Parallel logic programming systems

J. Chassin de Kergommeaux, Philippe Codognet

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

1992
25ème anniversaire

N° 1691

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

PARALLEL LOGIC PROGRAMMING SYSTEMS

Jacques CHASSIN de KERGOMMEAUX
Philippe CODOGNET

Mai 1992

RR.1691

Parallel Logic Programming Systems

Programmation Logique Parallèle

Jacques Chassin de Kergommeaux
IMAG/LGI
46, avenue Felix Viallet,
38031 Grenoble, FRANCE
chassin@imag.fr

Philippe Codognet
INRIA - Rocquencourt
B. P. 105
78153 Le Chesnay, FRANCE
codognet@minos.inria.fr

Abstract

Parallelizing logic programming has attracted much interest in the research community, because of the intrinsic OR- and AND-parallelisms of logic programs. One research stream aims at transparent exploitation of parallelism in existing logic programming languages such as Prolog while the family of Concurrent Logic Languages develops language constructs allowing programmers to express the concurrency, that is the communication and synchronization between parallel processes, inside their algorithms. This paper mainly concentrates on transparent exploitation of parallelism and surveys the most mature solutions to the problems to be solved in order to obtain efficient implementations. These solutions have been implemented and the most efficient parallel logic programming systems reach effective speedups over state-of-the-art sequential Prolog implementations. The paper also addresses current and prospective research issues aiming to extend the applicability and the efficiency of existing systems, such as models merging the transparent parallelism and the concurrent logic languages approaches, combination of constraint logic programming with parallelism and use of highly parallel architectures.

Résumé

La Parallélisation des langages logiques est un domaine de recherche très actif, du fait des parallélismes OU et ET intrinsèques aux programmes logiques. Une voie de recherche s'intéresse à l'exploitation d'un parallélisme transparent dans les langages logiques existants tel Prolog, tandis qu'une autre direction est prise par les Langages Logiques Concurrents qui developpent des primitives pour permettre d'exprimer la concurrence, c'est à dire la communication et la synchronisation, à l'interieur des programmes. Cet article s'intéressera surtout à l'exploitation transparente du parallélisme et présentera les solutions les plus avancées pour obtenir des mises en oeuvre efficaces. Ces solutions ont été implémentées et les systèmes de programmation logique parallèle les plus efficaces atteignent des "speedups" réelles par rapport aux implémentations séquentielles de Prolog. Nous présenterons également les thèmes de recherches actuels et prospectifs destinés à étendre l'applicabilité et l'efficacité des systèmes existants, tels les modèles qui combinent parallélisme transparent et langages logiques concurrents, l'extension de la programmation logique avec contraintes par le parallélisme, et l'utilisation des machines massivement parallèles.

Parallel Logic Programming Systems

Jacques Chassin de Kergommeaux
CMaP Group,
IMAG/LGI,
46 avenue Félix Viallet,
F-38031 Grenoble Cedex, France.
chassin@imag.fr

Philippe Codognet
INRIA,
Domaine de Voluceau,
Rocquencourt,
F-78153 Le Chesnay Cedex
codognet@minos.inria.fr

Abstract

Parallelizing logic programming has attracted much interest in the research community, because of the intrinsic OR- and AND-parallelisms of logic programs. One research stream aims at transparent exploitation of parallelism in existing logic programming languages such as Prolog while the family of Concurrent Logic Languages develops language constructs allowing programmers to express the concurrency, that is the communication and synchronization between parallel processes, within their algorithms. This paper concentrates mainly on transparent exploitation of parallelism and surveys the most mature solutions to the problems to be solved in order to obtain efficient implementations. These solutions have been implemented and the most efficient parallel logic programming systems reach effective speedups over state-of-the-art sequential Prolog implementations. The paper also addresses current and prospective research issues aiming to extend the applicability and the efficiency of existing systems, such as models merging the transparent parallelism and the concurrent logic languages approaches, combination of constraint logic programming with parallelism and use of highly parallel architectures.

1 Introduction

Logic programs can be computed sequentially or in parallel without changing their declarative semantics. For this reason, they are often considered as well suited to programming multiprocessors. At the same time, parallel architectures represent the most promising solution to increase the computing power of computers in the future. Since multiprocessors remain difficult to use efficiently, an implicitly parallel programming language offers a very attractive mean of exercising the parallelism of multiprocessors of today and tomorrow.

Logic programming languages are very high level languages enabling programs to be developed more rapidly and concisely than using imperative languages. However, in spite of important progresses of the compilation techniques for these languages, they remain less efficient than imperative languages and their use is mainly constrained to prototyping. Increasing the efficiency of logic programming to the level of imperative languages would certainly enlarge their domain of use and could therefore contribute to solve the so-called "software crisis" by raising the productivity of programmers.

These reasons have persuaded the ICOT to choose logic programming as the basic programming language of the Fifth Generation Computer Systems project (FGCS). One aim of this project is to produce multiprocessors delivering more than one giga-lips, one

"lip" being one logical inference per second and one inference being similar to a procedure call of an imperative language. The giga-lips level of performance seemed out of scope when the FGCS project started in 1982, since the most efficient Prolog systems of this period were limited to several kilo-lips of performance. This is not the case any more, mainly because of the spectacular progresses made by the hardware technology. The performances of the most efficient Prolog implementations exceed one mega-lips on the most powerful RISC micro-processors of today while massively parallel multiprocessors can include more than 1000 of them. However, achieving giga-lips level of performance on massively parallel multiprocessors will only be possible if parallelizing techniques capture enough parallelism of logic programs while keeping limited the overhead of parallel execution. In addition, it is not clear whether a large number of logic programs can benefit from massive parallelism.

There exists two main schools of thought in the logic programming community, considering either that parallelism should be made explicit or on the contrary that it should be kept implicit. Explicit parallelism is the approach developed in concurrent logic languages, aimed at being the basic programming languages of future multiprocessors and already successfully used to program parallel problems such as operating systems and parallel simulators. The aim of implicit parallelism is instead to speedup existing or future logic programs without troubling programmers with parallelism. Both approaches will be presented in this paper, although more emphasis will be placed on systems exploiting parallelism implicitly.

Parallel logic programming has benefited from a considerable amount of research activities which resulted in the definition of a large number of parallel computational models. These models cannot be all presented in this paper which will concentrate on the most mature ones, already used in efficient parallel implementations.

The organisation of the paper is the following. The first sections are introductive, defining the different types of parallelism that can be exploited in logic programming, introducing the language issues involved in implicit versus explicit parallelism and raising implementation issues involved in implementing efficiently logic programming in parallel. The two following sections survey representative systems exploiting one type of parallelism or combining several types of them. Several important research topics and perspectives are then sketched before the conclusion of the paper.

2 Parallelisms in Logic Programming

Due to their declarative and logical essence, different kinds of parallelism naturally arise in logic programs. Each type of parallelism will indeed lead to a specific model of execution which builds the foundations for the systems that will be presented later.

2.1 Logic programs

This brief and intuitive presentation of the basic notions of logic programs is intended to make the paper self-contained. A more complete introduction to logic programming can be found in John Lloyd's book [65].

A *logic program* is a set of definite clauses such as:

$$A \leftarrow B_1, ..., B_n \quad (1)$$

$$A \leftarrow \quad (2)$$

and one *query*

$$\leftarrow Q_1, ..., Q_p \tag{3}$$

The A_i and B_j are atomic formulas such as $p(X_1, ..., X_m)$, where p is a predicate symbol and $X_1, ..., X_m$ are compound terms, constants or simple variables. (1) can be logically read as " if B_1 and B_2 ... and B_n are true, then A is true ", and (2) as " A is a fact always true ". In (1), A is the *head* of the clause while the conjunction of the B_i is called its *body*. Each of the B_i is called a *goal*. Executing a logic program P amounts to proving that (3) is a *logical consequence* of the conjunction of the clauses of P.

The procedural interpretation of Horn clauses forms the basis of the operational semantics of logic programming. Indeed, the set of all clauses with the same head predicate symbol p can be considered as the definition of the *procedure* p. Each goal of the body of a clause can thus be considered as a procedure call. At a given step of the execution of a logic program, the set of goals that remain to be executed, i.e. the continuation of the computation, is called the current *resolvent*. The execution of a logic program starts by taking the query as initial resolvent and proceeds by transforming the current resolvent into a new one as follows. First select any goal G of the resolvent, and apply a *resolution step*, which consists in finding a clause of the program whose head unifies with G_1, and then replacing G by the body of the unifying clause to produce a new resolvent. The *bindings* of variables resulting from the unification apply to the whole resolvent. This process iterates until either the resolvent is empty, in which case the computation ends with *success*, or no unifying clause can be found during a resolution step, in which case a *failure* occurs and *backtracking* takes place. Backtracking consists in restoring the previous resolvent and selecting an alternative unifying clause in order to perform an alternative resolution step. If all possible alternatives have been tried unsuccessfully and it is not possible to backtrack further, the computation terminates with failure.

Let us detail this machinery by a simple example.

Example 2.1

Consider the following "genealogy" program, written with Prolog's identifier convention : variables begin with a capital letter, while predicate and function symbols begin with a lower-case letter.

```
grandfather(X, Z) ← father(X, Y), father(Y, Z)

father(john, philip)
father(peter, andy)
father(andy, mark)
```

with the query

```
← grandfather(X, mark)
```

The query is equivalent to the request : "Find a person, whose grandfather is Mark".

Let us detail the execution of this program by examining the resolvents produced at each step.

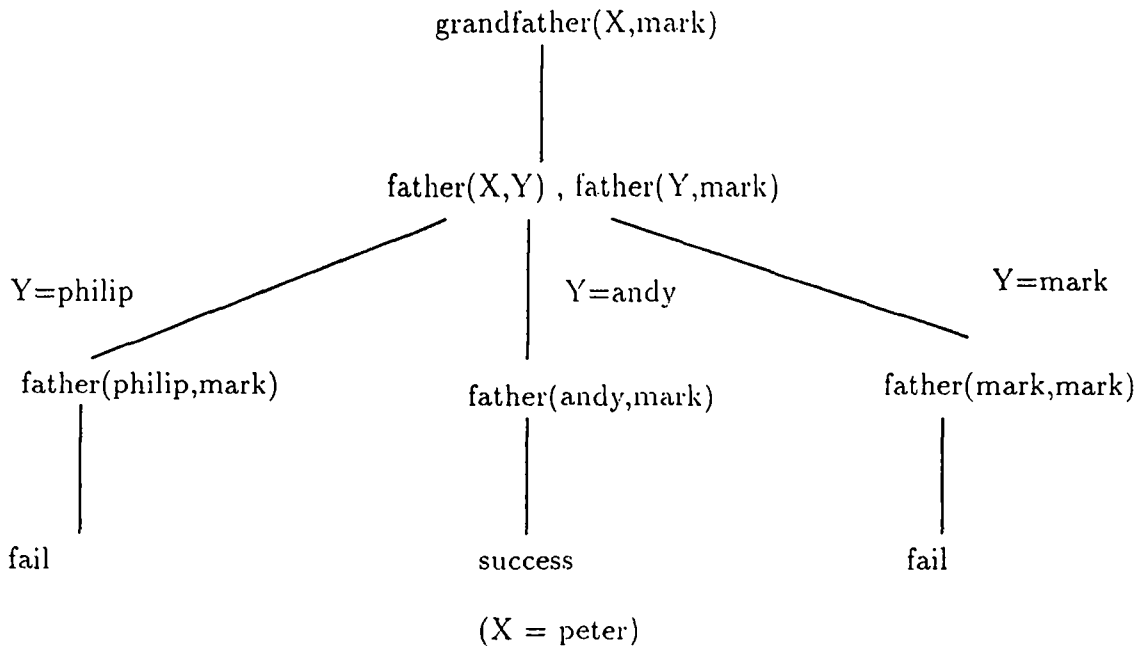

Figure 1: Search tree for tutorial example

First resolvent: ← `grandfather(X, mark)`
Second resolvent: ← `father(X, Y), father(Y, mark)`
Third resolvent: ← `father(philip, mark)`
No head clause of the program unifies with the unique goal of the resolvent. Therefore backtracking occurs and the last choice, the second resolvent, is restored :
← `father(X, Y), father(Y, mark)`
Fourth resolvent: `father(andy, mark)`
Fifth resolvent: □ (empty clause)

The execution of this logic program is successful and the result is the binding of the variable of the query, that is $X = peter$, which is called an *answer substitution*.

This computation can be graphically summarized by depicting the associated *search tree*, shown in Figure 1. The nodes of the tree are the resolvents occuring in the computation, and the arcs are labeled by the set of bindings of the corresponding resolution step.

A sequential computation according to Prolog's strategy simply consists in a depth-first left-to-right search of the tree.

2.2 Sources of parallelism in logic programs

There are two intrinsic sources of parallelism in the above execution model, that correspond to the two choices that have to be made to enforce the sequentialization of an inherently parallel model.

The first choice is the selection of a goal in the resolvent in order to perform a reso-

lution step. One may envisage selecting several goals of the resolvent and performing all resolution steps simultaneously. Such parallelism is called *AND-parallelism*, as it consists of developing in parallel goals that are connected by a logical AND operator.

The second one is the selection of a unifying clause in the program when performing a resolution step and proceeding to a new resolvent. If there exist several such unifying clauses, it is then possible to perform several alternative resolution steps in parallel, creating therefore several new resolvents upon which the computation to proceed. Such a mechanism is called *OR-parallelism*, as it amounts to develop simultaneously different alternative computations, and more precisely different OR-branches of the search tree as will now be detailed.

2.3 OR-parallelism in Prolog

OR-parallelism consists in the simultaneous development of several resolvents that would be computed successively by backtracking in a sequential execution. Examining the search tree (Figure 1), an OR-parallel search consists of exploring in parallel each branch of the tree, these branches being indeed OR-branches representing alternative computations. In our example, the computation may split into three OR-branches when computing the second resolvent, using the three clauses of the *father* procedure. Of course, only the second branch succeeds at the next computation step.

It is worth noticing that in our definition of OR-parallelism, simultaneous independent execution is not limited to a single resolution step, and thus reduced to a mere database parallelism while searching for a matching clause, but applies to the entire computation of *complete resolvents*. Thus, if the OR-split occurs early enough in the computation, the remaining alternative resolvents might involve large computations, yielding therefore coarse grain parallelism.

The main problem in OR-parallelism is to manage different *independent* resolvents in parallel, in each of the OR branches. This is not a problem in the example above where the resolvents are very small but in general care must be taken to avoid inefficient copying of large data structures, such as the current set of variables bindings. Sophisticated algorithms and data structures have been designed to overcome this problem, which will be presented later. Another issue to be addressed in this "eager evaluation" of alternative choices is to tune the amount of parallelism in order to avoid the system to collapse under too many processes. Efficient solutions mix backtracking and on-demand OR-parallelism (by idle processors) in so-called "multi-sequential" models.

2.4 AND-parallelism in Prolog

AND-parallelism consists in the simultaneous computation of several goals of a resolvent. Nevertheless, if each subcomputation is followed independently, one needs to ensure the compatibility of the binding sets produced by each parallel branch.

Consider again the computation in example 2.1, and more precisely the second subgoal ← `father(X, Y), father(Y, mark)` , which has potential AND-parallelism. The execution of the two parallel AND branches results in the following candidate solution sets:

Branch 1: goal : `father(X, Y)`
Variable bindings: `{X=john, Y=philip}, {X=peter, Y=andy}, {X=andy, Y=mark}`
Branch 2: goal : `father(Y, mark)`

Variable binding:{Y=andy}
Obviously, only the second set of bindings of the first branch is compatible with the unique binding produced by the second branch. One must "join" binding sets coming from different AND-branches in order to form a valid solution for the entire resolvent.

The main difficulty with AND-parallelism is indeed to obtain coherent bindings for the variables shared by several goals executed in parallel. Runtime checks can be very expensive. This has lead to design execution models where goals which may possibly bind shared variables to conflicting values are either serialized or synchronized. The first solution leads to *independent* AND-parallelism: only independent goals which do not share any variable are developed in parallel. The second solution leads to *dependent* AND-parallelism: concurrent execution of goals sharing variables is possible. Such goals are synchronized by producer/consumer relationships on the shared variables. This approach has been developed in "Concurrent Logic Programming languages" such as Parlog, KL1 and Concurrent Prolog.

3 Language issues

The first attempts in the early 80's to parallelize Prolog and to define new models of execution for logic programs raised a number of issues, including some key points in language design. The main debate can be summarized by the following question: "Do we need a new language with specific constructs to express concurrency and parallelism or should we stick to Prolog and exploit parallelism transparently?"

Prolog has been in use for nearly two decades and has established itself as a useful tool for a wide range of problems. Declarative languages are always in demand for more and more computing power, due to the displacement of the complexity of programming from the user to the internal computation mechanism. Efficient parallel implementations of existing languages are thus desirable to speed applications that are already developped, and they can moreover lead logic programming a step further in covering effective "real-world" applications.

On the other hand one may sometimes prefer a more flexible control of the search mechanism to the the simple but somehow rigid search control of Prolog. This brings up the problem of communication and synchronization in logic programs that calls for new features and changes in language design. Thus was born the new domain of concurrent logic languages, also called sometimes committed-choice languages. It has paved the way to a wide range of new applications that are more easily modeled by multiple cooperating concurrent processes.

3.1 Parallelizing Prolog

Prolog was originally designed with a sequential target machine in mind. However, parallelizing the language without change is an appealling approach for the following reasons:

- parallelism can be transparently achieved with "minimal" changes to the initial model, at least at the abstract level, as described in the previous section. All the implementation technology developed for sequential Prolog systems, such as the WAM abstract machine [102], can thus be reused. The most successful offsprings of this approach are the multi-sequential models for both OR and AND parallelism that will be presented in section 4.

- parallel execution does not add any complexity to the programming language; it helps the user to concentrate on declarative statements without bothering with control issues. This view keeps up with the separation of logic and control advocated by Kowalski in his well-known formula "program = logic + control" [60].

- the corpus of Prolog programs already developed can be executed without any modification to the programs' source code by parallel systems.

Some problems however arise when considering the Prolog constructs that are order-sensitive. Most side-effect primitives, as for instance *read* or *write*, require what could be called AND-sequentiality, and are thus necessary sequentialization points for AND-parallel models. OR-parallel models however, which keep the sequential order of execution of goals, are not sensitive to the problems due to AND-sequentiality.

Some OR-sequentiality is nevertheless required by some "impure" or "non-logical" features such as the cut operator or the assert/retract primitives which dynamically manipulate program clauses. These constructs will cause problems in OR-parallel execution, because they are sensitive to the order in which OR-branches are explored. The cut operator is used to make determinate parts of the computation. It appears syntactically as a special goal in the body of a clause and will affect, when executed, the part of the computation performed after the entry of the clause, by removing all choice-points that have been created since. No backtracking in this part of the computation is then possible. All the computation work corresponding to the development of the alternative clauses, that may be pruned by the cut, is called *speculative work*. Scheduling speculative work is difficult in OR-parallel models [48], since anticipation may amount to useless computation while expectancy may prevent from parallel execution. To avoid the inherent sequentiality of the cut operator, it is useful to introduce in OR-parallel Prolog a "symmetric cut" that prunes *all* the alternatives of a predicate, whether they appear before of after the clause containing the operator. Another possibility is to enforce a discipline of programming in order to use cut operators in all clauses of a predicate or in none of them, thus introducing the "commit" operator of concurrent logic languages.

3.2 From Coroutining to Concurrency

In Prolog, control of forward execution is given by the order of the goals inside a clause and that of the clauses in the program. However, a more flexible control strategy is sometimes needed. Dialects such as Prolog-II, MU-Prolog or Sicstus Prolog provide primitives to explicitely *delay* the execution of some goals. Roughly speaking, a predicate can be given a *wait* declaration specifying that it should not be executed before some of its arguments are instantiated. This corresponds to declaring this goal as consumer-only on those arguments.

A more direct and precise control mechanism has been introduced by the family of concurrent logic languages. All these languages have some syntactic way to declare that a process (goal) is either *producer* or *consumer* of some its arguments. Such declarations are thus used to produce a *synchronization* mechanism between active processes, i.e. goals of the resolvent. When unification leads a goal to instantiate a consumed variable, this goal is *suspended* until that variable is sufficiently instantiated (by some other goal).

3.3 Concurrent Logic Languages

The usual logic programming framework is limited to *transformational* systems, i.e. systems that, given an original input, transform it to yield some output at the end, and *as* the end. This framework is not well suited to *reactive* systems, i.e. "open" systems able to react to a continuous stream of inputs from the "real world". In other words, if Prolog is perfectly adequate for problem solving, it is not suited to "dynamic" or "reactive" systems, which are more easily modelized by interacting agents. To address these problems, communication and synchronization techniques derived from the concurrency theory [33] have been introduced in logic programming, giving rise to concurrent logic programming.

The basic notions for the integration of concurrency into logic programs can be traced back to Relational Language [19], the ancestor of (con)current languages such as Parlog [20], GHC and KL1 [97], Concurrent Prolog [89] and FCP [90], or CP [84]. The reader should consult [90] for a detailed genealogy, history and a complete presentation of this programming paradigm.

The main programming concepts behind those languages can be summarized up as follows:

- The *process interpretation* of logic programs [89] replaces the traditional procedural interpretation. Each goal is seen as a distinct process, AND-connected goals are assumed to run concurrently. This form of AND-parallelism is called dependent AND-parallelism, as goals that share variables (hence "linked" or dependent) can run in parallel.
- *Communication* is achieved through *logical variables*: each variable shared by several goals/processes act as a communication channel, leading to a very powerful and elegant communication mechanism.
- *Synchronization* is achieved by using a producer/consumer relationship between processes. A process may be blocked during a resolution step until the variables that it consumes are sufficiently instantiated (by some other processes).

Several new major language features are therefore introduced (see example below):

1. some way of stating a producer/consumer mode for each variable has to be introduced. Each language has his own way to declare the access mode (*Read* or *Write*) of a variable in the predicate declaration that we will not detail here. These modes are used dynamically to induce a producer/consumer relationship between active processes (goal of the resolvent) on each shared variable, i.e. on each communication channel. This technique amounts to replacing the *unification* procedure involved at each resolution step by a *matching* procedure for consumer goals. This notion can be in fact rephrased and generalized in the *Ask-and-Tell* mechanism of concurrent constraint languages that will be presented later.
2. the concept of *guard* introduced by Dijkstra and used for imperative languages such as ADA and Occam is adapted to logic programming. A guard is a switch construct of condition/action pairs, which delays execution until one of the conditions is true, and executes the corresponding action as soon as a single condition is verified. This translates to logic programs as follows: each clause is given an additional *guard* part which takes place between the head and the body. A guard is simply a sequence of

goals that must be executed successfuly before the body of the clause is entered (i.e. all body goals are spawned in parallel). Implementation considerations have led the last generation of languages such as KL1 or FCP (Flat Concurrent Prolog) to accept only *flat* guards, i.e. guards composed only with buit-in predicates. This ensures that no hierarchy of guard systems will ever be created (hence the name) and that guard check will be a reasonably fast and basic operation.

3. *"Don't care" non-determinism* replaces the traditional "Don't know" non determinism of logic languages. This means that at each clause try, only one alternative is pursued (and we don't care which) as opposed to pursuing all of them (as we don't know which one to choose). Alternative terminologies are "indeterminism" for Don't care non-determinism and "angelic non-determinism" for Don't know non-determinism. Hence no choice point is ever created and no backtracking ever takes place, resulting in a *deterministic* language. The only amount of non-determinism lies in the guard check, as all guards are evaluated in parallel. Among all satisfiable guards, one of them is chosen and the computation is *committed* to the corresponding clause, neglecting alternative ones. This mechanism is syntactically expressed by the presence of a *commit* operator between the guard and the body of each clause. The commit indeed corresponds to a generalized "cut" operator.

Concurrent Logic Languages hence depart from traditional logic languages in several ways, most notably by the replacement of unification by matching and by the abandonment of Don't know ("angelic") non-determinism. The former is the key mechanism for the synchronization mechanism and is the price to pay for a new programming style. The latter is however only supported by implementation reasons, as a simple backtracking scheme (such as the chronological backtracking of Prolog) was not easy in a concurrent environment.

To give a flavor of what a concurrent logic program is, let us consider the following *merge* program, written following the syntax of KL1 or FCP(—). This program merges the two input streams (lists) in its first two arguments to produce an output stream in its third argument.

```
merge([X|In1],In2,Out) <- true | Out=[X|Out'], merge(In1,In2,Out') .
merge(In1,[X|In2],Out) <- true | Out=[X|Out'], merge(In1,In2,Out') .
merge([],In2,Out) <- true | Out=In2 .
merge(In1,[],Out) <- true | Out=In1 .
```

This program is very simple: all guards being empty (`true` predicates indicate empty guard), synchronization will be enforced by the compound terms present in the head of the clauses. Since unification is replaced by matching, the presence of a compound term in the head of a clause for an argument place enforces a *consumer mode* for this argument. Thus a *merge* process will wait until some data arrives on one of its input streams (i.e. a "cons" is produced in one input list). When an input chanel is closed, the input stream is reduced to the empty list, *merge* simply produces what it is feeded with by the remaining input channel. This program preserves the order of the elements in both input streams, but guarantees nothing about the rate at which each stream will be served. More complex programs, such as a *fair* merger which guarantees that every data produced by one input stream will eventually be written on the output stream, are presented in [90].

3.4 Unifying Prolog and Concurrent Logic Languages

As described above, parallel Prolog and Concurrent Logic Languages address two distinct application areas:

- Prolog is more suited for declarative higher-level programs, such as non-deterministic, search-intensive problem solving.
- Concurrent logic languages are more suited when explicit control matters and fine-grain coordination is essential, when the problem is more easily modelled by communicating agents.

It was thus natural to try to encompass both paradigms in a single language, for the best of both worlds!

The *Andorra model* was proposed by D.H.D. Warren [100] to combine OR-parallelism and dependent AND-parallelism, and it has now bred a variety of idioms and extensions developed by different research groups. The essential idea is to execute determinate goals first and in parallel, delaying the execution of non-determinate goals until no determinate goal can proceed any more. This was inspired by the design of the concurrent language P-Prolog [107] where synchronization between goals was based on the concept of determinacy of guard systems. However the roots of such a concept can be trailed back further to the early developments of Prolog, as for instance in the *sidetracking* search procedure of [79] which favors the development of goals with the fewer alternatives (and hence determinate goals as a special case). This is indeed but another instance of the old "first fail" heuristic often used in problem solving. An interesting aspect of the Andorra principle, is its capability to reduce the size of the computation when compared to standard Prolog, as early execution of determinate goals can amount to an *a priori* pruning of the search space (see section 6.2).

The ability of delaying non-determinate goals as long as determinate ones can proceed amounts to a synchronization mechanism managed by determinacy. The programming style of concurrent logic languages, such as cooperation between a producer process and a consumer process, can thus be achieved by making the consumer non-determinate (and hence blocked) as long as the producer has not produced a value, which would then wake up the consumer by making him determinate. Andorra-based models, such as Andorra-I [25] or Andorra Kernel Language [45], thus support both Prolog and Concurrent Logic Languages styles of programming. Moreover, the Andorra Kernel Language is an attempt to fully encompass both Prolog and concurrent logic languages. The main new language feature is the introduction of the guard construct borrowed from the concurrent logic paradigm, and its associated guard operator. Both don't care and don't know non determinism are possible by using as guard operators either "commit", "cut" or "wait" (which does not prune alternatives). Guards are not restricted to build-in predicates but can be program goals, leading to non flat guards. The introduction of guards in the program's clauses is indeed a way to extend the determinacy test, which is at the core of the Andorra model, since a goal is determinate whenever a sole guard check succeeds, among all possible alternatives, and not only a sole head unification.

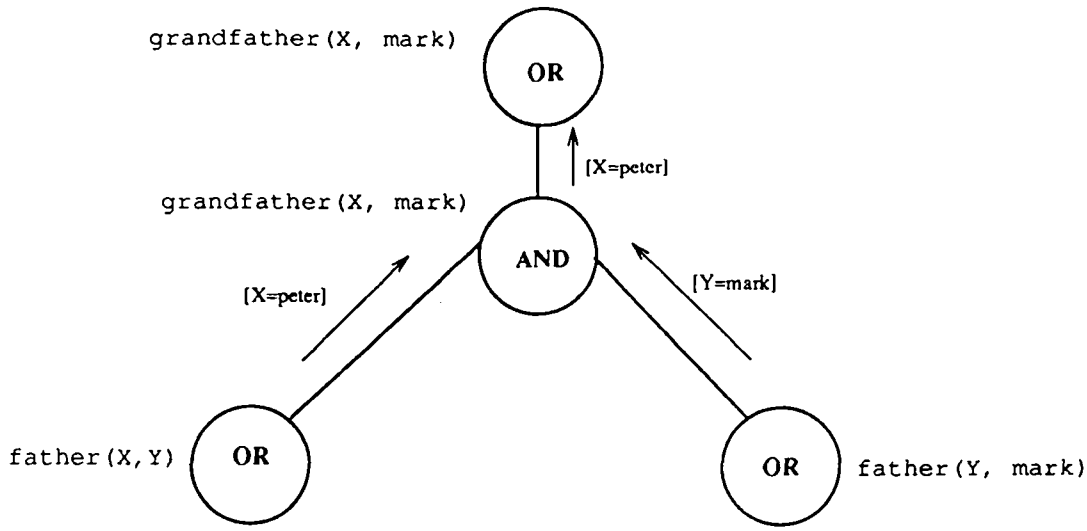

Figure 2: Processes generated by the AND-OR process model

The OR process computing *father(X, Y)* is not created by its father AND process until Y has been bound by the OR process computing *father(Y, mark)*

4 Implementation issues

This section emphasizes the main problems arising in implementing *efficiently* parallel logic programming systems.

4.1 Early models

Early computational models [24] [15] proposed to parallelize logic programming systems were based on "natural" data-flow process models. In the AND-OR process model [24], AND processes are created to compute the body of clauses while OR processes are created for each goal of clause bodies (see Figure 2). Each of these processes itself creates child processes, therefore building a tree of processes which exchange messages such as partial solutions. AND processes use complex algorithms to dynamically reorder the goals of their clause body, such that no two OR processes which could bind variables to conflicting values get activated at the same time.

Early models have in common the drawback of generating a large number of processes of small granularity which perform a considerable amount of runtime checks and unbounded length communications.

4.2 Multisequential systems

The aim of multisequential systems is to limit the overheads arising from process creation and communication and from runtime checks. In these systems, the amount of parallelism used is adapted automatically to the resources available. Computation is performed by a number of **workers**, the amount of which being close to the number of processing elements available at runtime. In the course of the computation, workers may be "active", that is

```
process_fathers(Y)      :- father(X, Y), long_computation(Y).
long_computation(P)     :- ...

father(john, philip).
father(peter, andy).
father(andy, mark).
```

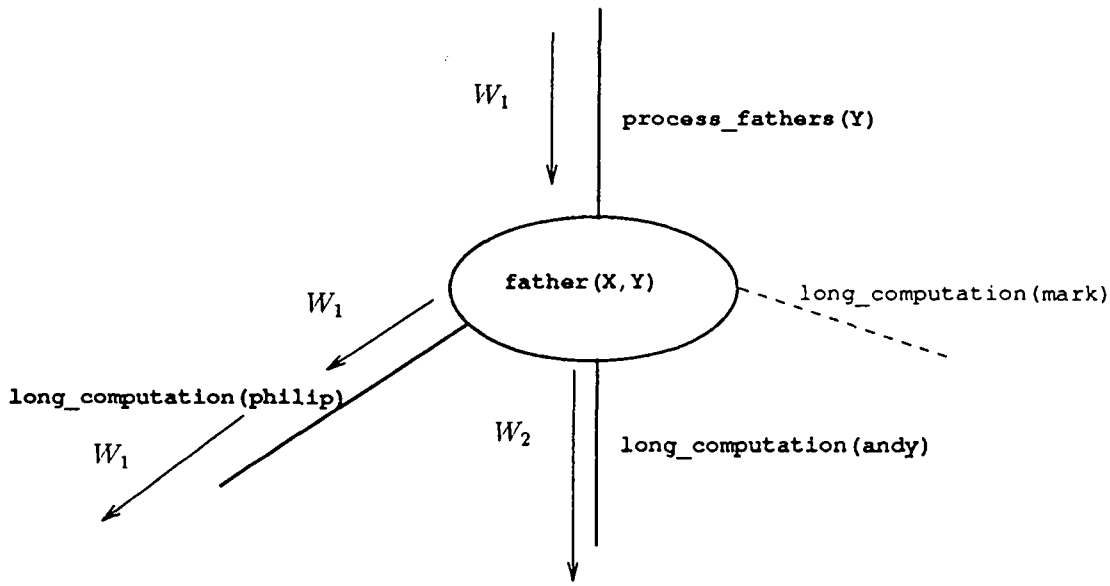

Figure 3: Multisequential OR-parallel computation by two workers

Workers W_1 and W_2 cooperate to process the computation tree of the program. The third branch originating from the *father* node will be processed by the first one of the two workers available to process it.

computing part of the program, or "idle", that is searching for work from the active workers or from other parts of the computational state. Untried alternatives, recorded at OR nodes of the search tree, represent potential OR-work (see Figure 3).

Multisequential systems use the most efficient implementation techniques of sequential Prolog systems. Most existing multisequential systems are based on the compilation of source programs into a particular abstract machine called Warren Abstract Machine or *WAM* [102]. Each worker of a multisequential system includes an efficient sequential logic computation engine.

The WAM is composed of a set of instructions operating on registers and data structures managed in several stacks. Source Prolog programs are compiled into instructions which may be interpreted or translated into target machine code. Most of the instructions correspond to compiling the unification, which is the core operation of Prolog systems. In addition, indexing instructions compile the selection of the candidate clause (callee) depending on the arguments of the current goal (caller). The other instructions are used for control, the first type of control being similar to the procedure call of imperative languages, the other type, backtracking, being specific to logic programming.

The WAM manipulates three stacks called *local, global* and *trail* stacks (see Figure 4). The local and global stacks are similar to the stack and heap used in the implementations of imperative languages. The local stack contains the clause activation records called *environments*. The global stack is used to store persistent bindings and is often managed as a heap. In addition, several data structures are needed to manage the Prolog *don't know* non-determinism, that is to perform backtracking efficiently. When a backtrackable choice is performed in the course of the computation of a Prolog program, the values of the registers, describing the state of the computation, are pushed on the local stack, in a data structure called *choice-point*. When there is no candidate clause for the current goal, the registers are restored from the latest choice-point, local and global stacks are popped to their previous contents and computation proceeds with an untried alternative. However, several bindings of the remaining sections of the local and global stacks may also need to be undone if they did not exist in the previous state being restored. The aim of the trail stack is to keep track of such bindings. Whenever a variable, older than the latest choice-point, is bound on a stack, its address is recorded in the trail stack. When backtracking occurs, all variables whose addresses have been recorded in the trail since the last choice-point, are reset.

4.3 Problems with OR-parallelism

The main problem to be solved by multisequential OR-parallel logic systems originates in the implementation techniques used for efficiency reasons by Prolog implementations. In theory, a new resolvent is generated at each unification of a goal against the head of a clause, by renaming all the variables of the current resolvent. In practice, resolvents are not copied and the same memory locations are used to compute alternative resolvents. This is made possible by the backtracking operation (see section 2.1). The backtracking operation restores a previous state of the computation and in particular pops the stacks.

In most sequential Prolog implementations, variables are identified by their memory locations. At any given time during the computation, no more than one variable is associated to a variable location in the (local or global) stack. When backtracking occurs, some variable locations do not belong to the stack space being popped and need to be reset so that they can be reused to store **new** variables. The addresses of the variables needing to be reset are stored in the trail stack (see Figure 4).

In OR-parallel systems, concurrent computations may be performed instead of a series of sequential computations interleaved with backtracking operations. Several workers may therefore associate the same variable location of a stack to different logical variables. Several solutions have been proposed to cope with this problem, one family of solutions being based on copying of stacks, another family of solutions sharing the stacks instead, the latest solution reconstructing the stacks by recomputation.

4.3.1 Copying of stacks

In the stack copying scheme, each worker maintains a complete copy of the stacks in its workspace. When getting work, an idle worker copies the stacks of the computation state down to the node providing work. In the example of Figure 3 and Figure 4, a worker grabbing work from the *father* node would have to copy the stacks from the root to the *father* node.

```
process_fathers(Y)      :- father(X, Y), long_computation(Y).
long_computation(P)     :- ...

father(john, philip).
father(peter, andy).
father(andy, mark).
```

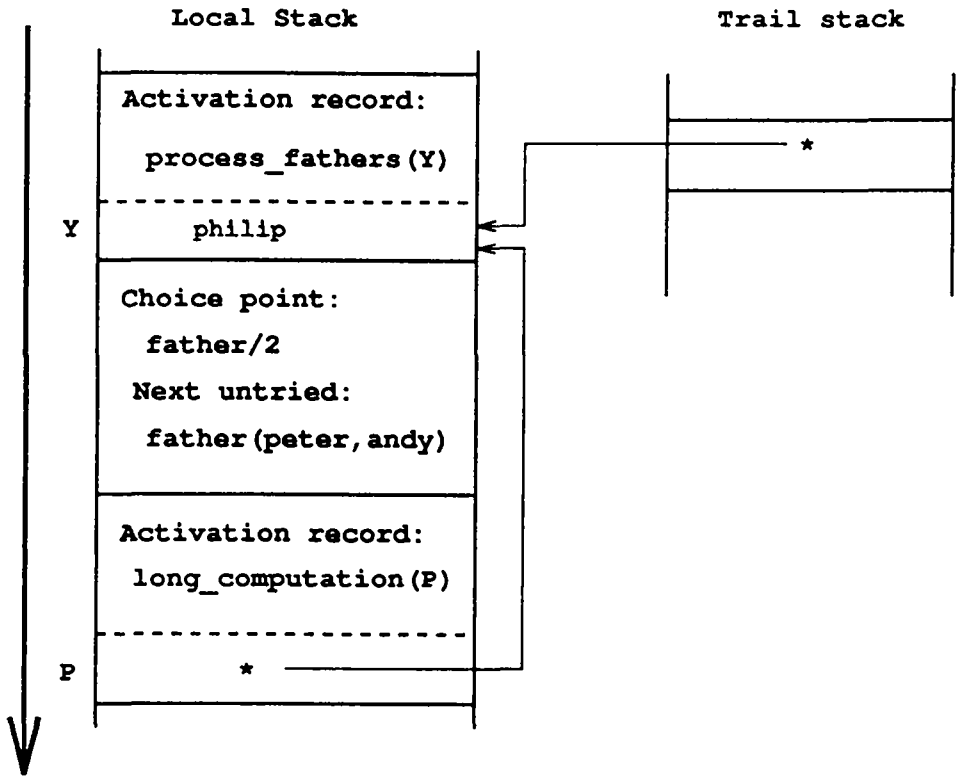

Figure 4: Management of Prolog variables in OR parallelism

The location of Y, in the local stack, is used to store three different logical variables corresponding to each possible clause unifiable with the *father(X, Y)* goal. If the *choice-point father/2* is used to provide work to an idle worker, location Y will be associated to more than one logical variable. The global stack is not used in the toy program shown above.

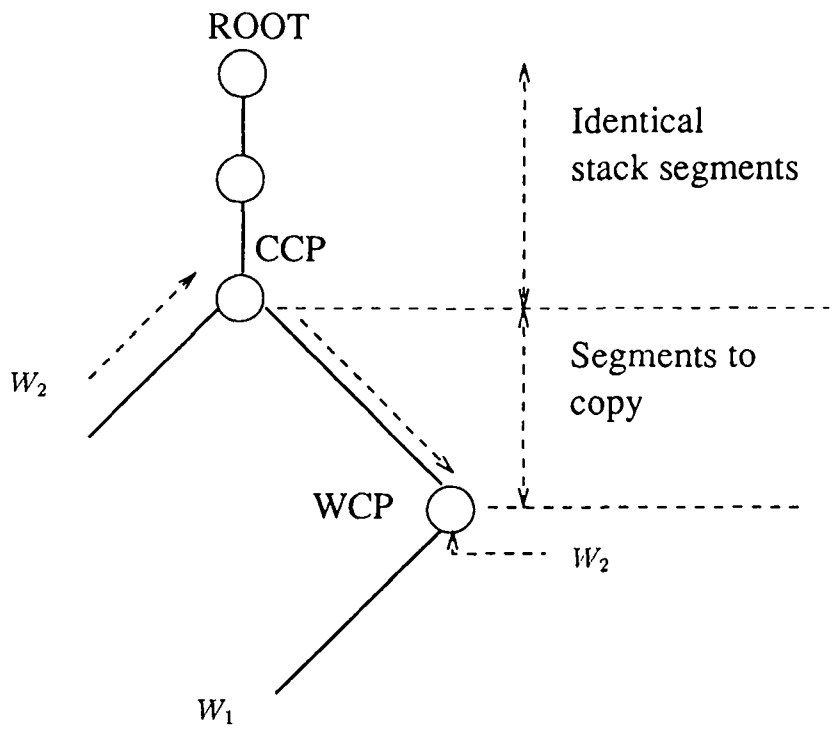

Figure 5: Incremental copying of the stacks

One problem occurs with stack copying: the worker grabbing work needs to restore the stacks to their previous state, when the OR node was created. In the example of Figure 3 and Figure 4, the binding of Y to *philip* has to be reset when W_2 copies the stacks of W_1. Two solutions have been proposed to solve this problem: the first one associates to each binding the value of a counter of the number of OR nodes, such a counter being managed by each worker [70]; the second solution uses the trail stack to restore the state of the stack of the worker grabbing work [2]. The first solution results in some time and space overhead to associate counter values with each binding while, in the second solution, an idle worker copying the stacks from an active worker, slows down the latter since both workers need to synchronize during the copying of the trail stack.

The overhead of stack copying can be reduced by remarking that any idle worker always shares a part of the program search tree with any active worker. So, when an idle worker W_2 gets work from an active one, there is no need to copy the complete stacks of the active worker providing work W_1, since W_1 and W_2 already share a part of the program search tree [2]. When incremental copying is used from an active worker W_1 to an idle worker W_2, W_2 backtracks to the last common choice-point (CCP) before copying the segments of the stacks of W_1 that are *younger* than CCP and *older* than the choice-point WCP used to provide work (see Figure 5). In addition, bindings performed by W_1 in the common parts of the stacks, between the creations of CCP and WCP, must be installed in the stacks of W_2, using the trail stack of W_1.

4.3.2 Sharing of stacks

In the stack-sharing scheme, workers share the parts of the stacks that they have in common. Using this solution, several logical variables may have the same identification, which is a location in a shared stack. In the example of Figure 3 and Figure 4, if enough computing resources are available, up to three different logical variables can be associated at the same time to the Y location of the stack.

In parallel logic systems sharing stacks, special data structure are used to store logical variables that may have the same stack identification as other logical variables, used simultaneously by another worker. Two types of data structures have been proposed: *binding arrays* [103] and *hash-windows* [7] [34] [106].

A binding array is an array, private to each worker, which contains the logical variables used by the worker and which have the same identification (stack address) as other logical variables used by other workers. Instead of containing a value, a stack location shared by several variables contains an index. This index refers to several different logical variables, one in the binding array of each worker sharing this section of the stack (see Figure 6).

In the alternative solution, variables sharing the same identifications are stored in data structures called hash-windows and attached to the branches of the search tree. An entry in a hash-window is computed by hashing the variable shared identification (address in the stack) of the variable. The concept of hash-window has led to several implementations. All the bindings stored in the hash-windows attached to branches located in the path leading from the root of the search tree to the current branch of the computation are valid for the worker computing this branch. Therefore, hash-windows are linked and variable lookup may imply to search a chain of hash-windows of unbounded length (see Figure 7).

An extension to this scheme restricts the creation of hash-windows to cases where parallelism is actually used [14] [106] [29], when several branches of a node are computed simultaneously. No hash-window is created when computation proceeds sequentially: variables are then stored in the stack by the worker computing the first branch, in such a way that other workers will recognize the cell as unbound and search their hash-windows. In [106], a scheme using time-stamps to distinguish the bindings valid for all branches is described.

Another problem arises in stack-sharing schemes, since parallelism mixes breadth-first strategy with the usual Prolog depth-first strategy. The computational state is no more a stack but a *cactus stack*, since several branches of some nodes may be computed simultaneously. It is not always possible to pop the stack on backtracking (see Figure 8) and the segments of memory that would be popped in a sequential system usually remains unused (*garbage slot* problem), resulting in an increased memory consumption. This problem happens also in AND-parallel systems and is presented in [51] together with possible solutions and associated tradeoffs.

4.3.3 Recomputation of stacks

To avoid the overheads associated with the two previous solutions, it is possible to have the stacks recomputed by each of the workers contributing to the computation [21]. Each worker computes a pre-determined path of the search tree described by an "oracle" allocated by a specialized process called "controler". Programs are rewritten to obtain an arity 2 search tree so that oracles can be efficiently encoded as bitstrings (see Figure 9).

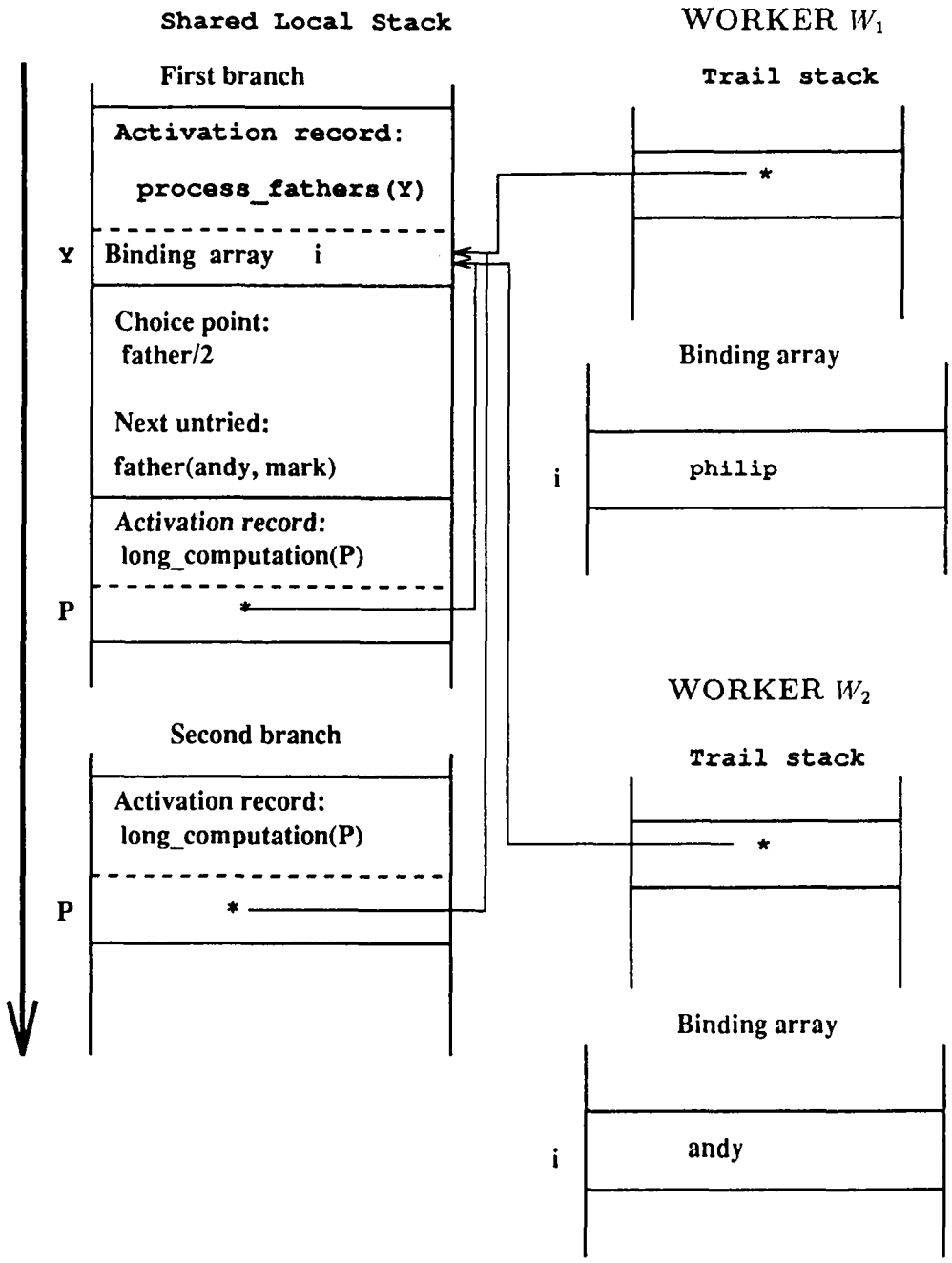

Figure 6: Use of binding array

This simple representation of the computational state of the example of Figure 3 examplifies the use of binding arrays. Two workers W_1 and W_2 share the portions of the stacks containing the value cell located at address Y. The logical variable of the branch processed by W_1, identified by the address Y, is bound to *philip*, while the logical variable of the branch processed by W_2, also identified by Y, is bound to *andy*.

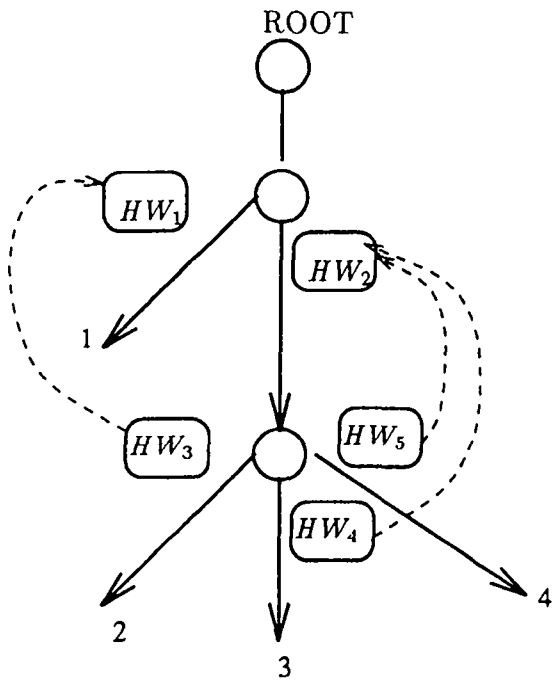

Figure 7: Hash-windows

Hash-windows are attached only to parallel branches. Variable lookup in branches 2, 3 and 4 may involve searching two hash-windows.

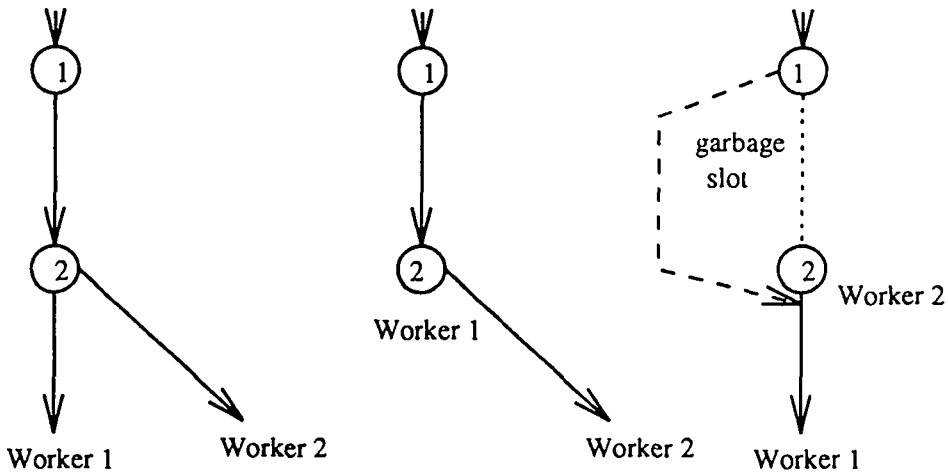

Figure 8: Garbage slot problem example

On the left-hand scheme of the example above, workers 1 and 2 compute two branches of node 2. On the scheme of the centre, worker 1 backtracks to node 2 and since there is no more work available on node 2, extends the stack to compute another branch of node 1. When worker 2 completes the computation of its branch, it cannot pop the stack segment comprised between node 1 and node 2 since this segment is not on the top of the stack.

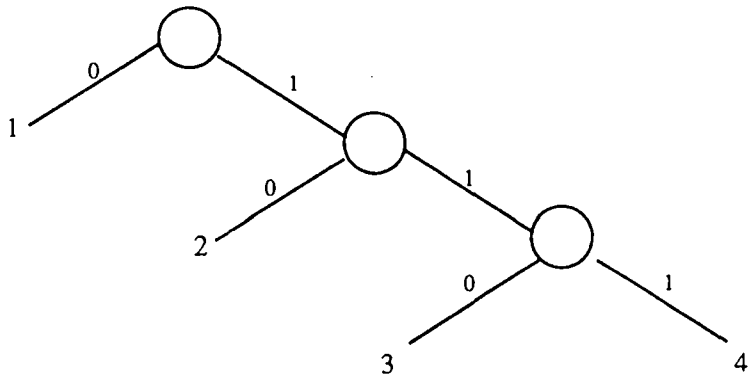

Figure 9: Bitstring encoding of oracles

The oracle leading to branch 3 can be encoded as the bitstring 110.

4.4 Problems with AND-parallelism

In systems exploiting AND-parallelism, several workers may bind the same logical variables to conflicting values. Systems exploiting *independent AND-parallelism* avoid to join dynamically partial AND-solutions and rather compute in parallel only independent goals. The other AND-parallel systems exploit *dependent AND-parallelism* instead.

4.4.1 Independent AND-parallelism

The main problem is the detection of independence between goals. Independence detection can be done during the execution or in advance at compile time or partly at compile time and partly during the execution.

Complex algorithms have been designed for runtime detection of parallelism [24]. If several goals have ground arguments, they can be executed in AND-parallel. Independence tests can be compiled as simple bit vector operations to be executed at runtime [64].

To limit the runtime overhead, it has been proposed to perform half of the detection work at compile time [32]. In the *Restricted AND parallelism*, programs are compiled into graphs called *Conditional Graph Expressions (CGEs)*, including simple runtime independence tests such as testing for the groundness of a variable. For example, the clause:

```
f(X) :- p(X), q(X), s(X).
```

will be compiled into the following graph expression:

```
f(X) =  (ground(X)
         p(X),
         (ground(X) q(X), s(X)).
```

If the argument of ground(X) is ground, the two following goals are computed in parallel. Otherwise they are executed sequentially. Depending on the instantiation state of X at runtime, three possible execution graphs may occur at runtime (see Figure 10).

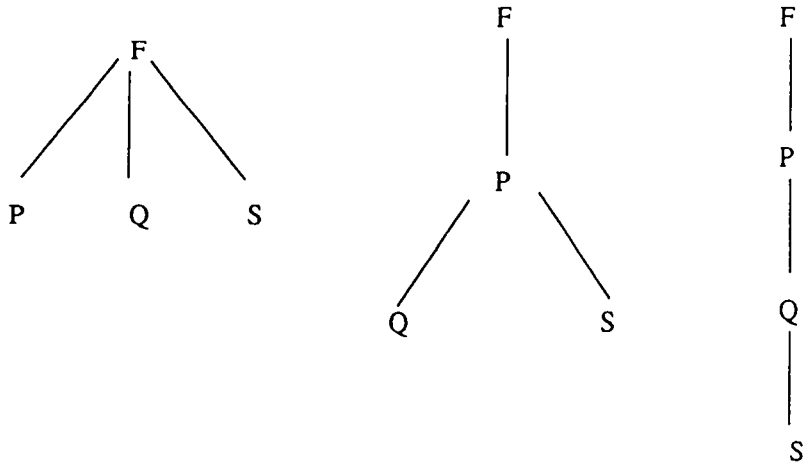

Figure 10: Possible execution graphs resulting from Conditional Graph Expressions

If X is ground after f(X) is called, all three goals can be computed in parallel. If X is grounded by the computation of p(X) instead, the two last subgoals *q(X) and s(X)* can be executed in parallel. Otherwise, all three goals are executed sequentially.

Conditional Graph Expressions may fail at capturing the potential independence between goals sharing the same variables. This occurs notably in programs where two goals share a variable which is instantiated by only one of them such as in the quicksort program:

$quicksort([X|L], SortedList, Acc) :-$ $partition(L, X, Littles, Bigs)$,
$quicksort(Littles, SortedList, [X|SortedBigs])$,
$quicksort(Bigs, SortedBigs, Acc)$.

$quicksort([], SortedList, SortedList)$.

In this example, no parallelism can be detected automatically between the two recursive calls to *quicksort* in the first clause and systems exploiting only *strict independence* of goals would fail at parallelizing this clause. However, there are situations where goals sharing variables can be rewritten in such a way that no variables are shared between these goals which can then be executed in parallel while preserving the correctness and efficiency of sequential computation [53]. Such AND-parallelism is called *non-strict independent AND-parallelism*. It can be exploited in the example above which can be rewritten as:

$quicksort([X|L], SortedList, Acc) :-$ $partition(L, X, Littles, Bigs)$,
$quicksort(Littles, SortedList, [X|Temp])$,
$quicksort(Bigs, SortedBigs, Acc)$,
$Temp = SortedBigs$.

$quicksort([], SortedList, SortedList)$.

In practice, even simple runtime checks can be very expensive, such as testing for groundness of a complex nested Prolog term. To avoid runtime tests, current research is being concentrated on purely static detection of the independence between subgoals [75]. Early results indicate that these methods capture almost as much parallelism as the compilation into CGEs.

4.4.2 Dependent AND-parallelism

In the following we will assume that dependent AND-parallelism is only used between determinate goals. This is the case with flat concurrent logic languages and with the Andorra model of computation.

Most systems implementing dependent AND-parallelism use the goal process model. A goal process is responsible for trying each candidate clause until one is found that successfully commits and then creating the goal processes for the body goals of the clause. The goal being determinate, at most one candidate clause head will unify with the goal. Tentative unification of a goal process with a clause head will either fail, succeed or suspend on a variable. Dependent AND-parallel systems need therefore to create, suspend, restart and kill a potentially high number of fine-grained processes. These basic operations on processes may be the source of considerable runtime overhead and need to be limited to obtain an efficient implementation [27]. Process switching overhead can be limited by reducing the use of the general scheduling procedure to assign a process to a worker: after completing the last child of a process, a worker continues with the parent; similarly, after spawning processes, a worker continues with the execution of its last child.

Another very important problem arising in concurrent logic programming systems is the garbage collection. Since these systems do not support the Prolog *don't know* non-determinism, most of the garbage collection performed at backtracking in Prolog systems cannot be done in these systems, resulting in considerable memory consumption and poor data locality. General garbage collection algorithms, which compact the stack and the heap, require that all workers synchronize on entering and exiting garbage collection [27].

In the PIM project, a less complete but more efficient incremental garbage collection is used instead, to increase the locality of references and obtain a better cache-hit ratio. In the Multiple Reference Bit (MRB) algorithm, used inside a processing element or a group of processing elements sharing the same memory (cluster), each pointer has one bit information to indicate whether it is the only pointer to the referenced data [18]. MRB is supported efficiently in time and space by the hardware of the PIM machine [39] while reclaiming more than 60 % of the garbage cells of stream parallel programs.

4.5 Scheduling of parallel tasks

The aim of the scheduling of parallel tasks in parallel logic systems is to optimize the global behaviour of the parallel system. Therefore schedulers aim to balance the load between workers while limiting as much as possible the overhead arising from task switching. This overhead occurs when workers have exhausted their work and need to move in the computation tree to grab a new task. In order to keep this overhead low, schedulers attempt to limit the number of task switching by maximizing the granularity of parallel tasks and to minimize each task switching overhead. In addition, to avoid unnecessary computations, schedulers need to manage carefully speculative work

Maximizing the granularity of parallel tasks would require the ability to estimate either at compile time or at runtime the computation cost of goals (AND-parallelism) or alternative clauses (OR-parallelism). Current research (see section on Abstract Interpretation) mainly concentrates on compile time analysis of granularity [95] [83]. Because of the limited results obtained so far in task granularity analysis, most schedulers use heuristics to select work. One common heuristic used by OR-parallel systems [4] [66] is that idle workers are given the highest uncomputed alternative in the computation tree, there-

fore mixing breadth-first exploration strategy between workers to the depth first Prolog computation rule within each worker.

To limit the number of task switches between workers, several schedulers *share* the computation of several nodes of the tree between workers [3] [6]. The scheduler then attempts to maximize the amount of shared work. Contrary to the granularity of an unexplored alternative (OR-parallelism) or of a goal waiting to be computed, the amount of sharable work in a branch can be easily estimated by the number of unexplored alternatives. Once an active worker has performed sharing of all its available work with an idle one, both workers resume the normal depth-first search of Prolog computation.

Workers of independent AND-parallel systems distribute in priority the closest work from their current position in the computation tree: goals waiting to be solved are put on a stack accessible from other workers [50].

In most OR-parallel logic systems, an idle worker switching to a new task needs to install its binding array or to copy parts of the stacks of the active worker providing work. In these systems, the cost of task switching depends on the relative positions of both workers providing and receiving work. Schedulers designed for these systems attempt to minimize the cost of task switching by selecting the closest possible work in the computation tree [16], [13] [3]. This is not necessary for systems designed to provide task switching costs independent from the relative positions of both workers involved in this operation [4].

Speculative work may be pruned by pending cuts. To avoid performing unnecessary work, a scheduler should give preference to non-speculative work and if all available work is speculative to the least speculative work available [47]. Speculative work is better handled by schedulers performing dispatching of work based on sharing [3] [6] than topmost dispatching of work [16] since the control of the former is closer to the depth-first left-most computation of sequential Prolog systems.

5 Systems exploiting one type of parallelism

A large number of models have already been proposed to parallelize Prolog. In this section, we will concentrate on the systems exploiting one type of parallelism which have been efficiently implemented based on the multisequential approach.

5.1 Systems exploiting OR-parallelism

The Kabu-Wake[1] system [70] was the first to copy stacks when switching task and to use time-stamps to discard invalid bindings. In Kabu-Wake, the worker giving work suspends its execution to copy its stacks into the idle worker's local memory, apparently without the incremental copying optimization. The Kabu-Wake implementation, on special purpose hardware, provides nearly linear speed-ups for large problems computed in parallel. However, it is based on a rather slow interpreter.

The ANL-WAM system from the Argonne National Laboratories [14] [34] was the first parallel Prolog system based on compiling techniques and implemented on shared memory multiprocessor. All trailed logical variables are stored in hash-windows, even when no parallel computation takes place. Being the first efficient parallel Prolog implementation,

[1] Kabu-Wake names a transplanting technique used to grow bonzai trees.

the ANL-WAM system has been widely used for experimental purpose. The performances of the ANL-WAM system are limited by the rather low efficiency of the sequential WAM engine used and by the overhead arising from its hash-window model.

In PEPSys [106] [4], hash-windows are only used if needed: when several logical variables having the same identification are bound simultaneously by several workers. Most accesses to the values of variables are as efficient as sequentially but some accesses may involve searching of hash-window chains of unbounded length. The cost of task installation is independent of the respective positions of the work and of the idle worker in the search tree. In spite of the sharing of the stacks, PEPSys does not require a global address space and has been simulated on a scalable architecture combining shared memory in clusters and message passing between clusters [5]. PEPSys has also been efficiently implemented on shared-memory multiprocessor. Sequentially, the WAM-based PEPSys implementation runs 30 % to 40 % slower than SICStus Prolog. In parallel, it provides almost linear speedups for programs having a large enough search space [4]. Other experimental results [28] indicate that long hash-window chains are rare and do not compromise the efficiency of the implementation.

The Aurora system is based on the SRI model [103] where bindings to logical variables sharing the same identification are performed in binding arrays (see previous section). Accessing to such variables is a constant overhead but contrary to PEPSys, it is a constant time operation. An Aurora prototype, based on SICStus Prolog, has been implemented on several commercial shared-memory multiprocessors. Four different schedulers have been implemented: three of them use various techniques for the idle worker to find the closest top-most node of the search tree with available work [13] [16] [9]. Since experimental results [93] indicate that the work installation overhead remains fairly limited, the Bristol scheduler [6] performs sharing of work similar to the Muse system [3], the objective being to maximise the granularity of parallel tasks and limit the task switching overhead. All schedulers support the full Prolog language including side-effects. The system has successfully run a large number of Prolog programs and the performances of some of them have been reported in [93] [6] (see Table 1 and Figure 11). Sequentially, Aurora is slightly more efficient than PEPSys and in average it also provides better speedups in parallel.

In the Muse model [2], active workers share, with otherwise idle workers, several choice points containing unprocessed alternatives. Sharing is performed by the active worker, which creates an image of a portion of its choice-points stack, in a workspace *shared* with previously idle workers. The stacks of the active worker are then copied from its local memory to the local memory of the idle worker, most of the copying being performed in parallel by the two workers, using incremental copying of the portions of the stacks which are not similar. Unwinding and installation of bindings in the shared section uses the trail stack instead of time-stamps in Kabu-Wake. Muse has been implemented on several commercial multiprocessors with uniform and non uniform memory addressing (see Section 7.4) and on the BC-Machine prototype constructed at SICS and providing both shared and private memory [2] [3]. The Muse implementation is based on SICStus Prolog [17]. It combines a very low sequential overhead over SICStus (5%) with parallel speed-ups similar to Aurora.

K-LEAF [8] is a parallel Prolog system implemented on a Transputer-based multiprocessor. Contrary to other multisequential systems, it creates all possible OR-parallel tasks. Combinatorial explosion of the number of tasks is avoided by using some language constructs to ensure sufficient grain size of parallel tasks. K-LEAF has been implemented on

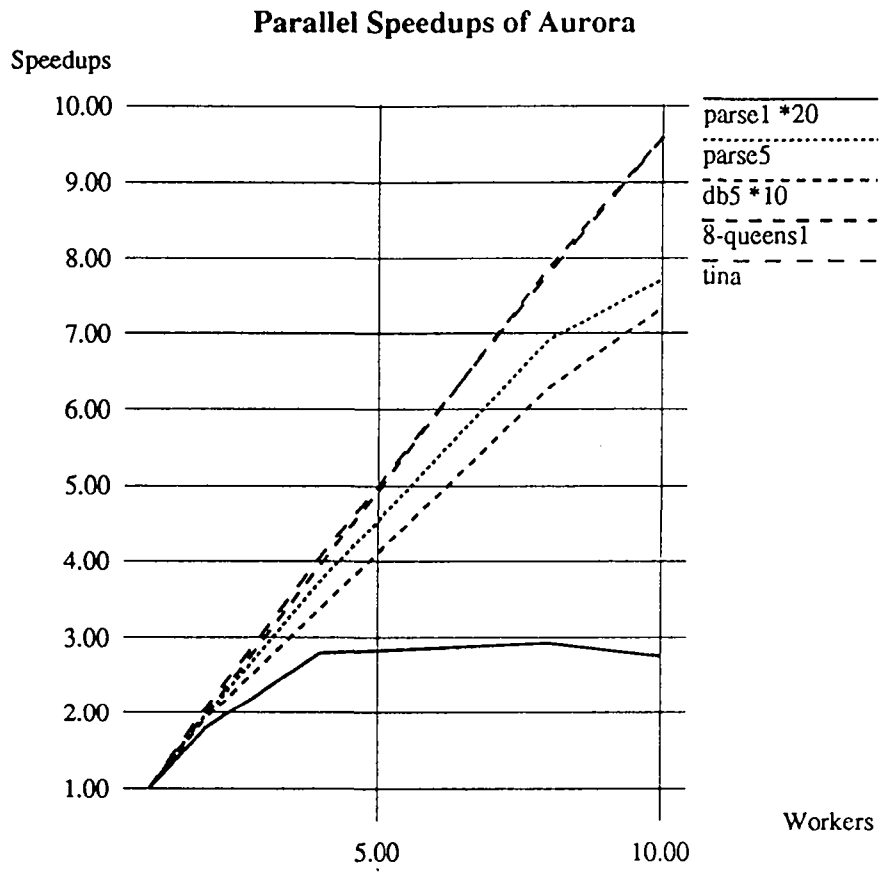

Figure 11: Parallel speedups of Aurora

Parallel speedups of Aurora with Bristol scheduler, executing the same benchmarks as in table 1.

Table 1: Execution times of Aurora

The benchmark programs are executed by Aurora using the Bristol scheduler and SICStus implementations on Sequent Symmetry. *parse1*, *parse5*, and *db5* are parsing and data-base queries of the *Chat80* natural language data-base front-end program. *8-queens1* computes all the solutions to the 8-queens problem and *tina* is a tourism information program.

Program[* times]	1	2	4	8	10	SICStus 0.6
parse1 *20	1.97	1.09	0.70	0.67	0.71	1.57
	1	1.80	2.80	2.93	2.76	1.25
parse5	4.87	2.47	1.30	0.70	0.63	3.82
	1	1.97	3.74	6.92	7.72	1.27
db5 *10	3.74	1.94	1.04	0.59	0.51	2.73
	1	1.93	3.58	6.29	7.34	1.37
8-queens1	8.46	4.26	2.14	1.07	0.88	6.77
	1	1.99	3.95	7.88	9.60	1.25
tina	19.79	9.54	4.87	2.53	2.06	13.78
	1	2.07	4.06	7.81	9.59	1.43

an experimental Transputer-based multiprocessor providing a virtual global address space. This implementation is based on the WAM, using the binding array technique. The WAM code is either emulated or expanded into C, and then compiled using a commercial C compiler, the latter solution being five times more efficient than the former.

5.2 System exploiting independent AND-parallelism: &-Prolog

Independent AND-parallelism has mainly been exploited by the &-Prolog system [52] which covers several aspects of the problem, from the independence detection at compile-time to the efficient implementation of AND-parallelism on shared memory multiprocessors.

Strict as well as non strict independent AND-parallelism are expressed using the &-Prolog system. &-Prolog programs can also be generated automatically through compile-time analysis of ordinary Prolog programs. &-Prolog is very similar to Prolog with the addition of the parallel conjunction operator "&" and of several builtin primitives to perform the independence runtime tests of conditional graph expressions (see previous section). For example the Prolog program

$$p(X) : -q(X), r(X).$$

could be written in &-Prolog as:

$$p(X) : -(ground(X) \Rightarrow q(X) \& r(X)).$$

The compiler performs a number of static analyses to transform plain Prolog programs into &-Prolog programs before compiling the latter into an extension of the WAM called PWAM [50]. The static analyses aim at detecting the independence of litterals, even in

Table 2: Execution times of &-Prolog

The benchmark programs are executed by &-Prolog and SICStus implementations on Sequent Balance and Quintus 2.2 on SUN3-110. The first benchmark is a matrix multiplication. The three next benchmarks are different versions of the *quicksort* of a list of 1000 elements using the predicate *append* or *difference-lists* exploiting strict independent AND-parallelism (si) and non-strict independent AND-parallelism (nsi). The last benchmark *annotator* is a "realistic" program of 1800 lines, the annotator being one of the static analysers used in the &-Prolog compiler.

Program	1	4	8	10	SICStus 0.5	Quintus Sun3-110
matrix(50)-int	103	25.9	13.0	10.45	99.8	7.98
qs(1000)-app	13.23	4.43	3.08	3.03	13.6	1.7
qs(1000)-dl-si	11.9	11.9	11.9	12.0	11.13	1.61
qs(1000)-dl-nsi	12.51	4.06	2.91	2.78	11.13	1.61
annotator(100)	4.09	1.27	0.75	0.64	3.87	0.62

presence of side-effects [74] [73]. The main difference between the PWAM and the WAM is the addition of a goal stack in the PWAM. The PWAM assumes a shared-memory multiprocessor.

To adjust the computing resources to the amount of parallel work available, the &-Prolog scheduler organises PWAMs in a ring. A worker searches for an idled PWAM in the ring. If no idle PWAM is found and enough memory is available, the worker creates a new PWAM, links it to the ring and look for work in the goal stacks of the other PWAMs.

The &-Prolog run-time systems is based on SICStus and has been implemented on shared memory multiprocessors. The overhead of the &-Prolog system running sequentially over SICStus arises mainly from the use of the goal stack and remains very limited (less than 5%) for the majority of the benchmarks using unconditional parallelism (without runtime tests which are actually fairly expensive). Parallel speedups depend on the benchmark program but even when no parallelism is available, such as in the *quicksort* program using *difference-lists* and exploiting only strict independent parallelism (see Table 2), the &-Prolog system remains almost as efficient as the SICStus sequential Prolog implementation.

5.3 Systems exploiting dependent AND-parallelism

Because of the importance of the efforts dedicated to concurrent logic languages, it is only possible to mention here some of the implementations of these languages.

5.3.1 Parlog

The JAM abstract machine based on the WAM has been designed to support concurrent logic languages and the full Parlog language in particular. It has been implemented on a shared-memory multiprocessor [27], reaching half of the efficiency of SICStus Prolog. In parallel, the benchmarks containing important dependent AND-parallelism show speedups of between 12 and 15 on 20 processors while benchmarks containing no dependent AND-

Table 3: Execution times of the JAM on Sequent Balance B21

Program	no. of calls	1	5	10	19
qsort	4708	3.7	0.9	0.6	0.5
nrev	80601	30.0	6.5	3.6	2.4
lqueens	23531	37.8	7.6	3.9	2.4
tak	63609	71.4	15.0	7.9	4.8
queens	62082	85.9	87.6	88.7	90.1

parallelism such as the *queens* program of Table 3, run at the same speed in parallel as sequentially (see Table 3).

5.3.2 Flat Concurrent Prolog

FCP has been implemented on distributed memory multiprocessors Intel iPSC1 [94] [35]. The implementation provides limited speedups for fine-grained systolic computations and good speedups for large-grained programs.

5.3.3 GHC and KL1

A large number of GHC and KL1 implementations have been done in the framework of the Japanese Fifth Generation Computer Systems Project lead by the ICOT and it is only possible to mention a few of them in this paper. KL1 has been implemented on commercial shared-memory multiprocessor [88], this implementation being 2 to 9 times slower than Aurora [96]. The most significant prototypes involve both hardware and software development with the implementation of several highly parallel machines dedicated to the execution of the concurrent logic language KL1. These machines, known as Multi-PSI and PIM-p are presented in another section of this paper (see section 7.4).

5.3.4 Commercial Concurrent Logic Language: Strand

A commercial product has been derived from Parlog and FCP: Strand (STReam AND-parallelism) [36]. In the Strand language, there is no more unification but matching of the head of a clause against a goal. The rule guards reduce to simple tests. An assignment operation can be used in the bodies of the rules but, as in other logical languages, Strand variables are single assignment variables. Processor allocation pragmas can be used by programmers to control the degree of parallelism of their programs. A foreign interface language enables the calling of Fortran or C modules from a Strand program. Strand can thus be used to coordinate the execution of existing sequential programs on multiprocessors, therefore avoiding to rewrite these programs to parallelize them.

The implementation of Strand is based on the Strand Abstract Machine or SAM, designed to limit the target-dependent parts of the implementation. SAM is divided into a reduction component, similar to a concurrent logic language simplified implementation and a communication component performing matching (read) and assignments (write). All machine specific features of the implementation are localized into the communication

component. Strand has been implemented on a variety of shared and distributed memory multiprocessors. The Strand implementation runs approximately two times faster than Parlog and three times faster than FCP.

5.4 Performances of systems exploiting one type of parallelism

The systems described above have demonstrated that it is possible to exploit efficiently OR- and independent AND- parallelism on shared-memory multiprocessors including a limited number (up to several tens) of processors. Efficiently means that in programs which exhibit the suitable type of parallelism, almost linear speedups can be obtained whereas in the worst case, when no parallelism can be exploited, its efficiency remains similar to a state-of-the-art sequential Prolog implementation such as SICStus Prolog. Dependent AND-parallel systems are less efficient but they enable to exploit finer-grained parallelism than OR- and independent AND-parallel systems.

The majority of the parallel systems presented in this section compile logic programs into abstract machine instructions which are most of the time executed by an interpreter written in C, as it is the case in SICStus Prolog. Some commercially available Prolog systems execute abstract machine instructions more efficiently by using an abstract machine interpreter written in assembly language (Quintus Prolog) or by generating target machine code from the abstract machine instructions (BIM Prolog). Parallelizing techniques developed in the systems presented above remain valid to parallelize these efficient commercial systems, although improving the efficiency of the sequential engine will probably decrease the parallel speedup.

Whether it is cost-effective to use non scalable shared-memory multiprocessors to solve symbolic problems programmed in logic languages is still questionable. This situation may change when the multiprocessor technology becomes more mature and multiprocessor workstations become as common as uniprocessor ones. Then a parallel logic programming system, providing significant speedups in the best cases and no speed-down otherwise, will be a good way of exploiting the computing power delivered by the multiprocessor platform. The use of more scalable higly and massively parallel multiprocessors, to solve larger size problems currently untractable sequentially, is discussed in section 7.4.

6 Systems combining several types of parallelism

Important benefits can be expected from the combination of several types of parallelisms into a single parallel logic programming system implementation: these are mainly an increase in the number of programs that can benefit from parallelism and a reduction of the search space of some of them.

Although systems exploiting one type of parallelism have demonstrated their ability to reach effective speedups over state-of-the-art sequential Prolog systems, a large amount of parallel programs cannot benefit from speed improvements: this is the case of deterministic programs in OR-parallel systems and of large search programs in AND-parallel systems. In addition, while OR- and independent AND-parallel systems mainly exploit coarse-grained parallelism, dependent AND-parallel systems have developed techniques to exploit fine-grained parallelism, present in a large number of applications.

Combining OR- with AND-parallelism also results in significant reductions in the program search space, when several recomputations of the same independent AND-branch,

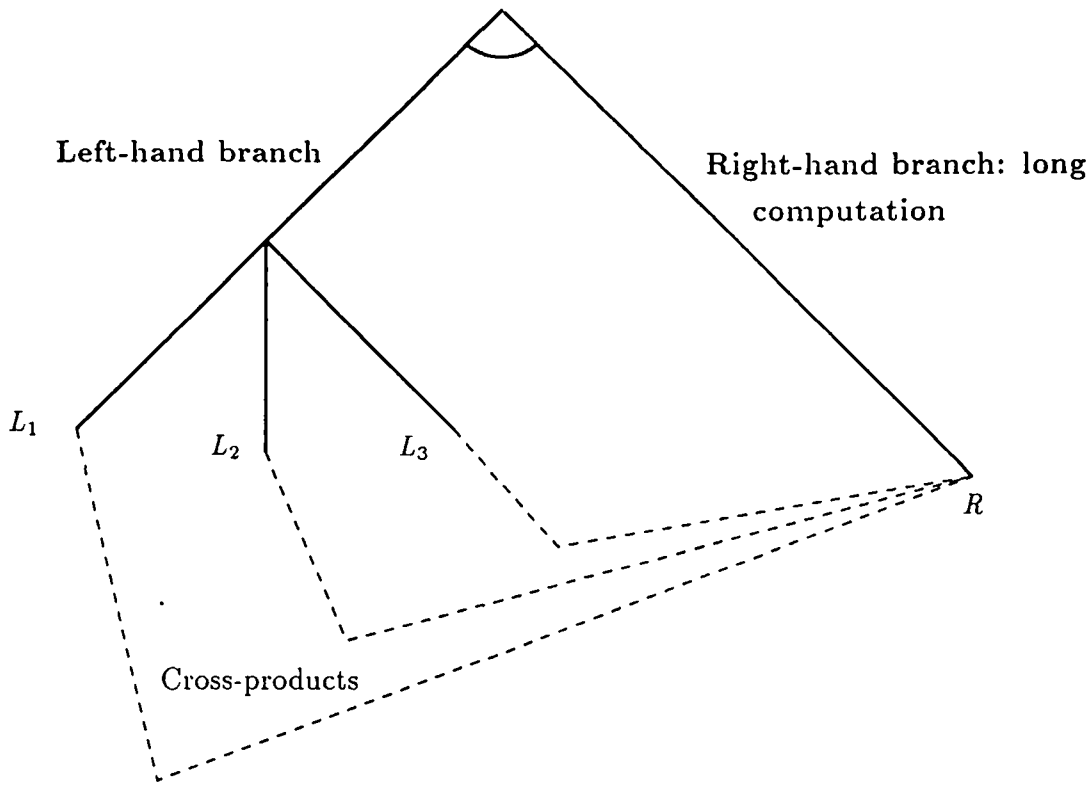

Figure 12: Reduction of the search space by use of cross-product

In this example, two goals are computed in AND-parallel, the computation of the left-hand branch being splitted into three OR-parallel branches. Combining AND with OR parallelism and cross-producing the solutions produced by the right- and the left-hand branches saves two re-computations of the right-hand branch, which would be recomputed with each of the left-hand branches.

due to backtracking into a previous AND branch, can be avoided by "reusing" the solutions already produced in each AND-branch (see Figure 12).

However, the combination of AND- with OR- parallelism raises difficult problems. Control of the execution must guarantee that all solutions to the AND-branches are cross-produced without introducing too much overhead to store partial solutions or to synchronize workers. Merging of partial solutions during the cross-product of AND-branches is a complex operation whatever binding scheme is being used: stack copying or stack sharing using binding arrays or hash-windows. The solutions proposed to solve these problems are too complex to be presented in this paper [106] [38] [41] [58] [25]. Indeed, the idea of "sharing" solutions between OR-branches is not proper to parallel execution models, and also appears in a few sequential Prolog systems that attempt to "memorize" and "reuse" partial solutions in order to avoid recomputations, such as [30]. Such systems are not yet mature and competitive enough with traditional Prolog systems to serve as a basis for parallel models. Moreover, simulations of a variety of programs [92] indicate that only few of them would benefit from reuse of partial solutions of independent AND-branches.

6.1 Systems combining independent AND- with OR-parallelism

The ROLOG system implements the Reduce OR Process Model (ROPM) [98] which combines independent AND- with OR-parallelism. Programs are compiled into an abstract machine inspired from the WAM [81]. The implementation uses a machine independent run-time kernel called CHARE kernel which made possible the porting of ROLOG on a variety of shared and distributed-memory parallel machines. Because of the complexity of the execution model, sequential efficiency of ROLOG is several times lower than efficient parallel logic systems exploiting one type of parallelism. In parallel, it provides linear speedups for benchmarks where programmer annotations ensure sufficient granularity of tasks. Experimental results also indicate that significant reductions of the search space can also be obtained in several programs, by avoiding the recomputation of AND-parallel branches due to backtracking [80].

The ACE model [42] aims to combine the independent AND-parallel approach of the &-Prolog system with the OR-parallel approach of the Muse sytem. Thus no effort is made to reuse results of independent AND-subcomputations in different OR-branches, as described above, but the strength of this model lies in its (re)use of simple techniques whose efficiency has already been proved.

6.2 Systems Combining dependent AND- with OR-parallelisms

As presented in section 3.4, execution models based on the Andorra principle has been proposed to combine OR- and dependent AND-parallelism. They intend to subsume both the Prolog and the concurrent programming paradigms. There exist two main streams of research in this area, depending on the language being supported: the first one, based on the Basic Andorra Model, supports the Prolog language, while another one, based on the Andorra Kernel Language, integrates concurrent languages constructs.

6.2.1 Basic Andorra Model

The Basic Andorra Model has been initially proposed by D.H.D. Warren as an execution model for Prolog [100], in order to exploit both OR- and dependent AND-parallelism and to have a better pruning of the search space, by favoring deterministic computation over non-determinate one. It has been implemented in the Andorra-I prototype [25] which runs on both sequential machines and shared-memory multiprocessors such as the Sequent Symmetry. The current system consist of an interpreter, similar in speed with comparable Prolog interpreters such as C-Prolog, while a compiler version is under development. OR-parallelism is achieved by using techniques borrowed form the Aurora system, and the exploitation of dependent AND-parallelism is derived from Crammond's abstract machine [27] tailored to concurrent logic languages (see section 5.3.1).

Andorra-I programs are executed by teams of workers (abstract processing agents). Each team is associated with a separate OR-branch in the computation tree, and undertake the parallel execution of determinate goals in that branch, if any. Else a non-determinate goal is chosen, usually the leftmost goal to follow Prolog's stategy, and a choice-point is created. The team will then explore one of the OR-branches. If several teams are available, OR-branches are explored in parallel using the SRI model as in Aurora, the scheduler of which is also used to distribute work. Within a team, all workers share the same execution stacks, but manage their own run queue of goals waiting for execution.

When a local queue becomes empty, the worker will try to find work within the team, as in the implementation of Parlog [27].

The integration of full Prolog, including cut and side-effects, is possible in Andorra-I, but requires a careful preprocessing phase [25]. A program analysis, based on abstract interpretation techniques, is performed in order to determine, for each procedure, the *mode* of its arguments, i.e. the possible instanciation types. This information is used, together with the analysis of pruning operators and side-effects to generate sequencing instructions that should be taken into account by the execution model. The preprocessor also generates specialized code intented to efficiently regonize the determinacy of procedures.

Performance evaluation of Andorra-I on a Sequent Symmetry shows that it can exploit efficiently either pure AND- or pure OR-parallelism, the latter with results comparable with Aurora. For instance, with 11 processors AND-parallel speedups w.r.t. sequential implementation range from 5.2 to 9.1 for suitable programs, while OR-parallel speedups range from 5.2 to 10.3. The ability of Andorra to reduce the amount of computation was estimated by mesuring the total number of inferences executed for several problem solving examples[25]. The reduction can attain one order of magnitude. Therefore, when both AND- and OR-parallelism are exploited, the overall speedup is always greater than that achievable with one kind of parallelism alone.

The basic Andorra model has also been implemented as an extension of the Parallel NU-Prolog system [78] [76], which implements dependent AND-parallelism. A simple variant of the compilation techniques developed for concurrent logic languages [59] is used to construct a decision tree of the conditions under which a goal is determinate. First parallel experiments are limited to 4 processors, giving speedups comprised between 2 and 3.4 for simple test programs.

6.2.2 Extensions to the Basic Andorra Model

D. H. D. Warren has recently proposed a new execution model [101] that extends the basic one by allowing parallel execution between non-determinate goals as long as they perform "local" computations, i.e. as they do not instanciate non-local variables. In this way non-determinate goals are synchronized (i.e. blocked) only when they try to "guess" the value of some external variable. Observe that such extra-parallelism contains independent AND-parallelism as a subcase. The Extended Andorra Model combines thus all three kinds of parallelism: OR-, dependent AND- and independent AND-parallelism.

Another combination of the three kinds of parallelism is proposed by the IDIOM model [43]. It uses Conditional Graph Expressions (CGE) to express independent AND-parallelism as in &-Prolog, and execution proceeds as follows. First occur the Dependent AND-parallel phase where all determinate goals are evaluated in parallel. When no more determinate goals exist, the leftmost goal is selected for expansion: if it is a CGE, then an independent AND-parallel phase is entered, otherwise a choice-point is created as in the basic Andorra model. Solution sharing is handled by cross producing partial solutions of independent subcomputations.

The two extended models presented above have not yet been implemented.

6.2.3 Andorra Kernel Language

The Andorra Kernel Language (AKL) [56] extends the basic andorra model by borrowing some of the concurrent languages constructs (see section 3.4). The semantics of AKL [45]

is given by a set of simple rewrite rules on an AND/OR tree, which has led to the design of a simple abstract machine and sequential implementation. Parallel implementations are currently under developement, based on the integration of a MUSE-like mechanism for handling OR-parallelism.

Independently, a new abstract machine is developed from [78] to accomodate the new constructs of AKL. It is inspired by the JAM [27], as the Andorra-I implementation. Another execution model [44] exploits mostly OR-parallelism and uses hash-windows similar to Pepsys', which seem more suited to the OR-parallel execution of AKL than binding arrays.

7 Current research and perspectives

7.1 Static analysis of logic programs

An important topic recently developed in logic programming is that of static program analysis, which aims at uncovering various program properties. Compile-time analysis can be applied to parallel execution models to predict runtime behavior so that compilation of programs may be optimized. This analysis is usually based on the *abstract interpretation* techniques, introduced by the seminal work of the Cousot's [26] for flowchart languages and developed in the domain of logic programming since the mid-80's [71] [57] [12]. Abstract interpretation is a general scheme for static data-flow analysis of programs. Such analysis consists primarily of executing a program on some special domain called "abstract" because it abstracts only some desired property of the concrete domain of computation.

The use of abstract interpretation for parallel execution falls roughly into three major kinds of analysis :

- the *dependency* analysis, which aims at approximating data-dependencies between literals due to shared variables,
- the *granularity* analysis, which aims at approximating the size of computations,
- the *determinacy* analysis, which aims at identifying deterministic parts of the programs.

Various abstract domains have been designed that approximate values of logical variables into only some *groundness* and *sharing* information. This is currently used in independent AND-parallel models such as [52] in order to find out at compile-time which predicates will run in parallel [75], thus avoiding costly runtime tests (see section 4.4.1). Experimental results indicate that a large part of the potential parallelism of programs is captured, although some potential parallelism may be lost, compared to a method performing precise but costly runtime analysis. In dependent AND-parallel models, the same kind of information can be used to determine an advantageous scheduling order among active processes and to avoid useless process creation and suspension. Another useful application is static detection of deadlocks in concurrent programs [22]. Task granularity analysis, by discriminating large tasks from small tasks not worth parallelization, strives to improve the scheduling policy of AND and OR-parallel models. Research has only just begun on this important topic [83]. Determinacy analysis is required by Andorra based models, in order to simplify runtime tests.

As execution models become more and more complex with the combination of several kinds of parallelism, the need for compile-time analysis increases in order to compile programs more simply and more efficiently.

7.2 Parallelism and Constraint Logic Programming

Constraint Logic Programming (CLP) languages provide an attractive paradigm which combines the advantages of Logic Programming (declarative semantics, non-determinism, logical variables, partial answer solution) with the efficiency of special-purpose constraint-solving algorithms over specific domains such as reals, rationals, finite domains, or booleans. The key point of this paradigm is to extend Prolog with the power of dealing with domains other than (uninterpreted) terms and to replace the concept of unification with the general concept of constraint solving [55]. Several languages, such as CLP(R) [55], CHIP [67] [49], Prolog-III [23] show the usefulness of this approach for real industrial applications in various domains: combinatorial problems, scheduling, cutting-stock, circuit simulation, diagnosis, finance, etc. The use of constraints indeed leads to a more natural representation of complex problems.

A most promising perspective for CLP languages is the parallel execution of such languages, where both OR- and AND-parallelism can be applied. The simultaneous exploration of alternative paths in the search tree provided by OR-parallelism can be particularly useful in problems such as optimisation problems. Sequential CLP sytems use a branch-and-bound method that involves searching for one solution and then starting over with the added constraint that the next solution should be better than the first one, ensuring thus an *a priori* pruning of non-optimal branches. With OR-parallelism, such a scheme allows for super-linear speedups, as the simultaneous search is more likely to quickly find a better solution, therefore improving the pruning of the search space and the overall performance. Experiments done by combining the finite domains solver of CHIP and the Pepsys OR-parallel system [99] and the Elipsys system [82] already show that impressive speedups with respect to an efficient sequential implementation can be achieved.

AND-parallelism can also be used for better performance. Independent AND-parallelism corresponds to partitioning the global constraint system into several smaller (independent) subsystems that can be solved more easily. Since problems tackled by constraint systems are usually very large and NP-hard, breaking a global problem in several subproblems that can be solved independently, is an important issue.

Andorra-based models by favoring deterministic computations, can also greatly improve CLP. They provide a good heuristic to guide the constraint solving process. This is the case for instance when solving disjunctive constraints. In CLP languages, disjunctive constraints are treated by using the non-determinism of the underlying Prolog language. A choice-point is thus created for each disjunctive constraint, and different alternatives are then explored by backtracking. The (static) order in which the disjunctive constraints are stated will enforce the order in which the choice-points will be created and therefore the overall search mecanism. A poor order could lead to bad performance due to useless backtracking. The Andorra principle, by delaying choice-point creation, will amount to render deterministic the problem as much as possible before handling the disjunctive constraints. In this way, computations are factorized between alternatives and some disjunctions may even be rendered deterministic, thus reducing the complexity of the problem.

7.3 Concurrent Constraint Languages

The current CLP framework is based on Prolog-like languages and hence limited to transformational systems (see section 3). New research has begun to extend this to reactive systems, by investigating Concurrent Constraint Languages. These languages have been recently introduced by V. Saraswat [85] [86]. They are based on the ask-and-tell mechanism [69] and generalize in an obvious manner both concurrent logic languages and constraint logic languages. The key idea is to use constraints to extend the synchronization and control mechanisms of concurrent logic languages. Briefly, multiple agents (processes) run concurrently and interact by means of a shared *store*, i.e. a global set of constraints. Each agent can either add a new constraint to the store (*Tell* operation) or check wether or not the store entails a given constraint (*Ask* operation). Synchronization is achieved through a *blocking ask* : an asking agent may block and suspend if one cannot state if the asked constraint is entailed nor its negation is entailed. Thus nothing prevents the constraint to be entailed, but the store does not yet contain enough information. The agent is suspended until other concurrently running agents add (*Tell*) new constraints to the store to make it strong enough to decide.

Obviously, the instantiation of this framework to unification constraints rephrases nicely the usual concurrent logic languages of section 3, with matching corresponding indeed to the *Ask* operation. Also restricting the language to only *Tell* operations reproduces the previous CLP languages.

These languages have not yet been implemented, but current research shows that on some specific domains, such as the finite domains as proposed by CHIP [49], the implementation technologies and know-how developed for both concurrent languages and constraint languages can be merged and nicely integrated, so that efficient implementations seem now at hand.

7.4 Use of highly and massively parallel multiprocessors

An ever increasing number of highly (in the order of 10^2) and massively (in the order of 10^3) parallel multiprocessors are becoming available. Highly parallel multiprocessors may provide logically shared memory based on physically distributed memory, access time to the shared memory being non uniform (NUMA: Non Uniform Memory Acess). An example of such a multiprocessor is the BBN Butterfly, which includes up to 128 processing elements, Motorola M68020 in the GP1000 or M88100 in the TC2000. Massively parallel multiprocessors have the NORMA (NO Remote Memory Access) architecture, where inter-processor communication is performed by message passing. Among the commercial massively parallel multiprocessors appearing on the market, the Supernode produced by several European manufacturers and including up to 1,024 T800 Transputer processors [46], AP1000 from Fujitsu, including up to 1,024 Sparc processors, CM-5 from Thinking Machines including up to 16,000 Sparc processors, the Intel Paragon, including up to 4,000 processors, etc. Although intended primarily to solve number-crunching applications, these multiprocessors could help solving efficiently large symbolic applications, currently untractable on sequential computers. In order for parallel logic programming systems to use efficiently a large number of processing elements, it is necessary that the techniques used in these systems scale well with the number of processing elements.

Table 4: Execution times of the MUSE prototype on Butterfly TC2000

Execution times and speedups of a knowledge based system checking the design of a circuit board.

No. of PEs	1	10	20	30	37
Time (s)	105.97	10.81	5.56	3.93	3.29
Speedup	1	9.80	19.0	26.96	32.2

7.4.1 Processor-memory connection issue

Systems performing sharing of computation stacks and designed for UMA (Uniform Memory Access) shared-memory multiprocessors may not be appropriate for NUMA multiprocessors such as the Butterfly GP1000 and Butterfly TC2000. On these machines, accessing local memory on a node is one order of magnitude faster than accessing shared memory on a remote node. In addition since cache coherency is not provided, shared memory cannot be cached and local accesses to shared memory are slower than local accesses to private memory. This is the case for all accesses to the execution stacks which cannot be cached since they are shared amongst workers. This problem was exhibited by the Aurora prototype implementation on Butterfly [72] running sequentially 80 % slower than SICStus , to be compared with the 20 % overhead of Aurora running on UMA shared memory Sequent Symmetry. In spite of these problems, experiments performed demonstrated the possibility of obtaining almost linear speedups with 80 processing elements computing large size problems.

Systems performing stack copying are better suited for NUMA multiprocessors. Experimental results of the MUSE prototype running on Butterfly TC2000 [1] indicate that the overhead of MUSE running sequentially over SICStus Prolog remains limited to 5 % while almost linear speedups are provided up to 40 processors (see Table 4).

7.4.2 Design of specialized architectures

Since NUMA architectures are not suitable for models sharing the computation stacks, one track of research attempts to design a scalable UMA architecture. In the Data Diffusion Machine [104], the location of a datum is decoupled from its virtual address and data migrates automatically where it is needed: memory is actually organised like a very large cache. The hardware organisation consists of a hierarchy of buses and data controllers linking processors, each having a set-associative memory. The machine is scalable since the number of levels in the hierarchy is not fixed and it should be possible to build a multiprocessor including a large number of processing elements with a limited number of levels.

The most important research activity in this area is performed by the ICOT, in the Fifth Generation Computer Systems project. This project will deliver the PIM/p multiprocessor, including up to 512 specialized processors [40]. PIM/p is composed of up to 64 clusters, each cluster being similar to a (physically) shared memory multiprocessor including 8 processors. The implementation techniques of KL1 on the PIM/p machine have been experienced with the Multi-PSI/V2 distributed memory multiprocessor [77].

The intra-cluster issues of the PIM/p machine are explored inside a Multi-PSI/V2 processor while inter-cluster issues of PIM/p are experienced between Multi-PSI/V2 processing elements. This is the case for example with intra- [18] and inter-cluster [54] incremental garbage collection techniques.

7.4.3 Scheduling issue

Several research projects address the scalability issues raised by scheduling and load balancing over massively multiprocessors. In order to avoid bottlenecks that could arise in centralized or distributed solutions, multi-level hierarchical load-balancing schemes have been proposed [10] [68]. Bottlenecks are expected to arise in centralized solutions with a large number of processing elements accessing a central scheduler. Fully distributed solutions are expected to require some shared data and result in bottlenecks to access these shared data together with a large amount of inter-processor communications. Experimental results over 64 PEs Multi-PSI NORMA architectures, reported in [68], indicate that multi-level load-balancing increases the speedups when using more than 32 PEs: a speedup of 50, using 64 PEs, was obtained for a puzzle solving program.

An additionnal problem arising on massively parallel multiprocessors is the high cost of maintaining an exact global state of the system. A scheduling strategy, using an approximate state of the system, has been proposed in [11].

8 Conclusion

An important research activity is being dedicated to parallelizing logic programming, mainly because of the "intrinsic" parallelism of logic programming languages and of the computational needs of symbolic problems. Parallel logic programming systems exploit essentially OR-parallelism, simultaneous computation of several resolvents and AND-parallelism, simultaneous computation of several goals of a clause. AND-parallel execution can be restricted to goals which do not share any variable, in which case it is called independent. Otherwise, it is called dependent AND-parallelism.

Two main approaches can be distinguished: the first one aims at exploiting parallelism transparently by parallelizing the Prolog language, while the second one develops language constructs allowing programmers to express explicitly the parallelism of their applications. The second approach is mainly represented by the family of Concurrent Logic Languages, which aim at supporting the programming of applications not addressed by Prolog such such as reactive systems and operating system programming.

In spite of the inherent parallelism of logic languages, implementing them efficiently in parallel raises difficult problems. Optimisations arising from backtracking cannnot be applied for OR-parallel systems. AND-parallel systems need to check that goals executed in parallel assign their variables to coherent bindings. Independent AND-parallel goals cannot bind the same variables. Independence can be computed at compile time, at runtime or partly at compile time and partly at runtime. Most runtime independence tests are time consuming and the most efficient results are obtained when the goal independence can be computed at compile time. Dependent AND-parallel systems only compute determinate goals in parallel, determinacy being enforced by the language in concurrent logic systems or computed partly at compile time and partly at runtime in the Andorra model of computation.

A large number of parallel computational models have been proposed for logic programming and some of them have been implemented efficiently on multiprocessors. The most mature systems exploit one type of parallelism. Systems exploiting OR- and independent AND-parallelism obtain linear speedups for programs containing enough parallelism while programs containing no exploitable parallelism run almost as efficiently as if executed by an efficient Prolog system. Systems exploiting dependent AND-parallelism are usually several times less efficient but they can exploit more parallelism of finer grain in logic programs. Systems combining several types of parallelisms are not for the moment as efficient as systems exploiting a single type of parallelism but they may result in a reduction of the search space of programs by avoiding useless computations. In addition, these systems can speed up more logic programs than systems exploiting one type of parallelism.

Current and future research in the domain of logic programming is active in several domains. One track of research is improving existing systems, mainly by developing compile-time analysis techniques of logic programs to detect determinacy of goals and independence between goals and to anticipate the granularity of resolvents (OR-parallelism) or goals (AND-parallelism). Another fruitful direction of research is the combination of constraint logic programming with parallelism. The last direction of research mentioned in this paper is the use of highly and massively parallel multiprocessors. The most important activity in this area is performed by the Japanese Fifth Generation Computer Systems project, where the PIM/p multiprocessor to be demonstrated in 1992, will deliver several hundred millions of reductions per second (one reduction is equivalent to one lip) peak performance.

In spite of the excellent performance results achieved by parallel logic systems, the cost-effectiveness of parallel logic programming is still questionable. Parallel multiprocessors generally do not use the most recent microprocessors already used in the most powerful uniprocessor workstations. In other words, because of the rapid progresses in the VLSI technology, a sequential Prolog system running on the most recent type of workstation is hardly superseded by a parallel logic programming system running on a modestly parallel multiprocessor based on the previous generation of microprocessors. This situation might change rapidly since several parallel workstations using state-of-the-art microprocessors are becoming commercially available. There are not yet enough applications likely to benefit from highly and massively parallel systems, since the traditional use of logic programming is constrained to making prototypes. This situation may change with the advent of highly and massively parallel multiprocessors which will enable to address problems too large to be solved by the most efficient sequential logic programming systems.

Parallel logic programming systems may become more cost-effective if multiprocessor technology becomes more mature and incorporates the most recent microprocessors as soon as uniprocessors. A more radical change may occur when progresses in VLSI technology are limited by physical constraints such that parallel execution becomes the only possibility of increasing the performances of computers. Parallel logic programming systems will then be much more efficient than sequential systems for programs having exploitable parallelism and not less for the other ones. In addition, parallelism will be much easier to express with these languages than using more "traditional" imperative languages, whose semantics is much more sequential.

Acknowledgements

The authors would like to thank D.H.D. Warren for the idea of this paper and helpful suggestions and corrections. They would also like to thank Jacques Cohen for reading the paper in detail and proposing numerous improvements.

References

[1] K. Ali and R. Karlsson. OR-Parallel Speedups in a Knowledge Based system: on Muse and Aurora. Submitted to the International Conference on Fifth Generation Computer Systems 1992, 1992.

[2] K. A. M. Ali and R. Karlsson. The Muse or-parallel Prolog model and its performance. In *Proceedings of the North American Conference on Logic Programming NACLP'90*, Austin, 1990. MIT Press series on Logic Programming.

[3] Khayri A. M. Ali and Roland Karlsson. Scheduling Or-parallelism in muse. In Furukawa [37], pages 807–821.

[4] U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.C. Syre, and H. Westphal. The parallel ecrc Prolog system PEPSys: An overview and evaluation results. In *Proceedings FGCS'88*, Tokyo, Nov-Dec 1988. International Conference on Fifth Generation Computer Systems.

[5] U.C. Baron, B. Ing, M. Ratcliffe, and P. Robert. A distributed architecture for the PEPSys parallel logic programming system. In *Proceedings ICPP'88*, Chicago, August 1988. International Conference on Parallel Processing.

[6] A. Beaumont, S. Muthu Raman, P. Szeredi, and D.H.D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE'91 Parallel Architectures and Languages Europe*, pages 403–420, Eindhoven, The Netherlands, 1991. Springer-Verlag, Lecture Notes in Computer Science No. 506.

[7] Peter Borgwardt. Parallel prolog using stack segments on shared memory multiprocessors. In *84 Int. Symposium on Logic Programming*, pages 2–11. IEEE, February 1984.

[8] P.G. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi. Logic and functional programming on distributed memory architectures. In *Proceedings of the* 6^{th} *International Conference on Logic Programming*, pages 325–339, Jerusalem, June 1990.

[9] P. Brand. Wavefront scheduling. Internal report gigalips project, SICS, 1988.

[10] J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. Scheduling of or-parallel prolog on a scalable, reconfigurable, distributed memory multiprocessor. In *PARLE'91 Parallel Architectures and Languages Europe*, pages 385–402, Eindhoven, The Netherlands, 1991. Springer-Verlag, Lecture Notes in Computer Science No. 506.

[11] J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. OPERA: OR-Parallel Prolog System on Supernode. In P. Kacsuk and M. Wise, editors, *Implementation of Distributed Prolog*. John Wiley and Sons, to appear in 1992.

[12] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):98–124, 1991. revised version of K.U.L. technical report CW-62, 1987.

[13] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling or-parallelism: An Argonne perspective. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605, Seattle, August 1988.

[14] R. Butler, E.L. Lusk, R. Olson, and R.A. Overbeek. ANLWAM: A Parallel Implementation of the Warren Abstract Machine. Technical report, Argonne National Laboratories, August 1986.

[15] L. Selle C. Percebois, N. Signes. An Actor-Oriented Computer for Logic and Its Application. In P. Kacsuk and M. Wise, editors, *Implementation of Distributed Prolog*. John Wiley and Sons, to appear in 1992.

[16] A. Calderwood and P. Szeredi. Scheduling or-parallelism in Aurora. In *Proceedings of the 6^{th} International Conference on Logic Programming*, Lisboa, June 1989.

[17] M. Carlsson and J. Widen. SICStus Prolog user manual. Research report, SICS, 1988.

[18] T. Chikayama and Y. Kimura. Multiple reference management in flat GHC. In Lassez [62], pages 276–293.

[19] K. Clark and S. Gregory. A relational language for parallel programming. In *ACM conference on Functional Languages and Computer Architecture*. ACM Press, 1981.

[20] K. Clark and S. Gregory. PARLOG : Parallel programming in logic. *ACM TOPLAS*, 8 (1), January 1986.

[21] W.F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.

[22] Christian Codognet, Philippe Codognet, and Marc-Michel Corsini. Abstract interpretation for concurrent logic languages. In Debray and Hermenegildo [31], pages 215–232.

[23] A. Colmerauer. An introduction to Prolog-III. *communications of the ACM*, 1990.

[24] J. S. Conery. *The AND/OR Process Model for Parallel Execution of Logic Programs*. PhD thesis, Univ. of California, Irvine, 1983. Tech. Report 204, Dept. of Computer and Information Science, UCI.

[25] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic andorra model. In Furukawa [37], pages 825–839.

[26] P. Cousot and R. Cousot. Abstract interpretation : a unified framework for static analysis of programs by approximation of fixpoint. In *4th ACM Symposium on Principles of Programming Languages*. ACM Press, 1977.

[27] J. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors*. PhD thesis, Department of Computer Science, Herriot-Watt University, Edinburgh, May, 1988.

[28] J. Chassin de Kergommeaux. Measures of the PEPSys implementation on the MX500. Technical Report CA-44, ECRC, January 1989.

[29] J. Chassin de Kergommeaux and P. Robert. An abstract machine to implement efficiently or-and parallel prolog. *Journal of Logic Programming*, 8(3), 1990.

[30] E. Villemonte de la Clergerie. *Dyalog: complete evaluation of Horn clauses by dynamic programming.* PhD thesis, INRIA, to appear in 1992.

[31] Saumya Debray and Manuel Hermenegildo, editors. *Proceedings of the 1990 North American Conference on Logic Programming.* MIT Press, 1990.

[32] Doug DeGroot. Restricted and-parallelism. In *Proc. of FGCS'84*, pages 471–478. ICOT, November 1984.

[33] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *communications of the ACM*, 18 (8), 1975.

[34] T. Disz, E. Lusk, and R. Overbeek. Experiments with or-parallel logic programs. In 4_{th} *Int. Conf. on Logic Programming*, pages 576,599, Melbourne, May 1987.

[35] E. Shapiro (ed.). *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge, Massachussets, 1987.

[36] Ian Foster and Steven Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[37] Koichi Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming.* MIT Press, 1991.

[38] Gupta Gopal and Bharat Jayaraman. Optimizing And-Or Parallel implementations. In Debray and Hermenegildo [31], pages 605–623.

[39] A. Goto, T. Shinogi, T. Chikayama, K. Kumon, and A. Hattori. Processor Element Architecture for a Parallel Inference Machine, PIM/p. *Journal of Information Processing*, 13(2), 1990.

[40] A. Goto, T. Shinogi, T. Chikayama, K. Kumon, and A. Hattori. Processor Element Architecture for a Parallel Inference Machine, PIM/p. *Journal of Information Processing*, 13(2):174–182, 1990.

[41] G. Gupta and M. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *ICLP'91 Workshop on Parallel Execution of Logic Programs, Springer-Verlag*, Paris, 1991.

[42] G. Gupta and M. Hermenegildo. Ace: And/Or-parallel copying-based execution of logic programs. Technical report, Universidad Politécnica de Madrid, 1991.

[43] Gopal Gupta, Vítor Santos Costa, Rong Yang, and Manuel V. Hermenegildo. IDIOM: Integrating dependent And-, independent And-, and Or-parallelism. In Saraswat and Ueda [87], pages 152–166.

[44] R. Moolenar H. Van Hecker and B. Demoen. A parallel implementation of AKL. In *ILPS 91 workshop on Implementation of Parallel Logic Programming Systems*, 1991.

[45] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In Warren and Szeredi [105], pages 31–46.

[46] J.G. Harp, C.R. Jesshope, T. Muntean, and C. Whitby-Stevens. The development and application of a low cost high performance multiprocessor machine. In *Proceedings ESPRIT'86: Results and Achievements.* Elsevier Science Publishers, 1986.

[47] B. Hausman. Pruning and scheduling speculative work in or-parallel system. In *PARLE'89 Parallel Architectures and Languages Europe*, Eindhoven, june 1989. Springer-Verlag, Lecture Notes in Computer Science, No. 366.

[48] B. Hausman. *Pruning and Speculative Work in OR-Parallel Prolog.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1990. SICS research report D-90-9001.

[49] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[50] M. V. Hermenegildo. An abstract machine for restricted and-parallel execution of logic programs. In 3^{rd} *Int. Conf. on Logic Programming*, pages 25–39, London, July 1986.

[51] M. V. Hermenegildo. Relating goal scheduling, precedence, and memory management in and-parallel execution of logic programs. In 4^{th} *Int. Conf. on Logic Programming*, pages 556–575, Melbourne, May 1987.

[52] M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In Warren and Szeredi [105], pages 253–268.

[53] Manuel Hermenegildo and Francesca Rossi. Non-strict independent and-parallelism. In Warren and Szeredi [105], pages 237–252.

[54] N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. *New Generation Computing*, 7:159–177, 1990.

[55] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *13th ACM symposium on Principles of Programming Languages, POPL 87.* ACM Press, 1987.

[56] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Saraswat and Ueda [87], pages 167–186.

[57] N. Jones and H. Sondergaard. A semantic-based framework forthe abstract interpretation of prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declartative Languages*, pages 123–142. Ellis Horwood, 1987.

[58] Laxmikant V. Kalé and Balkrishna Ramkumar. Joining AND Parallel solutions in AND/OR Parallel systems. In Debray and Hermenegildo [31], pages 624–641.

[59] Shmuel Klinger and Ehud Shapiro. A decision tree compilation algorithm for FCP (— , : , ?). In Kowalski and Bowen [61], pages 1315–1336.

[60] R.A. Kowalski. *Logic for Problem Solving.* Elsevier Science Publishing, 1979.

[61] Robert A. Kowalski and Kenneth A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming.* MIT Press, 1988.

[62] Jean-Louis Lassez, editor. *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming. MIT Press, 1987.

[63] Giorgio Levi and Maurizio Martelli, editors. *Proceedings of the Sixth International Conference on Logic Programming.* MIT Press, 1989.

[64] Y. J. Lin and V. Kumar. And-parallel execution of logic programs on a shared memory multiprocessor: A summary of results. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141, Seattle, August 1988.

[65] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, New-York, London, Paris, Tokyo, 1987.

[66] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwodd, P. Szeredi, S. Haridi, P. Brand, M. Carlson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. In *Proceedings FGCS'88*, Tokyo, Nov-Dec 1988. International Conference on Fifth Generation Computer Systems.

[67] H. Simonis M. Dincbas and P. van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8 (1-2), 1990.

[68] A N. Ichiyoshi M. Furuichi, K. Taki. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. *ACM SIGPLAN NOTICES*, 25(3):50–59, march 1990.

[69] M. J. Maher. Logic semantics for a class of committed-choice programs. In Lassez [62], pages 858–876.

[70] H. Masuzawa and et al. Kabu wake parallel inference mechanism and its evaluation. In *1986 FJCC*, pages 955–962. IEEE, November 1986.

[71] C. S. Mellish. Abstract interpretation of Prolog programs. In Shapiro [91], pages 463–474.

[72] Shyam Mudambi. Performances of aurora on NUMA machines. In Furukawa [37], pages 793–806.

[73] K. Muthukumar and M. Hermenegildo. Complete and efficient methods for supporting side-effects in independent/restrict ed AND-Parallelism. In Levi and Martelli [63], pages 80–97.

[74] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information through Abstract Interpretation. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 166–188, 1989.

[75] K. Muthukumar and M. V. Hermenegildo. The DCG, UDG, and MEL methods for automatic compile-time parallelization of logic programs for independent and-parallelism. In Warren and Szeredi [105], pages 221–236.

[76] Lee Naish. Parallelizing NU-Prolog. In Kowalski and Bowen [61], pages 1546–1564.

[77] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In 6^{th} *International Conference on Logic Programming*, Lisboa, 1989.

[78] Doug Palmer and Lee Naish. NUA-Prolog: An extension to the WAM for parallel andorra. In Furukawa [37], pages 429–442.

[79] L. M. Pereira and A. Porto. Intelligent backtracking and sidetracking in horn clause programs. Technical report, Universitade Nuova de Lisboa, 1979. CINUL 2/79.

[80] B. Ramkumar. *Machine Independent "AND" and "OR" Parallel Execution of Logic Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[81] B. Ramkumar and L. V. Kalé. Compiled Execution of the Reduce-OR Process Model on Multiprocessors In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 313–331, 1989.

[82] K. Schuerman S. A. Delgado-Rannauro and J. Xu. The elipsys computation model. Technical report, ECRC, Munich, 1990.

[83] M. Hermenegildo S. K. Debray, N-W Lin. Task granularity analysis in logic programs. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 20-22 1990.

[84] V. A. Saraswat. The concurrent logic programming language cp: Definition and operational semantics. In *13th ACM symposium on Principles of Programming Languages, POPL 87.* ACM Press, 1987.

[85] V. A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnagie-Mellon University, 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press 1991.

[86] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *16th ACM symposium on Principles of Programming Languages. POPL 90.* ACM Press, 1990.

[87] Vijay Saraswat and Kazunori Ueda, editors. *Proceedings of the 1991 International Logic Programming Symposium.* MIT Press, 1991.

[88] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on shared memory multiprocessor. In *Proceedings of the IFIP Working Conference on Parallel Processing.* North-Holland, may 1988.

[89] E. Shapiro. A subset of concurrent prolog and its interpreter. Technical report, Weizmann Institute, Rehovot, Israel, 1983.

[90] E. Shapiro. The family of concurrent logic programming languages. *ACM computing surveys*, 21(3), september 1989.

[91] Ehud Shapiro, editor. *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1986.

[92] Kish Shen and Manuel V. Hermenegildo. A simulation study of Or- and independent And-parallelism. In Saraswat and Ueda [87], pages 135–151.

[93] Peter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceeding of the North American Conference on Logic Programming NACLP'89*, Cleveland, October 1989.

[94] S. Taylor, S. Safra, and E. Shapiro. A parallel implementation of Flat Concurrent Prolog. *Journal of Parallel Programming*, 15(3), 245–275.

[95] E. Tick. Compile-Time Granularity Analysis for Parallel Logic Programming Languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS'88*, pages 994–1000, Tokyo, 1988.

[96] E. Tick. Comparing two Parallel Logic Programming Architectures. *IEEE Software*, July 1989.

[97] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *Computer Journal*, 33 (6), 1990.

[98] Kalé L. V. The REDUCE-OR process model for parallel evaluation of logic programs. In Lassez [62], pages 616–632.

[99] P. Van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of chip within PEPSys. In Levi and Martelli [63], pages 165–180.

[100] D. H. D. Warren. The andorra principle. Internal report, Gigalips Group, 1988.

[101] D. H. D. Warren. The Extended Andorra Model with implicit control. In *ICLP 90 workshop on Parallel Logic Programming*, 1990. presentation slides.

[102] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Report tn309, SRI, October 1983.

[103] D.H.D. Warren. The SRI model for or-parallel execution of Prolog. abstract design and implementation issues. In 4^{th} *Symposium on Logic Programming*, pages 46–53, San Fransisco, Sept. 1987.

[104] D.H.D. Warren and S. Haridi. Data Diffusion Machine - A scalable shared virtual memory multiprocessor. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 943–952, Tokyo, 1988.

[105] D.H.D. Warren and P. Szeredi, editors. *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, 1990.

[106] H. Westphal, P. Robert, J. Chassin, and J.-C. Syre. The pepsys model: Combining backtracking, and- and or-parallelism. In 4^{th} *Symposium on Logic Programming*, pages 436–448, San Fransisco, Sept. 1987.

[107] Rong Yang and Hideo Aiso. P-prolog: a parallel logic language based on exclusive relation. In Shapiro [91], pages 255–269.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399