect* result; that is, it violates the *correctness* condition above (this is an example of a *flow-dependency*).[14] A slightly different, but closely related situation occurs in (b): reversing the order of function application would result in a run-time error (one of the arguments to a function would be missing).

Interestingly, reversing the order of statements s_1 and s_2 in (d) does yield the correct result (`X=a`). In fact, this is an instance of a more general rule: if no side effects are involved, reordering statements does not affect *correctness*

[14]To complete the discussion above, note that output-dependencies do not appear in functional or logic and constraint programs because single assignment is generally used; we consider this a minor point of difference since one of the standard techniques for parallelizing imperative programs is to perform a transformation to a single assignment program before performing the parallelization.

in a logic program. The fact that (at least in pure segments of programs) the order of statements in logic programming does not affect the result[15] led in early models to the proposal of execution strategies where parallelism was exploited "fully" (i.e., all statements were eligible for parallelization). However, the problem is that such parallelization often violates the principle of efficiency: for a finite number of processors, the parallelized program can be arbitrarily slower than the sequential program, even under ideal assumptions regarding run-time overheads. For instance, in the last example, reversing the order of the calls to `p` and `q` in the body of `main` implies that the call `q(X)` (`X` at this point is free, i.e., a pointer to an empty cell) will first enter its first alternative, performing the large computation. Upon return of `q` (with `X` pointing to the constant `b`) the call to `p` will *fail* and the system will backtrack to the second alternative of `q`, after which `p` will succeed with `X=a`. On the other hand, the sequential execution would terminate in two or three steps, without performing the large computation. The fundamental observation is that, in the sequential execution, `p` *affects* `q`, in the sense that it *prunes* (limits) its choices. Executing `q` before executing `p` results in performing *speculative choices* with respect to the sequential execution. Note that this is in fact very related to executing conditionals in parallel (or ahead of time) in traditional languages (note that `q` above could also be (loosely) written as "`q(X) :- if X=b then` *`large computation`* `else if X=a then true else fail.`").

Something very similar occurs in case (c) above, which corresponds to a *constraint logic program*: while execution of the two constraints in the original order involves two additions and two assignments (the same set of operations as those of the imperative or functional programs), executing them in reversed order involves first adding an equation to the system, corresponding to statement s_2, and then solving it against s_1, which is more expensive. The obvious conclusion is that, in general, even for pure programs, arbitrary parallelization does not guarantee that the *two* conditions (*correctness* and *efficiency*) are met.[16] We return to the very interesting issue of what notions of parallelism are appropriate for constraint logic programming in Section 8.

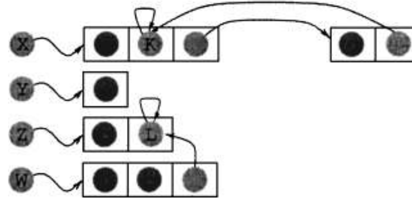
Contrary to early beliefs held in the field, most work in the last decade has considered that violating the efficiency condition is as much a "sign of dependence" among goals as violating the correctness condition. As a result, interesting notions of independence have been developed that capture these two issues of correctness and efficiency at the same time: independent goals as those whose run-time behavior, if parallelized, produces the same results as their sequential execution and an increase (or, at least, no decrease) in performance. To separate issues better, we discuss the issue under the assumption of ideal run-time conditions, that is, no task creation and scheduling overheads (we deal with

[15]Note that in practical languages, however, termination characteristics may change, but termination can actually also be seen as an extreme effect of the other problem to be discussed: efficiency.
[16]In fact, this is similar to the phenomenon that occurs in or-parallelism where arbitrarily parallelizing branches of the search does not produce incorrect results, but, if looking for only one solution to a problem (or, more generally, in the presence of *pruning operators*) results in speculative computations that can have a negative effect of efficiency.

overheads later). Note that, even under these ideal conditions, the goals in (c) and (d) are clearly *dependent* using the definition.

A fundamental question then is how to guarantee independence (without having to actually run the goals, as suggested by the definition given above). A fundamental result in this context is the fact that, if only the Herbrand constraint system is used (as in the Prolog language), a goal or procedure call, q, *cannot be affected* by another, p, if it does not share logical variables with it at the point in time just before execution (i.e., in the substitution represented by s_1). that is, in those cases correctness and efficiency hold and no-slowdown is guaranteed. In practice, the condition implies that there are no shared free variables (pointers to empty structure fields) between the run-time data structures passed to q and the data structures passed to p. This condition is called *strict independence* [DeGroot 1984; Hermenegildo and Rossi 1995].[17] For example, in the following program:

```
main :-  X=f(K,g(K)),
         Y=a,
         Z=g(L),
         W=h(b,L),
         p(X,Y),
         q(Y,Z),
         r(W).
```



p and q are *strictly independent*, because, at the point in execution just before calling p (the situation depicted in the right part of the figure), X and Z point to data structures that do not point to each other, and, even though Y is a pointer which is shared between p and q, Y points to a fixed value, which p cannot change (note again that we are dealing with single assignment languages). As a result, the execution of p cannot affect q in any way and q can be safely run ahead of time in parallel with p (and, again assuming no run-time overheads, no-slowdown is guaranteed). Furthermore, no locking or copying of the intervening data structures is required (which helps bring the implementation closer to the ideal situation). Similarly, q and r are not strictly independent, because there is a pointer in common (L) among the data structures they have access to and thus the execution of q could affect that of r.

Unfortunately, it is not always easy to determine independence by simply looking at one procedure, as above. For example, in the program below,

```
main :- t(X,Y),
        p(X),
        q(Y).
```

it is possible to determine that p and q are not (strictly) independent of t, since, upon entering the body of the procedure, X and Y are free variables that are

[17]To be completely precise, in order to avoid creating speculative parallelism, some nonfailure conditions are also required of the goals executed in parallel, but we knowingly overlook this issue at this point to simplify the discussion.

shared with t. On the other hand, after execution of t the situation is unknown since perhaps the structures created by t (and pointed to by X and Y) do not share variables. Unfortunately, in order to determine this for sure a global (interprocedural) analysis of the program (in this case, to determine the behavior of t) must be performed. Alternatively, a *run-time test* can be performed just after the execution of t to detect independence of p and q. This has the undesirable side-effect that then the no-slowdown property does not automatically hold, because of the overhead involved in the test, but it is still potentially useful.

A number of approaches have been proposed for addressing the data dependency detection issues discussed above. They range from purely compile-time techniques to purely run-time ones. There is obviously a trade-off between the amount of and-parallelism exploited and data dependency analysis overhead incurred at run-time: 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 which prevents the system from achieving the theoretical *efficiency* results. However, data dependencies cannot always be detected entirely at compile time, although compile-time analysis tools can uncover a significant portion of such dependencies. The various approaches are briefly described below.

(1) *Input/Output Modes:* One way to overcome the data dependency problem is to require the user to specify the "mode" of the variables, that is, whether an argument of a predicate is an input or 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 that of Prolog).

(2) *Static Data Dependency Analysis:* In this technique the goal and the program clauses are globally analyzed at compile time, assuming a worst case for subgoal dependencies. No checks are done at run-time. This approach was first attempted in Chang et al. [1985]. However, the relatively simple compile-time analysis techniques used, combined with no run-time checking means that a lot of parallelism may be lost. The advantage is, of course, that no overhead is incurred at run-time.

(3) *Run-Time Dependency Graphs:* Another approach is to generate the *dependency graph* at run-time. This involves examining bindings of relevant variables every time a subgoal finishes executing. This approach has been adopted, for example, by Conery in his and/or model [Conery and Kibler 1981, 1983; Conery 1987b]. The approach has prohibitive run-time 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 run-time). 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.

(4) A fourth approach, which is midway between (2) and (3), encapsulates the dependency information in the code generated by the compiler along with the addition of some extra conditions (tests) on the variables. In this way simple run-time checks can be done to check for dependency. This technique, called Restricted (or Fork/Join) And-Parallelism (RAP), was first proposed by DeGroot [1984]. Hermenegildo [1986a] defined a *source-level* language (Conditional Graph Expressions—CGEs) in which the conditions and parallel expressions can be expressed either by the user or by the compiler. The advantage of this approach is that it makes it possible for the compiler to express the parallelization process in a user-readable form and for the user to participate in the process. This effectively eliminates the dichotomy between manual and automatic parallelization. Hermenegildo, Nasr, Rossi, and García de la Banda formalized and enhanced the Restricted And-Parallelism model further by providing backtracking semantics, a formal model, and correctness and efficiency results, showing the conditions under which the "no-slowdown" property (i.e., that parallel execution is no slower than sequential execution) holds [Hermenegildo 1986a, 1987; Hermenegildo and Nasr 1986; Hermenegildo and Rossi 1995; García de la Banda et al. 2000]. A typical CGE has the form:

$$(\textit{conditions} \texttt{ => goal}_1 \texttt{ \& ... \& goal}_n)$$

equivalent to (using Prolog's if-then-else):

$$(\textit{conditions} \texttt{ -> goal}_1 \texttt{ \& ... \& goal}_n \texttt{ ; goal}_1\texttt{, ..., goal}_n)$$

where "&" indicates a *parallel conjunction*, that is, subgoals that can be solved concurrently (while "," is maintained to represent *sequential conjunction*, i.e., to indicate that the subgoals should be solved sequentially). The Restricted And-Parallelism model is discussed in more detail in Section 4.3. Although Restricted And-Parallelism may not capture all the instances of independent and-parallelism present in the program, in practice it can exploit a substantial part of it.

Approach (1) differs from the rest in that the programmer has to explicitly specify the dependencies, using annotations. Approach (4) is a nice compromise between (2), where extensive compile-time analysis is done to get suboptimal parallelism, and (3), where a costly run-time analysis is needed to get maximal parallelism. The annotations of (4) can be generated by the compiler [DeGroot 1987a] and the technique has been shown to be successful when powerful global analysis (generally based on the technique of abstract interpretation [Cousot and Cousot 1977, 1992]) is used [Hermenegildo and Warren 1987; Winsborough and Waern 1988; Muthukumar and Hermenegildo 1990, 1992a; Giannotti and Hermenegildo 1991; Hermenegildo et al. 1992, 2000; Jacobs and Langen 1992; Bueno et al. 1994, 1999; Muthukumar et al. 1999; Puebla and Hermenegildo 1999, 1996].

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 as 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 nonground 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. Hermenegildo [1987] showed the relationship between scheduling and memory management, and provided ideas on using more sophisticated scheduling techniques for guaranteeing a better match between the logical organization of the computation and its physical distribution on the stacks, with the aim of simplifying backtracking and memory performance. This issue has been studied further in Shen and Hermenegildo [1994, 1996a], where flexible related scheduling and memory management approaches are studied. Related research on scheduling for independent and-parallel systems has also been proposed by Dutra [1994]. In Pontelli and Gupta [1995b] a methodology is described which adapts scheduling mechanisms developed for or-parallel systems to the case of independent and-parallel systems. In the same way in which an 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 support intelligent forms of backtracking over and-parallel calls, which allow us 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 [1995b] to support this perspective. This notion is also close to the *first-fail principle* widely used in constraint programming [Haralick and Elliot 1980]. The importance of determining goals that will not fail and/or are deterministic was studied also in Hermenegildo [1986a], Pontelli et al. [1996], Hermenegildo and Rossi [1995], and García de la Banda et al. [2000], and techniques have been devised for detecting deterministic and nonfailing computations at compile-time [Debray and Warren 1989; Debray et al. 1997].

4.1.3 *Backward Execution Phase.* The need for a backward execution phase arises from the nondeterministic 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

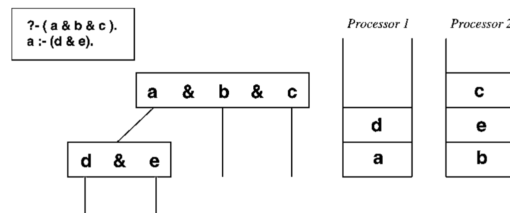


Fig. 11. Lack of correspondence between physical and logical computation.

should backtrack is determined, the machine state is restored, and forward execution of the selected subgoal is initiated.

As mentioned before, Hermenegildo [1986a] showed that, in the 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 that are one after the other in the resolvent) may be *physically* located in noncontiguous 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 a hazard during backtracking, since Prolog semantics require first exploring 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.

Considerably different approaches have been adopted in the literature to handle the backward execution phase. The simplest approach, as adopted in models such as 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 solution reuse. For example, 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

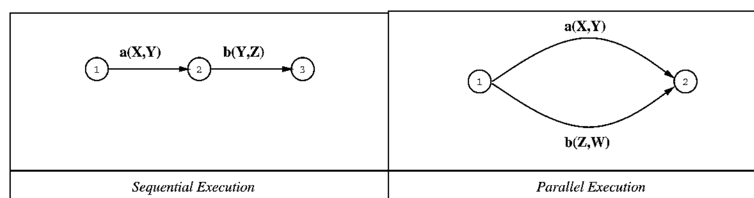


Fig. 12. Solution reuse.

cross-product operation. Although intuitively simple, this approach suffers from major drawbacks, including the extreme complexity of recreating Prolog semantics, that is, 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; for example, in a goal such as `?- p,q`, the goal `q` is completely recomputed for each solution of `p`.

In the context of independent and-parallel systems based on recomputation (such as those proposed in Hermenegildo [1986b], Lin and Kumar [1988], and Pontelli et al. [1996]), a number of different backtracking algorithms have been proposed. In the past, backtracking algorithms have been proposed that later turned out to be incomplete [Woo and Choe 1986].

The earliest and most widely used correct backtracking algorithm for IAP has been presented by Hermenegildo and Nasr [1986] and efficiently developed in &-Prolog [Hermenegildo and Greene 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,

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

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 that

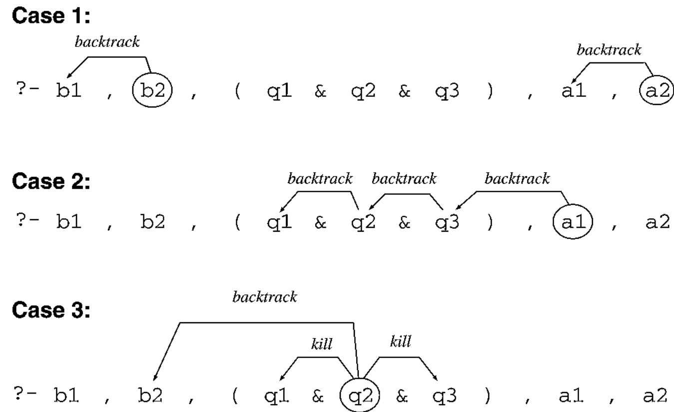


Fig. 13. Backtracking on And-parallel Calls.

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
  (a) the subgoals q_j ($j > i$) should be removed;
  (b) 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 by 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 nondeterminate 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 that 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 intelligent 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 1990; Winsborough 1987] have been proposed and implemented. In particular, the work by Codognet and Codognet [1990] 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 et al. 1988] to support both forward and backward and-parallel execution.

A further distinction has been made in the literature [Pontelli et al. 1996; Shen and Hermenegildo 1994, 1996a] 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 been 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 of where it resides; it can also backtrack on computations lying on the stack of a different worker.

Private backtracking has been adopted in various systems [Hermenegildo and Greene 1991; Shen 1992a]. It has the advantage of allowing each worker to have complete control of the parts of the computation that 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 these message-passing activities (e.g., to avoid the risk of deadlocks) makes the implementation very complex [Shen 1992b; Pontelli et al. 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 in And-Parallelism

As in the case of or-parallel systems, some researchers have favored supporting Prolog's sequential semantics in independent and-parallel systems [DeGroot 1987b; Muthukumar and Hermenegildo 1989; Chang and Chiang 1989]. This imposes some constraints on how backtracking as well as forward execution takes 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 extralogical predicates (e.g., input/output operations) that would be executed had we backtracked into `b`. A form of intelligent backtracking can be maintained and applied to the subgoals lying on the right of the failing one. In the same way as or-parallel systems, these systems also include useful "concurrent" versions of order-sensitive predicates, whose semantics do not require sequencing. In addition, supporting full Prolog also introduces challenges in other parts of and-parallel systems, such as, for example, in parallelizing compilers that perform global analysis [Bueno et al. 1996].

The issue of speculative computation also arises in independent and-parallel systems [Tebra 1987; Hermenegildo and Rossi 1995; García de la Banda et al. 2000]. 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 be wasted (in sequential Prolog the goal `b` will never be executed). Therefore, not too many resources (workers) should be invested in goals to the right. Once again, it should be stressed that the design of the work-scheduler is very important for a parallel logic programming system. Also, and as pointed out before, issues such as nonfailure and determinism analysis can provide important performance gains.

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) The And-Parallel Execution (APEX) model of Lin and Kumar [1988]; and,
(3) The Restricted And-Parallel (RAP) Model, introduced in DeGroot [1984], and extended in Hermenegildo and Nasr [1986b], Hermenegildo [1986b], and in Pontelli et al. [1995].

4.3.1 *Conery's Model.* In this method [Conery and Kibler 1983], a dataflow graph is constructed during the ordering phase making the producer–consumer relationships between subgoals explicit. If a set of subgoals has 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 dataflow graph, in which an arc is drawn from the producer of a variable to all its consumers.

Once the dataflow graph is determined, the forward execution phase ensues. In this phase, independent and-parallel execution of subgoals that do not have

any arcs incident on them in the dataflow 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 dataflow graph. If a producer creates a nonground term during execution, the ordering algorithm must be invoked again to incrementally redraw the dataflow 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 dataflow graph) for the purpose of re-solving.

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

4.3.2 *APEX Model.* The APEX (And-Parallel EXecution) model has been devised by 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 is more than one executable subgoal 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 that 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, $\mathtt{P}_j = \text{head}(\textit{B-list}(\mathtt{P}_i))$ is selected as the subgoal to which to backtrack. 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 run-time costs since the *B-lists* are created, merged, and manipulated at run-time. APEX has been implemented on shared memory multiprocessors for pure logic programs [Lin and Kumar 1988].

4.3.3 *RAP Model.* As mentioned before, in the standard version of this model program clauses are compiled into conditional graph expressions (CGEs) of the form:

$$(\textit{condition} => \mathtt{goal}_1 \ \& \ \mathtt{goal}_2 \ \& \ \cdots \ \& \ \mathtt{goal}_n),$$

meaning that, if *condition* is true, goals $\mathtt{goal}_1 \cdots \mathtt{goal}_n$ should be evaluated in parallel, otherwise they should be evaluated sequentially. The *condition* is a conjunction of tests of two types: *ground*($[v_1, \ldots, v_n]$) checks whether all of the variables $v_1, \ldots, v_n$ are bound to ground terms. *independent*(v_1, v_2) checks whether the set of variables reachable from v_1 is disjoint from the set of variables reachable from v_2. The *condition* can also be the constant *true*, which

means the goals can be unconditionally executed in parallel. The groundness and independence conditions are in principle evaluated at run-time. A simple technique that keeps track of groundness and independence properties of variables through tags associated with 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]). Enhanced parallelism operators and CGE expressions that eliminate this restriction while preserving backtracking semantics have been proposed in Cabeza and Hermenegildo [1996].

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 1986b; Pontelli et al. 1995] as the &-Prolog/Ciao system [Hermenegildo and Greene 1991], as the &ACE/ACE system [Pontelli et al. 1995, 1996], and as part of the dependent and-parallel DASWAM system [Shen 1992b,a].

A considerable body of work exists on the task of automatically parallelizing programs at compile time and generating CGEs. Global program analysis (generally based on the technique of abstract interpretation [Cousot and Cousot 1977, 1992]) has been shown useful at guiding the parallelization process and reducing the conditions in the CGEs, generating simpler run-time tests or even unconditional parallelism [Winsborough and Waern 1988; Muthukumar and Hermenegildo 1992, 1991, 1990; Giannotti and Hermenegildo 1991; Hermenegildo et al. 1992; Jacobs and Langen 1992; Bueno et al. 1994, 1999; Muthukumar et al. 1999; Puebla and Hermenegildo 1999]. A detailed overview of this automatic parallelization work is beyond the scope of this article. See Hermenegildo [2000] for a tutorial introduction and pointers to literature.

4.4 Experimental Systems

4.4.1 *The &-Prolog AND-Parallel Prolog System.* &-Prolog [Hermenegildo 1986a, 1986b; Hermenegildo and Greene 1991] is a prototype Prolog implementation, built as an extension of SICStus Prolog 0.5 (and, later, 0.6–0.7) 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 and exploited. 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 online 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

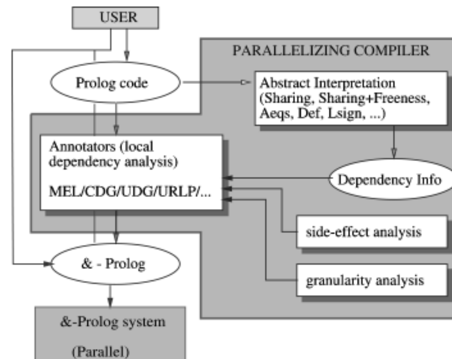


Fig. 14. &-Prolog parallelizer structure.

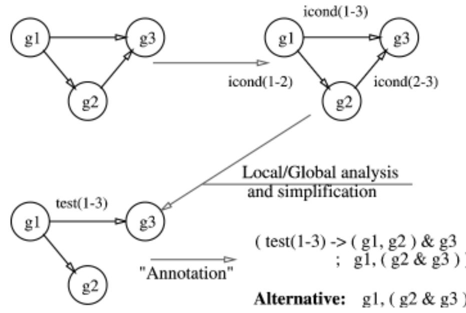


Fig. 15. Parallelization process in &-Prolog (for `p :- g1, g2, g3.`).

(implemented as "Prolog flags") determine whether user code will be parallelized and through which type of analysis. If the user chooses to parallelize some of the code, the compiler still helps by checking the supplied annotations for correctness, and providing the results of global analysis to aid in the dependency analysis task.

&-Prolog was originally designed for global addressing space systems and it has been implemented on a number of shared memory multiprocessors, including Sequent Balance, Sequent Symmetry, and Sun Galaxy systems (and it has been implemented on distributed systems as well [Hermenegildo 1994; Cabeza and Hermenegildo 1996]). 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 online visualization system (based on the X-windows standard) which provides a graphical representation of the parallel execution and which 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 [Hermenegildo and Warren 1987] (Figures 14 and 15): The *Annotator*, or "parallelizer," performs a (local) *dependency* analysis on the

input code, using a conditional graph-based approach. This is illustrated in Figure 15 representing the parallelization of "$g_1(\cdots)$, $g_2(\cdots)$, $g_3(\cdots)$". The bodies of procedures are explored looking for statements and procedure calls that are candidates for parallelization. A dependency graph is first built which in principle reflects the total ordering of statements and calls given by the sequential semantics. Each edge in the graph is then labeled with the run-time data conditions (the run-time check) that would guarantee independence of the statements joined by the edge. If the appropriate option is selected, the annotator obtains information about the possible run-time substitutions ("variable bindings") at all points in the program as well as other types of information from the *Global Analyzer* (described below). This information is used to prove the conditions in the graph statically true or false (Figure 15). If a condition is proved to be true, then the corresponding edge in the dependency graph is eliminated. If proved false, then an unconditional edge (i.e., a static dependency) is left. Still, in other edges conditions may remain, but possibly simplified.

The annotator also receives information from the *Side-Effect Analyzer* on whether each nonbuilt-in predicate and clause of the given program is *pure*, or contains a *side-effect*. This information adds dependencies to correctly sequence such side-effects [Muthukumar and Hermenegildo 1989].

The annotator then encodes the resulting graph using the `&` operator producing an "annotated" (parallelized) &-Prolog program. The techniques proposed for performing this process depend on many factors including whether arbitrary parallelism or only fork-join structures are allowed and also whether run-time independence tests are allowed. As an example, Figure 15 presents two possible encodings in &-Prolog of the (schematic) dependency graph obtained after analysis. The parallel expressions generated in this case use only fork-join structures, one with run-time checks and the other without them. The parallelizer also receives information from the *granularity analyzer* regarding the size of the computation associated with a given goal [Debray et al. 1990, 1997, 1994; López-García et al. 1996]. This information is used in an additional pass aimed at introducing granularity control, implemented using dynamic term size computation techniques [Hermenegildo and López-García 1995]. The information from global analysis is also used to eliminate loop invariants and repetitive checks, using the technique described in Giannotti and Hermenegildo [1991], and Puebla and Hermenegildo [1999]. 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.

Some of the techniques and heuristics used in the annotator, including techniques for compilation of *conditional* nonplanar dependency graphs into fork-join structures, and other, non graph-based techniques, are described in Muthukumar and Hermenegildo [1990], Codish et al. [1995], Bueno et al. [1994], Muthukumar et al. [1999], and Cabeza and Hermenegildo [1994]. The global analysis mentioned above is performed by using the technique of "abstract interpretation" [Cousot and Cousot 1977, 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 et al. 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 [1991] and the specialized fixpoint algorithms described in Muthukumar and Hermenegildo [1992a], Bueno et al. [1996], Hermenegildo et al. [2000], and Puebla and Hermenegildo [1996]. 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 Hermenegildo [1991], Hermenegildo et al. [1992], Codish et al. [1995], and Bueno et al. [1999]. Finally, Hermenegildo et al. provide an overview of CiaoPP, the Ciao system preprocessor, which shows other applications of the types of analyses performed by the PLAI system.

&-Prolog Run-Time System. The &-Prolog run-time system is based on the Parallel WAM (PWAM) model [Hermenegildo and Greene 1991], an evolution of RAP-WAM [Hermenegildo 1986a, 1986b; Tick 1991], itself an extension of the Warren Abstract Machine [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 with 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 that are ready to be executed in parallel are pushed onto the goal stack. The goals are then available to be executed on any PWAM (including the PWAM that 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 parallel conjunction is encountered (e.g., in a CGE with a satisfiable condition part). The parcall frame contains, among other entries, 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

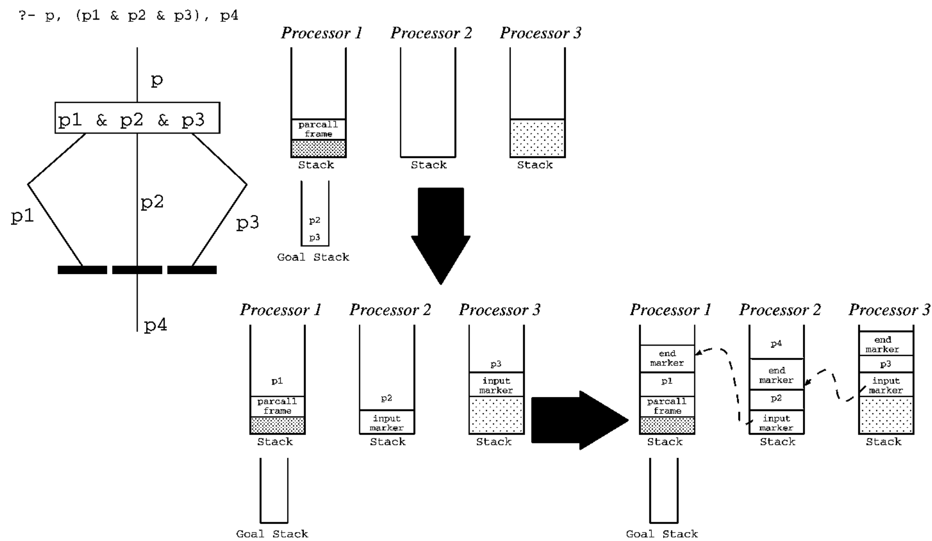


Fig. 16. Organization of computation in PWAM.

backtracking of parallel goals in a similar manner to choice points for sequential goals [Hermenegildo 1986b; Shen and Hermenegildo 1994, 1996a]. As illustrated in Figure 16, 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 (Prolog) order.

Figure 16 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 Worker 2 is 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 that are available for remote execution.

The &-Prolog run-time system architecture comprises a ring of stack sets, a collection of agents, and a shared code area (Figure 17). The agents (Unix

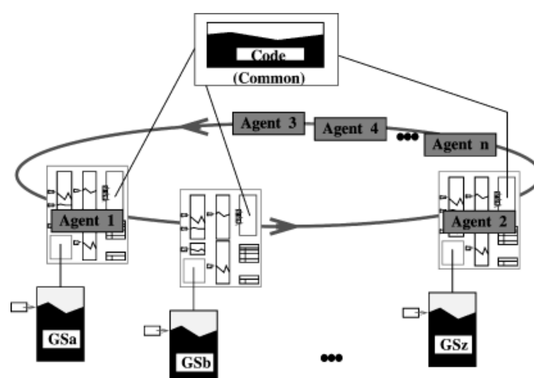


Fig. 17. The ring of PWAMs.

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 [Hermenegildo 1987; Hermenegildo and Greene 1991; Shen and Hermenegildo 1994].

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 of run-time tests using the MA^3 analyzer was reported in Hermenegildo et al. [1992]. 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 and the figures obtained close to what is possible manually.

Experimental results regarding the performance of the second generation parallelizing compiler in terms of attainable program speedups were reported in Codish et al. [1995] and Bueno et al. [1994, 1999] both without global analysis and also with sharing and sharing + freeness analysis running in the PLAI framework [Muthukumar and Hermenegildo 1992; Muthukumar et al. 1999]. Speedups were obtained from the run-time system itself and also using the IDRA system [Fernández et al. 1996], which collects traces from sequential executions and uses them to simulate an ideal parallel execution of the same program.[18] A much more extensive study covering numerous domains and situations, a much larger class of programs, and the effects of the three

[18]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.

annotation algorithms described in Muthukumar and Hermenegildo [1990] (UDG/MEL/CDG), can be found in Bueno et al. [1999] and García de la Banda et al. [1996b]. Although work still remains to be done, especially in the area of detecting nonstrict independence,[19] 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, 1996b].

Early 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 (for the time) comparatively large applications (e.g., parts of the abstract interpreter). Results were also compared to the performance of the sequential programs under &-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.[20]

The &-Prolog system (or, more precisely, the abstract machine underlying the system [Hermenegildo 1986a,b; Hermenegildo and Greene 1991]) is arguably the earliest proposed parallel execution system for logic programs which was shown consistently to produce speedups over state of the art sequential systems.

The &-Prolog system has been extended to support full concurrency in the language [Cabeza and Hermenegildo 1996; Carro and Hermenegildo 1999], other types of parallelism (such as nonstrict [Cabeza and Hermenegildo 1994] and dependent AND-parallelism [Hermenegildo et al. 1995]), AND-parallelism in constraint logic programs, and distributed execution [Hermenegildo 1994; Cabeza and Hermenegildo 1996]. These extensions are mentioned in the appropriate sections later in the article. The development of the &-Prolog system continues at present in Ciao, a next-generation logic programming system [Hermenegildo et al. 1999a; Bueno et al. 1997].

4.4.2 *The* &ACE *System.* The &ACE [Pontelli et al. 1995, 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). As does &-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

[19]The notion of nonstrict independence is described in Section 5.3.3.

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

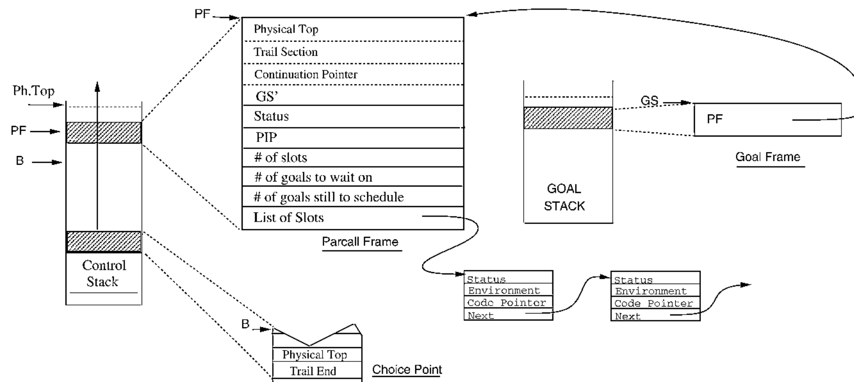


Fig. 18. Parcall frames and goals in &ACE.

stack, differently from &-Prolog, which allocates parcall frames on the environment stack; this allows for easier memory management[21] (e.g., facilitates the use of last-call optimization) and for application of various determinacy-driven optimizations [Pontelli et al. 1996] and alternative scheduling mechanisms [Pontelli et al. 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 parcall frame; this has been demonstrated [Pontelli et al. 1995, 1996] to be more effective and flexible than actual goal stacking. These data structures are described in Figure 18.

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 linking choice points lying in different stacks in the correct logical order; this allows limiting 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 that indicates the beginning of the continuation of the parallel call. Novel uses of the trail stack (by trailing status flags in the subgoal slots) allows the integratation of 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 us to take full advantage of the semi-intelligent backtracking phase during inside backtracking. Given a subgoal of the form

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

[21]&ACE is built on top of the SICStus WAM, that performs on-the-fly computation of the top of the stack register. The presence of parcall frames on the same stack creates enormous complications in the correct management of such a register.

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 the killing of subgoals and backtracking 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]. The use of backtracking independence allows the system to recover the full power of intelligent backtracking; in Pontelli and Gupta [1998] results are presented that show improvements of up to 400% in execution time over traditional point-backtracking.

&ACE has been developed by modifying the SICStus WAM and currently runs on Sparc- and Pentium-based multiprocessors. The use of the new memory management scheme, combined with a plethora of optimizations [Gupta and Pontelli 1997; Pontelli et al. 1996], allows &ACE to be very effective in exploiting parallelism, even from rather fine-grained applications [Pontelli et al. 1995; Gupta and Pontelli 1997]. The performance of the system is on average within 5% of 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 led to various cases of superlinear 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)`[22] where the two subgoals may potentially compete (or cooperate) in the creation of a binding for the unbound variable `X`.

Unrestricted parallel execution of the above query (in Prolog) is likely to produce nondeterministic 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) observable Prolog semantics in this case. This amounts to

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

—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; that is, they differ in how they conservatively approximate such an undecidable property.

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

The question then arises whether dependent and-parallelism is fruitful at all. Typically in a query such as the above, p will produce a binding for X while q will process (or consume) it. If this order between production of the 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 a significant amount of computation before it needs the binding of X; this computation can be overlapped with computation of p, because it does not 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 use the following terminology. Unbound variables that 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 that appear in the tree. We 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. For example, 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 such as 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 article we focus on the DAP execution of Prolog programs, thus, the ultimate goal of the DAP execution models, as far as this article is concerned, is to speed up 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; and
(4) *backtracking:* Management of nondeterminism in the presence of DAP executions.

In the rest of this section, we deal with all these issues except for issue 2: management of subgoals does not present any new challenge with respect to 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 (as described in Section 4.1.1). This should come as no surprise: it was already mentioned that DAP is nothing more 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 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]. Techniques have also been proposed for detecting nonstrict independent and-parallelism at compile-time [Cabeza and Hermenegildo 1994]. This includes new annotation algorithms that use sharing and freeness information before and after each literal and new, specialized run-time tests. These techniques have been implemented in a practical parallelizer for the &-Prolog and &ACE systems by extending the original &-Prolog/&ACE PLAI parallelizer.

Automatic and semiautomatic detection of potential valid sources of unrestricted DAP in logic programs has been proposed and implemented in Pontelli et al. [1997], the implementation also being an extension of the &-Prolog/&ACE PLAI parallelizer. This proposal generates code annotations that are extensions of the CGE format (similar to those originally introduced by Shen [1992a] and used also in Hermenegildo et al. [1995]): 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,

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

$$\big(\langle Cond\rangle \Rightarrow \$and_goal\big(\theta_1, G_1^{\theta_1}\big)\ \&\ \ldots \&\ \$and_goal\big(\theta_n, G_n^{\theta_n}\big)\big), \ldots.$$

where

—$X_1, \ldots, X_m$ are the *shared* variables for subgoals $G_1, \ldots, G_n$, that is, 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 θ_j is a renaming substitution for the variables $X_i^j (1 \leq i \leq k_j)$, that is, a substitution that replaces each X_i^j with a brand new variable. This allows each subgoal in the conjunction to have 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, which will be evaluated at run-time (e.g., for checking groundness, independence, or comparing dynamically computed grain-sizes to thresholds).

A DAP-annotated version of the recursive clause in the program for *naive reverse* will look like

```
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 et al. [1997a] operates through successive refinements:

(1) identification of clauses having a structure compatible with the exploitation of DAP: that is, they contain at least one group of consecutive nonbuilt-in predicates. Each maximal group of contiguous and nonbuilt-in goals is called a *partition*;
(2) use of sharing and freeness [Cabeza and Hermenegildo 1994; Muthukumar 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, and
    —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; and
    —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. 1997; López-García et al. 1996; Tick and Zhong 1993], while the second one is dealt with in Pontelli et al. [1997a] through the use of *instantiation analysis*, based on the estimation of the size of the computation that precedes the binding of shared variables.

Further improvements have been devised in Pontelli et al. [1997a] through the use of sharing and freeness to detect at compile-time subgoals that will definitely bind dependent variables, that is, 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 certain 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*, that is, 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; that is, all the threads of computation access the same memory area that 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—for example, through a `get_structure` instruction—and successively the subterms of the binding are constructed. This makes the binding of the variable a nonatomic 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 during 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 of whether this will be assigned to a dependent variable.

Another solution has been proposed in *Andorra-I* [Santos Costa et al. 1996]; in this system, terms that 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 with 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. 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; and

—a check needs to be performed during each operation that constructs a term, in order to understand whether the term has been assigned to a dependent variable, 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.

Another 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 situation. 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 that 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 in such a way that Prolog semantics is respected. We can classify the different approaches according to certain orthogonal criteria [Pontelli 1997; Pontelli and Gupta 1997a, 1997b]:

(1) *validation time*: the existing proposals either
   (a) remove inconsistencies on binding shared variables only once a conflict appears and threatens Prolog semantics (*curative schemes*); or

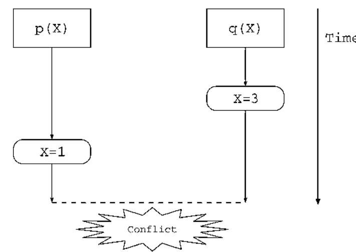


Fig. 19. Goal level curative approach.

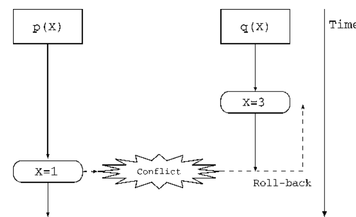


Fig. 20. Binding level curative approach.

   (b) prevent inconsistencies by appropriately delaying and ordering shared variable bindings (*preventive schemes*);
(2) *validation resolution*: the existing proposals either
   (a) perform the validation activity at the level of the parallel subgoals (*goal-level validation*); or
   (b) 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 19), this 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, such as 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 terminate in a time longer than the corresponding sequential ones; and
(3) it makes it extremely difficult to enforce Prolog semantics.

Performed at the binding level (see Figure 20), 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 [1987] and by Drakos [1989]. They both can be 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 models have various 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:

(1) *NonStrict Independent And-Parallelism (NSI) and Other Extended Notions of Independence:* The idea of these extensions of the notion of independence is to greatly extend the scope of independent and-parallelism while still ensuring correctness and efficiency/"no-slowdown" of the parallelization [Hermenegildo and Rossi 1995; Cabeza and Hermenegildo 1994]. The simplest concept of nonstrict independence allows execution of subgoals that have variables in common, provided *at most one subgoal can bind each shared variable.*[23] This kind of independence cannot be determined in general a priori (i.e., by inspecting the state of the computation prior to executing the goals to be parallelized) and thus necessarily requires a global analysis of the program. However, it is very interesting because it appears often in programs that manipulate "open" data structures, such as difference lists, dictionaries, and the like. An example of this is the following `flatten` example, which eliminates nestings in lists (`[X|Xs]` represents the list whose head is `X` and whose tail is `Xs` and `[]` represents the empty list):

```
flatten(Xs,Ys) :-
          flatten(Xs,Ys,[]).

flatten([], Xs, Xs).
flatten([X|Xs],Ys,Zs) :-
          flatten(X,Ys,Ys1),
          flatten(Xs,Ys1,Zs).
flatten(X, [X|Xs], Xs) :-
          atomic(X), X = [].
```

[23]The condition used in the case of impure goals is that the bindings of a goal will not *affect* the computation of the remaining subgoals to its right.

This program unnests a list without copying by creating open-ended lists and passing a pointer to the end of the list (`Ys1`) to the recursive call. Since this pointer is not bound by the first call to `flatten/3` in the body of the recursive clause, the calls to `flatten(X,Ys,Ys1)` and `flatten(Xs,Ys1,Zs)` are (nonstrictly) independent and *all the recursions can be run in parallel*. In fact, it is possible to detect this automatically [Cabeza and Hermenegildo 1994]. A number of (also correct and efficient) further generalizations of the concept of nonstrict independence have been proposed, based on notions of equivalence of search spaces [Hermenegildo and Rossi 1995; García de la Banda et al. 2000; García de la Banda 1994]. These ehancements allow goals to share variables and bind them, provided the bindings made by these goals are either *deterministic* (in a similar way to the "Andorra" models reviewed below) or *consistent* (in the constraint logic programming sense). These enhancements have allowed extending independence to logic programs with delays [García de la Banda et al. 1996b, 2000] and constraint logic programs [García de la Banda et al. 2000], as shown in Section 8.

(2) The *basic Andorra model* [Haridi 1990; Warren 1987a; Santos Costa et al. 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 a guarantee that there exists at most one single matching clause. In the basic Andorra model, subgoals can be executed ahead of their turn ("turn" in the sense of Prolog's depth-first search) in parallel *if they are determinate*, that is, if at most one clause matches the subgoal (the *determinate phase*). These determinate goals can be dependent on each other. If no determinate goal can be found for execution, a choice point is created for the leftmost goal in the goal list (the *nondeterminate phase*) and parallel execution of determinate goals along each alternative of the choice 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 that 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 nondeterminism to an otherwise committed choice language. In Pandora, clauses are classified as either "*don't-care*" or *"don't-know"*. As with 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, as 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 would 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:

(1) *Committed Choice Languages:* We only deal briefly with the notion of committed choice languages in this article, since they implement a semantics that is radically different from Prolog. Committed choice languages [Tick 1995] disallow (to a large extent) nondeterminism 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 [Shapiro 1987].

(2) *Binding-level nonstrict independence:* The application of the generalized (*consistency*- and *determinacy*-based) notions of independence [Hermenegildo and Rossi 1995; García de la Banda et al. 2000; García de la Banda 1994] at the finest granularity level—the level of individual bindings and even the individual steps of the constraint solver—has been studied formally in Bueno et al. [1994, 1998]. This work arguably represents the finest grained and "most parallel" model for logic and constraint logic programming capable of preserving correctness and theoretical efficiency proposed to date. While this model has not been implemented directly it serves as a theoretical basis for a number of other schemes.

(3) *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 that has access to such a variable. All remaining subgoals are restricted to read-only accesses to the shared variable (*consumers*); each attempt by a consumer to bind 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 [Shen 1992b].

    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* [Shen 1992b, 1996b] and the *filtered-binding model* [Pontelli and Gupta 1997a, 1997b] 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 et al. [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 because it allows a distributed implementation and it shares some commonalities with the filtered binding model presented in Section 5.5.3. Type-based optimizations of the approach have been proposed in Lamma et al. [1997].

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, and

—*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 et al. 1997b; Ranjan et al. 2000a]. 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 whether a binding for the variable is allowed. 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 the all the operations in the absence of shared variables aliasing.

The classification of the different models according to the complexity of the three key operations is illustrated in Figure 21. Unrestricted DAP means DAP with possible aliasing of unbound shared variables.

5.4 Backtracking

Maintaining Prolog semantics during parallel execution also means supporting nondeterministic computations, that is, 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 et al. 1991a]. This is largely due to the complexity of dealing with distributed backtracking. Nevertheless, it has been shown [Shen 1992b, 1996b] 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 nondeterministic goals.

Backtracking in the context of DAP is more complex than in the case of independent and-parallelism. While outside backtracking remains unchanged,

Fig. 21. Classification of DAP models.

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 (Section 4.1.3). 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 conjunction such as `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` that has just been untrailed.

Implementations of this scheme have been proposed in Shen [1992b,a] and 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 that support dependent and-parallelism. Some other systems that use dependent and-parallelism in conjunction with other forms of parallelism (e.g., ROPM) are described in Section 6. In this section we do not discuss committed choice languages: their sequential and parallel execution models have been described in detail in other surveys (e.g., Tick [1995] and Shapiro [1987]).

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, 1987c]. Due to its similarity to Aurora as far as or-parallelism is concerned, Andorra-I is able to use the schedulers built for Aurora. The current version of Andorra-I is compiled [Yang et al. 1993] and is a descendant of the earlier interpreted version [Santos Costa et al. 1991a].

As a result of exploitation of determinate dependent and-parallelism and the accompanying coroutining, not only can Andorra-I exploit parallelism from logic programs, but 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 nondeterminate—and execution of the nondeterminate 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, that is, after all determinate goals in the current goal list, irrespective of their order in this list, have been solved. Any goal that is nondeterminate (i.e., has more than one potentially matching clause) will be suspended in the determinate phase. Solving determinate goals early constrains the search space much more than using the standard sequential Prolog execution order (e.g., 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 nondeterminate 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 run-time [Santos Costa et al. 1996, 1991c].

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 et al. 1996, 1991c]. This is achieved by analyzing 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.[24]

The Andorra-I system speedsup execution in two ways: by reducing the number of inferences performed at run-time, and, 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 et al. 1991b; 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 is 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 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

[24]In spite of this, there are cases where Andorra-I and Prolog lead to different behavior; in particular, there are nonterminating Prolog programs that will terminate in Andorra-I and vice versa.

(i.e., each team behaves as a single Aurora worker). Determinate dependent and-parallelism is exploited by workers within a team; that is, workers within a team will cooperatively solve a goal along the or-branch picked up by the team. 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 et al. 1993, 1994b] and the PBA 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 [1991]. An extension of Andorra-I incorporating independent and-parallelism, called IDIOM, has also been proposed [Gupta et al. 1991]. Compile-time techniques have been used to allow automatic exploitation of nondeterminate independent and-parallelism in a system implementing the basic Andorra model [Olmedilla et al. 1993]. Work has also been done on implementing (the inference step reduction part of) the basic Andorra model by compilation into a standard Prolog system supporting delay declarations, with promising results [Bueno et al. 1995; Hermenegildo and CLIP Group 1994].

5.5.2 *DASWAM.* DASWAM [Shen 1992a, 1992b, 1996b] 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 [Shen and Hermenegildo 1994].

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 determined via a search operation, performed at the time of variable binding. Each dependent variable identifies the parcall frame that 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 regarding a suspended consumer is recorded. The process is illustrated in Figure 22. Each dependent cell maintains pointers to the parcall frame that introduced that dependent variable. In addition, 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 1992a; 1992b].

Efficient implementations of DASWAM on Sequent Symmetry, Sun Enterprise, and KSR-1 platforms have been developed [Shen 1996a; 1996b] the performance of the system has been validated on a large variety of benchmarks. Detailed performance analysis has been proposed in Shen [1992b].

5.5.3 *ACE.* The ACE system supports dependent and-parallelism using a method called the *Filtered Binding Model*. The Filtered Binding Model is

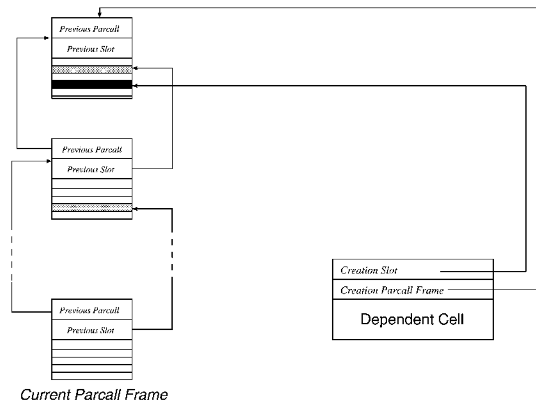


Fig. 22. DASWAM implementation.

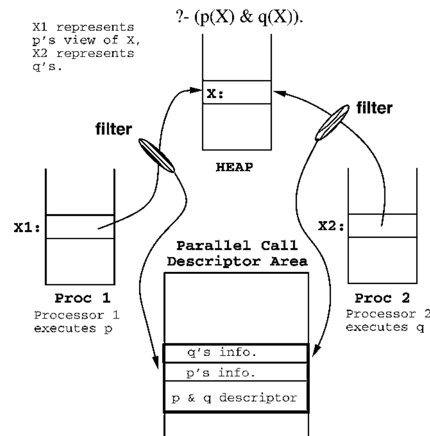


Fig. 23. The filtered binding model.

an instance of the class of models that use *preventive binding-level* validation. 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 23 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 a 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 nonconstant-time operation (the cost is always bounded by the number of dependent variables detected by the compiler in that parallel call[25]). 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 nonconstant overhead will be produced in the variable access operation; in such a case a nonconstant-time overhead is unavoidable. The current implementation, realized in the ACE system [Gupta et al. 1994a; Pontelli et al. 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 the 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, 1997b] 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 have 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

[25]We are also working under the assumption that the compiler marks goals for DAP execution conservatively; that is, 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.

(see also Section 10). Public domain parallel logic programming systems are available (e.g., YapOr [Rocha et al. 1999], KLIC [Chikayama et al. 1994], Ciao [Bueno et al. 1997], which includes &-Prolog, DASWAM [Shen 1996b]). For some time, a number of commercial parallel Prolog systems have also appeared on the market, such as 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 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 (exploiting multiple forms of parallelism) have been proposed, such as 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.[26] The implementation of the basic Andorra model [Haridi 1990; Warren 1987a], namely, Andorra-I [Santos Costa et al. 1991b] 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 the binding arrays technique [Lusk et al. 1990; Warren 1987c]. Likewise, the PEPSys model [Westphal et al. 1987; Baron et al. 1988], the AO-WAM [Gupta and Jayaraman 1993b], ROPM [Kalé 1985; Ramkumar and Kalé 1989, 1992], ACE [Gupta et al. 1994b, 1993], the PBA models [Gupta et al. 1994b, 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 from one another 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 1987b] 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 et al. 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 a 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

[26]Simulations have shown that indeed better speedups will be achieved if more than one source of parallelism is exploited [Shen 1992b; Shen and Hermenegildo 1991, 1996b].

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 antithetical requirements. 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:

(1) consider only pure Prolog programs for parallel execution; this was the approach taken by many early proposals, for example, AO-WAM [Gupta and Jayaraman 1993b] and ROPM [Kalé 1985]; or
(2) devise a new language that will allow extralogical features but in a controlled way, for example, PEPSys [Ratcliffe and Syre 1987; Westphal et al. 1987; Chassin de Kergommeaux and Robert 1990].

The disadvantage of 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. 1999; Ranjan et al. 2000a; Gupta et al. 1994a, 1994b; Santos Costa et al. 1991c].

Another issue that arises in systems that exploit independent and-parallelism is whether to *recompute* solutions of independent goals, or to *reuse* them. For example, consider the following program for finding "cousins at the same generation" taken from Ullman [1988],

```
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, a purely independent and-parallel, or a sequential implementation, the goal `parent(john, Yp)` will be recomputed for every solution to `parent(fred, Xp)`.[27] 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 cross-product (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 cross-product, assuming the cost of computing the cross-product is negligible.[28] However, for independent goals with very small granularity, the gain from solution sharing may be

[27]Respecting Prolog semantics, a purely independent and-parallel system can avoid recomputation of independent goals but most existing ones do not.
[28]This, as practice suggests, will not always be the case.

overshadowed by the cost of computing the cross-product, and so on, therefore, such goals should either be executed serially, or they should be recomputed instead of being shared [Gupta et al. 1993]. Independent goals that contain side-effects and extralogical predicates should also be treated similarly [Gupta et al. 1993; Gupta and Santos Costa 1996]. 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 would have to be recomputed. This is indeed the approach adopted by systems such as ACE [Gupta et al. 1994a] and the PBA model [Gupta et al. 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 et al. 1994b, 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 the 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, that is, determining whether an idle worker should perform or-parallel work or and-parallel work. The problem has been studied in depth by Dutra [1994, 1996]. The solution, which has been integrated in the Andorra-I system [Santos Costa et al. 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, that is, the percentage of time they spend 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 work 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 seem to scale up better with increasing number of workers, reducing idle time and number of reconfigurations.

6.3 Models for And/Or-Parallelism

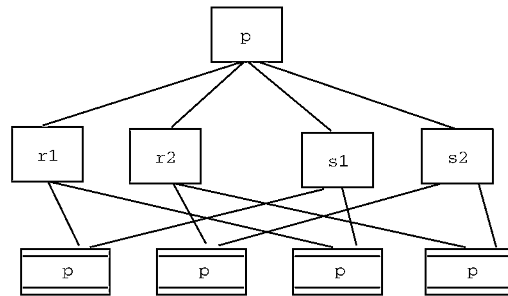
We now briefly describe the systems that combine more than one source of parallelism in logic programming.

6.3.1 *The PEPSys Model.* The PEPSys model [Westphal et al. 1987; Baron et al. 1988; Chassin de Kergommeaux and Robert 1990] combines and- and or-parallelism using a combination of techniques of timestamping 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 timestamped and stored in that process' hash window. Any PEPSys process can access the stacks and hash windows of its ancestor processes. The timestamp 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 must 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 24). 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 coexist 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 timestamping and hash windows for environment representation. This doesn't permit constant-time access to conditional variables. Therefore, access to conditional variables is expensive. However,



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. 24. Join cells.

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. dissertation [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 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:

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

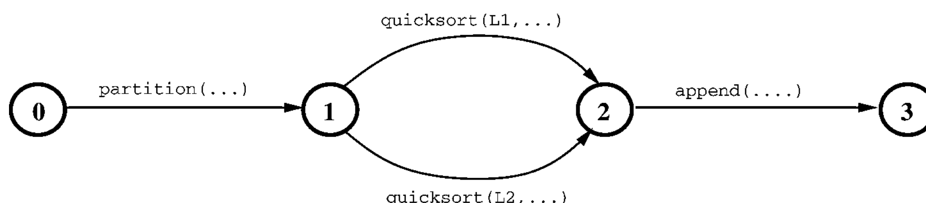


Fig. 25. An example data join graph.

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

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

(c) Any entry from the PSS of the reduce node can be added to the PSS of its parent or node. A substitution can be added to the PSS of a reduce node R representing a composite goal Q if it is a consistent composition of the substitutions, one for each literal of Q, from the PSSs 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 arc of the graph denotes a literal in the body of the clause (Figure 25).

DJGs are a means of expressing and-parallelism and are similar in spirit to Conery's dataflow 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 the 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 nonempty.

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 a process-based model rather than a stack-based one. As a result, there is no backtracking, and hence no memory reclamation that is normally associated with backtracking. Computing the join is an expensive operation since the actual bindings of variables have to be cross-produced to generate the tuple relations of the node (as opposed to using symbolic addresses to represent solutions, as done in PEPSys [Westphal et al. 1987] and AO-WAM [Gupta and Jayaraman 1993b]), 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.

The 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 et al. 1988b]. However, although ROLOG yields very good speedups, 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 1987a] 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, for example, OPAL [Conery 1992], where execution is governed by a set of and and or processes: such 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.* This model [Gupta and Jayaraman 1993b; 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, 1987c].

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 subtrees 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 descendant 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 26). Also, on switching nodes, a worker must update its binding array and base array with the help of the trail, as in Aurora.

6.3.4 *The ACE Model.* ACE (And/Or-parallel Copying-based Execution of logic programs) [Gupta et al. 1994a, Pontelli and Gupta 1997b] is another model that has been proposed for exploiting or- and independent and-parallelism simultaneously. ACE[29] 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 &ACE [Pontelli et al. 1995] 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 &ACE and MUSE, respectively. When or-parallelism and independent and-parallelism are present together, both are simultaneously exploited.

Multiple environments are maintained by stack copying as in MUSE. In ACE, available workers are divided into teams as in 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 as in &ACE until a choice point is created, at which point other teams may steal the untried alternatives from this choice point. Before doing so, the stealing team has to copy the appropriate stacks from the team from which the alternative was picked. When the choice point 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 choice point in the scope of a CGE is stolen by a team. To illustrate this, consider the case where a team selects an untried

[29]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.

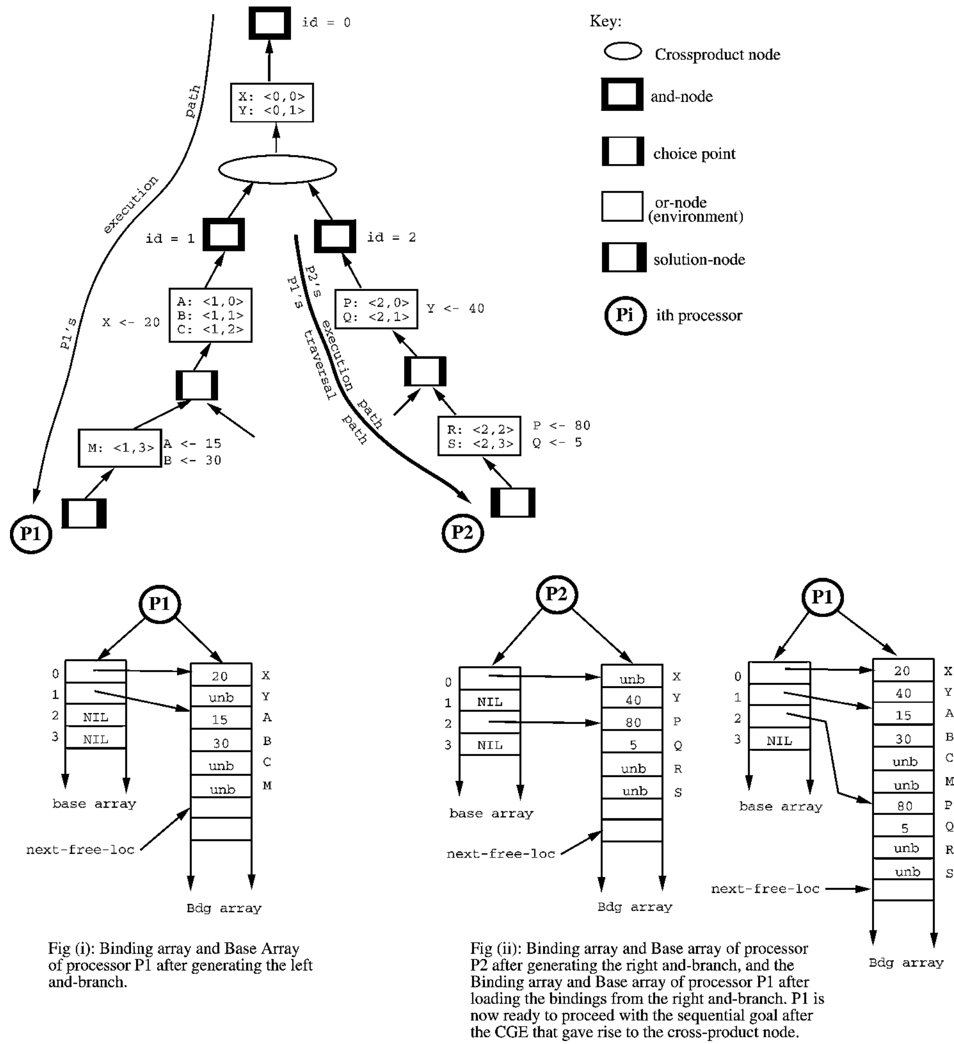


Fig (i): Binding array and Base Array of processor P1 after generating the left and-branch.

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. 26. Execution in the AO-WAM.

alternative from a choice point created during execution of a goal g_i inside the CGE (*true* $\Rightarrow$ $g_1\& \cdots \& g_n$). This team will copy all the stack segments in the branch from the root to the CGE including the *parcall* frame.[30] It will also have to copy the stack segments corresponding to the goals $g_1 \cdots g_{i-1}$ (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

[30]As mentioned earlier, the parcall frame [Hermenegildo 1986b] records the control information for the CGE and its independent and-parallel goals.

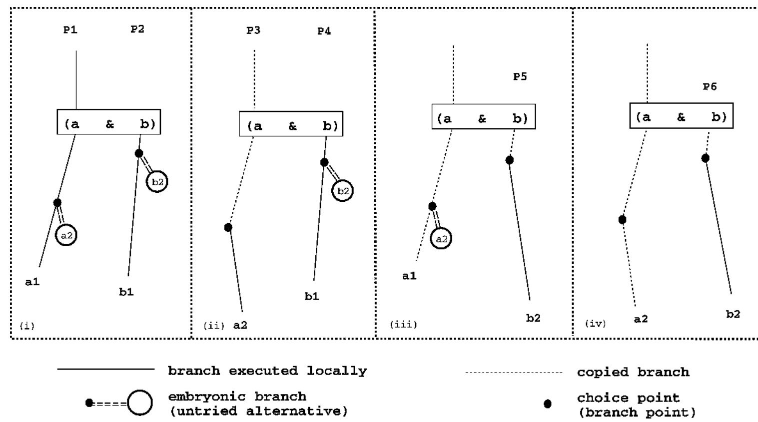


Fig. 27. Execution in ACE.

the CGE might bind a variable defined in one of the goals $g_1 \cdots g_{i-1}$ differently. The stack segments of the goal g_i from the CGE up to the choice point 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 choice point that is in the scope of the CGE only if goals to the left, i.e., $g_1 \cdots g_{i-1}$, 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 et al. 1993, 1994a; Gupta and Santos Costa 1996].

This is also illustrated in Figure 27. The four frames represent four teams working on the computation. The second team recomputes the goal b, while the third and fourth teams take 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 operation. In systems such as MUSE and ACE, this has been achieved by using memory mapping techniques that 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 that 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.

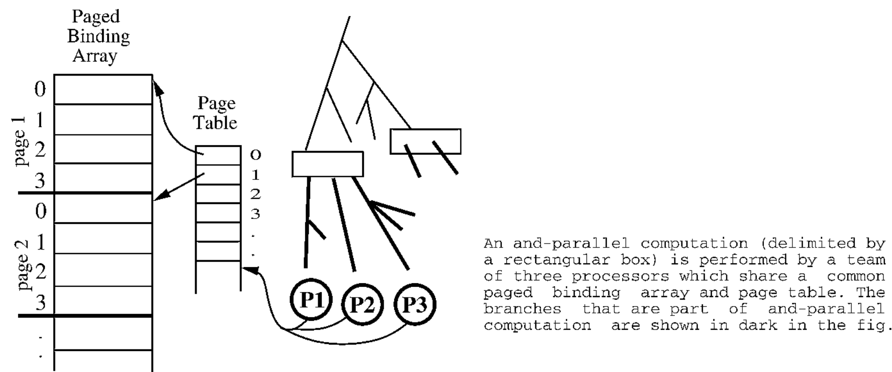


Fig. 28. The Paged Binding Array.

A simple solution to these issues has been recently proposed by V. Santos Costa [1999] in the *copy-on-write for logic programs (COWL)* methods. In COWL, each team occupies 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 them 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 also been extended to optimize the copying by avoiding wasted computation locally performed in the team and reusable after the copying operation (i.e., avoiding one team's copying 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/&ACE with MUSE, while preserving Prolog semantics. In a similar vein, one can combine &-Prolog/&ACE 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 (see Figure 28). 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 paged binding array and the page table are common to all workers on 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 et al. 1993; Gupta and Santos Costa 1996]. 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 choice point 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 choice point 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 [Gupat et al. 1994b].

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 et al. 1991a]. Other combined independent and- and or- parallel models have also been proposed [Biswas et al. 1988; Gupta et al. 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 very ambitious. Orthogonality implies that:

(1) each worker should be able to backtrack to a shared choice point and be aware only of or-parallelism;
(2) whenever a worker enters the public part of the or-tree, the other workers on 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 29.

Let us assume the following configuration.

(1) 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 is used before choice point *C1* while page 3 is used after that choice point, and worker *W1,2* makes use of page 2.
(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

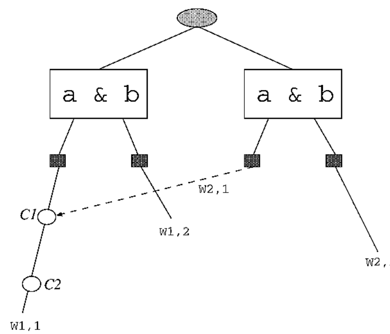


Fig. 29. Lack of orthogonality in PBA.

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 team 1 after the choice point *C1*. But for team 2, 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. Some of the schemes proposed are

—the *shared paged binding array (SPBA)* [Gupta et al. 1994b] which extends the PBA scheme by requiring the use of a global and shared paged binding array;

—the *sparse binding array* [Correia et al. 1997] in which each conditional variable is guaranteed to have a binding array index that is unique in the whole computation tree and relies 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 of these three schemes has been presented in Santos Costa et al. [2000].

6.3.8 *The Extended Andorra Model.* The extended Andorra model (EAM) [Warren 1987a; Haridi and Janson 1990; Gupta and Warren 1992] and the Andorra Kernel Language (AKL) (later renamed Agent Kernel Language) [Haridi and Janson 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 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 that 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 that 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 "nondeterminate 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 leftmost occurrence of that variable (as in the Basic Andorra Model). An alternative way of looking at the EAM is to view it as an extension of the basic Andorra model where nondeterminate 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" goals `p`, `q`, and `r` (this is depicted by existential quantification of `X` and `Y` in Figure 30). 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 *nondeterminate* promotion step will have to be performed. The nondeterminate promotion step will match all alternatives of `p`

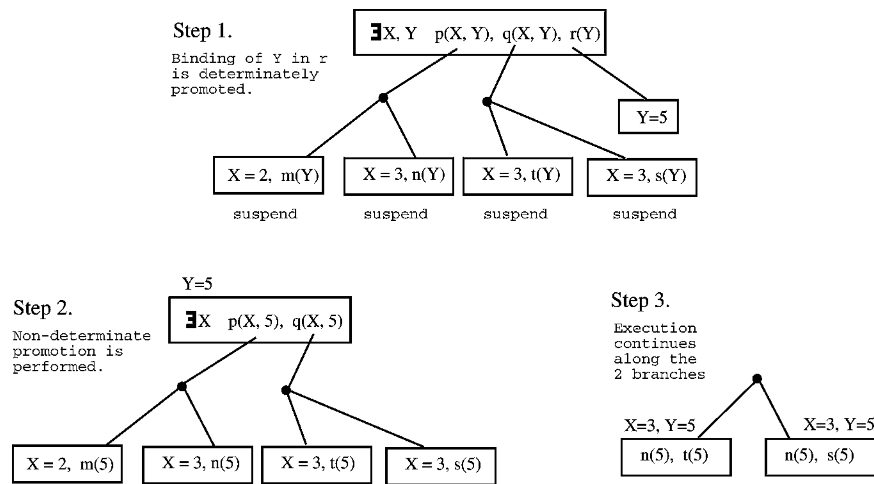


Fig. 30. Execution in EAM.

with those for q, resulting in only two combinations remaining active (the rest having failed because of nonmatching bindings of X). These steps are shown in Figure 30.

The above is a very coarse description of the EAM; a full description of the model is beyond the scope of this article. More details can be found elsewhere [Warren 1987a; Haridi and Janson 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 nondeterminate 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 EAM subsumes both the committed choice logic programming (with nonflat as well as flat guards) and nondeterministic logic programming, that is, general Prolog.

The generality and the power of the Extended Andorra Model make 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 the Swedish Institute of Computer Science [Janson and Montelius 1991]. A parallel implementation has also been undertaken by Moolenaar and Demoen [1993]. A very efficient parallel implementation of AKL has been proposed by Montelius in the Penny system [Montelius 1997; Montelius and Ali 1996]. This implementation combines techniques from or-parallelism and committed choice languages. Although AKL includes nondeterminism, it differs from Prolog both in syntax and semantics. However, automatic translators that transform Prolog programs into AKL programs have been constructed [Bueno and Hermenegildo 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 et al. 1998; Popov 1997].

More faithful models to support the execution of the EAM have been recently described and are currently under implementation, for example, the BEAM model [Lopes and Santos Costa 1999]. The literature also contains proposals of extensions of Prolog that try to more naturally integrate an EAM style of computation. One example is represented by the *Extended Dynamic Dependent* scheme [Gupta and Pontelli 1999a]. 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 VERSUS 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 that are simultaneously applied to a large set of data. This pattern of execution occurs often in logic programs, as exemplified by frequently used predicates such as the following (simplified) `map` predicate,

```
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 dataflow, 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 with respect to 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 speedups.

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 the presence of certain instances of data or-parallelism.

7.2 Data And-Parallelism

The idea of data parallel execution can also be 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, that is, 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,

$$\begin{array}{l} \mathrm{p}(\bar{X}) \ :- \ \Delta. \\ \mathrm{p}(\bar{X}) \ :- \ \Phi, \ \ \mathrm{p}(\bar{X}'), \Psi. \end{array}$$

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

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

where Φ_i and Ψ_i are the instances of goals Φ and Ψ obtained at the ith level of recursion. This clause can be executed by first running, in parallel, the goals $\Phi_1, \ldots, \Phi_n$, then executing $\mathrm{p}(\bar{b})$ (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 of the need to perform complex backtracking on parallel calls. Compile-time analysis tools have been proposed 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 et al. 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., φ) for the different values in the domain of the quantifiers (e.g., the different values in the set S).

Both traditional and-parallelism and data-parallelism offer advantages and disadvantages. Traditional and-parallel models offer generality, being able to exploit parallelism in a large class of programs (including the parallelism exploited by data parallelism techniques). Data and-parallelism techniques on the other hand offer increased performance for a restricted class of programs. As a result, various authors have worked on integrating data and-parallelism

into more traditional and-parallelism schemes [Debray and Jain 1994; Pontelli and Gupta 1995a; Hermenegildo and Carro 1996]. The basic idea is to identify instances of data and-parallelism in generic and-parallel programs, and to use specialized and more efficient execution mechanisms to handle these cases within the more general and-parallel systems. These techniques have been shown to allow obtaining the advantages of both types of parallelism within the same system.

8. PARALLEL CONSTRAINT LOGIC PROGRAMMING

Although the main focus of this survey is parallel execution of Prolog programs, we briefly overview in this section the most relevant efforts that 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 and, on the other hand, the study of parallelism in CLP has led to important generalizations in the concepts and techniques developed for traditional logic programming.

8.1 Or-Parallel Constraint Logic Programming

A parallel implementation of Chip [Van Hentenryck 1989] 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 (which is in effect a form of data or-parallelism). The results reported in Van Hentenryck [1989] are encouraging, and prove that or-parallel techniques are also quite suitable in the context of CLP execution. Experiments in the parallelization of ECLiPSe using a recomputation-based approach have also been presented [Mudambi and Schimpf 1994]. In Gregory and Yang [1992] finite domain constraint solving operations are mapped to the parallel execution mechanisms of Andorra-I.

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 *nondeterministic* 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 vectors 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].

Other recent work studies the direct parallelization of the sources of nondeterminism inherent in the operational semantics of CLP solvers. The work in Pontelli and El-Kathib [2001] presents a methodology for exploring in parallel the alternative elements of a constraint domain, while Ruiz-Andino et al. [1999] revisit the techniques used to parallelize arc-consistency algorithms (e.g.,

parallel AC3 [Samal and Henderson 1987] and AC4 [Nguyen and Deville 1998]) and apply them to the specific case of indexical constraints in CLP over finite domains. Similar work exploring interactions between search strategies in constraint logic programming and parallelism has also been presented [Schulte 2000; Perron 1999].

8.2 And-Parallel Constraint Logic Programming

An interesting issue that appears in the context of and-parallel constraint logic programming is that the traditional notions of independence do not hold. Consider, for example, the parallelization of two procedure calls `p(X),q(Z)` in the following situations,

(a) `main :- X > Y, Z > Y, p(X) & q(Z), ...`

(b) `main :- X > Y, Y > Z, p(X) & q(Z), ....`

In case (a), the store contains `(X>Y,Z>Y)` before calling `P` and `q`, whereas, in case (b), the store contains `(X>Y,Y>Z)`. The simple pointer aliasing reasoning implied by the definition of strict independence does not apply directly. However, `p` cannot in any way affect `q` in case (a), while this could be possible in case (b), that is, the two calls are clearly independent in case (a) while they are (potentially) dependent in case (b).

Notions of independence that apply to general constraint programming (and can thus deal with the situation above) have been proposed by García de la Banda et al. [2000] and García de la Banda [1994]. For example, two goals `p` and `q` are independent if all constraints posed during the execution of `q` are consistent with the output constraints of `p`.[31] The following is a sufficient condition for the previous definition but which only needs to look at the state of the store prior to the execution of the calls to be parallelized (e.g., using run-time tests that explore the store (c)), in the same spirit as the strict-independence condition for the Herbrand case. Assuming the calls are $p(\bar{x})$ and $q(\bar{y})$ then the condition is:

$$(\bar{x} \cap \bar{y} \subseteq \mathit{def}(c)) \text{ and } (\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow \ \exists_{-\bar{y} \cup \bar{x}} c),$$

where $\bar{x}$ is the set of arguments of `p`, def(c) is the set of variables constrained to a unique value in c, and $\exists_{-\bar{x}}$ represents the projection of the store on the variables $\bar{x}$ (the notion of projection is predefined for each constraint system). The first condition states that the variables which are shared between the goals in the program text must be bound at run-time to unique values. The second condition is perhaps best illustrated through an example. In the two cases above, for (a) $c = \{\mathtt{X} > \mathtt{Y}, \mathtt{Z} > \mathtt{Y}\}$ we have $\exists_{-\{\mathtt{X}\}} c = \exists_{-\{\mathtt{Z}\}} c = \exists_{-\{\mathtt{X},\mathtt{Z}\}} c = \mathit{true}$ and therefore `p` and `q` are independent. For (b) $c = \{\mathtt{X} > \mathtt{Y}, \mathtt{Y} > \mathtt{Z}\}$ we have $\exists_{-\{\mathtt{X}\}} c = \exists_{-\{\mathtt{Z}\}} c = \mathit{true}$ while $\exists_{\{\mathtt{X},\mathtt{Z}\}} c = \mathtt{X} > \mathtt{Z}$ and therefore `p` and `q` are not independent. While checking these conditions accurately and directly can be

[31]As mentioned earlier, this actually implies a better result even for Prolog programs since its projection on the Herbrand domain is a strict generalization of previous notions of nonstrict independence. For example, the sequence `p(X), q(X)` can be parallelized if `p` is defined, for example, as `p(a)` and `q` is defined as `q(a)`.

inefficient in practice, the process can be approximated at compile-time via analysis or at run-time via simplified checks on the store. A first and-parallel CLP system, based on an extension of the &-Prolog/Ciao system, and using the notions of independence presented has been reported in Garcia de la Banda et al. [1996a, 2000], showing promising performance results. Also, as mentioned earlier, applying the notions of constraint independence at the finest granularity level–the level of individual bindings and even the individual steps of the constraint solver–has been studied formally in Bueno et al. [1994b, 1998], leading to what is believed to be the "most parallel" model for logic and constraint logic programming proposed to date that preserves correctness and theoretical efficiency.

Another reported proposal is 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 [1989]. GDCC provides two levels in the exploitation of parallelism: 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; and gdcc has been interfaced to a number of constraint solvers (e.g., algebraic solvers for nonlinear equations), which are themselves capable of solving constraint in parallel.

9. IMPLEMENTATION AND EFFICIENCY ISSUES

Engineering an efficient, practical parallel logic programming system is by no means an easy task.[32] There are numerous issues one has to consider; some general ones are discussed below.

9.1 Process-Based Versus 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 [1987b] And-Or Process Model 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 somewhat more suited for implementation on nonshared memory MIMD processors,[33] at least from a conceptual point of view, since different processes can be mapped to different processors at run-time quite easily.

In processor-based (or "multisequential") approaches, multiple threads are created ahead of time that run in parallel to produce answers to the top level

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

[33]Some more process-based proposals for distributed execution of logic programs can be found in Kacsuk [1990].

query by being assigned parts of the computation, and, typically, each thread is a WAM-like processor. Examples of processor-based systems are &-Prolog, Aurora, MUSE, Andorra-I, PEPSys, AO-WAM, DDAS, ACE, PBA, and so on. Processor-based systems are more suited for shared memory machines, although techniques such as stack copying and stack splitting show a high degree of locality in memory reference behavior and hence are suited for nonshared memory machines as well [Ali 1988; Ali et al. 1992; Gupta and Pontelli 1999c]. 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 nonshared memory machine using stack copying [Villaverde et al. 2001; Gupta et al. 1992]. Alternatively, one could employ scalable virtual shared memory architectures that have been proposed [Warren and Haridi 1988] and built (e.g., KSR, SGI Origin, IBM NUMA-Q).

Ideally, a parallel logic programming system is expected to satisfy the following two requirements [Hermenegildo 1986b; Gupta and Pontelli 1997; Pontelli 1997].

—On a single processor, the performance of the parallel system should be comparable to sequential logic programming implementations (i.e., there should be limited slowdown compared to a sequential system). To this end, the parallel system should be able to take advantage of the sequential compilation technology [Warren 1983; Aït-Kaci 1991; Van Roy 1990] that has advanced rapidly in the last two decades, and thus the basic implementation should be WAM-based.

—Parallel task creation and management should introduce a small overhead (which implies using a limited number of processes and efficient scheduling algorithms).

Systems such as &-Prolog, Aurora, MUSE, and ACE indeed get very close to achieving these goals. Experience has shown that process-based systems 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 that is enforced by the concurrent semantics of the language); indeed the fastest parallel implementations of committed choice languages (e.g., Crammond [1992] and Rokusawa et al. [1996]) rely on a processor-based implementation. In the context of Prolog, the presence of backtracking makes the process model too complex for nondeterministic parallel logic programming. Furthermore, the process-based approaches typically exploit parallelism at a level that is too fine-grained, resulting in high parallel overhead and unpromising absolute performances (but good speedups 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 [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, and the like, 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 rules always hold in a traditional sequential system.

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

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

As a result of these two rules, space is always reclaimed from the top of the stacks during backtracking in logic programming systems which perform a depth-first search of the computation tree, as Prolog does.

However, as shown in Lusk et al. [1990], Ali and Karlsson [1990b] and Hermenegildo [1987], in parallel logic programming systems things are more complicated. First, these rules may not hold: 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, which results in the LIFO nature of stacks being destroyed.

There are many possible solutions to these problems [Hermenegildo 1986b; Pontelli et al. 1995; Shen and Hermenegildo 1994, 1996a]. 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 goal descriptors—corresponding to a given goal becoming noncontiguous). Garbage slots created are marked as such, and are reclaimed when everything above

them has also turned into garbage. This technique is also 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 et al. [1985], Appleby et al. [1988], Older and Rummell [1992], and Bekkers et al. [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 indicate 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 [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 [1995], which extends the work by Ali into a generational copying garbage collector (objects are divided into generations, where newer generations contain objects more recently created; the new generation is garbage collected more often than 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 et al. 1990; Xu et al. 1989].

9.3 Optimizations

A system that builds an and-or tree to solve a problem with nondeterminism 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 slowdown 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 nondeterministic 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], and Pontelli et al. [1996]). Nevertheless, limited effort has been placed in determining overall principles that 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 and Pontelli [1997]. The proposal presents a number of general optimization principles that can be employed by implementors of parallel nondeterministic 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 et al. 1996] (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 predetermined 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 dataflow (e.g., variable 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 nondeterminism 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 nondeterminism 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 et al. 1996].

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 preserves the desired execution behavior). This principle has been implemented, for example, in the *Backtracking Families Optimization* [Pontelli et al. 1996];
(2) avoiding allocation of unnecessary structures: most of the new data structures introduced in a parallel computation serve two purposes: to support the management of the parallel parts of the computation, and to support the management of nondeterminism. This principle has been implemented in various optimizations, including the *Shallow Backtracking Optimization* [Carlsson 1989] and the *Shallow Parallelism Optimization* [Pontelli et al. 1996].

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 that 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 nondeterministic 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 disallowed, then the scheduler has to take this into account and at any moment schedule only those goals meeting 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 1994, 1996; Ueda and Montelius 1996].

9.5 Granularity

The implementation techniques mentioned before for both or- and and-parallelism have proven sufficient for keeping the overheads of communication, scheduling, and memory management low and obtaining significant speedups in a wide variety of applications on shared memory multiprocessors (starting from the early paradigmatic examples: the Sequent Balance and Symmetry series). However, current trends point towards larger multiprocessors but with less uniform shared memory access times. Controlling in some way the granularity (execution time and space) of the tasks to be executed in parallel can be a useful optimization in such machines, and is in any case a necessity when parallelizing for machines with slower interconnections. The latter include, for example, networks of workstations or distribution of work over the Internet. 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.

The idea of granularity control is to replace parallel execution with sequential execution or vice versa based on knowledge (actual data, bounds, or estimations) of task size and overheads. The problem is challenging because, while the basic communication overhead parameters of a system can be determined experimentally, the computational cost of the tasks (e.g., procedure calls) being parallelized, as well as the amount of data that needs to be transferred before and after a parallel call, usually depend on dynamic characteristics of the input data. In the following example, we consider for parallel execution `q` (which, assuming it is called with `X` bound to a list of numbers, adds one to each element of the list);

```
..., r(X) & q(X,Y), ...

q([],[]).
q([I|Is],[I1|Os]):- I1 is I+1, q(Is,Os).
```

The computational cost of a call to q (and also the communication overheads) are obviously proportional to the number of elements in the list. The characterization of input data required has made the problem difficult to solve (well) completely at compile-time.

The Aurora and MUSE or-parallel systems keep track of granularity by tracking the *richness* of nodes, that is, 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. The Aurora and MUSE systems also make a distinction between the *private* and *public* parts 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 is a part of the private region of some worker made public. In these systems, granularity control is completely performed at run-time.

Modern systems [López-García et al. 1996; Shen et al. 1998; Tick and Zhong 1993] implement granularity control using the two-phase process proposed in Debray et al. [1990] and López-Gercía et al. [1996]:

(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, as a function of certain measures of input data;
(2) at *run-time* the cost estimates are instantiated, before execution of the task and compared with predetermined thresholds; parallel execution of the task is allowed only if the cost estimate is above the threshold.

Programs are then transformed at compile-time into semantically equivalent counterparts but which automatically control granularity at run-time based on such functions, following the scheme,

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

where the m subgoals will be allowed in a parallel execution only if the result of the *cost_estimate* is above the threshold τ. The parameters of *cost_estimate* are those goal input arguments that directly determine the time-complexity of the parallel subgoals, as identified by the global analysis phase. In the example above, these tools derive cost functions such as, for example, $2 * \mathit{length}(X) + 1$ for q (i.e., the unit of cost is in this case a procedure call, where the addition is counted for simplicity as one procedure call). If we assume that we should parallelize when the total computation cost is larger than "100," then we can transform the parallel call to p and q above into:

```
..., Cost=2*length(X)+1, (Cost>100 -> r(X) & q(X,Y)
                                     ;  r(X) , q(X,Y)), ...
```

(using an if-then-else). Clearly, many issues arise. For example, the cost of performing granularity control can be factored into the decisions. The cost functions can be simplified and related back to data structure sizes, list length in the case above; that is, the call will only be parallelized if the length of the list is larger than a statically precomputed value:

```
..., (length_greater_than(X,50) -> r(X) & q(X,Y)
                                 ;  r(X) , q(X,Y)), ...
```

This in turn has inspired the development of algorithms for keeping track of data sizes at run-time [Hermenegildo and López-García 1995]. As another example, a modified annotation for the recursive clause of Fibonacci may look like:

```
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).

Also, the same techniques used for cost bounding allow deriving upper and lower bounds on the sizes of the structures being passed as arguments [López-García et al. 1996]. This information can be factored into parallelization decisions (it affects the threshold). For example, in the example above, the argument size analysis (assuming that C is the cost of sending one element of a list, and a distributed setting where data are sent and returned eagerly) will infer that the communication cost is $2 * length(X) * C$. Interestingly, the $Computation > Overhead$ condition $(2 * length(X) + 1 > 2 * length(X) * C)$ can be determined statically to be always true (and parallelize unconditionally) or false (and never parallelize) depending only on the value of C, which in turn can perhaps be determined experimentally in a simple way. Performance improvements have been shown to result from the incorporation of this type of grain size control, especially for systems with medium to large parallel execution overheads [López-García et al. 1996].

Clearly, there are many interesting issues involved: techniques for derivation of data measures, data size functions, and task cost functions, program transformations, program optimizations, and so on. Typically, the techniques are proved correct, again typically using the notions of approximation and bounding, formalized as abstract interpretations. The key problem is clearly the automatic derivation of the functions that bound the time-complexity of the given tasks. The first proposals in this regard are those made by Debray et al. [1990] and Tick and Zhong [1993]. Both schemes are capable of deriving cost estimations that represent upper bounds for the time-complexity of the selected tasks.

The use of upper bounds is suboptimal 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 [Debray et al. 1994]. For this reason more recent efforts have focused on the derivation of lower-bound estimates [Debray et al. 1997; King et al. 1997]. A very effective implementation of some of these techniques, both for and- and or-parallelism, have been realized in the GraCos system [Debray et al. 1990; López-García et al. 1996]. This system adds mode and type analysis to the "upper-bounds" CASLOG system [Debray and Lin 1993] (and modifies it to compute lower bounds following Debray et al. [1997b]) and has been integrated in the Ciao logic programming system [Hermenegildo et al. [1999a]. Lower-bound analysis is considerably more complex than upper-bound analysis. First of all, it requires the ability to determine properties of tasks with respect to failure [Debray et al. 1997]. 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 that is guaranteed to not fail, and

—$h(n)$ 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 the 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. [1997] the problem is reduced to the computation of the chromatic polynomial of a graph.

In King et al. [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 et al. [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 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 *postmortem* 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 et al. 1997]. The existing systems can be distinguished according to the following criteria.

—*Static Versus 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 timestamps of the trace events.

—*Global Versus Local:* Global visualization tools provide a single representation that captures all the different observable characteristics of the parallel execution; local visualization tools instead allow the user to focus on specific characteristics.

The first visualization tools for parallel logic programs were developed for the Argonne Model [Disz and Lusk 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 31 shows a snapshot of Must: circles denote choice points and the numbers denote the position of the workers in the computation tree.

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 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[34]—and they are aimed at producing a static representation of the distribution of work among the available workers. Figure 32 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

[34]VisAndOr, however, can depict Andorra-I executions: that is, or-parallelism and deterministic dependent and-parallelism.

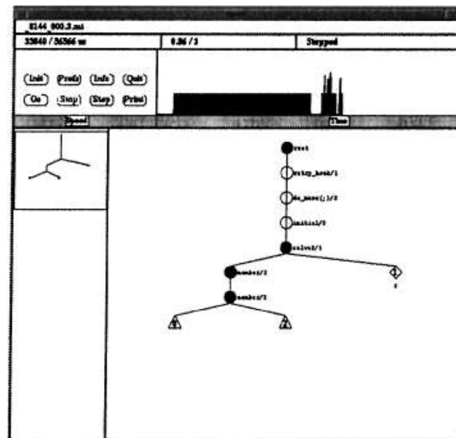


Fig. 31. Snapshot of Must.

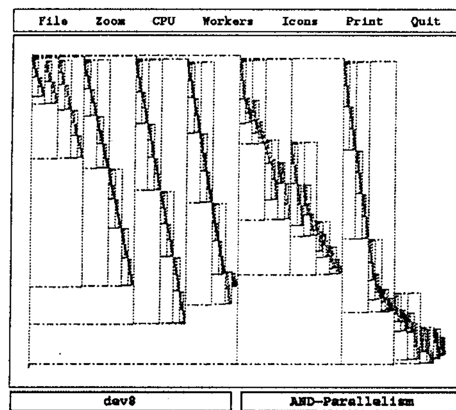


Fig. 32. Snapshot of VisAndOr.

the trace format adopted by various other systems [Vaupel et al. 1997; Kusalik and Prestwich 1996; Fonseca et al. 1998]. Must and VisAndOr have been integrated in the ViMust system; a timeline 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, for example, VACE [Vaupel et al. 1997], based on the notion of C-trees [Gupta et al. 1994], and VisAll [Fonseca et al. 1998]. Figure 33 shows a snapshot of VACE.

A final note is for the *VisAll* system [Fonseca et al. 1998]. VisAll provides a universal visualization tool that subsumes the features offered by most of

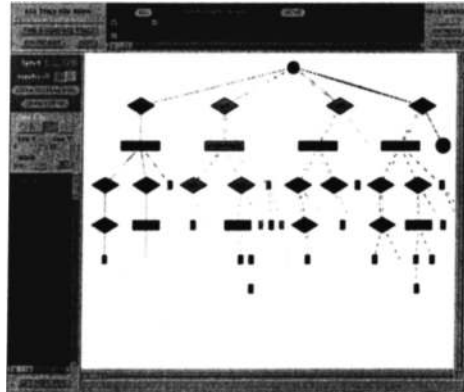


Fig. 33. Snapshot of VACE.

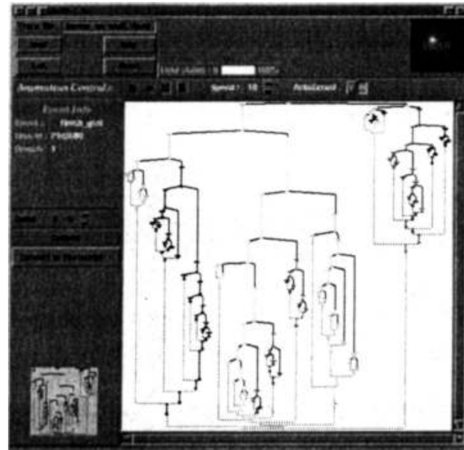


Fig. 34. Snapshot of VisAll.

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 34 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 scheduling solutions.

9.7 Compile-Time Support

As should be clear at this point, compile-time support is crucial for the 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. Without attempting to

be exhaustive, we point out some examples. 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 1992, 1991; Jacobs and Langen 1992]. ACE makes use of abstract interpretation techniques to build extended CGEs for dependent and-parallelism [Pontelli et al. 1997a] The Andorra-I system relies on determinacy analysis done at compile-time for detecting determinacy of goals at run-time [Santos Costa et al. 1991c; 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, for example, detection of determinacy, generation of CGEs, and the like. Compile-time analysis has also been used for transforming Prolog programs into AKL programs [Bueno and Hermenegildo 1992], and has also been used for supporting Prolog semantics in parallel systems that contain dependent and-parallelism, such as Andorra-I [Santos Costa et al. 1991c]. 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], and so on.

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 nonspeculative regions at compile-time, detecting whether a side-effect will ever be executed at run-time, detecting producer and consumer instances of variables, detecting whether a variable is conditional, and so on. 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 programs 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 a 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.

(1) Hermenegildo and Tick [1989; Tick 1987] proposed various studies estimating the performance of and-parallel systems on shared memory machines taking into account different cache coherence algorithms, cache sizes, bus widths, and so on. These early studies allowed predicting, for example, that &-Prolog would later produce speedups over state of the art sequential systems even on quite fine-grained computations on shared-memory machines that were not commercially available at the time.

(2) Montelius and Haridi [1997] have proposed detailed performance analysis of the Penny system, mostly using the SIMICS Sparc processor simulator;

(3) Gupta and Pontelli [1999c] 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;
(4) Santos Costa et al. [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, for example, distributed shared memory machines). In Santos Costa et al. [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 et al. 2000].
(5) The impact of cache coherence protocols on the performance of parallel Prolog systems has been studied in more detail in Dutra et al. [2000], Silva et al. [1999], and Calegario and Dutra [1999].

These works tend to agree on the importance of considering architectural parameters in the design of parallel logic programming systems. For example, the results achieved by Costa et al. [1997] 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 et al. [2000] indicates a high-risk of false-sharing in the presence of blocks larger than 64 bytes;
—Dutra et al. [2000] compare the effect of Write Invalidate versus. Write Update as cache coherence protocols. The study confirms the early results of Hermenegildo and Tick [1989] and Tick [1987] and extends them underlining the superiority of a particular version of the Write update algorithm (a 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 that a large body of research has been devoted to 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 presented in Shen and Hermenegildo [1991, 1996b]. This work considered a comparatively large pool of applications 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 form of parallelism; that is, 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.

—Many applications 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 speed up execution of various large real-life applications. These include:

—independent and dependent and-parallelism have been successfully extracted from Prolog-to-WAM compilers (e.g., the PLM compiler) [Pontelli et al. 1996];

—and-parallelism has been exploited from static analyzers for Prolog programs [Hermenegildo and Greene 1991; Pontelli et al. 1996];

—natural language processing applications have been very successfully parallelized extracting both or- and and-parallelism, for example, the Chat system [Santos Costa et al. 1991a; Shen 1992b], the automatic translator Ultra [Pontelli et al. 1998], and the word-disambiguation application Artwork [Pontelli et al. 1998];

—computational biology applications: for example, Aurora has been used to parallelize Prolog applications for DNA sequencing [Lusk et al. 1993];

—both Aurora and ACE have been applied to provide parallel and concurrent backbones for Internet-related applications [Szeredi et al. 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. Andorra-I has also been used in a variety of other telecommunication applications [Crabtree 1991; Santos Costa et al. 1991b];

—Aurora has been used to develop a number of concrete applications. Particularly important are those developed in the context of the Cubiq project:

  (1) the EMRM system, a medical record management system, which supports collection of medical information following the SOAP medical knowledge model [Szeredi and Farkas 1996]; and
  (2) the CONSULT credit rating system, which makes use of rule-based specification of credit assessment procedures [IQSoft Inc. 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 article 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 challenges 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 and presented a series of theoretical results ranging from formal notions of independence to limits on implementation efficiency. Parallel logic programming is a challenging area of research and will continue to be so until the objective of *efficiently* exploiting all sources of parallelism present in logic programs in the most cost-effective way is realized. This objective involves challenges at many levels, from run-time systems and execution models to compile-time technology and support tools.

From the point of view of run-time systems and execution models it can be argued that, when compared with work done in other fields, particularly strong progress has been made in the context of logic programming in abstract machines, efficient task representation techniques, dynamic scheduling algorithms, and formal definition of the advanced notions of independence (and guaranteed no-slowdown conditions) that are needed to deal with the irregularity and speculation occuring in search-based applications. As a result, the current state of the art is that there are very efficiently engineered systems such as &-Prolog and &ACE for independent and-parallelism, Aurora, MUSE, YAP, and ElipSys for or-parallelism, DASWAM and ACE for dependent and-parallelism (and some efficient implementations of committed choice languages [Shapiro 1987; Hirata et al. 1992]) which have been proved successful at achieving speedups over the state of the art sequential implementations available at the time of their development.

The systems mentioned above exploit a single form of parallelism. A few systems exist that efficiently exploit more than one source of parallelism (e.g., Andorra-I) although new promising ones are currently being designed and built [Gupta et al. 1994b; Correia et al. 1997; Santos Costa 1999]. However, no system exists that efficiently exploits *all* sources of parallelism present in logic programs. Efforts are already under way to remedy this [Montelius 1997; Santos Costa 1999; Gupta et al. 1994b; Pontelli and Gupta 1997b; Correia et al. 1997; Castro et al. 1999] and we believe that this is one of the areas in which much of the research in parallel logic programming may lie in the future. One

approach to achieving this goal, inspired by the duality [Pontelli and Gupta 1995b] and orthogonality [Correia et al. 1997] principles and by views such as those argued in Hermenegildo and CLIP Group [1994], would be to configure an ideal parallel logic programming system as a true "plug-and-play" system, where a basic Prolog kernel engine can be incrementally extended with different modules implementing different parallelization and scheduling strategies, and the like (as well as other functionality not related to parallelism, of course) depending on the needs of the user. We hope that with enough research effort this ideal can be achieved.

From the point of view of compile-time technology, the result of the work outlined in previous sections is that quite robust *parallelizing compilers* exist for various generalizations of independent and dependent and-parallelism, which automatically exploit parallelism in complex applications. The accuracy, speed, and robustness of these compilers have also been instrumental in demonstrating that *abstract interpretation* provides a very adequate framework for developing provably correct, powerful, and efficient global analyzers and, consequently, parallelizers. It can be argued that, when compared with work done in other fields, particularly strong progress has been made in the context of logic programming in developing techniques for interprocedural analysis and parallelization of programs with dynamic data structures and pointers, in parallelization using conditional dependency graphs (combining compile-time optimization with run-time independence tests), and in domains for the abstraction of the advanced notions of independence that are needed in the presence of speculative computations. More recently, independence notions, analysis techniques, and practical tools have also been developed for the parallelization of constraint logic programs and logic programs with dynamic execution reordering ("delays") [García de la Banda et al. 2000].

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 can potentially constitute a viable solution to this problem. However, considerable research in the design and implementation of parallel logic programming systems on distributed memory multiprocessors is still needed before competitive speedups can be obtained routinely. Distributed implementation of parallel logic programming systems is another direction where we feel future research effort should be invested. There are many challenges in the efficient implementation of distributed unification and maintaining program-coordinated execution state and data economically in a noncentralized way, as well as in the development of adequate compilation technology (e.g., for granularity control). Fortunately, this is an area where logic programming has already produced results clearly ahead of those in other areas. As we have overviewed, interesting techniques have been proposed for the effective management of computations in a distributed setting, for intelligent scheduling of different forms of parallelism, as well as for static inference of task cost functions and their application to static and dynamic

control of the granularity of tasks. Nevertheless, much work still remains to be done.

Further research is still needed also 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, and building better 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 tackled in a complex context due to the nature of the computations, which are typically symbolic (implying high irregularity, dynamically allocated data structures, etc.) and involving search (implying speculativeness).

Finally, the techniques developed in the context of parallel execution of Prolog have 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 et al. 19995; Rocha et al. 1999a], which opens the doors to parallelization of applications in a number of interesting application areas, such as model checking and database cleaning;

—parallelization of the computation of models of a theory in the context of *nonmonotonic reasoning* [Pontelli and El-Kathib 2001; Finkel et al. 2001];

—use of parallelism in the execution of *inductive logic programs* [Page 2000; Ohwada et al. 2000].

We also believe there are good opportunities for transference of many of the techniques developed in the context of parallel execution of Prolog programs and their automatic parallelization to other programming paradigms [Hermenegildo 2000].

ACKNOWLEDGMENTS

Thanks are due to Bharat Jayaraman for helping with an earlier article on which this article is based. Thanks to Manuel Carro and Vitor Santos Costa, who read drafts of this survey. Our deepest thanks to the anonymous referees whose comments tremendously improved the article.

REFERENCES

AIKAWA, S., KAMIKO, M., KUBO, H., MATSUZAWA, F., AND CHIKAYAMA, T. 1992. Paragraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proceedings of the Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. IOS Press, Tokyo, Japan, 286–293.

AÏT-KACI, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Cambridge, MA. www.isg.sfu.ca/~hak/documents/wam.html.

AÏT-KACI, H. 1993. An Introduction to LIFE: Programming with Logic, Inheritance, Functions, and Equations. In *International Logic Programming Symposium*, D. Miller, Ed. MIT Press, Cambridge, MA, 52–68.

ALI, K. 1988. Or-Parallel Execution of Prolog on BC-machine. In *Proceedings of the International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, Seattle, 1531–1545.

ALI, K. 1995. A Parallel Copying Garbage Collection Scheme for Shared-Memory Multiprocessors. In *Proceedings of the ICOT/NSF Workshop on Parallel Logic Programming and Its Programming Environments*, E. Tick and T. Chikayama, Eds. Number CSI-TR-94-04. University of Oregon, Eugene, OR, 93–96.

ALI, K. AND KARLSSON, R. 1990a. Full Prolog and Scheduling Or-Parallelism in Muse. *International Journal of Parallel Programming 19*, 6, 445–475.

ALI, K. AND KARLSSON, R. 1990b. The MUSE Approach to Or-Parallel Prolog. *International Journal of Parallel Programming 19*, 2, 129–162.

ALI, K. AND KARLSSON, R. 1992a. OR-Parallel Speedups in a Knowledge Based System: On MUSE and Aurora. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. IOS Press, Tokyo, Japan, 739–745.

ALI, K. AND KARLSSON, R. 1992b. Scheduling Speculative Work in MUSE and Performance Results. *International Journal of Parallel Programming 21*, 6.

ALI, K., KARLSSON, R., AND MUDAMBI, S. 1992. Performance of MUSE on Switch-Based Multiprocessor Machines. *New Generation Computing 11*, 1/4, 81–103.

ALMASI, G. AND GOTTLIEB, A. 1994. *Highly Parallel Computing*. Benjamin/Cummings, San Francisco, CA.

APPLEBY, K., CARLSSON, M., HARIDI, S., AND SAHLIN, D. 1988. Garbage Collection for Prolog Based on WAM. *Communications of the ACM 31*, 6 (June), 719–741.

ARAUJO, L. AND RUZ, J. 1998. A Parallel Prolog System for Distributed Memory. *Journal of Logic Programming 33*, 1, 49–79.

BADER, D. AND JAJA, J. 1997. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors. Tech. rep., University of Maryland.

BAHGAT, R. 1993. Pandora: Non-Deterministic Parallel Logic Programming. Ph.D. thesis, Department of Computing, Imperial College. Published by World Scientific Publishing.

BANSAL, A. AND POTTER, J. 1992. An Associative Model for Minimizing Matching and Backtracking Overhead in Logic Programs with Large Knowledge Bases. *Engineering Applications of Artificial Intelligence 5*, 3, 247–262.

BARKLUND, J. 1990. Parallel Unification. Ph.D. thesis, Uppsala University. Uppsala Theses in Computing Science 9.

BARKLUND, J. AND MILLROTH, H. 1992. Providing Iteration and Concurrency in Logic Program Through Bounded Quantifications. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. IOS Press, Tokyo, Japan, 817–824.

BARON, U., CHASSIN DE KERGOMMEAUX, J., HAILPERIN, M., RATCLIFFE, M., ROBERT, P., SYRE, J.-C., AND WETPHAL, H. 1988. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. IOS Press, Tokyo, Japan, 841–850.

BEAUMONT, A. 1991. Scheduling Strategies and Speculative Work. In *Parallel Execution of Logic Programs*, A. Beaumont and G. Gupta, Eds. Lecture Notes in Computer Science, Vol. 569. Springer-Verlag, Heidelberg, 120–131.

BEAUMONT, A. AND WARREN, D. H. D. 1993. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *Proceedings of the International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, MA, 135–149.

BEAUMONT, A., MUTHU RAMAN, S., SZEREDI, P., AND WARREN, D. H. D. 1991. Flexible scheduling or-parallelism in Aurora: The Bristol scheduler. In *PARLE 91, Conference on Parallel Architectures and Languages Europe*, E. Aarts, J. van Leeuwen, and M. Rem, Eds. LNCS, Vol. 506. Springer Verlag, Heidelberg, 421–438.

BEKKERS, Y., RIDOUX, O., AND UNGARO, L. 1992. Dynamic Memory Management for Sequential Logic Programming Languages. In *Proceedings of the International Workshop on Memory Management*, Y. Bekkers and J. Cohen, Eds. Springer-Verlag, Heidelberg, 82–102.

BEN-AMRAM, A. 1995. What is a Pointer Machine? Tech. rep., DIKU, University of Copenhagen.

BENJUMEA, V. AND TROYA, J. 1993. An OR Parallel Prolog Model for Distributed Memory Systems. In *International Symposium on Programming Languages Implementations and Logic Programming*, M. Bruynooghe and J. Penjam, Eds. Springer-Verlag, Heidelberg, 291–301.

BEVEMYR, J. 1995. A Generational Parallel Copying Garbage Collection for Shared Memory Prolog. In *Workshop on Parallel Logic Programming Systems*. University of Porto, Portland, OR.

BEVEMYR, J., LINDGREN, T., AND MILLROTH, H. 1993. Reform Prolog: The Language and Its Implementation. In *Proceedings of the International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, MA, 283–298.

BISWAS, P., SU, S., AND YUN, D. 1988. A Scalable Abstract Machine Model to Support Limited-OR Restricted AND parallelism in Logic Programs. In *Proceedings of the International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, Cambridge, MA, 1160–1179.

BORGWARDT, P. 1984. Parallel Prolog Using Stack Segments on Shared Memory Multiprocessors. In *International Symposium on Logic Programming*. Atlantic City, IEEE Computer Society, Silver Spring, MD, 2–12.

BRAND, P. 1988. Wavefront Scheduling. Tech. rep., SICS, Gigalips Project.

BRIAT, J., FAVRE, M., GEYER, C., AND CHASSIN DE KERGOMMEAUX, J. 1992. OPERA: Or-Parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, P. Kacsuk and M. Wise, Eds. J. Wiley & Sons, New York, 45–64.

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

BUENO, F. AND HERMENEGILDO, M. 1992. An Automatic Translation Scheme from Prolog to the Andorra Kernel Language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. IOS Press, Tokyo, Japan, 759–769.

BUENO, F., CABEZA, D., CARRO, M., HERMENEGILDO, M., LÓPEZ-GARCÍA, P., AND PUEBLA, G. 1997. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM). August. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

BUENO, F., CABEZA, D., HERMENEGILDO, M., AND PUEBLA, G. 1996. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*. Number 1058 in LNCS. Springer-Verlag, Sweden, 108–124.

BUENO, F., DEBRAY, S., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1995. Transformation-Based Implementation and Optimization of Programs Exploiting the Basic Andorra Model. Technical Report CLIP11/95.0, Facultad de Informática, UPM. May.

BUENO, F., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1994a. A Comparative Study of Methods for Automatic Compile-Time Parallelization of Logic Programs. In *Parallel Symbolic Computation*. World Scientific Publishing Company, 63–73.

BUENO, F., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1999. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems 21*, 2, 189–239.

BUENO, F., HERMENEGILDO, M., MONTANARI, U., AND ROSSI, F. 1994b. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*. Number 850 in LNCS. Springer-Verlag, 114–132.

BUENO, F., HERMENEGILDO, M., MONTANARI, U., AND ROSSI, F. 1998. Partial Order and Contextual Net Semantics for Atomic and Locally Atomic CC Programs. *Science of Computer Programming 30*, 51–82.

BUTLER, R., DISZ, T., LUSK, E., OLSON, R., OVERBEEK, R., AND STEVENS, R. 1988. Scheduling Or-Parallelism: An Argonne Perspective. In *Proceedings of the International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, Cambridge, MA, 1565–1577.

BUTLER, R., LUSK, E., MCCUNE, W., AND OVERBEEK, R. 1986. Parallel Logic Programming for Numerical Applications. In *Proceedings of the Third International Conference on Logic Programming*, E. Shapiro, Ed. Springer-Verlag, Heidelberg, 357–388.

CABEZA, D. AND HERMENEGILDO, M. 1994. Extracting Non-Strict Independent And-Parallelism Using Sharing and Freeness Information. In *International Static Analysis Symposium*, B. Le Charlier, Ed. LNCS. Springer-Verlag, Heidelberg, 297–313.

CABEZA, D. AND HERMENEGILDO, M. 1996. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proceedings of the AGP'96 Joint conference on Declarative Programming*. U. of the Basque Country, San Sebastian, Spain, 67–78. Available from `http://www.clip.dia.fi.upm.es/`.

CALDERWOOD, A. AND SZEREDI, P. 1989. Scheduling Or-Parallelism in Aurora: The Manchester Scheduler. In *Proceedings of the International Conference on Logic Programming*, G. Levi and M. Martelli, Eds. MIT Press, Cambridge, MA, 419–435.

CALEGARIO, V. AND DUTRA, I. C. 1999. Performance Evaluation of Or-Parallel Logic Programming Systems on Distributed Shared Memory Architectures. In *Proceedings of EuroPar*, P. Amestoy et al., Ed. Springer-Verlag, Heidelberg, 1484–1491.

CARLSSON, M., WIDEN, J., AND BRAND, P. 1989. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In *Sixth International Conference on Logic Programming*, G. Levi and M. Martelli, Eds. MIT Press, Cambridge, MA, 3–16.

CARLSSON, M. 1990. Design and Implementation of an OR-Parallel Prolog Engine. Ph.D. thesis, Royal Institute of Technology, Stockholm.

CARLSSON M., WIDEN, J., AND BRAND, P. 1995. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science.

CARRO, M. AND HERMENEGILDO, M. 1999. Concurrency in Prolog Using Threads and a Shared Database. In *International Conference on Logic Programming*, D. De Schreye, Ed. MIT Press, Cambridge, MA, 320–334.

CARRO, M., GÓMEZ, L., AND HERMENEGILDO, M. 1993. Some Event-Driven Paradigms for the Visualization of Logic Programs. In *Proceedings of the International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, MA, 184–200.

CASTRO, L., SANTOS COSTA, V., GEYER, C., SILVA, F., VARGAS, P., AND CORREIA, M. 1999. DAOS: Scalable And-Or Parallelism. In *Proceedings of EuroPar*, D. Pritchard and J. Reeve, Eds. Springer-Verlag, Heidelberg, 899–908.

CHANG, J.-H., DESPAIN, A., AND DEGROOT, D. 1985. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Digest of Papers of Compcon Spring 1985*. IEEE Computer Society, Los Alamitos, CA, 218–225.

CHANG, S.-E. AND CHIANG, Y. 1989. Restricted AND-Parallelism Execution Model with Side-Effects. In *Proceedings of the North American Conference on Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cambridge, MA, 350–368.

CHASSIN DE KERGOMMEAUX, J. 1989. Measures of the PEPSys Implementation on the MX500. Tech. Rep. CA-44, ECRC.

CHASSIN DE KERGOMMEAUX, J. AND CODOGNET, P. 1994. Parallel logic programming systems. *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. *Journal of Logic Programming 8*, 3, 249–264.

CHIKAYAMA, T., FUJISE, T., AND SEKITA, D. 1994. A Portable and Efficient Implementation of KL1. In *Proceedings of the Symposium on Programming Languages Implementation and Logic Programming*, M. Hermenegildo and J. Penjam, Eds. Springer-Verlag, Heidelberg, 25–39.

CIANCARINI, P. 1990. Blackboard Programming in Shared Prolog. In *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Eds. MIT Press, Cambridge, MA, 170–185.

CIEPIELEWSKI, A. 1992. Scheduling in Or-Parallel Prolog Systems: Survey and Open Problems. *International Journal of Parallel Programming 20*, 6, 421–451.

CIEPIELEWSKI, A. AND HARIDI, S. 1983. A Formal Model for OR-parallel Execution of Logic Programs. In *Proceedings of IFIP*, P. Mason, Ed. North Holland, Amsterdam, 299–305.

CIEPIELEWSKI, A. AND HAUSMAN, B. 1986. Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs. In *Proceedings of the Symposium on Logic Programming*. IEEE Computer Society, Los Alamitos, CA, 246–257.

CLARK, K. AND GREGORY, S. 1986. Parlog: Parallel Programming in Logic. *Transactions on Programming Languages and Systems 8*, 1 (January), 1–49.

CLOCKSIN, W. AND ALSHAWI, H. 1988. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing 5*, 361–376.

CODISH, M., MULKERS, A., BRUYNOOGHE, M., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1995. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems 17*, 1, 28–44.

CODOGNET, C. AND CODOGNET, P. 1990. Non-Deterministic Stream and-Parallelism Based on Intelligent Backtracking. In *Proceedings of the International Conference on Logic Programming*, G. Levi and M. Martelli, Eds. MIT Press, Cambridge, MA, 63–79.

CODOGNET, C., CODOGNET, P., AND FILÉ, G. 1988. Yet Another Intelligent Backtracking Method. In *Proceedings of the International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, Cambridge, MA, 447–465.

CONERY, J. 1987a. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *International Symposium on Logic Programming*. IEEE Computer Society, Los Alamitos, CA, 457–467.

CONERY, J. 1987b. *Parallel Interpretation of Logic Programs*. Kluwer Academic, Norwell, MA.

CONERY, J. 1992. The OPAL Machine. In *Implementations of Distributed Prolog*, P. Kacsuk and D. S. Wise, Eds. J. Wiley & Sons, New York, 159–185.

CONERY, J. AND KIBLER, D. 1981. Parallel Interpretation of Logic Programs. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture* (1981). ACM Press, New York, 163–170.

CONERY, J. AND KIBLER, D. 1983. And Parallelism in Logic Programs. In *Proceedings of the International Joint Conference on AI*, A. Bundy, Ed. William Kaufmann, Los Altos, CA, 539–543.

CORREIA, E., SILVA, F., AND SANTOS COSTA, V. 1997. The SBA: Exploiting Orthogonality in And-or Parallel System. In *Proceedings of the International Symposium on Logic Programming*, J. Małuszyński, Ed. MIT Press, Cambridge, MA, 117–131.

COUSOT, P. AND COUSOT, R. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Records of the ACM Symposium on Principles of Programming Languages*. ACM Press, New York, 238–252.

COUSOT, P. AND COUSOT, R. 1992. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming 13*, 2–3, 103–179.

COX, P. 1984. Finding backtrack points for intelligent backtracking. In *Implementations of Prolog*, J. Campbell, Ed. Ellis Horwood, Hemel Hempstead.

CRABTREE, B. 1991. A Clustering System to Network Control. Tech. rep., British Telecom.

CRAMMOND, J. 1985. A Comparative Study of Unification Algorithms for Or-Parallel Execution of Logic Languages. *IEEE Transactions on Computers 34*, 10, 911–971.

CRAMMOND, J. 1992. The Abstract Machine and Implementation of Parallel Parlog. *New Generation Computing 10*, 4, 385–422.

DE BOSSCHERE, K. AND TARAU, P. 1996. Blackboard-Based Extensions in Prolog. *Software Practice & Experience 26*, 1, 46–69.

DEBRAY, S. AND JAIN, M. 1994. A Simple Program Transformation for Parallelism. In *Proceedings of the 1994 Symposium on Logic Programming*. MIT Press.

DEBRAY, S. AND LIN, N. 1993. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems 15*, 5, 826–875.

DEBRAY, S. AND WARREN, D. S. 1989. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems 11*, 3, 451–481.

DEBRAY, S., LÓPEZ-GARCÍA, P., AND HERMENEGILDO, M. 1997. Non-Failure Analysis for Logic Programs. In *International Conference on Logic Programming*, L. Naish, Ed. MIT Press, Cambridge, MA, 48–62.

DEBRAY, S., LÓPEZ-GARCÍA, P., HERMENEGILDO, M., AND LIN, N.-W. 1994. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*. Number 864 in LNCS. Springer-Verlag, Namur, Belgium, 255–265.

DEBRAY, S., LÓPEZ-GARCÍA, P., HERMENEGILDO, M., AND LIN, N.-W. 1997. Lower Bound Cost Estimation for Logic Programs. In *International Logic Programming Symposium*, J. Maluszyński, Ed. MIT Press, Cambridge, MA, 291–306.

DEBRAY, S. K., LIN, N.-W., AND HERMENEGILO, M. 1990. Task Granularity Analysis in Logic Programs. In *Proceedings of the 1990 ACM Conference on Programming Language Design and Implementation*. ACM Press, New York, 174–188.

DEGROOT, D. 1984. Restricted and-Parallelism. In *International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. IOS Press, Tokyo, Japan, 471–478.

DEGROOT, D. 1987a. A Technique for Compiling Execution Graph Expressions for Restricted AND-Parallelism in Logic Programs. In *Proceedings of the 1987 International Supercomputing Conference* Springer-Verlag, Athens, 80–89.

DEGROOT, D. 1987b. Restricted and-Parallelism and Side-Effects. In *International Symposium on Logic Programming*. San Francisco, IEEE Computer Society, Los Alamitos, CA, 80–89.

DELGADO-RANNAURO, S. 1992a. Or-Parallel Logic Computational Models. In *Implementations of Distributed Prolog*, P. Kacsuk and M. Wise, Eds. J. Wiley & Sons, New York, 3–26.

DELGADO-RANNAURO, S. 1992b. Restricted And- and And/Or-Parallel Logic Computational Models. In *Implementations of Distributed Prolog*, P. Kacsuk and M. Wise, Eds. J. Wiley & Sons, New York, 121–141.

DISZ, T. AND LUSK, E. 1987. A Graphical Tool for Observing the Behaviour of Parallel Logic Programs. In *Proceedings of the Symposium on Logic Programming*. IEEE Computer Society, Los Alamitos, CA, 46–53.

DISZ, T., LUSK, E., AND OVERBEEK, R. 1987. Experiments with OR-Parallel Logic Programs. In *Fourth International Conference on Logic Programming*, J. Lassez, Ed. University of Melbourne, MIT Press, Cambridge, MA, 576–600.

DOROCHEVSKY, M. AND XU, J. 1991. Parallel Execution Tracer. Tech. rep., ECRC.

DRAKOS, N. 1989. Unrestricted And-Parallel Execution of Logic Programs with Dependency Directed Backtracking. In *Proceedings of the International Joint Conference on Artificial Intelligence*, N. Sridharan, Ed. Morgan Kaufmann, New York, 157–162.

DUTRA, I. C. 1994. Strategies for Scheduling And- and Or-Parallel Work in Parallel Logic Programming Systems. In *International Logic Programming Symposium*, M. Bruynooghe, Ed. MIT Press, Cambridge, MA, 289–304.

DUTRA, I. C. 1995. Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System. Ph.D. thesis, University of Bristol.

DUTRA, I. C. 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, Los Alamitos, CA, 645–655.

DUTRA, I. C., SANTOS COSTA, V., AND BIANCHINI, R. 2000. The Impact of Cache Coherence Protocols on Parallel Logic Programming Systems. In *Proceedings of the International Conference on Computational Logic*, J. Lloyd et al., Ed. Springer-Verlag, Heidelberg, 1285–1299.

FERNÁNDEZ, M., CARRO, M., AND HERMENEGILDO, M. 1996. IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In *Proceedings of EuroPar*, L. Bouge et al., Ed. Springer-Verlag, Heidelberg, 724–733.

FINKEL, R., MAREK, V., MOORE, N., AND TRUSZCZYŃSKI, M. 2001. Computing Stable Models in Parallel. In *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, A. Provetti and S. Tran, Eds. AAAI/MIT Press, Cambridge, MA, 72–75.

FONSECA, N., SANTOS COSTA, V., AND DUTRA, I. C. 1998. VisAll: A Universal Tool to Visualize the Parallel Execution of Logic Programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, J. Jaffar, Ed. MIT Press, Cambridge, MA, 100–114.

FREIRE, J., HU, R., SWIFT, T., AND WARREN, D. S. 1995. Exploiting Parallelism in Tabled Evaluations. In *Proceedings of the Symposium on Programming Languages Implementations and Logic Programming*, M. Hermenegildo and S. Swierstra, Eds. Springer-Verlag, Heidelberg, 115–132.

FUTÓ, I. 1993. Prolog with Communicating Processes: From T-Prolog to CSR-Prolog. In *International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, MA, 3–17.

GANGULY, S., SILBERSCHATZ, A., AND TSUR, S. 1990. A Framework for the Parallel Processing of Datalog Queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, H. Garcia-Molina and H. Jagadish, Eds. ACM Press, New York.

GARCÍA DE LA BANDA, M. 1994. Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming. Ph.D. thesis, Universidad Politecnica de Madrid.

GARZÍA DE LA BANDA, M., BUENO, F., HERMENEGILDO, M. 1996a. Towards Independent And-parallelism in CLP. In *Proceedings of Programming Languages*: *Implementation*, *Logics and Programs.* Springer Verlag, Heidelberg, 77–91.

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

García de la Banda, M., Hermenegildo, M., and Marriott, K. 2000. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems 22*, 2 (March), 269–339.

Giacobazzi, R. and Ricci, L. 1990. Pipeline Optimizations in And-parallelism by Abstract Interpretation. In *Proceedings of International Conference on Logic Programming*, D. H. D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, MA, 291–305.

Giannotti, F. and Hermenegildo, M. 1991. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proceedings 3rd International Symposium on Programming Language Implementation and Logic Programming*. Number 528 in LNCS. Springer-Verlag, 323–335.

Gregory, S. and Yang, R. 1992. Parallel Constraint Solving in Andorra-I. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. IOS Press, Tokyo, Japan, 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*, D. S. Warren, Ed.

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

Gupta, G. and Jayaraman, B. 1993a. Analysis of Or-Parallel Execution Models. *ACM Transactions on Programming Languages and Systems 15*, 4, 659–680.

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

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

Gupta, G. and Pontelli, E. 1999a. Extended Dynamic Dependent And-Parallelism in ACE. *Journal of Functional and Logic Programming 99*,Special Issue 1.

Gupta, G. and Pontelli, E. 1999b. Last Alternative Optimization for Or-Parallel Logic Programming Systems. In *Parallelism and Implementation Technology for Constraint Logic Programming*, I. Dutra et al., Ed. Nova Science, Commack, NY, 107–132.

Gupta, G. and Pontelli, E. 1999c. Stack-Splitting: A Simple Technique for Implementing Or-Parallelism and And-Parallelism on Distributed Machines. In *International Conference on Logic Programming*, D. De Schreye, Ed. MIT Press, Cambridge, MA, 290–304.

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

Gupta, G., Hermenegildo, M., Pontelli, E., and Santos Costa, V. 1994. ACE: And/Or-Parallel Copying-Based Execution of Logic Programs. In *Proceedings of the International Conference on Logic Programming*, P. van Hentenryck, Ed. MIT Press, Cambridge, MA, 93–109.

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

Gupta, G., Hermenegildo, M., and Santos Costa, V. 1993. And-Or Parallel Prolog: A Recomputation Based Approach. *New Generation Computing 11*, 3–4, 297–322.

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

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

Gupta, G., Santos Costa, V., Yang, R., and Hermenegildo, M. 1991. IDIOM: A Model Intergrating Dependent-, Independent-, and Or-parallelism. In *International Logic Programming Symposium*, V. Saraswat and K. Ueda, Eds. MIT Press, Cambridge, MA, 152–166.

HALSTEAD, R. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the Symposium on LISP and Functional Programming*. ACM Press, New York, 9–17.

HARALICK, R. AND ELLIOT, G. 1980. Increasing Tree Search Efficiency for Constraint Satisfaction. *Artificial Intelligence 14*, 3, 263–313.

HARIDI, S. 1990. A Logic Programming Language Based on the Andorra Model. *New Generation Computing 7*, 2/3, 109–125.

HARIDI, S. AND JANSON, S. 1990. Kernel Andorra Prolog and Its Computation Model. In *Proceedings of the International Conference on Logic Programming*, D. H. D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, MA, 31–46.

HARIDI, S., VAN ROY, P., BRAND, P., AND SCHULTE, C. 1998. Programming Languages for Distributed Applications. *New Generation Computing 16*, 3, 223–261.

HASENBERGER, J. 1995. Modelling and Redesign the Advanced Traffic Management System in Andorra-I. In *Proceedings of the Workshop on Parallel Logic Programming Systems*, V. Santos Costa, Ed. University of Porto, Portland, OR.

HAUSMAN, B. 1989. Pruning and scheduling speculative work in or-parallel Prolog. In *Conference on Parallel Architectures and Languages Europe*, E. Odijk, M. Rem, and J.-C. Syre, Eds. Springer-Verlag, Heidelberg, 133–150.

HAUSMAN, B. 1990. Pruning and Speculative Work in OR-Parallel PROLOG. Ph.D. thesis, The Royal Institute of Technology, Stockholm.

HAUSMAN, B., CIEPIELEWSKI, A., AND CALDERWOOD, A. 1988. Cut and Side-Effects in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. Springer-Verlag, Tokyo, Japan, 831–840.

HAUSMAN, B., CIEPIELEWSKI, A., AND HARIDI, S. 1987. OR-Parallel Prolog Made Efficient on Shared Memory Multiprocessors. In *Symposium on Logic Programming*. IEEE Computer Society, Los Alamitos, CA, 69–79.

HERMENEGILDO, M. 1986a. An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel. Ph.D. thesis, U. of Texas at Austin.

HERMENEGILDO, M. 1986b. An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs. In *Proceedings of the International Conference on Logic Programming*, E. Shapiro, Ed. Springer-Verlag, Heidelberg, 25–40.

HERMENEGILDO, M. 1987. Relating Goal Scheduling, Precedence, and Memory Management in And-parallel Execution of Logic Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press, Cambridge, MA, 556–575.

HERMENEGILDO, M. 1994. A Simple, Distributed Version of the &-Prolog System. Technical report, School of Computer Science, Technical University of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid, Spain. April. Available from `http://www.clip.dia.fi.upm.es/`.

HERMENEGILDO, M. 2000. Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. *Parallel Computing 26*, 13–14, 1685–1708.

HERMENEGILDO, M., BUENO, F., CABEZA, D., CARRO, M., GARCÍA DE LA BANDA, M., LÓPEZ-GARCÍA, P., AND PUEBLA, G. 1999a. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*. Nova Science, Commack, NY, 65–85.

HERMENEGILDO, M., BUENO, F., PUEBLA, G., AND LÓPEZ-GARCÍA, P. 1999b. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*. MIT Press, Cambridge, MA, 52–66.

HERMENEGILDO, M., CABEZA, D., AND CARRO, M. 1995. Using Attributed Variables in the Implementation of Parallel and Concurrent Logic Programming Systems. In *Proceedings of the International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, Cambridge, MA, 631–645.

HERMENEGILDO, M. AND CARRO, M. 1996. Relating Data–Parallelism and (And–) Parallelism in Logic Programs. *The Computer Languages Journal 22*, 2/3 (July), 143–163.

HERMENEGILDO, M. AND CLIP GROUP, T. 1994. Some Methodological Issues in the Design of CIAO—A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*. Number 874 in LNCS. Springer-Verlag, 123–133.

HERMENEGILDO, M. AND GREENE, K. 1991. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing 9*, 3–4, 233–257.
HERMENEGILDO, M. AND LÓPEZ-GARCÍA, P. 1995. Efficient Term Size Computation for Granularity Control. In *Proceedings of the International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, Cambridge, MA, 647–661.
HERMENEGILDO, M. AND NASR, R. I. 1986. Efficient Management of Backtracking in AND-Parallelism. In *Third International Conference on Logic Programming*, E. Shapiro, Ed. Number 225 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 40–54.
HERMENEGILDO, M., PUEBLA, G., MARRIOTT, K., AND STUCKEY, P. 2000. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems 22*, 2 (March), 187–223.
HERMENEGILDO, M. AND ROSSI, F. 1995. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming 22*, 1, 1–45.
HERMENEGILDO, M. AND TICK, E. 1989. Memory Performance of AND-Parallel Prolog on Shared-Memory Architectures. *New Generation Computing 7*, 1 (October), 37–58.
HERMENEGILDO, M. AND WARREN, R. 1987. Designing a High-Performance Parallel Logic Programming System. *Computer Architecture News, Special Issue on Parallel Symbolic Programming 15*, 1 (March), 43–53.
HERMENEGILDO, M., WARREN, R., AND DEBRAY, S. 1992. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming 13*, 4, 349–367.
HEROLD, A. 1995. The Handbook of Parallel Constraint Logic Programming Applications. Tech. Rep., ECRC.
HERRARTE, V. AND LUSK, E. 1991. Studying Parallel Program Behaviours with Upshot. Tech. Rep. ANL-91/15, Argonne National Labs.
HICKEY, T. AND MUDAMBI, S. 1989. Global Compilation of Prolog. *Journal of Logic Programming 7*, 3, 193–230.
HIRATA, K., YAMAMOTO, R., IMAI, A., KAWAI, H., HIRANO, K., TAKAGI, T., TAKI, K., NAKASE, A., AND ROKUSAWA, K. 1992. Parallel and Distributed Implementation of Logic Programming Language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. Ohmsha Ltd., Tokyo, Japan, 436–459.
IQSoft Inc. 1992. *CUBIQ - Development and Application of Logic Programming Tools for Knowledge Based Systems*. IQSoft Inc. www.iqsoft.hu/projects/cubiq/cubiq.html.
JACOBS, D. AND LANGEN, A. 1992. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming 13*, 1–4, 291–314.
JANAKIRAM, V., AGARWAL, D., AND MALHOTRA, R. 1988. A Randomized Parallel Backtracking Algorithm. *IEEE Transactions on Computers 37*, 12, 1665–1676.
JANSON, S. AND MONTELIUS, J. 1991. A Sequential Implementation of AKL. In *Proceedings of ILPS'91 Workshop on Parallel Execution of Logic Programs*.
KACSUK, P. 1990. *Execution Models of Prolog for Parallel Computers*. MIT Press, Cambridge, MA.
KACSUK, P. AND WISE, M. 1992. *Implementation of Distributed Prolog*. J. Wiley & Sons., New York.
KALÉ, L. 1985. Parallel Architectures for Problem Solving. Ph.D. thesis, SUNY Stony Brook, Dept. Computer Science.
KALÉ, L. 1991. The REDUCE OR Process Model for Parallel Execution of Logic Programming. *Journal of Logic Programming 11*, 1, 55–84.
KALÉ, L., RAMKUMAR, B., AND SHU, W. 1988a. A Memory Organization Independent Binding Environment for AND and OR Parallel Execution of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programs*, R. Kowalski and K. Bowen, Eds. MIT Press, Cambridge, MA, 1223–1240.
KALÉ, L. V., PADUA, D. A., AND SEHR, D. C. 1988b. Or-Parallel Execution of Prolog with Side Effects. *Journal of Supercomputing 2*, 2, 209–223.
KARLSSON, R. 1992. A High Performance Or-Parallel Prolog System. Ph.D. thesis, Royal Institute of Technology, Stockholm.
KASIF, S., KOHLI, M., AND MINKER, J. 1983. PRISM: A Parallel Inference System for Problem Solving. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence* (1983), A. Bundy, Ed. Morgan Kaufman, San Francisco, CA, 544–546.

KING, A., SHEN, K., AND BENOY, F. 1997. Lower-Bound Time-Complexity Analysis of Logic Programs. In *Proceedings of the International Logic Programming Symposium*, J. Maluszyński, Ed. MIT Press, Cambridge, MA, 261–276.

KLUŹNIAK, F. 1990. Developing Applications for Aurora Or-Parallel System. Tech. Rep. TR-90-17, Dept. of Computer Science, University of Bristol.

KOWALSKI, R. 1979. *Logic for Problem Solving*. Elsevier North-Holland, Amsterdam.

KUSALIK, A. AND PRESTWICH, S. 1996. Visualizing Parallel Logic Program Execution for Performance Tuning. In *Proceedings of Joint International Conference and Symposium on Logic Programming*, M. Maher, Ed. MIT Press, Cambridge, MA, 498–512.

LAMMA, E., MELLO, P., STEFANELLI, C., AND HENTENRYCK, P. V. 1997. Improving Distributed Unification Through Type Analysis. In *Proceedings of Euro-Par 1997*. LNCS, Vol. 1300. Springer-Verlag, 1181–1190.

LE HUITOUZE, S. 1990. A new data structure for implementing extensions to Prolog. In *Symposium on Programming Languages Implementation and Logic Programming*, P. Deransart and J. Małuszyński, Eds. Springer-Verlag, Heidelberg, 136–150.

LIN, Y. J. 1988. A Parallel Implementation of Logic Programs. Ph.D. thesis, Dept. of Computer Science, University of Texas at Austin, Austin, TX.

LIN, Y. J. AND KUMAR, V. 1988. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, Cambridge, MA, 1123–1141.

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

LINDGREN, T., BEVEMYR, J., AND MILLROTH, H. 1995. Compiler Optimizations in Reform Prolog: Eperiments on the KSR-1 Multiprocessor. In *Proceedings of EuroPar*, S. Haridi and P. Magnusson, Eds. Springer-Verlag, Heidelberg, 553–564.

LINDSTROM, G. 1984. Or-Parallelism on Applicative Architectures. In *International Logic Programming Conference*, S. Tarnlund, Ed. Uppsala University, Uppsala, 159–170.

LLOYD, J. 1987. *Foundations of Logic Programming*. Springer-Verlag, Heidelberg.

LOPES, R. AND SANTOS COSTA, V. 1999. The BEAM: Towards a First EAM Implementation. In *Parallelism and Implementation Technology for Constraint Logic Programming*, I. Dutra et al., Ed. Nova Science, Commack, NY, 87–106.

LÓPEZ-GARCÍA, P., HERMENEGILDO, M., AND DEBRAY, S. 1996. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation 22*, 715–734.

LUSK, E., BUTLER, R., DISZ, T., OLSON, R., STEVENS, R., WARREN, D. H. D., CALDERWOOD, A., SZEREDI, P., BRAND, P., CARLSSON, M., CIEPIELEWSKI, A., HAUSMAN, B., AND HARIDI, S. 1990. The Aurora Or-Parallel Prolog System. *New Generation Computing 7*, 2/3, 243–271.

LUSK, E., MUDAMBI, S., OVERBEEK, R., AND SZEREDI, P. 1993. Applications of the Aurora Parallel Prolog System to Computational Molecular Biology. In *Proceedings of the International Logic Programming Symposium*, D. Miller, Ed. MIT Press, Cambridge, MA, 353–369.

MASUZAWA, H., KUMON, K., ITASHIKI, A., SATOH, K., AND SOHMA, Y. 1986. KABU-WAKE: A New Parallel Inference Method and Its Evaluation. In *Proceedings of the Fall Joint Computer Conference*. IEEE Computer Society, Los Alamitos, CA, 955–962.

MILLROTH, H. 1990. Reforming Compilation of Logic Programs. Ph.D. thesis, Uppsala University.

MONTELIUS, J. 1997. Exploiting Fine-Grain Parallelism in Concurrent Constraint Languages. Ph.D. thesis, Uppsala University.

MONTELIUS, J. AND ALI, K. 1996. A Parallel Implementation of AKL. *New Generation Computing 14*, 1, 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, New York, 46–57.

MOOLENAAR, R. AND DEMOEN, B. 1993. A Parallel Implementation for AKL. In *Proceedings of the Conference on Programming Languages Implementation and Logic Programming*, M. Bruynooghe and J. Penjam, Eds. Number 714 in LNCS. Springer-Verlag, Heidelberg, 246–261.

MUDAMBI, S. 1991. Performance of Aurora on NUMA Machines. In *Proceedings of the International Logic Programming Symposium*, V. Saraswat and K. Ueda, Eds. MIT Press, Cambridge, MA, 793–806.

MUDAMBI, S. AND SCHIMPF, J. 1994. Parallel CLP on Heterogenous Networks. In *Proceedings of the International Conference on Logic Programming*, P. V. Hentenryck, Ed. MIT Press, Cambridge, MA, 124–141.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1989. Efficient Methods for Supporting Side Effects in Independent And-Parallelism and Their Backtracking Semantics. In *Proceedings of the International Conference on Logic Programming*, G. Levi and M. Martelli, Eds. MIT Press, Cambridge, MA, 80–97.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1990. The CDG, UDG, and MEL Methods for Automatic Compile-Time Parallelization of Logic Programs for Independent And-Parallelism. In *1990 International Conference on Logic Programming*, D. H. D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, MA, 221–237.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *Proceedings of the International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, Cambridge, MA, 49–63.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1992. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming 13*, 2/3 (July), 315–347.

MUTHUKUMAR, K., BUENO, F., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1999. Automatic Compile-Time Parallelization of Logic Programs for Restricted, Goal-Level, Independent And-Parallelism. *Journal of Logic Programming 38*, 2, 165–218.

NAISH, L. 1988. Parallelizing NU-Prolog. In *Proceedings of the International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, Cambridge, MA, 1546–1564.

NGUYEN, T. AND DEVILLE, Y. 1998. A Distributed Arc-Consistency Algorithm. *Science of Computer Programming 30*, 1–2, 227–250.

OHWADA, H., NISHIYAMA, H., AND MIZOGUCHI, F. 2000. Concurrent Execution of Optimal Hypothesis Search for Inverse Entailment. In *Proceedings of the Inductive Logic Programming Conference*, J. Cussens and A. Frisch, Eds. Springer-Verlag, Heidelberg, 165–183.

OLDER, W. AND RUMMELL, J. 1992. An Incremental Garbage Collector for WAM-Based Prolog. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, K. Apt, Ed. MIT Press, Cambridge, MA, 369–383.

OLMEDILLA, M., BUENO, F., AND HERMENEGILDO, M. 1993. Automatic Exploitation of Non-Determinate Independent And-Parallelism in the Basic Andorra Model. In *Logic Program Synthesis and Transformation, 1993*. Workshops in Computing. Springer-Verlag, 177–195.

OZAWA, T., HOSOI, A., AND HATTORI, A. 1990. Generation Type Garbage Collection for Parallel Logic Languages. In *Proceedings of the North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, MA, 291–305.

PAGE, D. 2000. ILP: Just Do It. In *Proceedings of the Inductive Logic Programming Conference*, J. Cussens and A. Frisch, Eds. Springer-Verlag, Heidelberg, 21–39.

PALMER, D. AND NAISH, L. 1991. NUA Prolog: An Extension of the WAM for Parallel Andorra. In *Proceedings of the International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, Cambridge, MA, 429–442.

PEREIRA, L. M., MONTEIRO, L., CUNHA, J., AND APARÍCIO, J. N. 1986. Delta Prolog: A Distributed Backtracking Extension with Events. In *Third International Conference on Logic Programming*. Number 225 in Lecture Notes in Computer Science. Imperial College, Springer-Verlag, Heidelberg, 69–83.

PERRON, L. 1999. Search Procedures and Parallelism in Constraint Programming. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, J. Jaffar, Ed. LNCS, Vol. 1713. Springer-Verlag, Heidelberg, 346–360.

PETERSON, J. AND SILBERSCHATZ, A. 1986. *Operating Systems Concepts*. Addison-Wesley, Boston, MA.

PITTOMVILS, E., BRUYNOOGHE, M., AND WILLEMS, Y. 1985. Towards a Real-Time Garbage Collector for Prolog. In *Proceedings of the Symposium on Logic Programming*. IEEE Computer Society, Los Alamitos, CA, 185–198.

POLLARD, G. H. 1981. Parallel Execution of Horn Clause Programs. Ph.D. thesis, Imperial College, London. Dept. of Computing.

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 *23rd Hawaian International Conference of Computers and Systems Science*. IEEE Computer Society, Los Alamitos, CA.

PONTELLI, E. AND EL-KATHIB, O. 2001. Construction and Optimization of a Parallel Engine for Answer Set Programming. In *Practical Aspects of Declarative Languages*, I. V. Ramakrishnan, Ed. LNCS, Vol. 1990. Springer-Verlag, Heidelberg, 288–303.

PONTELLI, E. AND GUPTA, G. 1995a. Data And-Parallel Logic Programming in &ACE. In *Proceedings of the Symposium on Parallel and Distributed Processing*. IEEE Computer Society, Los Alamitos, CA, 424–431.

PONTELLI, E. AND GUPTA, G. 1995b. On the Duality Between And-Parallelism and Or-Parallelism. In *Proceedings of EuroPar*, S. Haridi and P. Magnusson, Eds. Springer-Verlag, Heidelberg, 43–54.

PONTELLI, E. AND GUPTA, G. 1997a. Implementation Mechanisms for Dependent And-Parallelism. In *Proceedings of the International Conference on Logic Programming*, L. Naish, Ed. MIT Press, Cambridge, MA, 123–137.

PONTELLI, E. AND GUPTA, G. 1997b. Parallel Symbolic Computation with ACE. *Annals of AI and Mathematics 21*, 2–4, 359–395.

PONTELLI, E. AND GUPTA, G. 1998. Efficient Backtracking in And-Parallel Implementations of Non-Deterministic Languages. In *Proceedings of the International Conference on Parallel Processing*, T. Lai, Ed. IEEE Computer Society, Los Alamitos, CA, 338–345.

PONTELLI, E., GUPTA, G., AND HERMENEGILDO, M. 1995. &ACE: A High-Performance Parallel Prolog System. In *Proceedings of the International Parallel Processing Symposium*. IEEE Computer Society, Los Alamitos, CA, 564–571.

PONTELLI, E., GUPTA, G., PULVIRENTI, F., AND FERRO, A. 1997a. Automatic Compile-Time Parallelization of Prolog Programs for Dependent And-Parallelism. In *International Conference on Logic Programming*, L. Naish, Ed. MIT Press, Cambridge, MA, 108–122.

PONTELLI, E., GUPTA, G., TANG, D., CARRO, M., AND HERMENEGILDO, M. 1996. Improving the Efficiency of Non-Deterministic Independent And-Parallel Systems. *Computer Languages 22*, 2/3, 115–142.

PONTELLI, E., GUPTA, G., WIEBE, J., AND FARWELL, D. 1998. Natural Language Multiprocessing: A Case Study. In *Proceedings of the Fifteenth National Conference on Artifical Intelligence*. AAAI/MIT Press, Cambridge, MA, 76–82.

PONTELLI, E., RANJAN, D., AND GUPTA, G. 1997b. On the Complexity of Parallel Implementation of Logic Programs. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar, Eds. Springer-Verlag, Heidelberg, 123–137.

POPOV, K. 1997. A Parallel Abstract Machine for the Thread-Based Concurrent Language Oz. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, E. Pontelli and V. Santos Costa, Eds. New Mexico State University.

PUEBLA, G. AND HERMENEGILDO, M. 1996. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*. Number 1145 in LNCS. Springer-Verlag, 270–284.

PUEBLA, G. AND HERMENEGILDO, M. 1999. Abstract Multiple Specialization and Its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs 41*, 2&3 (November), 279–316.

RAMESH, R., RAMAKRISHNAN, I. V., AND WARREN, D. S. 1990. Automata-Driven Indexing of Prolog Clauses. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press, New York, 281–290.

RAMKUMAR, B. AND KALÉ, L. 1989. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proceedings of the North American Conference on Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cambridge, MA, 313–331.

RAMKUMAR, B. AND KALÉ, L. 1990. And Parallel Solutions in And/Or Parallel Systems. In *Proceedings of the North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, MA, 624–641.

RAMKUMAR, B. AND KALÉ, L. 1992. Machine Independent AND and OR Parallel Execution of Logic Programs. Part i and ii. *IEEE Transactions on Parallel and Distributed Systems 2*, 5.

RANJAN, D., PONTELLI, E., AND GUPTA, G. 1999. On the Complexity of Or-Parallelism. *New Generation Computing 17*, 3, 285–308.

RANJAN, D., PONTELLI, E., AND GUPTA, G. 2000a. Data Structures for Order-Sensitive Predicates in Parallel Nondetermistic Systems. *ACTA Informatica 37*, 1, 21–43.

RANJAN, D., PONTELLI, E., LONGPRE, L., AND GUPTA, G. 2000b. The Temporal Precedence Problem. *Algorithmica 28*, 288–306.

RATCLIFFE, M. AND SYRE, J. C. 1987. A Parallel Logic Programming Language for PEPSys. In *Proceedings of IJCAI*, J. McDermott, Ed., Morgan-Kaufmann, San Francisco, CA, 48–55.

ROCHA, R., SILVA, F., AND SANTOS COSTA, V. 1999a. Or-Parallelism Within Tabling. In *Proceedings of the Symposium on Practical Aspects of Declarative Languages*, G. Gupta, Ed. Springer-Verlag, Heidelberg, 137–151.

ROCHA, R., SILVA, F., AND SANTOS COSTA, V. 1999b. YapOr: An Or-Parallel Prolog System Based on Environment Copying. In *LNAI 1695, Proceedings of EPPIA'99: The 9th Portuguese Conference on Artificial Intelligence*. Springer-Verlag LNAI Series, 178–192.

ROKUSAWA, K., NAKASE, A., AND CHIKAYAMA, T. 1996. Distributed Memory Implementation of KLIC. *New Generation Computing 14*, 3, 261–280.

RUIZ-ANDINO, A., ARAUJO, L., SÁENZ, F., AND RUZ, J. 1999. Parallel Execution Models for Constraint Programming over Finite Domains. In *Proceedings of the Conference on Principles and Practice of Declarative Programming*, G. Nadathur, Ed. Springer-Verlag, Heidelberg, 134–151.

SAMAL, A. AND HENDERSON, T. 1987. Parallel Consistent Labeling Algorithms. *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, Los Alamitos, CA, 720–727.

SANTOS COSTA, V. 2000. *Encyclopedia of Computer Science and Technology*. Vol. 42. Marcel Dekker Inc., New Yourk, Chapter Parallelism and Implementation Technology for Logic Programming Languages, 197–237.

SANTOS COSTA, V., BIANCHINI, R., AND DUTRA, I. C. 1997. Parallel Logic Programming Systems on Scalable Multiprocessors. In *Proceedings of the International Symposium on Parallel Symbolic Computation*. ACM Press, Los Alamitos, CA, 58–67.

SANTOS COSTA, V., BIANCHINI, R., AND DUTRA, I. C. 2000. Parallel Logic Programming Systems on Scalable Architectures. *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*. University of Porto. www.ncc.up.pt/~vsc/Yap.

SANTOS COSTA, V., ROCHA, R., AND SILVA, F. 2000. Novel Models for Or-Parallel Logic Programs: A Performance Analysis. In *Proceedings of EuroPar*, A. B. et al., Ed. Springer-Verlag, Heidelberg, 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 ACM Symposium on Principles and Practice of Parallel Programming*. ACM Press, New York, 83–93.

SANTOS COSTA, V., WARREN, D. H. D., AND YANG, R. 1991b. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *Proceedings of the International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, Cambridge, MA, 825–839.

SANTOS COSTA, V., WARREN, D. H. D., AND YANG, R. 1991c. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *Proceedings of the International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, Cambridge, MA, 443–456.

SANTOS COSTA, V., WARREN, D. H. D., AND YANG, R. 1996. Andorra-I Compilation. *New Generation Computing 14*, 1, 3–30.

SARASWAT, V. 1989. Concurrent Constraint Programming Languages. Ph.D. thesis, Carnegie Mellon, Pittsburgh. School of Computer Science.

SCHULTE, C. 2000. Parallel Search Made Simple. In *Proceedings of Techniques for Implementing Constraint Programming Systems, Post-conference workshop of CP 2000*, N. Beldiceanu et al., Ed. Number TRA9/00. University of Singapore, 41–57.

SHAPIRO, E. 1987. *Concurrent Prolog: Collected Papers*. MIT Press, Cambridge MA.

SHAPIRO, E. 1989. The Family of Concurrent Logic Programming Languages. *ACM Computing Suveys 21*, 3, 413–510.

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

SHEN, K. 1992b. Studies in And/Or Parallelism in Prolog. Ph.D. thesis, University of Cambridge.

SHEN, K. 1994. Improving the Execution of the Dependent And-Parallel Prolog DDAS. In *Proceedings of Parallel Architectures and Languages Europe*, C. Halatsis et al., Ed. Springer-Verlag, Heidelberg, 438–452.

SHEN, K. 1996a. Initial Results from the Parallel Implementation of DASWAM. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*. MIT Press, Cambridge, MA.

SHEN, K. 1996b. Overview of DASWAM: Exploitation of Dependent And-Parallelism. *Journal of Logic Programming 29*, 1/3, 245–293.

SHEN, K. 1997. A New Implementation Scheme for Combining And/Or Parallelism. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, E. Pontelli and V. Santos Costa, Eds. New Mexico State University, Dept. Computer Science.

SHEN, K. AND HERMENEGILDO, M. 1991. A Simulation Study of Or- and Independent And-Parallelism. In *Proceedings of the International Logic Programming Symposium*, V. Saraswat and K. Ueda, Eds. MIT Press, Cambridge, MA, 135–151.

SHEN, K. AND HERMENEGILDO, M. 1994. Divided We Stand: Parallel Distributed Stack Memory Management. In *Implementations of Logic Programming Systems*, E. Tick and G. Succi, Eds. Kluwer Academic Press, Boston, MA.

SHEN, K. AND HERMENEGILDO, M. 1996a. Flexible Scheduling for Non-Deterministic, And-Parallel Execution of Logic Programs. In *Proceedings of EuroPar'96*. Number 1124 in LNCS. Springer-Verlag, 635–640.

SHEN, K. AND HERMENEGILDO, M. 1996b. High-Level Characteristics of Or- and Independent And-Parallelism in Prolog. *International. Journal of Parallel Programming 24*, 5, 433–478.

SHEN, K., SANTOS COSTA, V., AND KING, A. 1998. Distance: A New Metric for Controlling Granularity for Parallel Execution. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, J. Jaffar, Ed. MIT Press, Cambridge, MA, 85–99.

SILVA, F. AND WATSON, P. 2000. Or-Parallel Prolog on a Distributed Memory Architecture. *Journal of Logic Programming 43*, 2, 173–186.

SILVA, M., DUTRA, I. C., BIANCHINI, R., AND SANTOS COSTA, V. 1999. The Influence of Computer Architectural Parameters on Parallel Logic Programming Systems. In *Proceedings of the Workshop on Practical Aspects of Declarative Languages*, G. Gupta, Ed. Springer-Verlag, Heidelberg, 122–136.

SINDAHA, R. 1992. The Dharma Scheduler — Definitive Scheduling in Aurora on Multiprocessor Architecture. In *Proceedings of the Symposium on Parallel and Distributed Processing*. IEEE Computer Society, Los Alamitos, CA, 296–303.

SINDAHA, R. 1993. Branch-Level Scheduling in Aurora: The Dharma Scheduler. In *Proceedings of International Logic Programming Symposium*, D. Miller, Ed. MIT Press, Cambridge, MA, 403–419.

SINGHAL, A. AND PATT, Y. 1989. Unification Parallelism: How Much Can We Exploit? In *Proceedings of the North American Conference on Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cambridge, MA, 1135–1147.

SMITH, D. 1996. MultiLog and Data Or-Parallelism. *Journal of Logic Programming 29*, 1–3, 195–244.

SMOLKA, G. 1996. Constraints in Oz. *ACM Computing Surveys 28*, 4 (December), 75–76.

STERLING, L. AND SHAPIRO, E. 1994. *The Art of Prolog*. MIT Press, Cambridge MA.

SZEREDI, P. 1989. Performance Analysis of the Aurora Or-Parallel Prolog System. In *Proceedings of the North American Conference on Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cambridge, MA, 713–732.

SZEREDI, P. 1991. Using Dynamic Predicates in an Or-Parallel Prolog System. In *Proceedings of the International Logic Programming Symposium*, V. Saraswat and K. Ueda, Eds. MIT Press, Cambridge, MA, 355–371.

SZEREDI, P. 1992. Exploiting Or-Parallelism in Optimization Problems. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, K. Apt, Ed. MIT Press, Cambridge, MA, 703–716.

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., CARLSSON, M., AND YANG, R. 1991. Interfacing Engines and Schedulers in Or-Parallel Prolog Systems. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, E. Aarts et al., Ed. LNCS, Vol. 506. Springer-Verlag, Heidelberg, 439–453.

SZEREDI, P., MOLNÁR, K., AND SCOTT, R. 1996. Serving Multiple HTML Clients from a Prolog Application. In *Workshop on Logic Programming Tools for Internet*. Bonn.

TAKEUCHI, A. 1992. *Parallel Logic Programming*. Kluwer Academic Press, Boston, MA.

TARAU, P. 1998. Inference and Computation Mobility with Jinni. Tech. rep., University of North Texas.

TAYLOR, A. 1991. High-Performance Prolog Implementation. Ph.D. thesis, Basser Dept. of COmputer Science, University of Sydney.

TEBRA, H. 1987. Optimistic And-Parallelism in Prolog. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, J. de Bakker, A. Nijman, and P. Treleaven, Eds. Springer-Verlag, Heidelberg, 420–431.

TERASAKI, S., HAWLEY, D., 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 *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. IOS Press, Tokyo, Japan, 330–346.

TICK, E. 1987. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, Norwell, MA 02061.

TICK, E. 1991. *Parallel Logic Programming*. MIT Press, Cambridge, MA.

TICK, E. 1992. Visualizing Parallel Logic Programming with VISTA. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. Ohmsha Ltd., Tokyo, Japan, 934–942.

TICK, E. 1995. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming 23*, 2, 89–123.

TICK, E. AND ZHONG, X. 1993. A Compile-Time Granularity Analysis Algorithm and Its Performance Evaluation. *New Generation Computing 11*, 3, 271–295.

TINKER, P. 1988. Performance of an OR-Parallel Logic Programming System. *International Journal of Parallel Programming 17*, 1, 59–92.

TONG, B. AND LEUNG, H. 1993. Concurrent Constraint Logic Programming on Massively Parallel SIMD Computers. In *Proceedings of the International Logic Programming Symposium*, D. Miller, Ed. MIT Press, Cambridge, MA, 388–402.

TONG, B. AND LEUNG, H. 1995. Performance of a Data-Parallel Concurrent Constraint Programming System. In *Proceedings of the Asian Computing Science Conference*, K. Kanchanasut and J.-J. Levy, Eds. Springer-Verlag, Heidelberg, 319–334.

TRAUB, K. 1989. Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM Press, New York, 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. *New Generation Computing 11*, 3/4, 323–341.

ULLMAN, J. D. 1988. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Maryland.

VAN HENTENRYCK, P. 1989. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In *Proceedings of the Sixth International Conference on Logic Programming*, G. Levi and M. Martelli, Eds. MIT Press, Cambridge, MA, 165–180.

VAN HENTENRYCK, P., SARASWAT, V., AND DEVILLE, Y. 1998. Design, Implementation and Evaluation of the Constraint Language cc(FD). *Journal of Logic Programming 37*, 1–3, 139–164.

VAN ROY, P. 1990. Can Logic Programming Execute as Fast as Imperative Programming? Ph.D. thesis, U.C. Berkeley.

VAN ROY, P. 1994. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming 19/20*, 385–441.

VAN ROY, P. AND DESPAIN, A. 1992. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer 25*, 1, 54–68.

VAUPEL, R., PONTELLI, E., AND GUPTA, G. 1997. Visualization of And/Or-Parallel Execution of Logic Programs. In *Proceedings of the International Conference on Logic Programming*, L. Naish, Ed. MIT Press, Cambridge, MA, 271–285.

VÉRON, A., SCHUERMAN, K., REEVE, M., AND LI, L.-L. 1993. Why and How in the ElipSys Or-Parallel CLP System. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, A. Bode, M. Reeve, and G. Wolf, Eds. Springer-Verlag, Heidelberg, 291–303.

VILLAVERDE, K., GUO, H.-F., PONTELLI, E., AND GUPTA, G. 2000. Incremental Stack Splitting. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, I. C. Dutra, Ed. Federal University of Rio de Janeiro, London.

VILLAVERDE, K., PONTELLI, E., GUPTA, G., AND GUO, H. 2001. Incremental Stack Splitting Mechanisms for Efficient Parallel Implementation of Search-Based Systems. In *International Conference on Parallel Processing*. IEEE Computer Society, Los Alamitos, CA.

WALLACE, M., NOVELLO, S., AND SCHIMPF, J. 1997. ECLiPSe: A Platform for Constraint Logic Programming. Tech. rep., IC-Parc, Imperial College.

WARREN, D. H. D. 1980. An Improved Prolog Implementation Which Optimises Tail Recursion. Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh.

WARREN, D. H. D. 1983. An Abstract Prolog Instruction Set. Technical Report 309, SRI International.

WARREN, D. H. D. 1987a. The Andorra Principle. Presented at Gigalips workshop, Unpublished.

WARREN, D. H. D. 1987b. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT*, H. E. et al., Ed. Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 243–259.

WARREN, D. H. D. 1987c. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *Proceedings of the Symposium on Logic Programming*. IEEE Computer Society, Los Alamitos, CA, 92–102.

WARREN, D. H. D. AND HARIDI, S. 1988. Data Diffusion Machine—A Scalable Shared Virtual Memory Multiprocessor. In *Proceedings of the International Conference on Fifth Generation Computer Systems*. ICOT, Springer-Verlag, Tokyo, Japan, 943–952.

WARREN, D. S. 1984. Efficient Prolog Memory Management for Flexible Control Strategies. In *Proceedings of the Symposium on Logic Programming*. IEEE Computer Society, Los Alamitos, CA, 198–203.

WEEMEEUW, P. AND DEMOEN, B. 1990. Memory Compaction for Shared Memory Multiprocessors, Design and Specification. In *Proceedings of the North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, MA, 306–320.

WESTPHAL, H., ROBERT, P., CHASSIN DE KERGOMMEAUX, J., AND SYRE, J. 1987. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Proceedings of the Symposium on Logic Programming*. IEEE Computer Society, Los Alamitos, CA, 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, Los Alamitos, CA, 134–152.

WINSBOROUGH, W. AND WAERN, A. 1988. Transparent And-Parallelism in the Presence of Shared Free Variables. In *Fifth International Conference and Symposium on Logic Programming*. 749–764.

WISE, D. S. 1986. *Prolog Multiprocessors*. Prentice-Hall, New Jersey.

WOLFSON, O. AND SILBERSCHATZ, A. 1988. Distributed Processing of Logic Programs. In *Proceedings of the SIGMOD International Conference on Management of Data*, H. Boral and P. Larson, Eds. ACM, ACM Press, New York, 329–336.

WOO, N. AND CHOE, K. 1986. Selecting the Backtrack Literal in And/Or Process Model. In *Proceedings of the Symposium on Logic Programming*. IEEE, Los Alamitos, CA, 200–210.

XU, L., KOIKE, H., AND TANAKA, H. 1989. Distributed Garbage Collection for the Parallel Inference Engine PIE64. In *Proceedings of the North American Conference on Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cambridge, MA, 922–943.

YANG, R. 1987. P-Prolog: A Parallel Logic Programming Language. Ph.D. thesis, Keio University.

YANG, R., BEAUMONT, A., DUTRA, I. C., SANTOS COSTA, V., AND WARREN, D. H. D. 1993. Performance of the Compiler-Based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, MA, 150–166.

ZHONG, X., TICK, E., DUVVURU, S., HANSEN, L., SASTRY, A., AND SUNDARARAJAN, R. 1992. Towards an Efficient Compile-Time Granularity Algorithm. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT Staff, Ed. Ohmsha Ltd., Tokyo, Japan, 809–816.

ZIMA, H. AND CHAPMAN, B. 1991. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York.

Received June 2000; revised October 2000; accepted March 2001