# Parallel logic programming

*This document contains an overview of the current state-of-the-art in the implementation of Logic Programming Languages, with particular focus on exploitation of parallelism.*

*The presentation is divided in two parts. The first part gives an overview of the most relevant implementation techniques currently used in parallel logic programming, along with a brief description of the most effective sequential and parallel systems available. The second part deals with the issue of efficiency of the implementation models: the major sources of inefficiency in the current systems are identified and possible solutions to improve them are considered.*

# Contents

## Part1



## Part2

# Part3



# Part4

# Introduction

Parallel logic programming (PLP) systems are sophisticated examples of symbolic computing systems. PLP systems address problems such as allocating dynamic memory, scheduling irregular computations, and managing different types of implicit parallelism. Most PLP systems have been developed for bus-based architectures. However, the complexity of PLP systems and the large amount of data they process raise the question of whether logic programming systems can still achieve good performance on modern scalable architectures, such as distributed shared-memory (DSM) systems. In this work we use execution-driven simulation of a cache-coherent DSM architecture to investigate the performance of Andorra-I, a state-of-the-art PLP system, on a modern multiprocessor. The results of this simulation show that Andorra-I exhibits reasonable running time performance, but it does not scale well. Our detailed analysis of cache misses and their sources expose several opportunities for improvements in Andorra-I. Based on this analysis, we modify Andorra-I using a set of simple techniques that led to significantly better running time and scalability. These results suggest that Andorra-I can and should perform well on modern multiprocessors. Furthermore, as Andorra-I shares its main data structures with several PLP systems, we conclude that the methodology and techniques used in our work can greatly benefit these other PLP systems.

Most research on parallel logic programming divides into (1) or-parallel implementations of Prolog, and (2) and-parallel implementations of committed choice variants of Prolog, the so-called concurrent logic languages. The reason is implementation efficiency: it is extremely complex to implement a combined and/or parallel system. This paper introduces the research on the concurrent and-parallel languages and their extensions, with an emphasis on Parlog. However, all the main concurrent languages are introduced and compared, and set in a historical context of precursor research. Recent work on extensions of the languages is described, particularly the Parlog extensions: Parlog + + and Polka for object-oriented programming, and Pandora for constrained search.

# History of Logic Programming

The logic programming paradigm has its roots in automated theorem proving from which it took the notion of a deduction. What is new is that in the process of deduction some values are computed. The creation of this programming paradigm is the outcome of a long history that for most of its course ran within logic and only later inside computer science. Logic programming is based on the syntax of first order logic, that was originally proposed in the second half of 19th century by Gottlob Frege and later modified to the currently used form by Giuseppe Peano and BertrandRussell.

In the 1930s Kurt Gödel and Jacques Herbrand studied the notion of computability based on derivations.

These works can be viewed as the origin of the "computation as deduction" paradigm. Additionally, Herbrand discussed in his PhD thesis a set of rules for manipulating algebraic equations on terms that can be viewed now as a sketch of a unification algorithm.

Some thirty years later in 1965 Alan Robinson published his fundamental paper that lies at the foundations of the field of automated deduction. In this paper he introduced the resolution principle, the notion of unification and a unification algorithm. Using the resolution method one can prove theorems of first-order logic, but another step was needed to see how one could compute within this framework. This was eventually achieved in 1974 by Robert Kowalski in his paper in which logic programs with a restricted form of resolution were introduced. The difference between this form of resolution and the one proposed by Robinson is that the syntax is more restricted, but proving now has a side effect in the form of a satisfying substitution. This substitution can be viewed as a result of a computation and consequently certain logical formulas can be interpreted as programs. In parallel, Alain Colmerauer with his colleagues worked on a programming language for natural language processing based on automated theorem proving. This ultimately

led to creation of Prolog in 1973. Kowalski and Colmerauer with his team often interacted in the period 1971–1973. This influenced their views and helped them to crystallize the ideas. Prolog can be seen as a practical realization of the idea of logic programs. It started as a programming language for applications in natural language processing, but soon after it was found that it can be used as a general purpose programming language, as well.

A number of other attempts to realize the computation as deduction paradigm were proposed around the same time, notably by Cordell Green and Carl Hewitt, but the logic programming proposal, probably because it was the simplest and most versatile, became most successful. Originally, Prolog was implemented by Philippe Roussel, a colleague of Colmerauer, in the form of an interpreter written in Algol-W. An important step forward was achieved by David H. Warren who proposed in 1983 an abstract machine, now called WAM (Warren Abstract Machine), that consists of a machine architecture with an instruction set which serves as a target for machine independent Prolog compilers. WAM became a standard basis for implementing Prolog and other logic programming languages. The logic programming paradigm influenced a number of developments in computer science. Already in the seventies it led to the creation of deductive databases that extend the relational databases by providing deduction capabilities. A further impetus to the subject came unexpectedly from the Japanese Fifth Generation Project for intelligent computing systems (1982–1991) in which logic programming was chosen as its basis. More recently, this paradigm led to constraint logic programming that realizes a general approach to computing in which the programming process is limited to a generation of constraints (requirements) and a solution of them, and to inductive logic programming, a logic based approach to machine learning. The above account of history of logic programming and Prolog shows its roots in logic and automated deduction.

In fact, Colmerauer and Roussel write in: “There is no question that Prolog is essentially a theorem prover ‘`a la Robinson.’ Our contribution was to transform that theorem prover into a programming language.” This origin of the logic paradigm probably impeded its acceptance within computer science in times when imperative programming got impetus thanks to the creation of Pascal and C, the fields of verification and semantics of imperative programs gained ground and when the artificial intelligence community already adopted Lisp as the language of their choice. Here we offer an alternative presentation of the subject by focusing on the ordinary programming concepts (often implicitly) present in logic programming and by relating various of its ideas to those present in the imperative and functional programming paradigms.

**Syntax of Logic Programming:.**

Skip this bit, if you already know something about logic programming. The introduction is intended for those who don't know anything about the subject.

Logic Programming is not, as you might expect, a very low-level part of computer science, working with gates. Logic Programming is rather an attempt to use predicate logic as the basis of a programming language, leading to some very high level languages.

**Declarative Programming**

Logic programming languages tend to be *declarative* programming languages, as opposed to the more traditional *imperative* programming languages, such as C or Pascal.

In a declarative language, you concentrate on the relationships between the parts of the problem, rather than on the exact sequence of steps required to solve the

problem. An example of a well-known declarative language is SQL; SQL queries are translated by the database engine into various lookup strategies, where the presence of indexes and the sizes of various tables and views are taken into account. All this detail doesn't matter too much to the programmer, who is only interested in the result of the query, rather than the exact steps taken to retrieve the result.

Imperative languages, in contrast to their declarative brethren, expect the programmer to provide an exact description of the steps required to solve some problem.

In a logic programming language, what is declared is the logical relationship between various entities. After that, if you supply some sort of query, the logic programming language can deduce the results of the query by tracing its path through the relationship.

**An Example**

As an example of how a set of logical statements can be used as a programming language, here is the ancestor relationship expressed logically:

Sue is-parent-of Bill.

Sue is-parent-of James.

Sue is-parent-of Edith.

Fred is-parent-of Bill.

Fred is-parent-of James.

Authur is-parent-of Edith.

Mary is-parent-of Kylie.

Mary is-parent-of Jason.

Mary is-parent-of Matilda.

James is-parent-of Kylie.

James is-parent-of Jason.

James is-parent-of Matilda.

Edith is-parent-of David.

William is-parent-of David.

*X* is-ancestor-of *Y* **if**

*X* is-parent-of *Y*

*X* is-ancestor-of *Y* **if**

**there is some** *Z* **such that**

*X* is-parent-of *Z* **and**

*Z* is-ancestor-of *Y*

This example contains two relationships: is-parent-of and is-ancestor-of. These relationships are called *predicates*.

The simpler predicate is is-parent-of. All the statements which make up is-parent-of are simple facts and can be easily interpreted. We can easily ask *"Is James the parent of Matilda?"* and, since James is-parent-of Matilda is present in the above list, the question will be answered with a "true". If we ask *"Is James the parent of David?"* then, since James is-parent-of David is not in the list, we will be answered with a "false".

is-ancestor-of is a more complex predicate, since it expresses the is-ancestor-of relationship as a pair of rules. Logically, the first relationship can be read as "Someone is the ancestor of somebody else if they are that person's parent." This is the simplest expression of what it means to be sombody's ancestor. The second part of the is-ancestor-of relationship is more complex. This part can be read as "Somebody is the ancestor of somebody else if there is another person who is the child of the first person, and this third person is the ancestor of the second person." If

this is a little confusing, try this example: *my grandmother is my ancestor, because she is the parent of my father and my father is my ancestor, because he is my parent.*

**Directionless Programming**

There are four ways in which we may want to use the above example:

- We can ask it a question. Eg. Sue is-ancestor-of David? is true (through the line Sue-Edith-David).
- We can ask it who the ancestors of somebody are. Eg. Who is-ancestor-of David? (Edith, William, Authur and Sue).
- We can ask it who the descendants of somebody is. Eg. Edith is-ancestor-of Who? (David).
- We can ask it for the complete set of ancestor relationships. Eg. Who is-ancestor-of Whom?

The way in which we have declared is-ancestor-of allows you to ask any of these questions. It's up to the logic programming language itself to provide you with all the answers.

**Logic Programming Resources**

Here is a list of useful URLs concerned with logic programming:

Virtual Library - Logic Programming

The WWW Virtual Library logic programming directory.

Database Systems and Logic Programming

A fairly complete search site for papers on logic programming.

The Journal of Functional and Logic Programming

An online journal.

**Prolog**

Prolog (**Pro**gramming in **Log**ic) is the first logic programming language and still the most common. Prolog now has a large number of commercial and free implementations.

**Syntax**

A Prolog predicate is written as *name*($arg_1$, ..., $arg_n$). For example, the parent relationship shown in the example would be written as parent(sue, bill). Predicate names start with lower-case letters

The arguments to predicates are constructed from *terms.*. The most primitive form of term is a *constant* which is represented by a word with a lower-case initial letter, eg. sue, walter, binary. Constants starting with upper-case letters or containing punctuation need to be surrounded by single quotes, eg. 'Sue', '#something', 'ancestor(sue, bill)'. Numbers are also constants.

The next most primitive form of term is a *variable.* Variables are represented by words beginning with an upper-case letter, or an underscore, eg. X, Alpha1, _IgnoreMe. Variables beginning with an underscore are *anonymous* variables; throw-away variables used as place-holders.

Complex terms are built from *function* symbols, consisting of a named function and some arguments in brackets, eg. g(X, Y), alpha(sue, matter), xor(1, 1), s(s(s(0))). In some cases, function symbols can be used as infix operators, eg. X + Y = '+'(X, Y).

*Lists* are used extensively in Prolog and have a special syntax. A list is represented by [ $elt_1$, ..., $elt_n$ ] eg. [1, 2, a, b, f(a, Y)] is a list with 5 elements. The empty list is represented as []. Lists are separated into head and tail elements by using the | symbol. [ $elt_1$, ..., $elt_n$ | $T$ ] is a list where the first $n$ elements are specified, and the

tail of the list is *T*. Eg. [a, b, f(a, b), 1, 6, g(2)] = [X, Y | Z] will give X = a, Y = b and Z = [f(a, b), 1, 6, g(2)]. In some code, a *dot notation* is used to separate head and tail; X.Y = [X | Y].

Predicates are defined by either stating facts, eg. parent(sue, bill) or rules, eg. ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y). In rules, the ':-' reads as an **if** and the comma reads as an **and**. Each fact or rule in a predicate is called a *clause.*

Queries in Prolog are expressed as **?-** $Goal_1$, ..., $Goal_n$. Eg. ?- parent(sue, X), ancestor(fred, X)

**The Example in Prolog**

The ancestor example, written in Prolog would look like:

parent(sue, bill).

parent(sue, james).

parent(sue, edith).

parent(fred, bill).

parent(fred, james).

parent(arthur, edith).

parent(mary, kylie).

parent(mary, jason).

parent(mary, matilda).

parent(james, kylie).

parent(james, jason).

parent(james, matilda).

parent(edith, david).

parent(william, david).

```
ancestor(X, Y) :-
        parent(X, Y).
ancestor(X, Y) :-
        parent(X, Z), ancestor(Z, Y).
```

An example query would be ?- ancestor(fred, david), *"Is fred the ancestor of david"* which would return a **no**.

Another example query would be ?- parent(sue, X), ancestor(fred, X), *"Is sue a parent of somebody who also has fred as an ancestor?"* In this case, Prolog will respond with **X=bill ?**; you may respond with a return to indicate that you are satisfied with this answer. If you respond with a ; then the system will search for another answer and return with **X=james ?**. If you respond with another ;, then you will get the response **no** indicating that there are no more answers.

**Prolog's Search Strategy**

When Prolog is searching for an answer to a query, it starts by choosing one of the goals that it has, to see if it can find an answer to that goal. Eg. if the query were ?- parent(sue, X), ancestor(X, david), then Prolog would choose parent(sue, X) as the first goal to try.

Once Prolog has chosen a goal, it starts to try and match the clauses of the goal, top-to-bottom until it finds one that fits. In the above example, parent(sue, bill) matches parent(sue, X) if X was *bound* to bill.

The Prolog search rule now tries the ancestor(X, david) part of the query, only, since X has been bound to bill the query is now really ancestor(bill, david).

The first clause of ancestor is now tried. This clause simply tries parent(bill, david). There is no parent(bill, david) clause, so the program *backtracks*. When a Prolog

program backtracks, it moves back to the next possible choice of clause that it has and tries that clause instead. Any variables that have been bound after that choice are undone, leaving them ready to try something else. In the current case, there are no variables to be unbound - X was bound *before* we started choosing clauses of ancestor.

The program now tries the next clause of ancestor, which leads to two further goals, parent(bill, Z), ancestor(Z, david). Prolog picks the leftmost goal again and tries parent(bill, Z).

Eventually, ancestor(bill, david) will fail. At this point, the program backtracks, unbinding X and choosing the next clause from parent(sue, X), which will lead to X being bound to james and the whole thing starting off again.

After another round of backtracking, X will be bound to edith and ancestor(edith, david) will succeed.

The Prolog search strategy, therefore, always picks the leftmost, innermost goal to try next and always tries clauses in top to bottom order. This strategy is called a *depth-first* search strategy, as it always searches down the possible goals, rather than across them. The depth-first strategy is very simple and efficient to implement, but can lead to some very odd infinite loops. For example:

```
p(a) :- p(Z).

q(b).

?- p(X), q(X)
```

In this example ?- p(X), q(X) should fail, since X will be bound to a by p and to b by q. However, the p(X) goal will go into an infinite loop, and q(X) will never be tried.

**Cut**

In ordinary use of Prolog, you are interested in any possible answer that can be computed. Occasionally, however, you want just a few answers, and then wish to *cut* away any further answers. This process is normally referred to as *pruning*.

In Prolog the cut operator, written as a ! allows pruning. When a cut operator is encountered in a clause, all possible alternatives below the clause are removed, as are any alternatives which have been computed to the left of the cut.

An example of the use of cut is where you have a series of alternatives, with a default at the end. If you encounter an alternative, then you want to ignore the default. As an example, the following table works out whether a foreign exchange deal is a spot deal or not5:

```
dealType(Date, Today, cad, _Currency2, spot) :-
        addDays(Today, 1, Date),
        !.
dealType(Date, Today, _Currency1, cad, spot) :-
        addDays(Today, 1, Date),
        !.
dealType(Date, Today, _Currency1, _Currency2, spot) :-
        addDays(Today, 2, Date),
        !.
dealType(_Date, _Today, _Currency1, _Currency2, outright).
```

addDays(*date*$_1$, *n*, *date*$_2$) is defined to be true when *Date*$_2$ is *n* days ahead of *Date*$_1$6. If we assume that dates are represented by a term: date(*day*, *month*, *year*), then we could define addDays in this way:

```
addDays(date(Day, Month, Year), Days, Date) :-
```

```
        plus(Day, Days, Day1),
        normalizeDate(date(Day1, Month, Year), Date).

normalizeDate(date(Day, Month, Year), Date) :-
        Day =< 0,
        !,
        nextMonthYear(Month1, Year1, Month, Year),
        daysInMonth(Month1, Year1, DIM),
        plus(Day, DIM, Day1),
        normalizeDate(date(Day1, Month1, Year1), Date).
normalizeDate(date(Day, Month, Year), Date) :-
        daysInMonth(Month, Year, DIM),
        Day > DIM,
        !,
        nextMonthYear(Month, Year, Month1, Year1),
        plus(Day1, DIM, Day),
        normalizeDate(date(Day1, Month1, Year1), Date).
normalizeDate(Date, Date).

daysInMonth(jan, _Year, 31).
daysInMonth(feb, Year, 29) :-
        isLeapYear(Year), !.
daysInMonth(feb, Year, 28).
daysInMonth(mar, _Year, 31).
daysInMonth(apr, _Year, 30).
daysInMonth(may, _Year, 31).
daysInMonth(jun, _Year, 30).
daysInMonth(jul, _Year, 31).
```

```
daysInMonth(aug, _Year, 31).
daysInMonth(sep, _Year, 30).
daysInMonth(oct, _Year, 31).
daysInMonth(nov, _Year, 30).
daysInMonth(dec, _Year, 31).

nextMonthYear(jan, Year, feb, Year).
nextMonthYear(feb, Year, mar, Year).
nextMonthYear(mar, Year, apr, Year).
nextMonthYear(apr, Year, may, Year).
nextMonthYear(may, Year, jun, Year).
nextMonthYear(jun, Year, jul, Year).
nextMonthYear(jul, Year, aug, Year).
nextMonthYear(aug, Year, sep, Year).
nextMonthYear(sep, Year, oct, Year).
nextMonthYear(oct, Year, nov, Year).
nextMonthYear(nov, Year, dec, Year).
nextMonthYear(dec, Year, jan, NewYear) :-
        plus(Year, 1, NewYear).

leapYear(Year) :-
        Year \\ 400 =:= 0, !.
leapYear(Year) :-
        Year \\ 4 =:= 0,
        Year \\ 100 =\= 0.
```

In this example, =:= tests for numerical equality, =\= tests for numerical inequality > tests for the greater-than relationship and =< tests for the less than or equal to relationship. \\ is the modulus operator. plus(*A, B, C*) succeeds if *A+B=C,* with one

of *A, B* or *C* being bound to the correct value if the other two arguments are numbers.

Cut damages the idea of directionless programming7. For example, the query ?-daysInMonth(M, 1996, DIM) will not give each month and the number of days in the month for each month; the cut in daysInMonth(feb, 1996, 29) eliminates any further choices.

**Prolog Resources**

Here is a list of handy URLs for getting various Prolog resources:

Quintus Prolog

A fully-fledged commercial Prolog system.

SICStus Prolog

Another commercial Prolog system, with academic and free student licenses. This page also contains a good set of links to Prolog libraries.

Prolog Library

A collection of Prolog routines.

jProlog

A Prolog to Java compiler??!

**Agents Kernel Language**

Prolog's search strategy can be very inefficient. Very often, many fruitless search branches are explored which could be ignored if more information from later in the query were made available. As an example, consider the following program and query:

```
number(0).
number(s(X)) :- number(X).
```

?- number(X), X = s(s(s(0))).

With a normal Prolog search strategy, the program would generate 0, s(0) and s(s(0)) before generating s(s(s(0))). Each value of X which is rejected causes the program to backtrack. If the program could somehow move ahead to the X = s(s(s(0))) part, then the call to number becomes a simple check.

The easiest goals in a query to execute are those with only one possible clause to execute. These goals are called *determinate* goals, in contrast to *nondeterminate* where several branches may have to be tried. Determinate goals represent "no regrets" goals; they have to be executed sometime, so the program should execute them as soon as possible and see if they bind enough variables to make another goal determinate.

The *Andorra Principle* essentially says "Do the determinate bits first". The Agents Kernel Language, or AKL is a Prolog-like language designed to take advantage of the Andorra principle.

**Syntax**

The AKL syntax is similar to Prolog-syntax, but somewhat more complex to allow the AKL to decide how much of a goal to try to see whether it is determinate or not.

The AKL is a *guarded* language. Clauses are written as *Head* :- *Guard, Operator, Body*, where *Guard* and *Body* are normal, comma-separated lists of goals and *Operator* is one of three guard operators:

Wait (?)

Implies a normal search, with the ? used to separate the guard from the body.

Conditional (->)

Works in a similar manner to the Prolog cut. If a guard completely succeeds, then all possible clauses below this clause are pruned - possibly leading to a determinate goal.

Commit (|)

If a guard completely succeeds then the commit operator prunes away all other possible branches, leading to a determinate goal.

All clauses in a predicate must have the same guard operator.

If no guard operator is specified, then a guard and guard operator of true ? is assumed. Facts are assumed to have a guard and body of true ? true, eg. parent(james, kylie) would be more correctly written as parent(james, kylie) :- true ? true..

Queries follow the Prolog model.

**The Example in AKL**

The ancestor example, written in AKL would look like:

```
parent(sue, bill).
parent(sue, james).
parent(sue, edith).
parent(fred, bill).
parent(fred, james).
parent(arthur, edith).
parent(mary, kylie).
parent(mary, jason).
parent(mary, matilda).
parent(james, kylie).
parent(james, jason).
```

```
parent(james, matilda).
parent(edith, david).
parent(william, david).

ancestor(X, Y) :-
        parent(X, Y) ? true.
ancestor(X, Y) :-
        parent(X, Z) ? ancestor(Z, Y).
```

Putting the parent part of the ancestor predicate in the guard allows the system to quickly check to see whether the clause is likely to succeed. A query such as ?-ancestor(sue, kylie) will immediately cause the first clause to fail, leaving the way open for the program to determinately commit to the second clause.

**Execution**

An AKL program essentially acts like a Prolog program, except that the leftmost, depthmost goal is not always selected. Preference is given to any goal where the AKL can detect determinism.

When a goal is selected, all the guards in each clause in the predicate are all tried. With a bit of luck, all the guards except one will fail and the goal can be *determinatly promoted*. A determinately promoted goal first allows its guard to complete, and then sets about executing the body of the clause.

If there are two or more possible clauses when the guards complete, then the goal suspends until further bindings cause enough guards to fail. Any variable bindings made by the guards are treated as speculative; they apply to the guard, but do not leak into the surrounding computation. Part of promotion is the promotion of speculative bindings into real bindings.

Clauses (and predicates) with conditional or commit guards must have *quiet* guards before the operators are allowed to prune. Quiet guards have no speculative bindings; either the guard doesn't care about the state of a variable, or the variable has been really bound. Quietness makes sure that pruning operators don't get activated by an over-enthusiastic guard. For example:

```
p(X) :- X = a | true.

p(X) :- X = b | true.


q(X) :- true ? X = b.


?- p(X), q(X).
```

In the example above, if quietness were not required, then each guard in p would speculatively bind X and then commit. Without quietness, therefore, there is a race to see whether X = a or X = b. Quietness ensures that no pruning occurs until q is called, which is determinate and binds X to b.

At some point, most AKL programs won't be able to find a determinate goal. At this point, the AKL program must fall back on the Prolog-style search. Since the an AKL program has usually tried many goals in the program, there is a wider variety of possible choices. The AKL picks a goal which is wait-guarded and then promotes the first clause, leaving the remaining clauses for backtracking. Hopefully, this *nondeterminate promotion* will cause something else to be determinate and start the ball rolling again.

**Concurrency and Parallelism**

The AKL chooses a goal to execute from a set of determinate goals. Since the order of execution of these goals does not matter, the AKL allows both *concurrent* programming and *parallelism*.

In concurrent programming, although each goal is evaluated one at a time, the system is free to switch between goals as it sees fit. This kind of behaviour is similar to a multitasking operating system, and concurrent systems allow logic programs to be viewed as a set of threads, all acting in concert.

Concurrent programming gives the appearance of parallelism, without actually having a parallel set of systems. In true parallelism, several determinate goals can be evaluated at once.

*Stream and-parallelism* views individual predicates as independent, communicating processes. The processes communicate by incrementally filling out a list; as one process binds the head of a list, another process can commit to a single clause and process the list element. As an example, the following program and query sums up a series of numbers:

```
numbers(0, X) :- true -> X = [].
numbers(N, X) :- true ->
        X = [N | X1],
        plus(N1, 1, N),
        numbers(N1, X1).
```

First-Order Logic:. First-order logic is symbolized reasoning in which each sentence, or statement, is broken down into a subject and a predicate. The predicate modifies or defines the properties of the subject. In first-order logic, a predicate can only refer to a single subject. First-order logic is also known as first-order predicate calculus or first-order functional calculus. A sentence in first-order logic is written in the form Px or P(x), where P is the predicate and x is the subject, represented as a variable. Complete sentences are logically combined and manipulated according to the same rules as those used in Boolean algebra. In first-order logic, a sentence can be structured using the universal quantifier (symbolized ) or the existential quantifier (). Consider a subject that is a variable represented by x. Let A be a predicate "is an apple," F be a predicate "is a fruit," S be a predicate "is sour"', and M be a predicate "is mushy." Then we can say

x : Ax Fx

which translates to "For all x, if x is an apple, then x is a fruit." We can also say such things as

x : Fx Ax

x : Ax Sx

x : Ax Mx

where the existential quantifier translates as "For some."

First-order logic can be useful in the creation of computer programs. It is also of interest to researchers in artificial intelligence (AI). There are more powerful forms of logic, but first-order logic is adequate for most everyday reasoning. The Incompleteness Theorem, proven in 1930, demonstrates that first-order logic is in general undecidable. That means there exist statements in this logic form that, under certain conditions, cannot be proven either true or false.

**Translating English to FOL**

- Every gardener likes the sun. (Ax) gardener(x) => likes(x,Sun)

- You can fool some of the people all of the time. (Ex) (person(x) ^ (At)(time(t) => can-fool(x,t)))

- You can fool all of the people some of the time. (Ax) (person(x) => (Et) (time(t) ^ can-fool(x,t)))

- All purple mushrooms are poisonous. (Ax) (mushroom(x) ^ purple(x)) => poisonous(x)

## 1. The parallel wave

Evolution of computer technology has been driven by one major factor:

The uninterrupted search for increased power and speed ,While technology for sequential processors is quickly approaching the physical limits of chip building, an explosive growth of interest in *Parallel Computing* has started dominating the computer science community. In parallel computing, the traditional *Von Neumann* view of a computer as a single processing unit, capable of executing a single stream of instructions at once, is replaced by a *cooperating* view, where multiple processing units cooperate in solving the problem--leading to what has been named the ``*Parallel Wave*" of computing .

On the other hand, while parallel hardware technology has improved at a high pace, the same cannot be said for the parallel software technology, opening the doors to a feared *(parallel) software crisis*. Mapping the parallelism implicit in applications onto a parallel computer, and coordinating the parallel executions add an extra layer of complexity to the task of programming. This is mainly related to the exponential growth of possible interactions of the different *threads* in which the parallel computation is articulated--which makes the explicit handling of the parallel execution a titanic task.

These considerations justify the effort of developing programming paradigms aimed at making parallel processing an activity whose cost is comparable to that of sequential processing.

**What is Parallelism?**

A strategy for performing large, complex tasks faster.

A large task can either be performed serially, one step following another, or can be decomposed into smaller tasks to be performed simultaneously, i.e., in parallel.

Parallelism is done by:

- Breaking up the task into smaller tasks
- Assigning the smaller tasks to multiple workers to work on simultaneously
- Coordinating the workers

Parallel problem solving is common. Examples: building construction; operating a large organization; automobile manufacturing plant.

Sequential Programming & parallel programming:.

**Sequential Programming**:

Traditionally, programs have been written for serial computers:

- One instruction executed at a time
- Using one processor
- Processing speed dependent on how fast data can move through hardware
    - Speed of Light = 30 cm/nanosecond
    - Limits of Copper Wire = 9 cm/nanosecond
- Fastest machines execute approximately 1 instruction in 9-12 billionths of a second .

**Parallel Computing:**

**Traditional Supercomputers**

Technology

- Single processors were created to be as fast as possible.
- Peak performance was achieved with good memory bandwidth.

Benefits

- Supports sequential programming (Which many people understand)
- 30+ years of compiler and tool development
- I/O is relatively simple

Limitations

- Single high performance processors are extremely expensive
- Significant cooling requirements
- Single processor performance is reaching its asymptotic limit

**Parallel Supercomputers**

Technology

- Applying many smaller cost efficient processors to work on a part of the same task
- Capitalizing on work done in the microprocessor and networking markets

Benefits

- Ability to achieve performance and work on problems impossible with traditional computers.
- Exploit "off the shelf" processors, memory, disks and tape systems.
- Ability to scale to problem.
- Ability to quickly integrate new elements into systems thus capitalizing on improvements made by other markets.
- Commonly much cheaper.

Limitations

- New technology. Programmers need to learn parallel programming approaches.
- Standard sequential codes will not "just run".
- Compilers and tools are often not mature.
- I/O is not as well understood yet.

Parallel computing requires:

- Multiple processors
  (The workers)
- Network
  (Link between workers)
- Environment to create and manage parallel processing
  - Operating System
    (Administrator of the system that knows how to handle multiple workers)
  - Parallel Programming Paradigm
    - Message Passing

        - MPI
        - PVM
    - Data Parallel
        - Fortran 90 / High Performance Fortran
    - Others
        - OpenMP
        - shmem
- A parallel algorithm and a parallel program (The decomposition of the problem into pieces that multiple workers can perform) .

**Parallel Programming**

- Parallel programming involves:
    - Decomposing an algorithm or data into parts
    - Distributing the parts as tasks which are worked on by multiple processors simultaneously
    - Coordinating work and communications of those processors
- Parallel programming considerations:
    - Type of parallel architecture being used
    - Type of processor communications used

**Steps for creating parallel system**

- If you are starting with an existing serial program, debug the serial code completely
- Identify the parts of the program that can be executed concurrently:
    - Requires a thorough understanding of the algorithm
    - Exploit any inherent parallelism which may exist.
    - May require restructuring of the program and/or algorithm. May require an entirely new algorithm.
- Decompose the program:
    - Functional Parallelism
    - Data Parallelism
    - Combination of both
- Code development
    - Code may be influenced/determined by machine architecture
    - Choose a programming paradigm
    - Determine communication
    - Add code to accomplish task control and communications
- Compile, Test, Debug
- Optimization
    - Measure Performance
    - Locate Problem Areas
    - Improve them

# 2. Paradigms for parallel processing

The evolution of parallel processing, even if slow, gave rise to a considerable variety of programming paradigms.

Independently from the specific paradigm considered, in order to execute a program which exploits parallelism, the programming language must supply the means to:

1. *identify parallelism*, by recognizing the components of the program execution that will be (potentially) performed by different processors;
2. *start and stop* parallel executions;
3. *coordinate* the parallel executions (e.g., specify and implement interactions between concurrent components).

It is custom to separate the approaches to parallel processing into *explicit* versus *implicit* parallelism.

## 2.1 Explicit Parallelism:

is characterized by the presence of explicit constructs in the programming language, aimed at describing (to a certain degree of detail) the way in which the parallel computation will take place. A wide range of solutions exists within this framework. One extreme is represented by the ``ancient'' use of basic, low level mechanisms to deal with parallelism--like fork/join primitives, semaphores, etc--eventually added to existing programming languages. Although this allows the highest degree of flexibility (any form of parallel control can be implemented in terms of the basic low level primitives), it leaves the additional layer of complexity completely on the shoulders of the programmer, making his task extremely complicate.

More sophisticated approaches have been proposed, supplying the users with tools for dealing with parallel computations at a higher level of abstraction. This goes from specialized libraries supplying a uniform set of communication primitives to hide the details of the computing environment to sophisticated languages .

Explicit parallelism has various advantages and disadvantages. The main advantage is its considerable flexibility, which allows to code a wide variety of patterns of execution, giving a considerable freedom in the choice of what should be run in parallel and how. On the other hand, the management of the parallelism--a very complex task--is left to the programmer. Activities like detecting the components of the parallel execution and guaranteeing a proper synchronization (e.g., absence of race conditions) can be more or less complex depending on the specific application.

## 2.2 Implicit Parallelism:

allows programmers to write their programs without any concern about the exploitation of parallelism. Exploitation of parallelism is instead *automatically* performed by the compiler and/or the runtime system. In this way the parallelism is *transparent* to the programmer (except, hopefully, for an improvement in execution performance), maintaining the complexity of software development at the same level of standard sequential programming.

The situation is brighter for *declarative languages*. Declarative Programming languages, and in particular *Functional* and *Logic* languages, are characterized by a very high level of abstraction, allowing the programmer to focus on *what* the problem is and leaving implicit many details of *how* the problem should be solved. Declarative languages have opened new doors to automatic exploitation of parallelism. Their focusing on a high level description of the problem and their mathematical nature turned into positive properties for implicit exploitation of parallelism. In particular:

- they are *referentially transparent*, which means that variables are seen as mathematical entities, whose value cannot be changed during execution (single-assignment languages).
- their operational semantics are based on some form of non-determinism (e.g., clause selection in logic languages, apply operator in functional languages);
- the possibility of using *eager* evaluation schemes allows to obtain dataflow-like computations (highly suitable to parallel execution).

Thanks to these reasons, the efforts in parallelizing declarative programming languages have been highly successful. The obvious advantages (automatic parallelization, no additional effort for the programmer) are balanced by some non-straightforward disadvantages, like:

- the system (typically) lacks knowledge of the size of the various components of a computation, and may exploit very fine grained parallelism, leading to slow-downs instead of speed-ups;
- the system may attempt to parallelize code which is only *apparently parallel*, being instead inherently sequential. This will introduce large amounts of synchronization points and lead to inefficient executions.

## 2.3 Ideal System:

The ideal situation would be an approach to parallelization which allows to get the advantages of both the two approaches above. Implicit and explicit parallelism are nothing else than the two extremes of a continuous line, as depicted in figure 1.

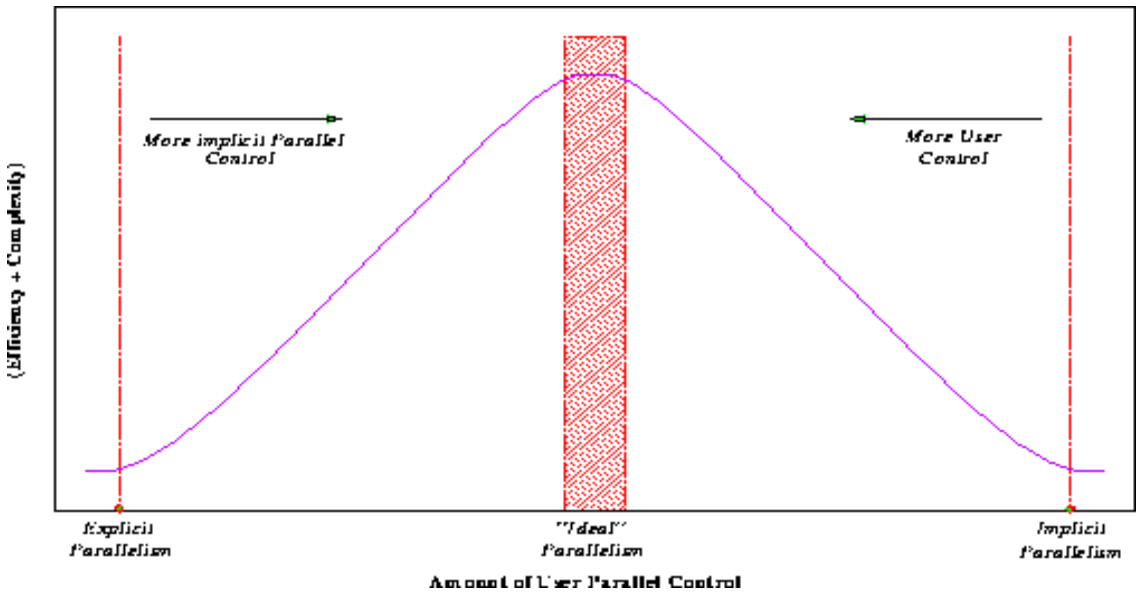



**Figure 1:** Schemes for Parallelism

An ideal system should allow for implicit exploitation of parallelism, but without preventing the user from eventually directing this activity--e.g., by indicating parts that have potential for parallelism or parts that are inherently sequential. The requirement of having potential implicit exploitation of parallelism restricts the choice of programming paradigms mainly to the declarative ones. On the other hand, the requirement of allowing facultative user intervention on parallelism detection does not seem to fit naturally within the semantics of certain classes of declarative languages.

Only few declarative paradigms seem to naturally offer the capabilities and the flexibility indicated--"Pure" Object Oriented Programming ,some forms of Functional Programming ,and *Logic Programming*. Of these, Logic Programming seems to be the most suitable programming paradigm to achieve the ideal parallel behaviour.

# 3. Parallel logic programming

## 3.1 Logic Programming and Prolog:

In the rest of this section we assume all the traditional definitions of mathematical logic .

*Logic Programming* is a well-known programming paradigm based on a subset of First Order Logic--named *Horn Clause Logic*. A program is composed by a set of clauses (i.e., implications) of the form $A \leftarrow B_1, \ldots, B_n$ and execution is driven by a *query* (or *goal*) of the form $\leftarrow B_1, \ldots, B_n$.

Given a program *P* and a query $\leftarrow B_1, \ldots, B_n$, the purpose of an *execution* is to verify under which conditions the conjunction $B_1 \wedge \ldots \wedge B_n$ is a logical consequence of the program *P*--i.e., $P \models B_1 \wedge \ldots \wedge B_n$.

The computation process is based on two basic mechanisms, mainly due to Robinson ,namely *Resolution* and *Unification*. Unification of two atoms *A* and *B* is the process of computing a substitution σsuch that $A^\sigma = B^\sigma$.

During resolution, given a query $\leftarrow B_1, \ldots, B_{i-1}, B_i, B_{i+1}, \ldots, B_n$, a literal $B_i$ is selected, and a clause $H \leftarrow \bar{S}$ such that *H* and $B_i$ unify is used to produce a new query (named *resolvent*)

$$\leftarrow (B_1, \ldots, B_{i-1}, \bar{S}, B_{i+1}, \ldots, B_n)^\sigma$$

where σis the (most general) unifier of *H* and $B_i$.

In general, no restrictions are imposed on the order in which the subgoals and the clauses to be used are selected--i.e., they are non-deterministic operations. In particular:

- since, in absence of failure, each subgoal in the query will be eventually selected, the non-determinism in the subgoal selection is indicated as *don't care* non-determinism;
- since selecting different clauses will possibly lead to different solutions, the non-determinism in the clause selection is indicated as *don't know* non-determinism;

Different languages and different semantics can be obtained by playing on the definition of these two selection operations (i.e. subgoal selection and clause selection).

*Prolog* is the most popular logic programming language. Prolog is based on Horn Clause Logic and its semantics are based on the previously described *resolution* + *unification* mechanisms. Prolog's operational semantics define the two select operations as follows:

- Prolog treats the query as a *list* of atoms and always selects the leftmost atom in the list;
- Prolog treats the program as a sequence of clauses and selects them in the order in which they are stored.

Prolog extends pure Horn clause logic with some additional features, aimed at making Prolog a ``complete'' programming language. The most relevant extensions are:

- **Meta-logical Predicates:** which are used to manipulate terms and to query the state of the proof (e.g., instantiation state of the variables).
- **Cut:** it represents the only non-logical operator in Prolog which affects the *control* of the execution. The cut allows to prune parts of the search space explored by the clause selection process.

- **Extra-logical Predicates:** they are used to perform I/O and to dynamically modify the structure of the program (e.g., dynamically adding new clauses to the program).

The semantics of Prolog are *heavily* based on the ordering adopted by the two selection functions--which is fundamental for the correct behaviour of the various extra-logical features of the language. From now on, whenever we talk about *Prolog semantics* we refer to this ordering of exploration of the search space. Because of this, we will often refer to those extra-logical predicates whose behaviour depends on their order of execution as *order-sensitive predicates*.

## 3.2 Parallelism in Logic Programming:

Logic Programming offers some unbeaten opportunities for implicit exploitation of parallelism. This is quite evident from the presentation of its operational semantics in the previous section. The resolution algorithm offers various degrees of non-determinacy, i.e. points of the execution where different alternatives are available to continue the computation. In any sequential implementation, these choices will be serialized--i.e. explored in some order (depending on the definition of the various selection operations). On the other hand, it is feasible to consider the possibility of exploring different alternatives concurrently, i.e., in parallel.

Furthermore, logic programming satisfies the second desired requirement--allowing natural ways to intervine on the exploitation of parallelism, e.g. through user annotations which do not ``disturb'' the structure of the program.

Various forms of parallelism can be exploited from logic programming languages, depending on which selection operations are transformed into ``parallel'' operations. It is custom to identify the following major forms of parallelism :

1. **Or-parallelism:** Or-parallelism emerges from executing in parallel the different clauses that can be used to solve the selected subgoal. Intuitively, each subgoal can be solved by using different clauses (all those whose head unifies with the subgoal), and each of them gives rise to an alternative execution path. This can be depicted as a tree (the *Or-tree*), having a node for each selected subgoal and a branch for each clause matching the subgoal. Figure 2 illustrates an example of Or-tree (on the right).

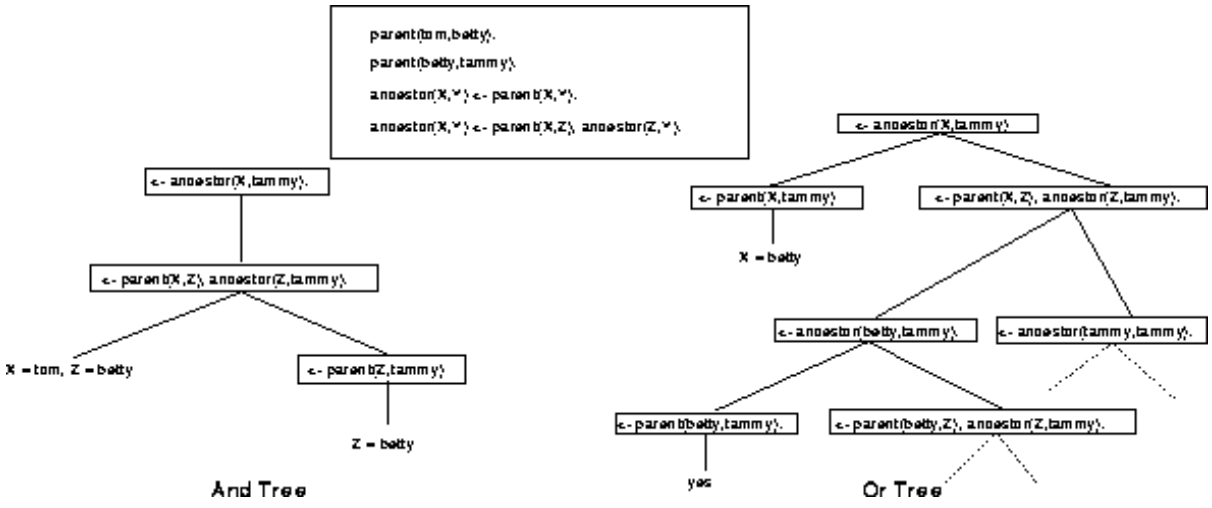



**Figure 2:** And-tree and Or-tree

In a sequential system the tree is visited in a predetermined order (e.g., in Prolog a left-to-right, depth-first search is used). Whenever a leaf of the tree is reached (a solution or a failure point) and a new solution is desired, a node with unexplored branches is selected and the new branch is traversed. This process is called *backtracking*. Nodes in an or-tree are usually named *choice-points*.

In an or-parallel system different computing agents are concurrently exploring different branches of the or-tree.

2. **And-parallelism:** a dual view of a computation can be obtained by considering the non-determinism present in the subgoal selection operation. This can be described using a tree, the *And-tree*: each node represents (part of) a query and a node has a successor for each subgoal present in the query associated to such node. Figure 2 (on the left) illustrates an example of and-tree.

   In a sequential execution subgoals are selected one at the time from the current query--and the computation will stop when the query is empty or when a failure is encountered. This is equivalent to a traversal of the and-tree performed following a predefined order. And-parallelism is obtained by allowing the concurrent resolution of different subgoals of a given query--i.e., allowing different computing agents to explore different parts of the and-tree.

Other forms of parallelism (e.g. unification parallelism) have been identified but, due to their fine granularity, they require specialized architectures in order to achieve results of any interest. We will not be concerned with them in the rest of this work.

## 3.3 Limits and Perspectives:

Models exploiting the forms of parallelism in logic programming described before have been developed and successfully implemented. Nevertheless various issues are still open and object of active research.

**Efficiency:** although the literature counts a large number of different proposals for exploiting parallelism from logic programming languages, only few of them have a practical meaning. This is related to the fact that a practical parallel system should be *efficient*, where by efficiency we intend the ability of *(i)* containing the amount of

overhead introduced to extract parallelism; and *(ii)* achieving a sequential efficiency comparable to that of the state-of-the-art in sequential implementations.

The typical requirement that we would like to put on a parallel implementation model is the *no-slowdown requirement*: this means that in no conditions the parallel execution should be slower than a corresponding sequential one. In practice the no-slowdown requirement can be hard to meet; exploitation of parallelism may introduce overheads that in certain situations can actually slowdown the computation, and certain operations may become more expensive in presence of parallelism (this is often the case for backtracking). Nevertheless, even if the no-slowdown cannot be completely achieved, we should still aim at its best possible approximation, limiting the cases in which slowdown appears and maintaining a good sequential efficiency. Designing efficient parallel logic programming systems is a considerably complex task that is still partly unresolved.

**Combinations:** although efficient implementations of logic programming languages exploiting a single form of parallelism are available , systems concurrently exploiting different forms of parallelism still elude us. Systems proposing ways of combining different forms of parallelism have been proposed, but they suffer various drawbacks, like:

- they are based on implementation schemes which lose the sequential efficiency available in current implementations--leading to inefficient implementations. One of the requirements of a parallel system, as already mentioned before, should be the ability to maintain a sequential efficiency comparable to that of the state-of-the-art sequential systems.
- they are limiting the amount of parallelism exploited;
- they are modifying the semantics of the language, in order to make exploitation of parallelism easier .

**Optimizations:** even models based on standard sequential implementation techniques (like the *Warren Abstract Machine (WAM)* may suffer considerable overheads due to the support of parallel execution. On the other hand the support for parallelism is often designed in order to tackle the *worst case* situations, which may seldomly occur. A wide lack of optimization principles which can be applied to tailor the cost of the exploitation of parallelism to the actual complexity of each specific instance of the problem is behind the poor performance of many implementations.

**Integration:** logic programming has rapidly evolved in the last years, bringing the attention on new computational models capable of solving in a more expressive way and more efficiently various categories of problems.

# 4. Parallel logic programming

In the previous section we have introduced the basic concepts of logic programming and we have intuitively justified the reasons that make logic programming languages extremely appealing in terms of parallel execution.

In this section we propose a more detailed description of the major forms of parallelism exploitable from logic programming languages, analyzing the major problems that need to be solved to obtain efficient implementations.

As mentioned previously, two major forms of parallelism are commonly identified in logic programming, or-parallelism and and-parallelism. Exploitation of these two forms of parallelism requires efficient solutions to different problems, related mainly to the management of the *control* of the computation (for and-parallelism) and the management of *environments* (for or-parallelism).

Before entering in the details of these issues, a general overview of the principles on which the most common sequential implementations of logic programming are based is due, in order to make the presentation more self-contained.

## 4.1 Sequential Implementation of Logic Programming:

The use of logic as a ``programming'' language, or at least as a meaning to mechanically express and manipulate reasoning, has been implicit since its creation .Nevertheless, it was not until the creation of Prolog, thanks to the brilliant intuitions of Colmeraur and Kowalski , that ``logic'' became ``logic programming''. Key issue in this evolution step is the understanding that a subset of first order logic, namely Horn Clause Logic, could be sufficiently expressive for practical programming purposes and still efficiently implementable.

Even though reasonably fast interpreters for Prolog, like the *C-Prolog* realized by Pereira, Damas, and Byrd had been developed, a myth of ``logic programming = slow implementations'' developed and still survives. Only with the advent of Prolog compilers this myth started declining and at present extremely fast implementations are available--capable on some specific programs of beating state-of-the-art C compilers .

The real breakthrough is represented by the introduction of the *Warren Abstract Machine (WAM)* ,which has become a standard de-facto for the implementation of Prolog and Logic Programming languages.

The WAM defines an abstract architecture whose instruction set allows an easy mapping from Prolog source code and to it is sufficiently low-level to allow an efficient emulation and/or translation to native machine code.

The WAM is a stack-based architecture, sharing some similarities with imperative languages implementation schemes (e.g., use of call/return instructions, use of frames for maintaining procedure's local environment), but extended in order to support the features peculiar to Logic Programming, namely *unification* and *backtracking*. The major data areas of the WAM are (as shown in figure 3):

- *Heap:* it is used to store the complex data structures (lists and Prolog's compound terms) created during the execution.
- *Local Stack:* it serves the same purpose of the control stack in the implementation of imperative languages--it contains control frames, called *environments*, which are created upon entering a new clause (i.e., a new ``procedure'') and are used to store the local variables of the clause and the control information required for ``returning'' from the call.
- *Choice Point Stack:* choice points encapsulate the execution state for backtracking purposes. A choice point is created whenever a call having

multiple possible solution paths (i.e., more than one clause successfully match the call) is encountered. Each choice point should contain sufficient information to restore the status of the execution at the time of creation of the choice point, and should keep track of the remaining unexplored alternatives.

- *Trail Stack:* during an execution variables can be instantiated. Nevertheless, during backtracking these bindings need to be undone (to restore the previous state of execution). In order to make this possible, bindings that can be subject to this operation are registered in the trail stack. Each choice point records the point of the trail where the undoing activity needs to stop.
- *Code Area:* it is used to store the compiled code of the program.

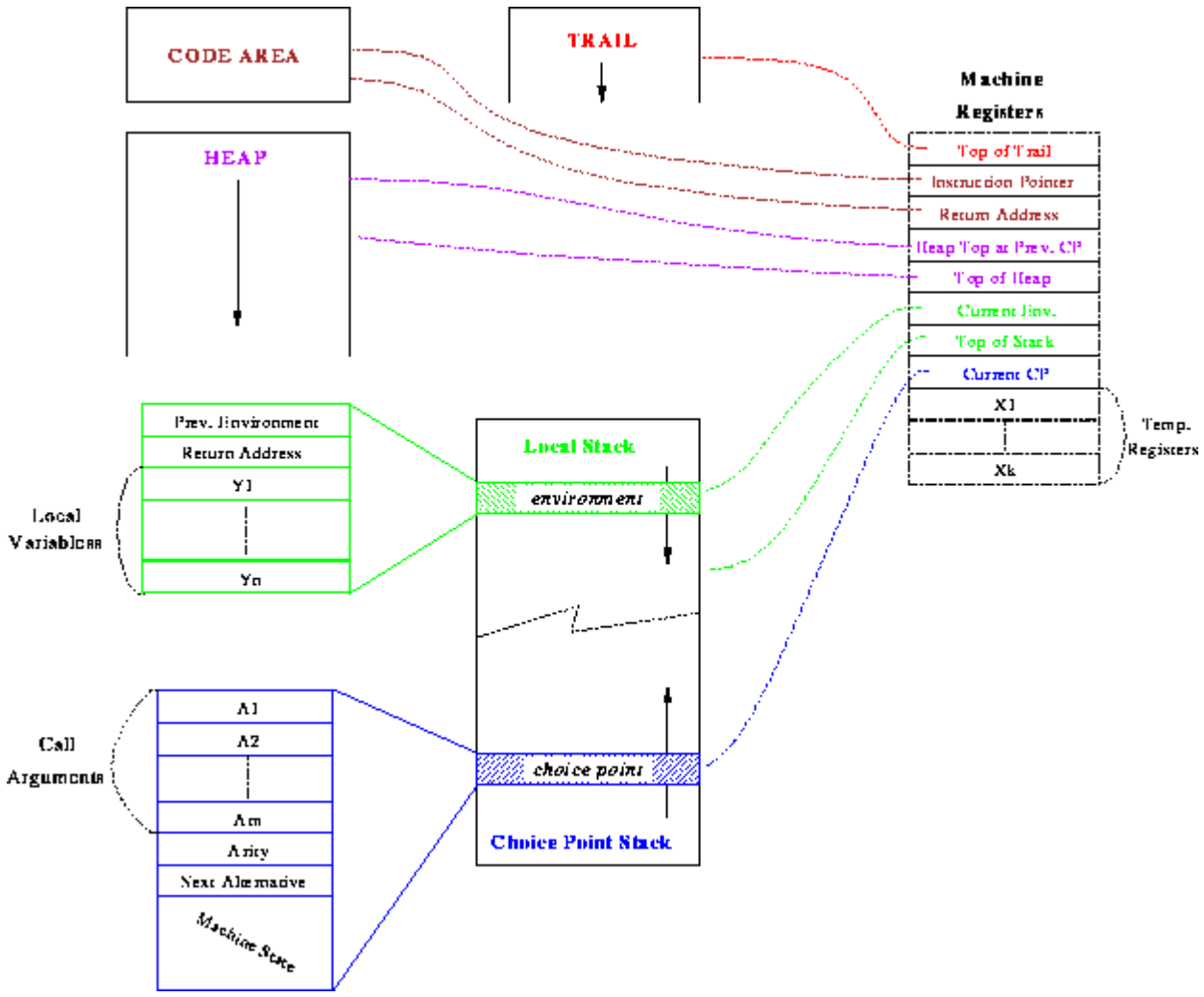



**Figure 3:** Organization of the WAM

Being a dynamically typed language, Prolog requires a type information to be associated with each data object. In the WAM, Prolog terms are represented as *tagged words*. Each word is composed by a tag, identifying the type of the object (e.g. atom, list, etc.), and by a value (e.g. atom name, pointer to the first molecule of a list, etc.).

### 4.1.1 Beating the Interpretation Blues:

The use of WAM as a target language for compilation of Prolog, and the direct execution of WAM code (using an emulator) can lead to interesting performance results--but still leaving a margin of difference w.r.t. the performance of languages based on different programming paradigms (e.g., *C*). This is mainly due to the existence of an intermediate interpretation step (execution of WAM code), which translates to overhead during execution.

Two main approaches have been followed trying to tackle this issue:

1. hardware solution;
2. compilation of WAM code.

#### 4.1.1.1 Hardware Solution:

The hardware solution consists of actually building specialized hardware capable of supporting WAM (or a similar low-level language). A considerable number of proposals have been made in the past for specialized hardware dedicated to support execution of logic programming languages.

The interest in hardware support for Prolog exploded mainly during the 1980's, and found a considerable source of inspiration in the work by Tick and Warren ,where the design of a microcoded WAM is presented.

The experience of developing ad hoc hardware for logic programming languages, as happened for functional languages, has been only partially satisfactory. The experience of using sophisticate compiling techniques, and the high pace at which RISC technology is improving general-purpose processors speed, proved that there is nothing inherent in the Prolog language that prevents from executing it with speed at least comparable with that of imperative languages--without the need of explicit hardware support. This has emerged in the experience of Parma and Aquarius (by using processors with the same clock-rate, Parma running on a general-purpose processor is achieving a greater performance than VLSI-BAM--the specialized hardware designed for Aquarius).

In recent years the interest for special-purpose architectures has progressively died, and the work on hardware support for Prolog has been mainly redirected towards the design of the next generation of general purpose processors.

#### 4.1.1.2 Native Code Compilation:

The reasonable other alternative in order to avoid the overhead of emulating the WAM code is to generate directly machine language code from the compilation of Prolog programs. This is what usually goes under the name of *Native Code Compilation*, a feature that is quickly becoming part of the most widely used implementations of Prolog (e.g., Quintus and SICStus).

The two more sophisticated compilers performing native code compilation are Aquarius and Parma . Both the compilers translates Prolog to an intermediate code--expressed in a language whose level is considerably lower than WAM. In the case of

Parma the intermediate code is expressed in a quite standard 3-address code; in the case of Aquarius the intermediate code (BAM code) is based on a simple language (based on load-store instructions, tagged addressing modes, pragmas, and some Prolog-specific instructions to manipulate unification, trail, dereferencing, environments, and choice-points).

In both cases the intermediate code is translated into machine code (MIPS for Parma, SPARC for Aquarius). Native code compilation has been proven to be most effective way to produce efficient executions of Prolog, especially if coupled with the use of sophisticate static analysis techniques.

### 4.1.2 Other Intermediate Languages:

In all the implementation schemes considered before, compilation goes through an independent intermediate code (WAM or similar), which is eventually translated into machine code. Nevertheless, this approach has some major drawbacks:

- *portability:* since the compiler is producing native code, this makes the back-end completely dependent on the specific machine (and hardly portable);
- *complexity:* native code generation is a very complex task, which requires a deep knowledge of the target architecture;
- *efficiency:* the complexity of certain operations (e.g., register allocation, peephole optimization) may endanger the efficiency of the translation process.

It would be appealing to have a big part of this work done by somebody else. Compiling to C (instead using a specialized intermediate language) and using a standard C compiler as back-end, represents a solution to these problems. The reason for choosing C instead of other programming languages is mainly related to

its wide availability on any platform, its high efficiency and level of sophistication of the compilers, and its ability to take advantage of low-level machine features.

Nevertheless, a straightforward compilation of Prolog to C (e.g., converting Prolog procedures to C procedures, etc.) would produce inefficient executions, due to the deep difference between the nature of the control structures available in the two languages (e.g., iteration in Prolog can be expressed only using recursion). The design of a compiler to C requires developing very smart solutions. Various issues need to be tackled when dealing with Prolog-to-C compilation. The most important is the selection of a proper mapping between Prolog procedures and C functions.

Various logic programming languages are currently compiled to C; this approach has been proven to be extremely effective to implement committed-choice languages--which is quite intuitive since supporting backtracking in a Prolog-to-C translation requires a considerable amount of effort.

## 5. Classification of Parallelism

In the previous section we briefly introduced the basic forms of parallelism commonly identified in Logic Programming languages. In order to better understand the problems connected with the exploitation of parallelism in Logic Programming, it is important to introduce an additional level of classification. The parameter in this classification is simply represented by the *level of interaction* existing between parallel threads of computation .This classification is independent from the specific form of parallelism exploited (i.e., either or- or and-parallelism). In order to make the description uniform, we generically talk about *threads* of computation to indicate potentially parallel pieces of computation (i.e., parallel subgoals in and-parallelism and alternatives from a parallel choice point in or-parallelism).

We will also use the generic term *dependency* between threads to indicate the *potential* ability of one thread of affecting the execution of the other threads. For example, an unbound variable accessible by different threads may represent a dependency.

Using this parameter, we can develop the following classification of parallelism:

- ***Independent Parallelism:*** it is characterized by the fact that the parallel execution is allowed only between threads which are *independent* from each others. This level of parallelism is clearly the simplest to implement, since the parallel execution requires little or no synchronization. The only mechanisms that are required are those to generate the parallel work (e.g., *fork* of parallel computations) and, eventually, a final barrier to detect termination of all the threads.

  Independent Parallelism is the weakest form of parallelism--nevertheless, it has been shown in many situations that independent parallelism can be

sufficient to produce considerable speed-ups in many real-life applications . Furthermore, the relative simplicity of the mechanisms required to support independent parallelism guarantees a limited overhead, which translates in very *efficient* exploitation of parallelism.

- ***Restricted Parallelism:*** parallel execution of threads having dependencies is allowed as long as there is guarantee that no *solvable* conflicts will take place during the execution. In simpler words, the dependencies that are allowed are such that no conflict will occur during the computation (i.e., the different threads always agree on the dependency) or, if a conflict occurs, then a failure of the whole computation should take place. Implementation of restricted parallelism is more complex, the presence of possible dependencies requires certain care, like the use of mutual exclusion mechanisms to avoid race conditions. Nevertheless, the guarantee that no solvable conflicts will occur allows to avoid the introduction of any mechanism to order the accesses to the sources of dependencies--i.e., there is no don't know non-determinism associated to the presence of dependencies.

  This level of parallelism (which strictly subsumes independent and-parallelism) allows to obtain speedups on a larger class of programs, hopefully maintaining a satisfactory degree of efficiency.

- ***Dependent Parallelism:*** in this class we collect all those models in which parallel execution of dependent threads is allowed. The presence of dependencies requires a careful design. The main issue to be tackled is to guarantee respect of the semantics of the language. The purpose of our research is to obtain transparent exploitation of parallelism: in this terms the user should be guaranteed that the outcome of its execution on a parallel machine is exactly identical to the outcome produced from a sequential

execution (modulo variations in execution time). Arbitrary execution of parallel threads in presence of dependencies may cause a different behaviour in the execution, leading to a lost of the original semantics of the computation. To avoid this the system needs to enforce a certain amount of control on the order in which the threads are accessing and executing operations related to the dependencies--i.e., reintroducing a slight amount of sequentiality, in the form of synchronization points between threads.

Implementing dependent parallel system is considerably more complex, because of the problems just mentioned. Nevertheless, only the systems belonging to this class guarantee the possibility of extracting parallelism from almost any kind of application.

The classification above applies to both or- and and-parallel systems, and in the rest of this section we will see how the classes above instantiate in the two cases (for or- and and-parallelism) and for their combinations.

## 6. Single Parallelism Systems

In this section we analyze the main problems and solutions in the design and implementation of parallel logic programming systems exploiting a single form of parallelism.

### 6.1 Or-parallelism:

As described in the introduction, or-parallelism arises whenever a subgoal can unify with the heads of more than one clause, and the bodies of these clauses are concurrently executed by different computing agents--that, from now on, will be referred to as *or-agents*.

Or-parallelism is present in almost any application, although certain application areas seem to offer it on a larger scale. This is the case of application areas like parsing ,optimization problems ,databases ,and others .

**Example 2.2** *An example of or-parallelism, from our* **integr** *problem is the following: let u*
*two rules*

1. **integr(X + Y, X' + Y')** ← **integr(X,X'),integr(Y,Y')**
   *expressing the identity* $\int g_1 + g_2\,dx = \int g_1\,dx + \int g_2\,dx$.
2. **integr(A + B, X × Y)** ← **A = X₁ × Y, B = X × Y₁, integr(X,X₁), integr(Y,Y**
   *expressing the identity* $\int (g_1 \times \frac{dg_2}{dx} + \frac{dg_1}{dx} \times g_2)dx = g_1 \times g_2$.

*Given a query like* ← **integr(5×x + ln x × x,Z)**, *both the clauses can actually be applied, successfully match the query. This query is a potential source for or-parallelism.*

It should be noted that, by taking different alternatives from a non-deterministic choice point, each or-agent is actually computing a *distinct* solution to the original query. This has often lead to the intuition that or-parallelism, at least in principle, could be easily implemented with straightforward modifications to existing sequential technology.

This is only true if we restrict our attention to very simple forms of parallelism (i.e., independent or restricted). Implementing or-parallelism in its most general form offers few complex challenges.

In particular, it is important to observe that the *independence* of the alternatives explored by the different or-agents exists only at the theoretical level. This can be understood from the following example: let us consider the query ←q(Y,X), p(Y,X) and let us assume that

- the computation of q succeeds without binding the variable X and binding the variable Y to a;
- p is defined by the two facts p(a,1) and p(a,2).

The Or-tree for this part of computation is represented in figure 4(i). On one hand the binding for the variable Y (Y->a) should be visible to both the alternative computations of p. On the other hand, such computations will produce different bindings for the variable X. This introduces a clear problem in the design of an implementation model - the standard approach in managing the environments is not anymore suitable to our needs. In a sequential system a single environment would be allocated for the query, containing a location for the variable X; since the two alternative computations are explored sequentially, they can easily reuse the same environment. In an or-parallel system, the two alternatives are concurrently explored, and they cannot share the same environment (and, in particular, the same location for X), as illustrated in figure 4(ii). The access, within parallel alternatives, to environments created *dependency* between parallel threads. Variables belonging to this kind of environments are usually named *conditional variables* (since their bindings may vary across different alternatives). The problem can be tackled in different ways, either imposing restrictions on the cases where parallelism is allowed, or by devising new environment representation schemes.

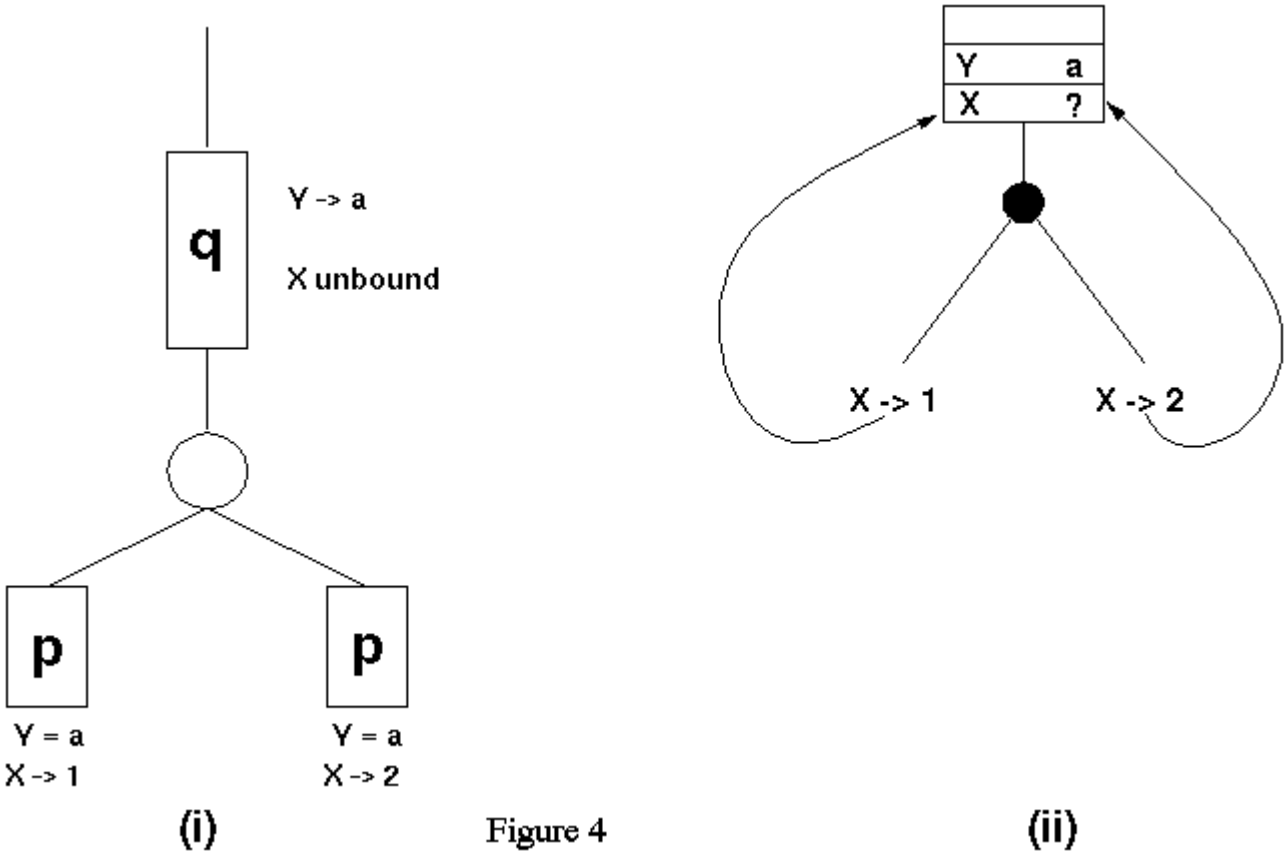



Figure 4

### 6.1.1 Independent Or-Parallelism:

this fashion of or-parallelism arises whenever the different alternatives in a parallel choice-point are independent of each others--i.e., no two alternatives belonging to a choice-point access the same conditional variables. The most common instance of independent or-parallelism occurs when the query which originates or-parallelism is *ground*, i.e. it does not contain any unbound variable.

Independent or-parallelism has the advantage of *not* requiring any change to the environment representation scheme adopted in standard sequential implementations. The amount of effort required to design a system exploiting independent or-parallelism is minimal, and guarantees a very high level of efficiency (overhead is

limited). On the other hand, its applicability is quite limited (typical example queries with all ground arguments).

Another example of independent or-parallelism comes from the area of deductive databases. One of the most commonly used approaches to parallelization of Datalog programs is based on the concept of *Decomposability* .The notion can be synthesized as follows: given the set of all possible *derivable* ground atoms $B_H$ and the Extensional database *E*, the program is *decomposable* if there exists a partition $\{M_1, \ldots, M_n\}$ of $B_H$ such that, if $g \in M_i$ and $\leftarrow g$ can be proved from the program, then the derivation tree of *g* contains only atoms from $M_i \cup E$. This notion allows partitioning of the database between different processors, with the guarantee that the computation of a query can be carried on in a processor without the need of any communication. This form of parallelism can be seen as another instance of the independent or-parallel model.

### 6.1.2 Restricted Or-Parallelism:

extends the notion of independent or-parallelism, by allowing parallel execution of alternatives accessing unbound conditional variables. The restriction imposed is that no conflicts should occur on such variables. This translates to the fact that either: *(i)* the conditional variables are read and not written; or *(ii)* conditional variables instantiated along one alternative are not actually used by the other alternatives (i.e. the dependency is only apparent).

The concept of restricted or-parallelism could be pushed to the limit, allowing parallel execution of alternatives capable of binding the same conditional variables, as long as *(i)* the bindings are *consistent*, that is the same value is assigned to the conditional variable across the different alternatives; *(ii)* the execution of the alternatives is time-insensitive with respect to the bindings to the conditional variables.

Figure 5 illustrates an example of restricted or-parallelism. It is important to observe that

- the binding to X does not affect the computation of the first alternative;
- the binding to Y is deterministic (i.e., both the alternative assign the same value) and it is not time-sensitive (i.e., the computation will not change depending on which of the two alternatives binds Y first).

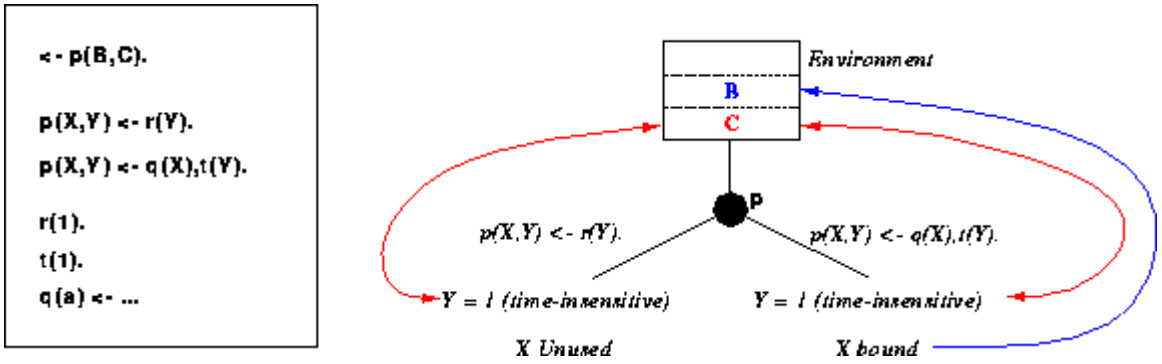



**Figure 5:** Restricted Or-Parallelism

At the implementation level, little more than independent or-parallelism is required to support restricted or-parallelism. The only relevant addition is related to the possibility of concurrent accesses to the same objects: accesses to conditional variables should be treated as critical sections, guaranteeing mutual exclusion.

The notion of restricted or-parallelism, to our knowledge, has not be practically explored in any system.

### 6.1.3 Dependent Or-Parallelism:

this represents the most general form of or-parallelism, in which no restrictions are imposed a priori on the exploitation of parallelism.

This general form of or-parallelism requires, as described before, the development of new environment representation techniques, in order to tackle the dependency problem in the access and binding of conditional variables. During the execution of a resolution step, new variables may be created (the variables local to the clause used in the resolution step). Later on, the computation may split, due to the presence of or-parallelism, and the different executions may clash in trying to read and/or write the variables created in the ``shared'' part of the computation.

**Example 2.3** *Considering again the clauses of example 2.2, given a query like*

$$\leftarrow \mathbf{integr}(5 \times x + \ln x \times x, \mathbf{Z})$$

*an or-parallel execution will attempt to bind differently variable* **Z** *(although at the end only one of the computations will be successful).*

The environments should be organized in such a way that, in spite of the fact that a part of the computation is ``shared'' (the part before the choice-point), the correct bindings applicable to each branch are easily identifiable. And this representation should be efficient, it should not add unbearable overheads.

The problem has to be solved by devising a mechanism where each branch of the or-tree has some private area, used to store the bindings to the conditional variables. A variety of schemes have been proposed to tackle this problem  present a thorough comparison of more than 20 models). They differ in the way in which creation and access to these private areas are organized.

Of the many approaches proposed, two have been proved to be superior in terms of efficiency and flexibility.

> **Stack-copying:** the environment representation problem is solved by performing copies of the ``shared'' environments. Each branch of the or-tree owns a private copy of all the environments (relevant to such branch), avoiding in such way any conflict in binding conditional variables.

Furthermore, with this technique the computation carried on by each or-agent resembles exactly a standard sequential execution. Figure 6(ii) illustrates this approach in the case of the query of example 2.3. The first processor (on the left) generates a choice point with unexplored alternatives; the second processor creates a local copy of the computation up to such choice point (in particular, as illustrated in the figure, the environments are duplicated) and continues with one of the unexplored alternatives.

**Binding Arrays:** in this scheme, an index is associated to each conditional variable, and each branch has an associated array where bindings to conditional variables will be stored. Whenever a conditional variable is accessed, its index is used to identify an entry in the local binding array, where the binding will be retrieved and/or stored. Whenever an or-agent starts executing an alternative from a choice-point, it must update its binding array in order to reflect the current state of the conditional variables. An example of use of binding arrays, on the usual query of example 2.3, is shown in figure 6(i). The variable Z is conditional and it is translated at its creation time in a reference to entry 0 of the binding arrays. Each processor will create the binding for Z in the entry 0 of its local binding array. This technique has been originally developed in the Aurora system ,and successively adopted by other systems .

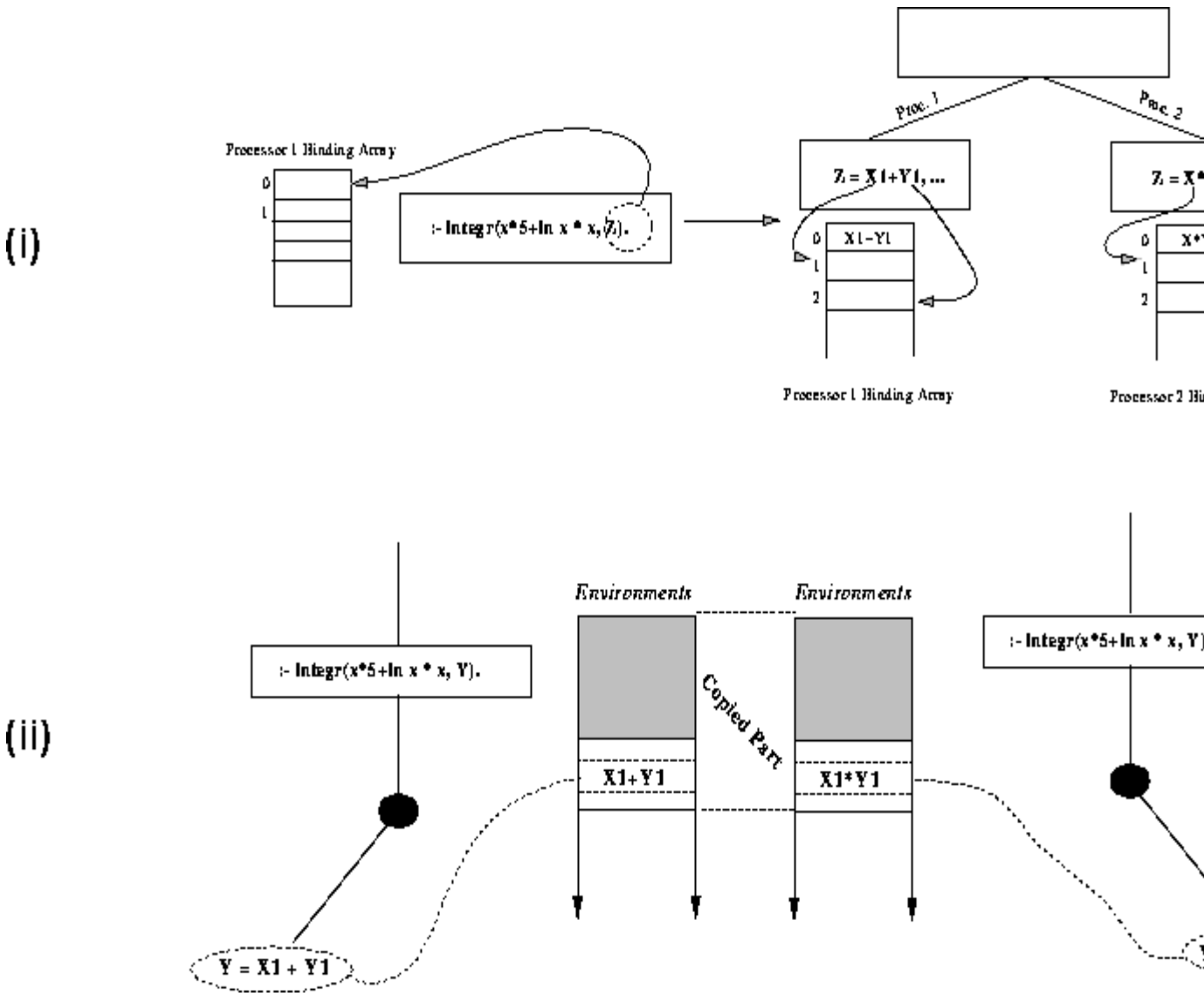



**Figure 6:** Stack-copying and Binding Arrays

## 6.2 And-parallelism:

And-parallelism arises whenever subgoals belonging to a given query are concurrently solved by different computing agents. Agents exploiting and-parallelism are named *and-agents*.

And-parallelism is probably the most promising form of parallelism. Every application contains a reasonable amount of it. The same is not true for or-parallelism (since programmers are often tempted to write determinate programs, in

which the amount of don't know non-determinism is extremely limited). Also for and-parallelism it is useful to distinguish different levels of exploitation of parallelism, depending on the degree of dependence allowed between parallel threads. In the case of and-parallelism, dependencies are originated by the presence of variables whose instantiation state can be modified by different concurrent threads of execution. In this way, the execution of a subgoal can affect the execution of others by binding one of these *shared variables* (also known as *dependent variables*). Furthermore, it is important to observe that the different and-agents are computing different parts of the same solution to the original query. For this reason it is necessary to create a sufficient amount of *sharing* between the different and-agents, in order to allow an efficient combination of the partial solutions computed by each of them.

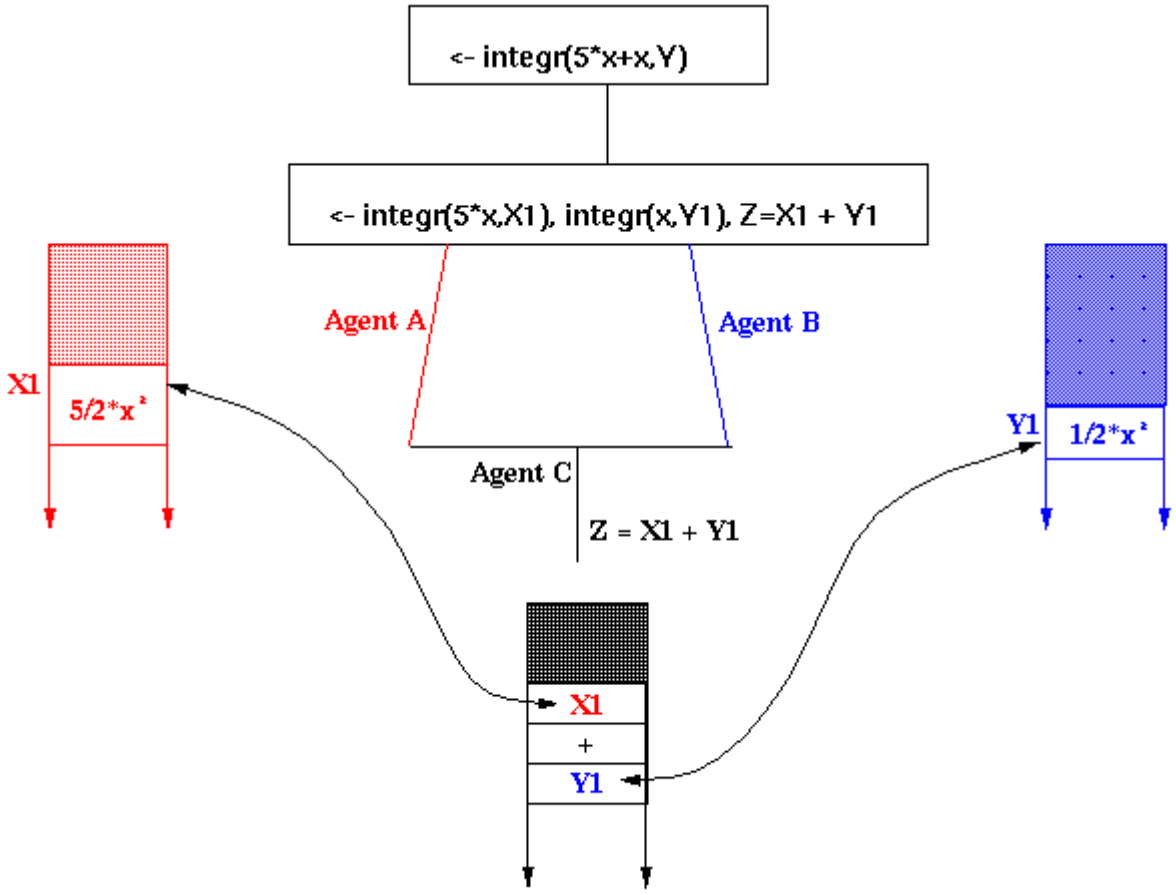



**Figure 7:** Need for Sharing of Data Structures

**Example 2.5** *Considering the example 2.4, figure 7 shows a possible pattern of and-parallel e execution carried on by the agents A and B will cause the allocation of new terms on their lo terms representing the values assigned to* $X_1$ *and* $Y_1$*). The continuation of the parallel call (i.e., of* $Z = X_1 + Y_1$*) will require to have free access to the heap of both the agents A and B in or the two terms. Agent C will require access to A's and B's heap.*

### 6.2.1 Independent And-parallelism:

this form of parallelism has been studied by various researchers and different implementations have been designed and developed. This wide interest in independent and-parallelism is mainly related to its apparent simplicity and the fact that many applications are naturally rich of independent and-parallelism (or they can be easily modified to expose it).

Independent and-parallelism is characterized by the fact that only subgoals which do not share any unbound variable at run-time are entitled to parallel execution. The condition guarantees that they will not affect each other's execution--removing the need of introducing any form of synchronization during parallel execution. The only synchronization activity required is a *barrier* placed at the end of the parallel part of the execution, needed to switch back to a sequential execution once all the parallel subgoals are completed.

Various steps are required in the exploitation of independent and-parallelism:

1. *Detection of Independence:* since only independent subgoals are allowed for and-parallel execution, some mechanisms are needed to allow detection of independence. Independency information may be either supplied by the user, detected through compile-time analysis, or detected at run-time.

   Granted that allowing the user to supply information about the subgoals can only improve the execution, the most attractive solution is based on a

compromise between compile-time analysis and run-time analysis--compromise which is necessary, since

- pure run-time analysis guarantees the best results but adds an unberable overhead to the execution;
- pure compile-time analysis avoids run-time overhead but is forced to make very conservative assumptions (with consequent loss of parallelism).

The solution, originally proposed by DeGroot and successively elaborated by Hermenegildo and others ,consists of generating at compile-time some *conditional annotations* (named *Conditional Graph Expressions (CGE)*), which identify potential independent subgoals and the conditions under which the independence is guaranteed. A CGE has the form ( $\langle conditions \rangle \Rightarrow$ G 1& . . .& G n), where $\langle conditions \rangle$ are some simple tests on the instantiation status of some variables, and G 1, . . ., G n are the potentially independent subgoals. At run-time, whenever a CGE is encountered, the $\langle conditions \rangle$ are evaluated and, if the evaluation is successful, the subgoals G 1, . . ., G n are executed in parallel. The symbol ``&" is used to denote a parallel conjunction (i.e., the subgoals connected by & can be run in parallel), while ``," will be maintained to represent sequential conjunction.

2. *Support for And-parallelism:* once a set of independent subgoals has been identified, the subgoals need to be prepared for parallel execution and the and-agents should be notified of the availability of parallel work;
3. *Distributed Backtracking:* one of the biggest challenges in the development of any and-parallel system is the design of a sound, complete, and efficient backtracking mechanism. The main difficulty is represented by the arbitrary spreading of the computation between the different agents, with the

possibility that more than one agent may be concurrently backtracking on the same part of computation. Different approaches to backtracking in this kind of framework have been proposed ,although, to our knowledge, only one full-blown implementation has been realized, within the ACE project ,

Systems exploiting independent and-parallelism have been successfully implemented; the most relevant proposals are &-Prolog ,APEX, and &ACE .

### 6.2.2 Restricted And-parallelism:

restricted and-parallelism emerges from allowing and-parallel execution of subgoals which share unbound variables, as long as there is guarantee that no solvable conflicts can emerge on binding such shared variables. Two main approaches have been proposed in this area:

***Non-strict Independence:***

proposed by Hermenegildo ,consists of allowing parallel execution of subgoals sharing variables, as long as the different computations do not ``compete" for their bindings (e.g., at most one computation will attempt to bind each of these variables).

This approach has the advantage of not requiring any additional run-time mechanism (only compile-time analysis needs to be improved to detect these cases). On the other hand it is still an open question whether this generalization does really improve the applicability of and-parallelism.

***Basic Andorra Model:***

developed by D.H.D. Warren and integrated in the Andorra-I system ,this model allows exploitation of and-parallelism between subgoals which are *determinate*, i.e., subgoals that will produce at most one solution. The different subgoals may have any kind of dependency but, being determinate, no solvable conflict may occur--if a conflict on binding a variable occurs,

then we are guaranteed that the execution does not have any solution. Results for the Andorra-I system have been proposed and they show considerable speed-ups. On the other hand various issues need to be carefully tackled, like detection of determinacy and management of subgoals. In particular, the approach taken by Andorra-I, where parallel execution may actually change the order in which subgoals are explored (determinate subgoals may be executed ahead of time, independently from their position in the query), requires very complex run-time mechanisms (backtrackable goal chains), and their management seems to have a noticeable negative effect on the overall performance.

**Example 2.6** *Given a program containing the facts* **p(1)**, **q(1)**, **r(1)**, *and* **r(2)** *Andorra-I will start by executing in and-parallel the deterministic subgoals*—**p(X)** *a compatible binding for* **X**, *allowing the execution of* **r(1)** *to start.*

### 6.2.3 Dependent And-parallelism:

this occurs when and-parallelism is allowed between subgoals which share unbound variables, even in presence of possible binding conflicts (i.e., different computations try to bind differently the same variables).

**Example 2.7** *Referring to the usual integration problem, a clause to express the* **integration** *stating*

$$\int f(x)\frac{dg(x)}{dx}dx = f(x)g(x) - \int g(x)\frac{df(x)}{dx}dx$$

*will be expressed as follows:*

**integr(X × Y,Z) ← integr(Y,Y1), differ(X,X1), integr(Y1 × X1,Z1), Z = X ×**
*where* **differ** *is a predicate computing the derivative of its first argument. The and-paralle the subgoals in the body of this clause will give rise to dependent and-parallelism, since the subgoals* **integr(Y,Y1)** *and* **differ(X,X1)** *will be accessed by the concurrently executing subg* **× Y1,Z1).**

Only few proposals have been made in the literature to tackle this class of parallelism, and this is mainly related to the complexity of the problem. In particular,

very few approaches have been described which are capable of maintaining Prolog semantics, while the large majority of the others introduce new languages with different semantics, in which exploitation of parallelism becomes simpler and more efficient.

***Prolog Implementations:***

in order to support Prolog semantics, a model for dependent and-parallelism needs to guarantee that the bindings to variables are produced in the *correct order* (i.e., in sequential Prolog order). This can be seen in the following example. Let's consider the program containing the three facts p(1), q(2), and q(1), and the query <- p(X) & q(X). In a sequential execution p(X) is executed first, producing a binding $\{X \mapsto 1\}$, and successively the subgoal q(1) will be completed. On the other hand, if q(X) is executed first, then it will produce a binding $\{X \mapsto 2\}$ and successively the subgoal p(2) will fail, leading to an unsuccessful execution. To tackle this problem within and-parallel systems, the notions of *producer* and *consumer* of a variable have been introduced. At every step of the execution, for each shared variable we identify exactly one producer (i.e., the subgoal which has the ``right'' to bind the variable). All the other subgoals accessing the shared variable are only allowed to read the value--i.e., they are consumers. A consumer accessing an unbound variable will suspend its execution (since it is not allowed to bind it), waiting either for a binding to be produced or for its turn to become itself a producer for such variable. The producer and consumer status is determined according to Prolog's operational semantics. Thus, the producer for a variable is the leftmost subgoal in the goal chain accessing such variable. All the other goals containing the same variable are designated as consumers. A consumer becomes a producer for a variable when the designated producer finishes execution without binding the shared variable

and the consumer in question is the leftmost among all potential consumers of the shared variable. The (few) models presented in the literature dealing with dependent and-parallelism (of which DDAS is the most relevant) can be simply distinguished depending on the way in which the role of producer/consumers is computed and handled.

***New Languages :***

the complexity of supporting full Prolog semantics in presence of dependent and-parallelism--especially concerning its interactions with don't know non-determinism--lead many researchers to consider simplified frameworks in which the exploitation of this form of parallelism could be realized without suffering excessive overheads.

The most interesting approach is represented by the class of *Committed-choice Languages (CCL)* .The languages belonging to this family (e.g., GHC ,Parlog ,Concurrent Prolog ,KL1 )are characterized by the fact that producer/consumer roles are statical (e.g., through mode declarations), and don't know nondeterminism has been eliminated (i.e., selection of a clause will not leave choice points behind).

These restrictions guarantee the possibility of efficiently implementing dependent and-parallelism--as confirmed by the practical experience .Nevertheless, these languages implement a semantic considerably different from that of Prolog (e.g., no don't know nondeterminism).

# 7. Multiple parallel systems

One of the long time goals of the research in the area of exploitation of parallelism from logic programming languages is the development of a single, general framework in which different forms of parallelism can be concurrently extracted.

Systems taking advantage of either and- or or-parallelism in an efficient way have been developed and they are showing excellent results, both in terms of speed and speedups, on a large variety of programs. Nevertheless, as explained in various experimental works :

1. many programs are quite rich in one of the two-forms of parallelism (either and- or or-parallelism). On the other hand neither of the two forms of parallelism seem to be prevalent on the other.
2. most ``real-life'' applications seem to present both forms of parallelism, but neither are ubiquitous. Indeed, some applications do not have much of either parallelism in them.

These observations lead to a natural conclusion: there is the need of being able of supporting both forms of parallelism within the same framework. Given a system of this kind, we would be able to obtain speedups from programs that have scarce parallelism (by taking advantage of it as much as possible), and we would be able to obtain considerable improvements on *any* kind of program, independently from whether it is more and- or or-parallel oriented.

Exploiting concurrently both and- and or-parallelism can be visualized as the development of an and/or-tree, which can be simply described as a union of the notions of and-tree and or-tree, as described in a previous section of this document.

**Example 2.8** *The symbolic integration problem developed in the previous sections is a typical application in which both or- and and-parallelism appears. The presence of clauses like*

1. `integr(X + Y, X' + Y') ← integr(X,X'),integr(Y,Y')`

2. `integr(A + B, X × Y) ← A = X1 × Y, B = X × Y1, integr(X,X1), integr(Y,Y`

*represents a powerful source of and/or-parallelism. Both clauses may be used to attempt sol the form $A + B$, where $A$ and $B$ are arbitrary complex terms, giving rise to or-parallelism, wh contain independent subgoals which can be executed in and-parallel.*

Various issues emerge when combination of different forms of parallelism is considered. In the rest of the section we will briefly analyze some of the possible choices and the consequent problems.

## 7.1 And- Under Or-parallelism:

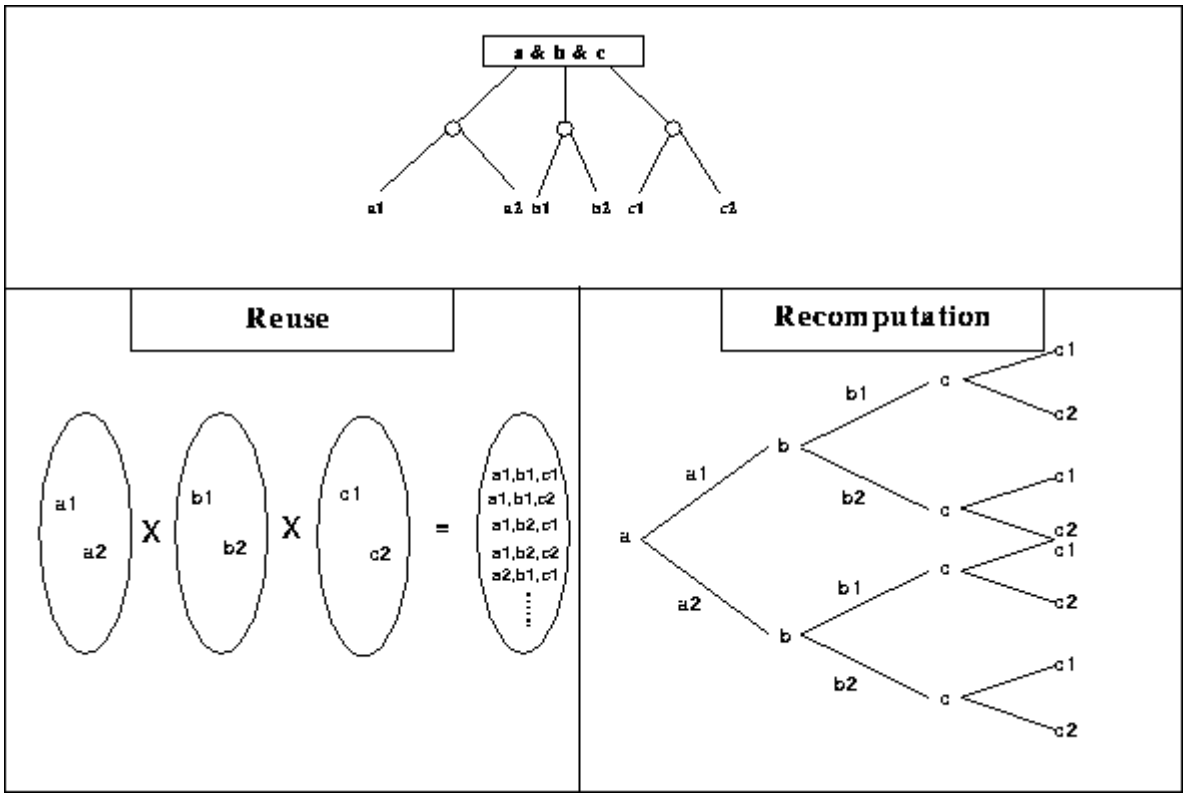



**Figure 9:** Reuse vs. Recomputation

As evident from figure 9, adopting reuse has some advantages, since the amount of work performed is considerably reduced (each subgoal is computed in its entirety only once). On the other hand, the reuse of subgoals creates many problems when order sensitive predicates are present in the subgoals. Reuse, in fact, does not guarantee anything regarding the relative order of execution of the various subgoals. Furthermore, in presence of predicates causing side-effects (e.g., write, read, etc.) the semantics of the program may change due to reuse--a program which performs one write has a different semantics from one performing five write operations. For these reasons (and others ),Reuse has not been considered as a viable approach to and/or-parallelism, and its use is limited to some old proposals (e.g., the Reduce Or-parallel Model by Kalé and Ramkumar ),Furthermore, the use of recomputation is substained by experimental studies ,showing that, on average, real-life Prolog programs contain a limited amount of recomputation--e.g., because the programmer is aware of the structure of the search space and writes his program in such a way to reduce the amount of backtracking.

## 7.2 Problems in Combining And- and Or-Parallelism:

Considering that parallel systems exploiting a single form of parallelism have been studied and efficiently implemented, it may seem that the task of combining the two forms of parallelism in a unique framework can be reduced to a simple engineering problem. This in consideration of the apparent orthogonal nature of the two forms of parallelism.

A more careful observation shows that this is not the case: combining different forms of parallelism introduces new complex issues to be solved, mainly related to environment representation and memory management. This can be illustrated quite easily. As we have shown in the previous sections, the different forms of parallelism

impose specific constraints on the way in which environments and data areas are managed:

- in and-parallelism, the different and-agents are computing *different* parts of the *same* solution to the original query--i.e., the different agents are working on the same branch of the or-tree. This leads to the need of allowing different agents to freely access each others' memory areas, in order to allow an efficient combination of the partial solutions computed by each agent. That is, the different and-agents should *share* their environments and data areas.
- in or-parallelism, the different or-agents are working on *different* solutions of the original query--i.e., they are working on different branches of the or-tree. As pointed out in a previous section, each agent's environments should be kept strictly separated from those of the other agents.

As a consequence, the requirements imposed by or- and and-parallelism are antithetical.

Furthermore, in order to guarantee a proper response to unbalanced distribution of or- and and-parallelism in the computation tree, the structure of the teams need to be dynamic, i.e., agents should be allowed to move from one team to another, or to create new teams in case of necessity. This requires the presence in the system of a *top-level scheduler*, whose task is to select the best teams configuration.

## 7.3 Existing Approaches to And/Or-Parallelism:

The few approaches available in literature for combining and- and or-parallelism in a single framework can be roughly distinguished in three categories. The first category includes all those systems which faithfully implement Prolog semantics, guaranteeing that any Prolog program will be executed with the same external behaviour as in a sequential execution. In this category we can find only very few proposals, and most of them either have not been implemented or are still in their

development stage. This is a clear indication of the challenging problems that need to be solved in order to achieve this objective. The most relevant proposals in this categories are *ACE* and the *Paged Binding Array (PBA)* scheme.

ACE is based on a generalization of the stack-copying scheme--described for or-parallelism in section. This model uses the layering scheme to and/or-parallelism described before. Computation within a team behaves exactly like a traditional and-parallel system (using recomputation). Or-parallelism is managed through stack-copying, with the novelty that the parts of the computation to be copied are spread across the stacks of the different agents belonging to the team. This makes the process of identifying the relevant parts of memory to be transferred extremely complex. Furthermore and-parallelism may lead to cases of ``trapped goals" (i.e., computations which appear on a stack in the order opposite to that in which they will be reclaimed), which will translate in copying of garbage during stack-copying.

Paged Binding Arrays extends the binding array method for or-parallelism, and it tackles the issues of memory organization by paging the binding arrays (following a methodology similar to the paging mechanisms used in operating systems). Both these systems are in their implementation stage.

Other proposals are IDIOM .. which extends the ACE model by allowing a restricted form of dependent and-parallelism--and Prometheus ..which generalizes the DDAS scheme for dependent and-parallelism by adding or-parallelism. The implementation of neither of these models has ever been attempted.

A second class of proposals is oriented to either allowing parallelism to be exploited only in programs having a certain structure, or relaxing the restriction of supporting full Prolog semantics. Various proposals have been made which allow exploitation of and/or-parallelism from *pure* Prolog programs, that is programs which do not contain any extra-logical operation. This is the case of the *Reduced Or-Parallel*

*Model (ROPM)* and AO-WAM .The already mentioned Andorra-I system exploits or-parallelism and determinate and-parallelism but, due to the particular nature of the Basic Andorra Model on which its computational model is based, it is unable to guarantee Prolog semantics in every case.

The third and final class of languages dealing with and/or-parallelism is characterized by the fact that the systems are based on implementing logic languages with semantics different from that of Prolog. The most relevant proposals in this area are represented by ANDOR-II ,in which GHC is extended with don't know nondeterminism (through the so-called *OR-predicates*) supported using a binding stamping mechanism, and AKL , based on the notion of Concurrent Constraint Programming (the computation is based on a constraint store accessed through *ask* and *tell* operations ).

## 7.4 The Ideal System:

The *Extended Andorra Model (EAM)* was designed in order to provide a formal framework and an abstract computational model for logic programming, in which different approaches to exploitation of parallelism can be described and compared. In EAM, computation is expressed through a set of *rewriting rules* acting on a state description.

The aim of the EAM is to detect the ``perfect'' computational model, where:

- *Work is Minimized:* all the answers to a logic programming query are generated with the minimum number of inferences;
- *Parallelism is Maximized:* the detection of the solutions to the problem is obtained performing as many steps as possible in parallel.
- *Subsumes other Languages:* it should support a language which is a superset of the most commonly used logic programming languages (e.g., Prolog,

Committed-Choice Languages, etc.), producing the same results with the same (or smaller) number of inferences and with the same (or larger) amount of parallelism exploited.

The first issue is quite intuitive, and translates into two subprinciples:

- branches of the computation tree should be *shared* whenever possible, instead of recomputing them.
- speculative computations should be avoided.

This last observation is quite crucial in understanding the complexity of the task: the two criteria of minimizing the number of inferences and exploiting maximal parallelism are antithetical. Almost any form of parallelism is prone to produce speculative computation, which contradicts the principle of work minimization.

The original EAM's description introduces a further aspect, the requirement of minimizing the amount of control information that the user is required to supply. This last principle have been deemed of secondary importance in some of the actual languages originated from EAM (e.g., AKL ).

The ideal nature of the EAM arises from the fact that the model may achieve the best balance between amount of parallelism exploited and the number of inference steps performed. On the other hand, as it is, the model is very abstract and far from a possible implementation. The major concern is that the abstract level of the model ignores many aspects involved with implementation (efficient management of environments, synchronization between agents, etc.). Nevertheless, the EAM is useful, since it supplies a general framework for describing and comparing different parallel execution models, where emphasis is put on *what* can be done in parallel (without yet loosing sight of the issue of efficiency), and outlining the ``perfect'' situation.

# Andrew systems

## Andrew 1 system

### Andorra-I Definition

A {parallel} {logic programming} language with the {OR-parallelism} of {Aurora} and the {AND-parallelism} of {Parlog}. ["Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism", V.S Costa et al, SIGPLAN Notices 26(7):83-93 (July 1991)]. [Imperial College? Who?] (1995-11-24).

A language combinging the OR parallelism of Aurora with the AND parallelism of Parlog.

"Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism", V. S. Costa et al, SIGPLAN Notices 26(7):83-93 (July 1991).

**Distributing And- Or-Work in the Andorra-I System**

Parallelism and logic programming are two fields that have been successfully combined, as is shown by recent implementations of parallel logic programming systems. There are two main sources of parallelism in logic programming, namely or- and and-parallelism. Or-parallelism is exploited when several alternative clauses to a goal are executed in parallel. And-parallelism is exploited when we execute two or more goals of the same clause simultaneously. Exploitation of full or- and and-parallelism is limited by the number of physical processors available in a system. And-parallelism is also limited by the interdependence among goals in a clause.

In parallel logic programming systems which exploit both and- and or-parallelism, a problem that arises is how to distribute processors between the dynamically varying

amounts of and- and or-work that are available. Solutions have been reported for distributing only or-work, or distributing only and-work, but the issue of distributing processors between both kinds of work has not yet been addressed. In this thesis we discuss the problem of distributing and- and or-work in the context of Andorra-I, a parallel logic programming system that exploits *determinate* and-parallelism and or-parallelism, and propose scheduling strategies that aim at efficiently distributing processors between and- and or-work.

We study general criteria that every reconfigurer should meet to reconfigure processors without incurring too high overheads in the Andorra-I system. We propose two different strategies to reconfigure processors between and- and or-work based on these criteria. One strategy, *work-guided*, guides its decisions by looking at the amount of current and- and or-work available in an application during execution. The other strategy, *efficiency-guided*, is designed around the intuition that processors should spend most of their execution time doing useful work, i.e., performing reductions.

Our benchmark set consists of programs that contain both and-parallelism and or-parallelism and programs that contain a single kind of parallelism. We compare the results produced by our strategies with the results produced by a version of Andorra-I that provides only a fixed division of processors between and- and or-work. Results show the benefit of adding a reconfigurer to Andorra-I and show that both strategies perform better than any plausible fixed configuration for all benchmarks. Although our work is focussed on the Andorra-I system, we believe that both strategies could be applied to other parallel logic programming systems that aim at exploiting both and- and or-parallelism.

**What is the Basic Andorra Model and AKL?**

The Basic Andorra Model is a way to execute definite clause programs that allows dependent and-parallelism to be exploited transparently. It also supports nice programming techniques for search programs. The idea is to first reduce all goals that match at most one clause. When no such goal exists, any goal (e.g., the left-most) may be chosen. The BAM was proposed by David H. D. Warren, and his group at Bristol has developed an AND-OR parallel implementation called Andorra-I, which also supports full Prolog. See, for example, Seif Haridi and Per Brand, "Andorra Prolog, an integration of Prolog and committed choice languages", in Proceedings of the FGCS 1988, ICOT, Tokyo, 1988. Vitor Santos Costa, David H. D. Warren, and Rong Yang, "Two papers on the Andorra-I engine and preprocessor", in Proceedings of the 8th ICLP. MIT Press, 1991. Steve Gregory and Rong Yang, "Parallel Constraint Solving in Andorra-I", in Proceedings of FGCS'92. ICOT, Tokyo, 1992. AKL (Andorra Kernel Language) is a concurrent constraint programming language that supports both Prolog-style programming and committed choice programming. Its control of don't-know nondeterminism is based on the Andorra model, which has been generalised to also deal with nondeterminism encapsulated in guards and aggregates (such as bagof) in a concurrent setting. See, for example, Sverker Janson and Seif Haridi, "Programming Paradigms of the Andorra Kernel Language", in Proceedings of ILPS'91. MIT Press, 1991. Torkel Franzen, "Logical Aspects of the Andorra Kernel Language", SICS Research Report R91:12, Swedish Institute of Computer Science, 1991. Torkel Franzen, Seif Haridi, and Sverker Janson, "An Overview of the Andorra Kernel Language", In LNAI (LNCS) 596, Springer-Verlag, 1992. Sverker Janson, Johan Montelius, and Seif Haridi, "Ports for Objects in Concurrent Logic Programs", in Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993 (forthcoming).

The Andrew system, developed at Carnegie- Mellon University, runs on thousands of computers distributed across the university campus. Its most notable component is the Andrew File System which now ties together file systems at sites distributed across the United States. Coda is a follow-on to Andrew, improving availability, especially in the face of network partitions.

The **Andrew Project** was a distributed computing environment begun in 1983, driven by the *Information Technology Center*, a joint Carnegie Mellon University and IBM project.

In its initial phase it involved both software and hardware, including wiring the campus for data and developing workstations to be distributed to students and faculty at Carnegie Mellon University and elsewhere. The proposed "3M" workstations included a million pixel display and a megabyte of memory, running at a million instructions per second. Unfortunately a fourth M, cost on the order of a megapenny, proved the 3M beyond the reach of students' budgets, so the initial hardware deployment in 1985 established a number of university-owned "clusters" of public workstations in various academic buildings and dormitories. The campus was fully wired and ready for the eventual availability of inexpensive personal computers.

Early software development within the Information Technology Center was organized into

- centralized tools (primarily a file server) and
- workstation tools (a window manager, editor, email, and file system client code),

called VICE (Vast Integrated Computing Environment) and VIRTUE (Virtue Is Reached Through Unix and Emacs) respectively.

The project was extended several times after 1985 in order to complete the software, and was renamed "Andrew" for Andrew Carnegie and Andrew Mellon, the founders of the institutions that eventually became Carnegie Mellon University. Mostly rewritten as a result of experience from early deployments, Andrew had four major software components:

- The **Andrew Toolkit** (ATK), a set of tools that allows users to create and distribute documents containing a variety of formatted and embedded objects,
- The **Andrew Messaging System** (AMS), an email and bulletin board system based on ATK, and
- The **Andrew File System** (AFS), a distributed file system emphasizing scalability for an academic and research environment.
- The **Andrew window manager** (WM), a tiled (non-overlapping windows) window system which allowed remote display of windows on a workstation display. WM was later replaced by X11 from MIT.

AFS moved out of the Information Technology Center to Transarc in 1988. AMS was fully decommissioned and replaced with the Cyrus IMAP server in 2002.

**The Andrew User Interface System**

After IBM's funding ended, Andrew continued as an open source project named the **Andrew User Interface System**. AUIS is a set of tools that allows users to create and distribute documents containing a variety of formatted and embedded objects. It is an open-source project run at the Department of Computer Science at Carnegie Mellon University. The Andrew Consortium governs and maintains the development and distribution of the Andrew User Interface System.

The Andrew User Interface System encompasses three primary components. The **Andrew User Environment (AUE)** contains the main editor, help system, user interface, and tools for rendering multimedia and embedded objects. The **Andrew Toolkit (ATK)** contains all of the formattable and embeddable objects, and allows a method for developers to design their own objects. ATK allows for multi-level object embedding, in which objects can be embedded in one another. For example, a raster image object can be embedded into a spreadsheet object. The **Andrew Message System (AMS)** provides a mail and bulletin board access, which allows the user to send, receive, and organize mail as well as post and read from message boards.

**Components of AUIS**

As of version 6.3, the following are all components of AUIS:

**Applications**

- Word processor (EZ)
- Drawing Editor (Figure)
- Mail and News Reader (Messages)
- Mail and News Sender (SendMessage)
- Font Editor (BDFfont)
- Documentation Browser (Help)
- Directory Browser (Bush)
- Schedule Maintainer (Chump)
- Shell Interface/Terminal (Console, TypeScript)
- AUIS Application Menu (Launch)
- Standard Output Viewer (PipeScript)
- Preferences Editor (PrefEd)

**Graphical and Interactive Editors**

- Equation Insert (EQ)
- Animation Editor (Fad)
- Drawing Editor (Figure)
- Insert Layout Insert (Layout)
- Display Two Adjacent Inserts (LSet)
- Extension and String Processing Language (Ness)
- Display and Edit Hierarchies (Org)
- Page Flipper (Page)
- Monochrome BMP Image Editor (Raster)
- Spreadsheet Insert (Table)
- Text, Document, and Program Editor (Text)

## Andrew Systems Inc

While Andrew Wireless Solutions is recognized as an industry leader in telecommunications infrastructure manufacturing, many companies are not aware that we also offer complete program management and installation services. Founded in 1972, Andrew Systems Inc. provides **program management** and **field installation services**, to the North American markets, of the same level of quality and reliability that customers have come to expect from Andrew products.

For maximum peace of mind, the ASI Program Management team can provide an entire Turnkey System, where all aspects of the project are handled by ASI start to finish. Laying out of the site, clearing the ground, grading, fencing, arranging utility services, verifying components against site drawing requirements, maintaining installation specifications, installing tower and shelter foundations, erecting the tower, installing mounts, installing antennas, installing RF transmission cables, shelter placement and setup, and testing the system for proper installation and operation.

Perhaps a turnkey system is more than you require. Andrew Systems Inc. crews can perform any combination of valuable services at your site. Crews are fully trained and certified on a variety of skills. They utilize a wide range of equipment, including RF sweeper mainframes, network analyzers, plotters / printers, directional bridges, Hybrid T reflectometers, Andrew path alignment transceivers and more.

Andrew Systems Inc. meets all legal regulations, holds all required State Contractor Licenses, are compliant to DOT Rules and Regulations, are compliant to OSHA Safety Regulations for Construction and Safety, and hold $2,000,000.00 General Liability Insurance.

For more information, please contact your local sales office and ask about Andrew Systems Inc.

# Andrew 2 system

Andrew is the central distributed computing service provided to the Carnegie Mellon campus by the Computing Services division. Among the many services provided on Andrew there is a central file store, electronic mail, notification services, and a large selection of application software. The Andrew user community consists of over ten thousand users. The Andrew services are biased towards UNIX clients, though local clients have been developed for the Macintosh and Windows/DOS machines on campus. Andrew II will more tightly integrate the Macintosh and Windows/DOS environment with that of the UNIX clients on campus.

**What is Andrew II?**

The Andrew II series of projects is replacing much of the original Andrew distributed computing infrastructure at Carnegie Mellon University. The Andrew II projects were initiated in response to the realization that many of the original Andrew services would need to be re-engineered when the Andrew File System (AFS) is replaced by the Distributed File System (DFS) from the Open Software Foundation (OSF). All of the computing infrastructure of the original Andrew system was built on top of AFS.

The current Andrew distributed computing environment has total and critical dependency on the Andrew File System (AFS). Hence if AFS is performing poorly, printing, mail, backup, and all such services will also suffer. Indeed, if the file or the network system is down, the Andrew workstation looks remarkably like a dumb terminal without a functioning host. Effectively, Andrew has the characteristics of a time sharing system.

A year or two ago, Transarc, the supplier of AFS, had submitted a new version of AFS (version 4) to the Open Software Foundation (OSF) for consideration as the file system for their Distributed Computing Environment (DCE). The submission was

warmly received and accepted as the as OSF's standard Distributed File System (DFS). DFS is expected to become widely available from vendors such as DEC, IBM, and HP in late 1993. We will most likely to be cutting over to this new and much improved file system sometime in 1994.

Unfortunately, the conversion from AFS to DFS will not be straightforward. Significant structural changes have been made to the current version of AFS. The Application Programming Interface (API) has also undergone substantial modifications. Any program written based upon assumptions of the specific internal structure of AFS 3 will most likely break. This is the case with virtually all the current locally developed Andrew applications.

Our alternatives are to (a) wait for DFS to become stable and modify all those applications accordingly or (b) take the opportunity to address the more fundamental architectural issue of a completely new file system dependent distributed services environment.

We decided on the latter approach. In the new scenario, the file system is one of many services attached to the network. Other services would include printing, mail, backup, authentication, directory services, etc. In a ideal situation, all services will be equal to each other with little or no cross dependency upon one another.

Hence, if a service fails, the client will be denied of that, and only that, service. All other services on the network should still be available. This is not actually a novel concept. It closely approximates that of the Macintosh networking environment. Our challenge is to make it scalable providing services ACROSS THE campus community instead of to a localized work group.

Service Overview

A brief overview of the main services we will be providing follows.

File service will be provided mainly by DFS. he Macintosh community will continue with AppleShare. The PC community will be using Netware from Novell. We will be looking at ways to better integrate the three different environments. The default, lowest common denominator between them will be FTP.

Printing services will be enhanced to provide for better monitoring of usage and control. Users should also be able to direct their output to any desired printer, subject to compatibility, access privileges and other controls as defined by the owner of the equipment.

Remote backup and archival services will be provided for owners of workstations, Macintoshes and PCs. Backup is the means to provide a general safeguard against disk failure, while archiving is DONE AT THE explicit request of a user to backup specific components. As the usage of multimedia increases, automatic migration of data from disk to optical storage technology, based on usage, will also be considered.

Mail and BBoard services will be redesigned to be structurally more modular, substantially reducing or removing altogether their dependency on the central file system. More focus will be placed on improving basic capabilities such as distribution list handling, attachments, etc.

The configuration of the UNIX workstation will fall into two categories: network dependent and network independent. Network dependent workstations would function the same way as the Andrew workstations today. They would likely be the configuration of choice for cluster workstations. Network independent workstations would have adequate local disk space to support standalone operation if necessary. Provisions will be made to allow users to move software packages from appropriate servers easily into the local disk, subject to licensing constraints. An intriguing area we will address is <169>network docking.<170> When a network independent

workstation, such as a laptop or notebook machine, is reconnected to the network, a user-definable set of activities will happen automatically. Hence the user will be advised if the software packages on his or her machine is out of date, spooled print jobs will automatically flow into the network, archival request and backup will happen, the user will be informed of pending mail, etc.

Note that because the servers will have minimum cross dependencies upon one another, any organization in the university should be able to bring up its own servers relatively easily. Access to those servers would, of course, be by the grace of their providers.

Approaches

This time around, we will be approaching the problems somewhat differently than the original Andrew project. While UNIX workstation support will remain a key element of this distributed computing environment, we will take providing service to Macintoshes and PCs into consideration from the outset. We will also explore the use of public domain and commercial components either directly or as a base for development whenever possible. Our challenge will be the graceful integration of the parts to form a relatively seamless and easy to manage system that can readily evolve over time.

The above project is fairly extensive in scope. Instead of approaching it alone, we have made considerable effort to form partnerships with other departments on campus, other universities and corporations. Besides getting potential human and/or financial support for our work, we would very much like to get long term support for our products from other quarters.

Timelines

The Computing & Communications Development groups have started working on this project indirectly since last Fall. The first objective is to achieve a stable, reliable, environment that performs reasonably and, most importantly, requires low maintenance so that more developers can be put onto the Phase II project. We have substantially achieved that goal. While we will continue to enhance the current Andrew environment, such as by providing support for new versions of operating systems and/or new platforms and increasing service capacities, most of the focus will be diverted to Andrew II. Because of the principle of minimum cross dependencies between servers, components of Andrew II will be introduced as they become available over the next two years. Indeed, one may consider LIS II, the recently introduced electronic library service derived from the Mercury project, as the first component of this new phase. For components that would be changed between Andrew I and Andrew II, careful considerations will be taken to ensure the migration occurs smoothly. The last component to be changed will likely be the cause of this whole exercise itself: AFS to DFS. That should happen no earlier than 1994.

**The Andrew II Projects Requirements Documents**

**Host Configuration: Workstation Administration**

The Host Configuration project addresses issues of workstation management. In defining the requirements for Host Configuration, three separate aspects are discussed:

- *Performance Management* - Performance Management provides the tools necessary to benchmark the operating system and commonly used applications. This is necessary for capacity planning. For example, do we need more machines now or can we wait a year? Performance Management

is also useful for testing purposes. If a change was made to a kernel configuration variable, can we test this change in a controlled environment rather than a production environment to see if there is a significant improvement.

- *Docking* - Docking addresses mobile computing as well as availability. Docking integrates services normally available only in a network-based, distributed environment with portable computers. Since many of the issues overlap, docking will also address how to minimize the disruption of network or server failure.
- *Workstation Administration* - Workstation Administration recognizes the increasing popularity and increasing number of workstations. This includes UNIX, Macintosh or PC(1) systems. As a result, the task of maintaining the configuration of each individual workstation becomes an increasingly difficult and time consuming chore. Not only do system administrators have to deal with more and more machines, there also tend to be many machines from different vendors, with each vendor having different management tools and practices. Finally, workstation owners are demanding custom configurations, often tailored to their specific need. Workstation Administration concentrates on how to install, distribute, customize and maintain the software and files that are to reside on the local disk of the workstation.

## Backups & Archiving

This Requirements Specification defines the criteria necessary to implement a backup and archiving system for the Carnegie Mellon central computing environment. It describes all of the features required in the new system and the factors essential in its evaluation and implementation. This Backup & Archiving System will also be available for Carnegie Mellon departmental and organizational computing facilities to implement independently.

This document will be used by the Backup and Archiving Project Group when evaluating possible backup and archiving systems. Vendors may use these Specifications to determine if their existing or future products are appropriate for the Carnegie Mellon system.

Carnegie Mellon is willing to work with vendors or other institutions in a joint development effort for a backup system that would handle Carnegie Mellon's needs as well as the needs of commercial users.

## Electronic Mail

This document describes the functional requirements of the Andrew II Electronic Mail project. The purpose of the project is to provide an electronic mail system for users of the Andrew system and any other organizational computing facility at Carnegie Mellon that wishes to use it.

Our experience with the Andrew Mail System (AMS) has shown a number of problems with its design, leading to problems with performance and difficulty in administration. This project will attempt to implement a replacement mail system which corrects many of these problems.

The Andrew Mail System has many useful features not found in other mail systems. Some of these features are important to the user community, others are of dubious value. Determining which features of the Andrew Mail System are important and reimplementing them in the new mail system is an important part of this project.

## Personal Systems

This document outlines the functional requirements for integrating personal systems into the Andrew II environment. Personal Systems traditionally have been PCs or

Macintoshes. These machines can run a rich suite of commercial software products and are heavily used on campus and in industry. Personal systems run stand-alone and offer users enough power and capability to do their tasks. Personal systems that require network access will generally do so for the value added services that the network provides. Personal systems host a variety of operating systems, such as DOS, Windows, and MAC OS. Please refer to Section 3.2, *Design Constraints*, for the list of supported hardware, operating systems, and software.

The Andrew II environment as a whole will be based on the Open Software Foundation's Distributed Computing Environment. The minimal offerings that the Andrew II environment will provide to the users of personal systems is:

- Mail and BBoard access
- Printing
- Secure network access to other services (ie: Backup)
- Remote filesystem access
- Backup and archiving of personal systems
- Notification services

## Printing

This document describes the functional requirements for the Andrew II Printing Project. The purpose of the project is to provide authenticated printing across the PC, Mac, and Unix environments at Carnegie Mellon. The solution must provide a native "look and feel" for all platforms, and be transparent to the user community.

The goals of the project also include enhancements to the current printing environment. These enhancements include improved tools for management, monitoring, and tracking.

# Andrew File System

The **Andrew File System (AFS)** is a distributed networked file system which uses a set of trusted servers to present a homogeneous, location-transparent file name space to all the client workstations. It was developed by Carnegie Mellon University as part of the Andrew Project. It is named after Andrew Carnegie and Andrew Mellon. Its primary use is in distributed computing.

## Features

AFS has several benefits over traditional networked file systems, particularly in the areas of security and scalability. It is not uncommon for enterprise AFS cells to exceed twenty five thousand clients. AFS uses Kerberos for authentication, and implements access control lists on directories for users and groups. Each client caches files on the local filesystem for increased speed on subsequent requests for the same file. This also allows limited filesystem access in the event of a server crash or a network outage.

Read and write operations on an open file are directed only to the locally cached copy. When a modified file is closed, the changed portions are copied back to the file server. Cache consistency is maintained by a mechanism called *callback*. When a file is cached, the server makes a note of this and promises to inform the client if the file is updated by someone else. Callbacks are discarded and must be re-established after any client, server, or network failure, including a time-out. Re-establishing a callback involves a status check and does not require re-reading the file itself.

A consequence of the file locking strategy is that AFS does not support large shared databases or record updating within files shared between client systems. This was a deliberate design decision based on the perceived needs of the university computing

environment. It leads, for example, to the use of a single file per message in the original email system for the Andrew Project, the Andrew Message System, rather than a single file per mailbox.

A significant feature of AFS is the volume, a tree of files, sub-directories and AFS mountpoints (links to other AFS volumes). Volumes are created by administrators and linked at a specific named path in an AFS cell. Once created, users of the filesystem may create directories and files as usual without concern for the physical location of the volume. A volume may have a quota assigned to it in order to limit the amount of space consumed. As needed, AFS administrators can move that volume to another server and disk location without the need to notify users; indeed the operation can occur while files in that volume are being used.

AFS volumes can be replicated to read-only cloned copies. When accessing files in a read-only volume, a client system will retrieve data from a particular read-only copy. If at some point that copy becomes unavailable, clients will look for any of the remaining copies. Again, users of that data are unaware of the location of the read-only copy; administrators can create and relocate such copies as needed. The AFS command suite guarantees that all read-only volumes contain exact copies of the original read-write volume at the time the read-only copy was created.

The file name space on an Andrew workstation is partitioned into a *shared* and *local* name space. The shared name space (usually mounted as /afs on the Unix filesystem) is identical on all workstations. The local name space is unique to each workstation. It only contains temporary files needed for workstation initialization and symbolic links to files in the shared name space.

The Andrew File System heavily influenced Version 4 of Sun Microsystems' popular Network File System (NFS). Additionally, a variant of AFS, the Distributed

File System (DFS) was adopted by the Open Software Foundation in 1989 as part of their Distributed Computing Environment.

**Implementations**

There are three major implementations, Transarc (IBM), OpenAFS and Arla, although the Transarc software is losing support and is deprecated. AFS (version two) is also the predecessor of the Coda file system.

A fourth implementation exists in the Linux kernel source code since at least version 2.6.10. Committed by Red Hat, this is a fairly simple implementation still in its early stages of development and therefore incomplete.

**Available permissions**

The following Access Control List permissions can be granted:

Lookup (l)

: allows a user to list the contents of the AFS directory, examine the ACL associated with the directory and access subdirectories.

Insert (i)

: allows a user to add new files or subdirectories to the directory.

Delete (d)

: allows a user to remove files and subdirectories from the directory.

Administer (a)

: allows a user to change the ACL for the directory. Users always have this right on their home directory, even if they accidentally remove themselves from the ACL.

Permissions that affect files and subdirectories include:

Read (r)

allows a user to look at the contents of files in a directory and list files in subdirectories. Files that are to be granted read access to any user, including the owner, need to have the standard UNIX "owner read" permission set.

Write (w)

allows a user to modify files in a directory. Files that are to be granted write access to any user, including the owner, need to have the standard UNIX "owner write" permission set.

Lock (k)

allows the processor to run programs that need to "flock" files in the directory.

## Parallelism vs. Efficiency:

The aim of developing parallel systems is that of being able of taking advantage of parallel architectures, obtaining faster executions.

Unfortunately, often the research have not focused in the proper direction. The main reason is that the ``search'' for parallelism has been often taken to an extreme. This translates to a very large amount of parallelism extracted from programs, but without taken into account some fundamental aspects, like:

- availability of resources: extracting more parallelism than the number of resources available for execution (i.e., processors) leads to the need of assigning multiple pieces of work to the same agent. This typically involves a cost that may impact the overall system efficiency. In this terms, it may be wiser to extract ``less'' but ``better'' parallelism (e.g. parallel tasks with larger granularity).
- system efficiency: this is an issue that, especially in the older systems, has not been taken sufficiently into account. It is well-known that supporting

parallel execution imposes certain costs (scheduling costs, work management costs, etc.), and often these costs are proportional to the amount of parallelism exploited. Furthermore, certain design choices made in parallel systems may end up disrupting the sequential efficiency of the system--i.e. the efficiency of the system while executing sequential parts of the computation.

The ideal situation is represented by the achievement of good speedups in presence of the *no-slowdown* property:the parallel system, in any case, should never take more time than a corresponding sequential execution. This, in particular, translates to having systems capable of maintaining the efficiency of the state-of-the-art sequential implementations.

The logical question is ``how to achieve good efficiency ?'' There are various issues that have to be tackled in the design of a parallel logic programming system in order to guarantee good efficiency, and the rest of this document will mainly focus on them. We can identify three major aspects that have to be analyzed:

- *Control Management:* Control management is an extremely delicate issue in parallel logic programming, especially for what concerns scheduling. Scheduling, i.e. deciding *what*, *when*, and *by whom* should be executed in parallel, is the key to the success (or failure) of any parallel system.

  Scheduling should ideally select tasks that are more likely to be useful (e.g. or-branches containing a successful derivation), having large grain, and guaranteeing execution with a minimal amount of communication between agents.

- *Memory Management:* many researchers have drawn the attention on the importance of an efficient and effective memory management. The

computation should try to minimize the amount of memory employed, aiming at avoiding allocation of data structures that are not strictly required. This will not only reduce the amount of memory used, but may allow considerable reductions in execution time (due to removal of the costs related to managing such data structures).

Furthermore, the way in which information are organized in the abstract machine has a profound impact on the parallel execution. As we will see in the next sections, adopting a different organization for certain data areas may considerably improve execution time--by removing part of the cost of searching for information spread in the various agents' stacks.

Logic programming languages are considerably eager in memory consumption, thus garbage collection is also an issue that cannot be ignored.

- *Optimizations:* the mechanisms available to support parallelism are typically designed to tackle the worst-case situation. The ability of identifying simpler cases allows the use of simpler execution schemes, with consequent reduction in overhead and improvement in speed. Optimizations can be distinguished in various categories:
    - Pure Compile-time: the opportunities for optimization are identified during compilation and the optimization is directly exploited by the compiler, producing an improved compiled code;
    - Pure Run-time: the possibility of improving the execution is detected during the execution itself;
    - Mixed Optimization: the compiler collects knowledge that will be used at run-time to verify the possibility of improving the execution.

In the rest of this document we will devote limited space to this topic, being largely unexplored in the area of Parallel Logic Programming .

# 8. Sequential Efficiency

Sequential technology for Prolog has reached a quite mature stage, actual systems are commercially used and offer performances comparable to those of the state-of-the-art systems for other declarative programming paradigms. Nevertheless, the aim of achieving performance figures at the level of imperative programming technology (e.g. C) has not been completely achieved yet. Various directions are currently pursued in trying to improve the execution performance of sequential systems.

## 8.1 Compile-time Analysis:

Most of the complexity of executing Prolog programs comes from the *lack of a-priori knowledge* regarding the development of the execution. This forces the engine to either

- make conservative choices; or
- introduce additional run-time checks to guarantee safe execution and direct the computation in the correct direction.

For example, during compilation, the lack of knowledge regarding the instantiation state of the arguments of a subgoal obliges the compiler to generate code capable of handling all the different situations (e.g., arguments are unbound variables, arguments are ground terms, etc.). The selection of the proper sequence of instructions will be done at run-time--with the cost of checking the status of each argument.

A great deal of improvement could be obtained by increasing the amount of knowledge available during program compilation. This knowledge can be either

supplied by the programmer, through declarations and pragmas in the source code (e.g. Mercury), or automatically extracted using abstract interpretation and other compile-time techniques .

The natural question at this point is the following: ``*which kind of information are useful to improve program compilation and execution ?*" A wide variety of information may actually turn out to be useful during program translation, but the main sources of improvement may be represented by having a good knowledge about the following features of the execution:

- **Types:** Prolog, as most of the other logic languages proposed, is a typeless language. It is based on a single-sort language--there is virtually no distinction between different classes of terms. The lack of types makes the language more elegant and more clear from the semantical point of view, it represents a considerable source of inefficiency at the implementation level. The term representation inside the abstract machine needs to explicitly maintain type information (to support type-checking, memory management, etc.)--actual values need to be ``*boxed*" with the type information. During execution values cannot be directly used, but they first need to be unboxed, and the eventual type checks performed. Knowledge about the type of the terms used in every point of the execution would allows various improvements, like avoidance of the boxing of terms, removal of run-time type checking and improved clause indexing. Various systems make use of type information (either collected through compile-time analysis or supplied by the programmer) to improve execution.
- **Modes:** in general terms, a mode of a predicate indicates how its arguments will be instantiated when that predicate is called. More formally, the mode of a predicate is a mapping from the initial state of instantiation of its arguments to their final state of instantiation. Given a tree representing a type, an

instantiation tree is obtained by associating a value of instantiatedness (either *free* or *bound*) to each or-node of the tree. This annotated tree (called instantiatedness tree in ) tells which variables are bound and which are not. A mode is a mapping between instantiatedness trees.

Knowledge of modes allows considerable optimizations during execution, like improved clause selection, transformation of unification to matching, and detection of producer/consumer relationships.

- **Determinism:** considering a given mode for a predicate, determinism knowledge is used to categorize the mode according to the *number of solutions* that can be returned by the predicate when called with such mode. The traditional classification distinguishes *determinate*--i.e., producing at most one solution--and *nondeterminate* modes--i.e., which may produce more than one solution. A subgoal which is known to be determinate can be executed very efficiently--it does not require any additional support with respect to execution of imperative programs. Furthermore, knowledge of determinacy allows many optimization of the execution (e.g., it could be feasible to *reuse* determinate subgoals on backtracking, instead of recomputing them each time). Certain systems go to the extent of adopting different execution algorithms depending on the nature of the subgoal to be executed (e.g., Mercury distinguishes four levels in the categorization based on determinacy, with three different execution schemes).

## 8.2 New Models:

While many implementations are based on the Warren Abstract Machine, or at least on models directly derived from this design, in recent years some novel execution models have been proposed, which substantially differ from the WAM. Two of the most successful ones are *BinProlog* and the *Vienna Abstract Machine (VAM)*.

**BinProlog:** Developed by P. Tarau, BinProlog is based on the idea of *continuation passing*. Prolog programs are transformed into binary logic programs--i.e. Prolog programs with at most one literal in the body. This is realized by making explicit the transfer of the continuation of each subgoal, as a new argument.

**Example 3.1** *Given the Prolog program:*

```
p(a,b).
p(X,Y) ← q(X,Z),r(Z,Y).
```

*it is transformed into the following binary program:*

```
p(a,b,Cont) ← call(Cont).
p(X,Y,Cont) ← q(X,Z,r(Z,Y,Cont)).
```

*As we can see, the continuation is passed along as an additional argument and executed whe*
*(i.e., clause with an empty body) is encountered.*

BinProlog has been implemented as a heap-only system (i.e., no concept of environment is introduced). The fact that the abstract machine has been specialized to binary programs allowed to considerably reduce the number of instructions required and facilitated the introduction of a wide variety of optimizations.

**Vienna Abstract Machine:** The Vienna Abstract Machine (VAM) developed by A. Krall maintains in many aspects the spirit of the WAM, curing some of the drawbacks in the original Warren's design. In particular, Krall's observation is that one of the main sources of inefficiency in the WAM is represented by the separation between the phase in which the arguments of a subgoal are prepared and the phase in

which head unification is performed. The original design of VAM (called VAM $^{2P}$) is based on an architecture with two instruction pointers. These are used to combine argument setup and head unification in a unique step (by fetching two instructions at the same time, one coming from the goal and one from the head of the clause). Furthermore the selection of the kind of action required is made very efficient by using a special instruction coding, which allows to identify the desired action by simply adding the codes of the two instructions fetched.

# 9. Parallelism Efficiency

The issue of efficiency is becoming of major importance in the area of parallel implementation of logic programming. It is relatively straightforward to produce a naive parallel models, eventually based on intuitive views of the execution model. This is the case, for example, of the *process view* adopted in some of the seminal works on parallel Prolog, in which each subgoal was treated as a separate process, and the shared variables used as communication channels--a useful view to understand the model, but a very inefficient approach if directly used as design principle.

As already mentioned in a previous section, one of the objectives of an efficient parallel implementation is its *sequential efficiency*, i.e. its ability to maintain during sequential execution a speed comparable to that of the state-of-the-art sequential implementations. Achieving this target is far from being easy, and very few systems can actually claim of being able to obtain similar results.

Supporting parallelism imposes various costs, which translates on overhead w.r.t. the standard sequential behaviour. Furthermore, due to the inability of the engine of realizing whether parallelism will be actually exploited or not, part of this overhead will be paid even if the actual execution is carried on by a single agent. Merit of a good parallel model is to force most of the necessary overhead to occur at the moment in which the actual parallelism is exploited--avoiding it in case of sequential execution.

Parallel execution in a logic programming system can be viewed as the parallel traversal of an and-or tree. An and-or tree has or-nodes and and-nodes. Or-nodes are created when there are multiple ways to solve a goal. An and-node is created when a goal invokes several conjunctive subgoals. The or- and and-nodes represent *sources*

*of parallelism*, i.e. points of the execution where it is possible to fork parallel computations.

Parallelism allows overlapping of exploration of different parts of the and-or tree (which are instead sequentialized in sequential computation). Nevertheless, as mentioned earlier, this does not always translate to an improvement in performance. This happens mainly because:

- the tree structure developed during the parallel computation needs to be explicitly maintained, in order to allow for proper management of nondeterminism and backtracking--this requires the use of additional data structures, not needed in sequential execution. Allocation and management of these data structures represent an overhead during parallel computation w.r.t. the sequential one.
- the tree structure of the computation needs to be repeatedly traversed in order to search for multiple alternatives and/or cure eventual failure of goals, and such traversal often requires synchronization between the computing agents. These traversal and synchronization activities are absent from a sequential computation (in which the management of nondeterminism is reduced to a linear and fast scan of the branches, following a predetermined order).

So far we have intuitively identified the main sources of overhead in a parallel computation--the management of additional data structures, and the need to traverse a complex tree structure.

Overheads in the traversal of the tree can be tackled by simplifying as much as possible the structure of the computation (e.g. reducing the level of nesting of parallelism), and by collecting information that may guide the traversal (and prune useless parts of the tree). Management of the data structures can be made more

effective by reusing data structures whenever possible and avoiding allocation of new structure until they are strictly necessary.

These observations can be actually concretized into various optimizations (some have been illustrated in various works ), allowing improvements in performance and reductions in parallel overhead.

In the rest of this section we will present a more specific discussion on overheads emerging in the different forms of parallelism.

## 9.1 Overheads in Or-Parallelism:

Or-Parallelism has been implemented with quite limited overhead. The two major implementations proposed, Muse and Aurora, respectively report an average overhead over SICStus Prolog (on which both the implementations are based) of 8-22% (depending on the architecture considered) and 25%. These results are extremely encouraging.

The main advantage of or-parallelism (w.r.t. and-parallelism) is the fact that in or-parallelism it becomes quite natural to limit the overhead to the actual exploitation of parallelism. The compiler may detect (eventually through the help of user declarations) which are the choice points which may become parallel, and they will be appropriately marked (an almost zero-cost operation) when created. As long as parallelism is not exploited, choice points are treated normally, without any overhead.

Overheads generated in an or-parallel model depend on the specific model employed. Nevertheless, it is possible to make some general considerations. Gupta and Jayaraman  have observed that any or-parallel model implementing Prolog semantics is going to suffer a non-constant overhead in *at least* one of the following three phases:

1. *Task Creation:* which involves creating the global environment and goal-list for a new alternative taken from a choice-point;
2. *Task Switching:* which involves moving one agent from one node in the or-tree to another;
3. *Variable Access:* which involves reading and/or binding a variable.

Since the frequency of two of these operations, variable access and task creation, is outside the scope of control of the engine, it would be desirable to have models where the non-constant overhead is located exclusively in the task switching phase (whose frequency can be controlled by the scheduler). This observation is actually confirmed by the practical experience: the most effective implementations of or-parallelism available are exactly based on this principle (e.g., Muse and Aurora).

### 9.1.1 Stack Copying:

The stack-copying approach has a very attractive feature, the fact that the local execution carried on by each agent is exactly a standard sequential execution, at least as long as this computation is limited to the ``private'' part of the tree (i.e., the computation is not backtracking back on the copied part of the execution).

Overheads appears only in three situations:

1. after a certain number of calls, each agent should check whether any request of work sharing have been sent to him by other (idle) agents. If such request is present, then a sharing operation takes place--involving detection of the areas to be copied and the actual copy operation. This sharing operation can be time consuming. In order to contain its cost, the usual implementations
    - try to share as much as possible at once, in order to reduce the frequency of sharing operations;
    - adopt incremental copying techniques, where only the difference between the two agent states is copied;

- try to parallelize the sharing operation.

2. whenever the execution falls back in a part of the computation that has been copied, special actions are required. Access to the copied choice points should be a critical section and the detection of no alternatives should lead to the invocation of the scheduler.
3. the execution of side-effects requires special attention: these are order-sensitive predicates across the branches of the or-tree, and the agents should guarantee that their execution is realized in the correct order. This typically involves delaying the execution as long as the current branch is not leftmost in the or-tree.

### 9.1.2 Binding Arrays:

many of the considerations made for stack-copying are valid also in the case of Binding Arrays. The execution model is quite similar, in the fact that the scheduler will be called upon backtracking in the public part of the tree (part that is shared between different agents), and it will eventually send a request to an agent for taking work from a new choice point. In the Binding Arrays case the operation of taking work is considerably faster, since the only activity required is updating the content of the binding arrays. On the other hand, the model has an additional overhead due to the fact that every conditional variable is now stored in binding arrays, and every access to it requires one step of indirection.

## 10. Overheads in And-Parallelism

The situation in terms of overhead is more serious in the case of and-parallelism. In or-parallelism there is the advantage that the data structures describing the list of alternative computations (i.e., the choice points) are already present in the sequential model itself, and can be easily updated to suit a parallel execution. For this reason we obtain almost zero overhead during sequential execution.

In and-parallelism the situation is more complex: the sequential model does not have an explicit representation of the list of subgoals (this is implicit in the next instruction pointers saved in the environments and in the code structure itself). Though, the sequential model (at least based on the WAM structure) does not supply sufficient information to allow an easy adaptation to and-parallelism. And-parallelism requires an explicit representation of the current resolvent, or at least of the part of the resolvent that is meant to be executed in parallel--this to allow the different agents to easily identify the presence of parallel work and take advantage of it.

The majority of and-parallel systems introduce a form of goal-stacking in their execution model (this will be described in detail in a successive section--for now it will be sufficient to know that goal-stacking is a computation model in which the current resolvent is completely described by a sequence of goal descriptors stored in a stack), either to completely replace environment stacking or limited to the parts of the resolvent subject to parallel execution. Furthermore, this should come together with some additional data structures to allow a proper control of the parallel execution (e.g., how many subgoals are still to be completed, where are they located, etc.).

The relevant point is that switching to goal stacking and paying its cost need to be performed a-priori, in the moment in which the resolvent is started, without any knowledge of whether the and-parallel execution will take place at all. This translates to pure overhead that may not be balanced by any speedup due to parallelism, with a possible slow-down with respect to sequential execution.

This has made and-parallel systems not extremely competitive with sequential technology--the amount of overhead may be considerable, especially in programs which offer a great amount of parallel work.

Backtracking offers another source of overhead. And-parallelism leads to a distribution of the execution across different agents. Backtracking requires traversal of the computation in a well-specified order (right-to-left) in order to guarantee the proper semantics. This imposes the requirement of being able to trace back the position of the different parts of the computation in the stacks of the various and-agents. This can be realized, in general, only by storing additional information that will be used during backtracking. For example, in the RAP-WAM model each parallel subgoal's execution is delimited by *markers*, and each parallel subgoal has a descriptor associated to it and stored in the descriptor of the parallel call. The subgoal's descriptor stores information regarding the location of the markers delimiting the execution. On backtracking the entry point in the subgoal will be detected by locating the marker which closes its execution.

If the argument moves towards dependent and-parallelism, we must register the presence of further overheads due to the complexity of managing accesses to shared variables (e.g., guaranteeing mutual exclusion).

Solutions to at least part of these problems can be found, although their engineering in an actual implementation is far from being straightforward. Parts of the idea of the Warren Abstract Machine need to be reconsidered--the WAM has been designed to

be an efficient sequential model and some of its choices are not suitable to parallel implementation. Leaving the WAM as a base for sequential implementation will not necessarily translate to a loss in performance since, as described in the previous section on sequential efficiency, new models have been recently proposed that are competitive with the WAM and are more ``goal-stacking'' oriented.

# 11. Overheads in And/Or-Parallelism

And/or-parallel system clearly suffer of the overheads coming from both and- and or-parallelism. Unfortunately, these overheads may get worse due the combination of the two forms of parallelism.

Adopting a model for exploitation of and/or-parallelism based on the layering approach, we can observe the following effects:

- and-parallelism is locally exploited within a team--and will not suffer of any additional overhead since its execution model is unchanged;
- or-parallelism becomes considerably more complex. The computation developed by an or-agent is not anymore nicely stored on a single stack, but it is spread across the stacks of the various and-agents belonging to the corresponding team. This makes the management of or-parallelism more complex.

  If stack-copying is the technique adopted to support or-parallelism, then the distributed nature of the computation makes very complicate to detect the areas that need to be transferred, making incremental copying very hard. Furthermore, and-parallelism can produce arbitrary intermixing of subgoals on the stacks, leaving trapped subgoals and empty gaps; during the copying operation, this will cause the transfer of areas which are not needed (and that will need to be properly marked as garbage).

  On the other hand, if binding arrays are used to support or-parallelism, various issues related on how binding arrays are to be managed emerges and

will often translates to additional overhead. If a technique like Paged Binding Arrays is used, then there will be the additional cost for managing pages and for supporting an extra level of indirection during variable access.

- scheduling becomes, in general, considerably more complex. Scheduling or-parallelism is complicated by the fact that the computation along each branch is spread between different processors and keeping track of the choice points created becomes a challenging task.

  Furthermore, the presence of both and- and or-parallelism requires an additional scheduling step, in order to decide which form of parallelism to exploit next.

  All these additional scheduling requirements can be achieved only through the scheduler.

# 12. Memory Management

The issue of memory management has been a very delicate one for many years in the area of programming languages implementation. This has driven research in this area towards implementation models capable of handling memory in an effective and efficient way. The WAM represents a good example of good balancing in memory usage; the abstract machine has been designed in such a way to make garbage collection a consequence of the execution process itself (e.g., backtracking allows direct recovery of memory used by the discarded parts of the computation).

This leaves two major questions open:

1. is memory management still an issue ?

2. are currently memory management techniques affecting performance ?

The answer to the first question is debatable. On one hand the technology trend has lead to architectures whose memory capacity go far beyond what are the needs of average real-life Prolog programs. This may allow for simpler memory management schemes with less sophisticated (and less expensive) garbage collection mechanisms. On the other hand, a careless memory management scheme may lead to an overuse of memory, which will itself negatively affect the performance of the system (by increasing the amount of virtual memory's page faults and introducing irregular cache behaviour). This is particularly true for parallel systems, where the amount of memory used tends to increase proportionally to the amount of parallelism exploited.

The considerations above can be synthesized in a simple answer to the second question: yes, memory management may have a profound impact on the system performance. In particular:

- good memory management, leading to a contained usage of memory can be positive in that it helps avoiding the side effects (e.g., virtual memory thrashing) mentioned above;
- too much emphasis in containing memory consumption can lead to considerable overhead in a parallel execution, which will end up slowing down the parallel execution (even those executions that do not create serious problems in memory usage).

The general observation is that a correct balance has to be found. Garbage collection is required but should not become an obstacle to efficiency. This is true, in particular, in the case of and-parallelism, where memory management is considerably more complex and garbage collection becomes a challenging and expensive task.

## 12.1 Memory Management in And-Parallelism:

As mentioned above, memory management becomes extremely complex in presence of and-parallelism, as illustrated in various works . The original proposals (e.g., RAPWAM ) were based on memory management schemes which were trying to mimic as close as possible memory management of the WAM. In particular, restrictions were imposed on the and-scheduler in order to maintain in the stacks the ``correct'' order--i.e., topmost objects on the stacks should be those that are going to be reclaimed first on backtracking. This, as illustrated from various experimental results, was leading to severe restrictions on the amount of parallelism exploited.

More recent models relax this restrictions, at the cost of more complex memory management schemes. And this cost becomes evident both in terms of memory consumption (more data structures are required in order to maintain the correspondence between physical and logical layout of the computation), and in terms of overhead (allocation, initialization, traversal, and removal of the new data

structures). The impact on the global efficiency can be considerable. Techniques have been studied to reduce these overheads . Furthermore, new models for and-parallelism are currently under considerations, where most of these additional operations are avoided, with the only drawback of having to introduce an explicit garbage collector.

In the rest of this section we will discuss the two main aspects related to memory management in and-parallel systems: representation of environments and trail management.

*Goal Stacking vs. Environment Stacking*

Since the earlier designs of Prolog engines, the implementors debated about the best machine organization to adopt in order to support the control behaviour of a Prolog execution. Prolog execution is essentially characterized by two phases:

1. *Forward Execution*: which corresponds to the application of SLD-resolution and the computation of the succession of resolvents of the initial query (management of *don't care* nondeterminism);
2. *Backward Execution*: which corresponds to the management of *don't know* nondeterminism--the set of mechanisms invoked upon occurrence of a failure in the current execution.

While the management of don't know nondeterminism is quite well-established, through the use of choice-points allocated on a proper control stack, the management of forward execution leaves various alternatives open. In particular, two major approaches have been considered and are currently used in various implementations:

1. *Environment Stacking:* this is the model which has been assumed in the previous sections. An environment is allocated whenever a clause is entered,

and used to maintain a (fairly implicit) representation of the current goals chain and to maintain the local environment of the clause.

Environment stacking has various advantages and appears to properly model the sequential pattern of execution. The use of environments guarantees a compact representation of forward execution, and allows incremental refinements of the environment during the execution (through *environment trimming*, a technique which allows to drop from the environment local variables which are not needed anymore).

2. *Goal Stacking:* in goal stacking, the system maintains an explicit representation of the goal list generated during the successive steps of resolution. The engine manipulates a *goal stack*. Each subgoal is represented by a data structure allocated in this stack and, at each moment, the current resolvent is represented by the succession of data structures present in the goal stack. In principle, no environments are created for the clauses, while each subgoal has its own arguments directly associated.

   Goal stacking has various advantages, like:

   1. by making explicit the structure of the current resolvent, it offers a more precise view of the status of the execution, which can be used for various purposes (e.g., it makes easier to implement different selection strategies);
   2. various optimizations can be applied more naturally (e.g., tail recursion optimization).
   3. as pointed out by Warren , goal stacking reduces the number of jumps in the code (good for pipelined architectures) and improves locality of the accesses to the code (reducing memory page faults).

On the other hand, goal stacking suffers of some drawbacks:

4. all the subgoals need to be created upon entering a clause--this work can get wasted in case of an early failure in the clause;
5. variables management becomes more complex. In particular for unbound variables--either frequent run-time checks are required to avoid dangling references or all of them need to be *globalized* (e.g., allocated on the heap), increasing memory consumption and making garbage collection more complex.

In most of the current sequential implementations, environment stacking has been preferred to goal stacking--this especially in view of the best management of variables offered.

On the other hand, switching to parallelism, and more specifically to and-parallelism, the situation becomes more complex. What is evident is that environment stacking becomes an obstacle in the exploitation of and-parallelism. First of all, and-parallelism requires the availability of an *explicit* representation of the resolvent (or, at least, of the part of the resolvent that is meant to be executed in and-parallel), in order to allow the various and-agents to identify the parallel work available, and take advantage of it. Using environment stacking, the structure of the resolvent is implicit in the code addresses stored in the environments and in the code itself--making the task of identifying the successive subgoals extremely hard and costly.

To gain efficiency, and-parallel systems need to introduce, at least up to a certain extent, a form of goal-stacking. Systems like ACE, &-Prolog, DDAS, APEX limit the introduction of goal stacking to those subgoals which can potentially be executed in parallel, while maintaining environment stacking to represent clauses' local environments and the sequential parts of the resolvent. Other systems, like the

parallel implementation of Parlog, extend the use of goal stacking to the whole execution. In various other systems (e.g. DDAS and CIAO Prolog ) the lack of an explicit goal chain representation leads to the need of recreating the sequence of subgoals through the use of special data structures properly linked--this is necessary to support the correct backtracking semantics. Nevertheless, the cost of maintaining these additional data structures (creation, update, traversal) seriously endangers the efficiency of the parallel execution, leading to actual slow downs for programs with particular execution behaviours (e.g., programs with heavy backtracking).

*Management of the Trail*

Even though memory management has been shown to be an important issue, it is opinion of the author that, as long as the implementation does not lead to a combinatorial explosion in memory consumption, efficiency of execution should not be sacrificed to reduce memory consumption or facilitate garbage collection. This is particularly true when dealing with parallel systems where

- overheads are introduced to support parallel execution;
- memory management becomes very complex--and, as a consequence, the techniques to recover memory and recollect garbage become more and more expensive.

When dealing with and-parallel systems, recovering memory upon backtracking is an awkward task. Backtracking can be realized in two different ways:

*(i). Private Backtracking:* each and-agent is allowed to backtrack only over the parts of the computation that it has generated. Let us consider the scenario in which a query of the form ←a & b & c is given, the agent $A$ has executed a and c, while agent $B$ has computed b. Assuming the use of a recomputation-based approach, backtracking will move from c, to b, and finally to a. $A$ will start the backtracking

activity, but it will stop once all the alternatives of c have been exhausted--not being allowed to backtrack over b, which is owned by $B$. Backtracking needs to be transferred to $B$.

This approach allows an easier recovery of memory (the agent backtracking owns the data structures, which can be immediately recollected), and it has been used in various systems, like the original &-Prolog and a preliminary version of ACE. On the other hand, the price paid for having an easier memory management is a considerably lack of efficiency during backtracking. Backtracking is not anymore a continuous activity, but is a sequence of *sequential* steps performed by different agents, with all the expenses incurred to ensure a proper synchronization and transfer of control.

*(ii). Public Backtracking:* every agent is allowed to backtrack over any memory area, independently from the actual owner. In the example above, if a new alternative is located in b, then the agent $A$will backtrack to it, even if b is owned by $B$. This method requires care in order to avoid conflicts (due to different agents backtracking on the same area)--which can be realized by adding simple locks and runtime checks upon failure. But, on the other hand, one agent will complete the whole backtracking, without any synchronization activity and without transferring the backtracking task between agents (as was happening in private backtracking). This makes the whole backtracking much faster (figure 12 shows an estimated comparison of speedups obtained using the two forms of backtracking).

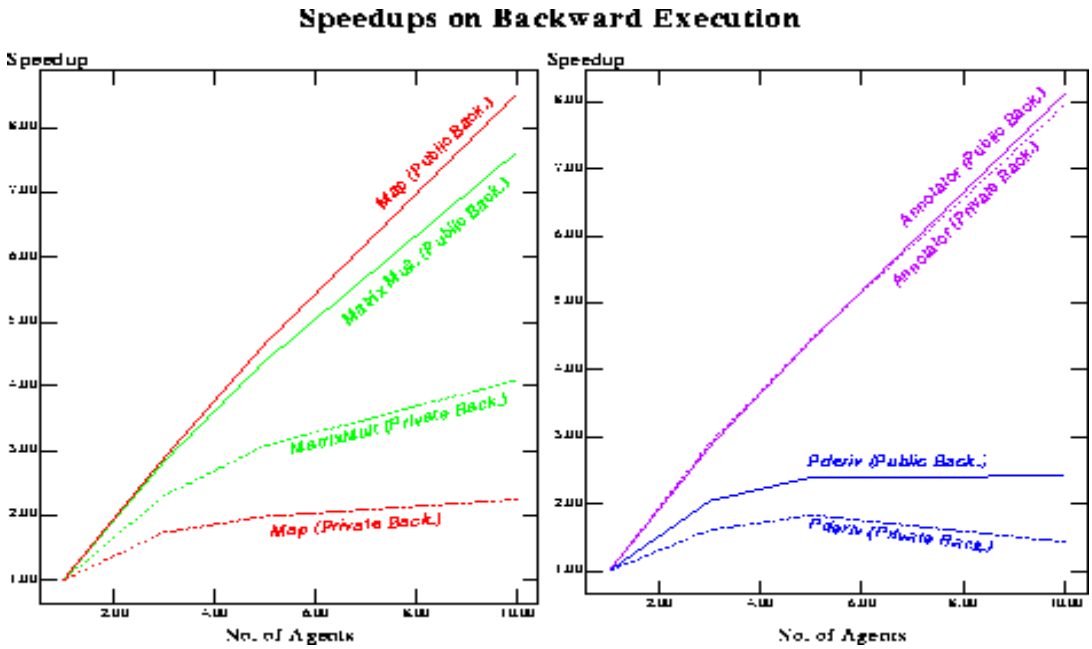



**Figure 12:** Comparison of Public and Private Backtracking.

The main disadvantage of public backtracking is the inability of performing garbage collection on the fly during backtracking--clearly an agent cannot recover memory belonging to another agent. On the other hand areas to be reclaimed can be marked during backtracking, and recovered in a successive, separate garbage collection phase.

The current version of ACE, and other systems, like DDAS, are based on this backtracking scheme.

The discussion above is useful to introduce another important issue in the implementation of and-parallelism. As we have seen, we can develop backtracking schemes where memory recovering is not affecting the system's efficiency. The natural question is then the following: which are the operations involved during backtracking that **must** be done and, on the other hand, which are those that can be avoided or delayed to a later stage of execution?

In the sequential models, backtracking represents the combination of the following activities:

- detection of unexplored alternatives in a choice point;
- detection of the backtracking points;
- removal of trailed bindings;
- recover of space on control stack, environment stack, and heap.

Of the four activities above, only three of them are strictly necessary, selection of backtracking points, extraction of alternatives, and removal of trailed bindings. While the first two steps can be realized efficiently and easily in an and-parallel system, the third point represents the main source of complexity on backtracking.

Identifying the bindings to be removed is straightforward in a sequential model, since there is a unique trail and there is a one-to-one correspondence between order of backtracking and order of the entries in the trail. This has the considerable advantage that, once the backtracking point has been detected, the set of bindings to be removed can be obtained in a single step--this thanks to the fact that a single piece of information is sufficient to detect the set of conditional bindings to remove.

The distributed nature of an and-parallel execution makes this whole process more complex: the trailed bindings are distributed in the trail stacks of the different agents, lacking the element that was allowing efficient untrailing in sequential system, i.e. the contiguity of the trailed bindings.

Most of the existing and-parallel systems use a memory organization similar to that of sequential systems, and are consequently obliged to *explicitly* compute at backtracking time the sequence of bindings to be removed. An improvement can be obtained by incrementally associating the parts of trail to the choice points, in such a way to facilitate their retrieval during backtracking.

Nevertheless, the perfect situation can be realized only by adopting a radically different memory organization, in which the contiguity of the trail is guaranteed. No models with this property have been proposed in the literature.

## 12.2 Memory Management in Or-parallelism:

The main issue to be tackled in the design of or-parallel systems is the development of an effective environment representation scheme, capable of associating to each or-branch the correct set of variable bindings. The two main approaches commonly used to solve this problem are stack-copying and binding arrays.

**Stack-Copying:** the stack-copying approach is characterized by a very intense memory usage. Since each or-agent has a whole copy of its branch, the same information appears repeated over and over in the stacks of different agents. Furthermore, in order to avoid repetition of computation, for each choice point belonging to the shared part of the tree (i.e., the part of the tree that has been copied by different agents) a shared frame has to be created, which stores information that are required by all the agents working on such choice-points (e.g., which are the alternatives still unexplored). Various issues can be raised regarding this model.

The first natural question is whether there is really the need of performing this large amount of copying. Copying involves all the data areas of the abstract machine (choice-point stack, environment stack, heap, and trail)--which translates to a considerable overhead during execution, considering also that during the copying operation both the agents involved (the agent copying and the agent the information are copied from) need to suspend their normal activity. Muse , the first system which introduced stack-copying, tries to improve the situation by:

- using incremental forms of stack copying, i.e. copying only the difference between the states of the two agents;

- avoiding copying information that will not be used (e.g., sequential choice points at the end of the branch);
- trying to parallelize as much as possible the copying activity, overlapping some of its steps.

On the other hand it seems intuitively possible to do better than this. Having distinct copies of certain data structures, like choice points, does not seem to have much sense, especially considering the fact that additional data structures (the shared frames) need to be introduced to create connections between the different copies of the same entity. On the other hand, sharing data structures which can be subject to updates during the execution can have the side effect of requiring mutual exclusion during accesses--which may become itself a source of overhead. It seems anyway possible to improve the situation by finding a proper compromise, in which information that are seldomly updated (e.g., choice points) are shared between computations, while those that either must be kept independent (e.g., conditional bindings) or are subject of frequent updates (e.g. trail) are duplicated. This can be interpreted as an intermediate point between two extremes, where one extreme is constituted by pure stack-copying and the other by pure sharing (e.g., Binding Arrays).

*Garbage Collection* is another complex issue in stack-copying. This may not be very apparent, since thanks to stack-copying the different agents can apply a standard WAM-like behaviour, carrying out local garbage collection on their own stacks. On the other hand, one of the key elements in the efficient implementation of stack-copying is the incrementality of the process: copying of the common state of the two agents is avoided. But if one of the two agents has performed garbage collection then finding out the common part of the two agents' state becomes very difficult. One possibility is to simply get rid of incremental copying in case of modification of the state due to garbage collection--quite undesirable due to the large amount of data

that may be copied. Alternatively, new garbage collection mechanisms need to be introduced, which fit in a better way in the stack-copying model (e.g. the segmented garbage collection mechanism used by Muse).

**Binding Arrays:** the Binding Arrays (B.A.) approach represents the symmetrical approach w.r.t. stack-copying regarding the amount of copying/sharing involved. In B.A. the maximal amount of sharing is enforced between the different agents. The only component should be kept separate across the different agents is the set of bindings to the conditional variables, which are kept in the private binding array of each agent. When one agent starts a new computation, it needs to perform a ``copy'' of the binding array in order to update its local state.

Various issues related to memory management emerge in the B.A. approach. First of all, the binding arrays need to be maintained in such a way to allow an efficient management. In the implementation of Aurora each agent is maintaining a local binding array which is updated whenever a new computation is started. Recent experiences by the group of Vitor Santos Costa have shown that using a unique binding array shared by the different agents may lead to improvements and various optimizations. This issue may become of fundamental importance when dealing with and/or-parallel systems (as described later on).

Furthermore, the current approach in updating the binding arrays consists of scanning the trail of the agent from which work is going to be taken, in order to detect the bindings that are missing from the current state. This can become quite time consuming--different techniques should be considered, like transforming this phase into a real ``copying'' operation, which can be performed quickly and holding the two agents synchronized for a shorter period of time.

Another problem related to memory management in the B.A. approach is garbage collection on the shared areas. Let us consider the situation exemplified in figure 13.

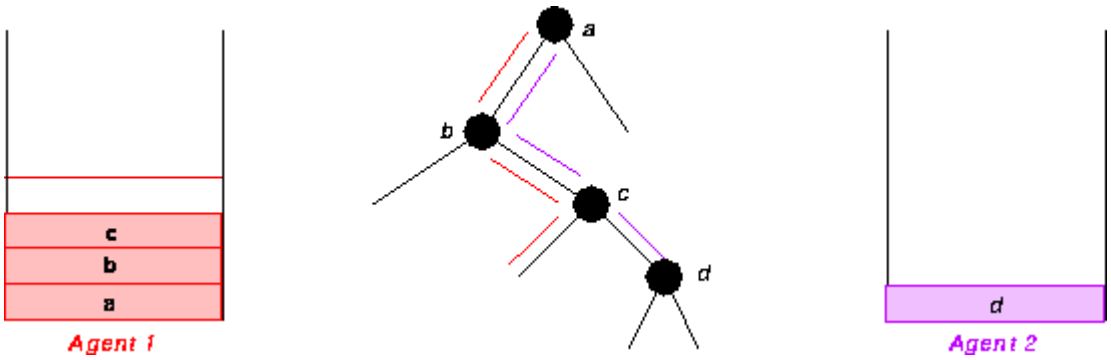



**Figure 13:** Garbage Collection in Binding Array

If Agent 1 backtracks over choice point *c*, it will not be allowed to reclaim space, since the same choice point is still needed by Agent 2. Agent 1 will be forced to allocate new data structures on the top of its stack, over *c*. If successively also Agent 2 backtracks, node *c* may become useless and a gap in Agent 1's stack will remain--this is called *ghost node* in Aurora's terminology. These gaps should be properly marked and recovered upon backtracking.

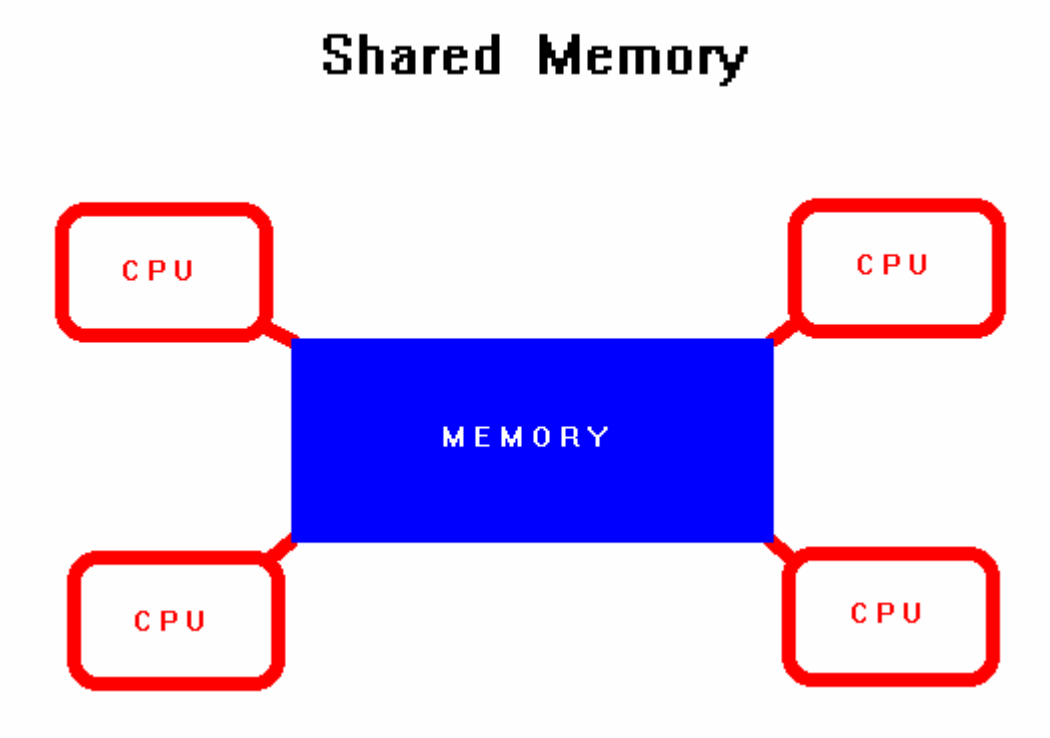
## 12.3 Memory Management in And/Or-Parallel Systems:

Memory management in an And/Or-parallel system becomes considerably more involved. Taking as reference a typical system exploiting and/or-parallelism based on the layering approach (e.g. ACE, Andorra-I), an organization scheme is visualized in figure 14(i). As we can see, the logical address space need to be partitioned between the different teams and agents.

## 12.4 memory architectures:

- The way processors communicate is dependent upon memory architecture, which, in turn, will affect how you write your parallel program
- The primary memory architectures are:
    - Shared Memory
    - Distributed Memory

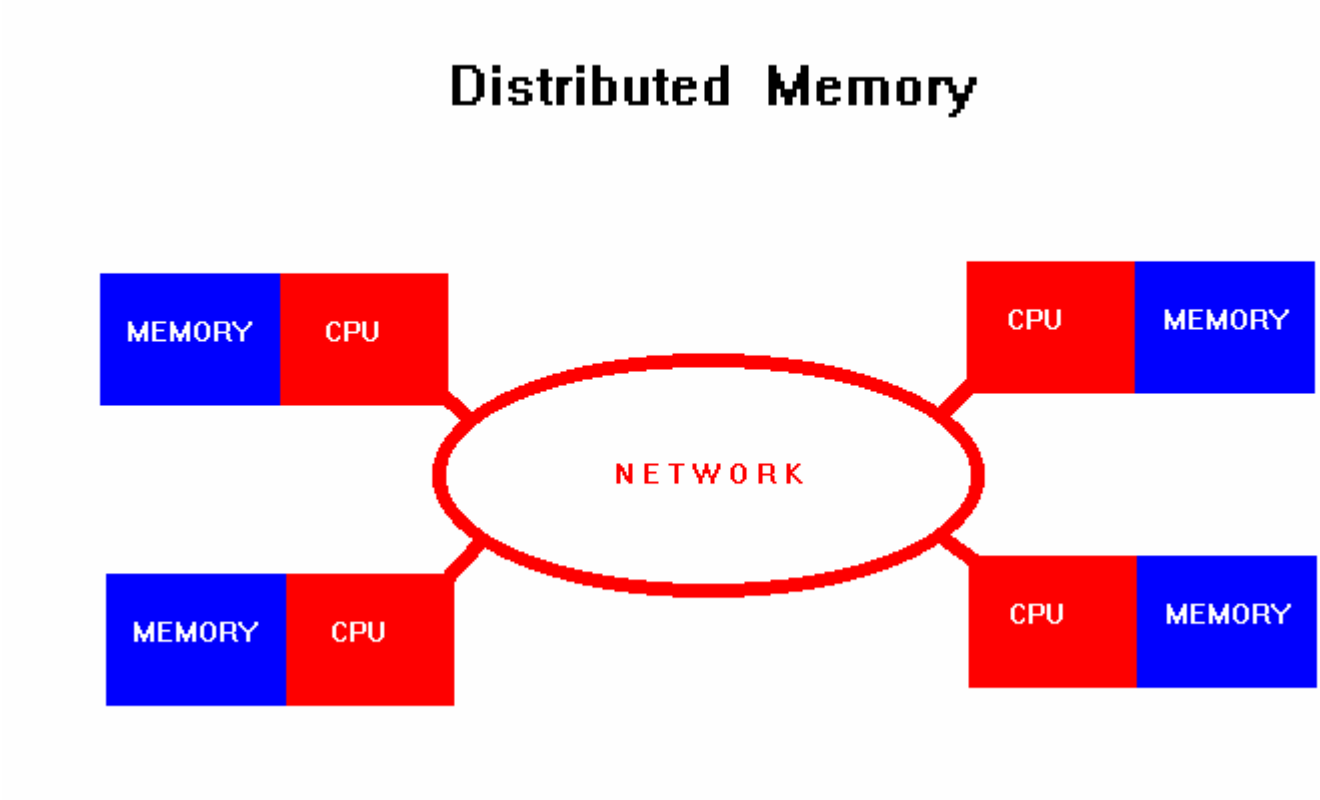
### 12.4.1 Shared Memory




- Multiple processors operate independently but share the same memory resources
- Only one processor can access the shared memory location at a time
- Synchronization achieved by controlling tasks' reading from and writing to the shared memory

- Advantages
    - Easy for user to use efficiently
    - Data sharing among tasks is fast (speed of memory access)
- Disadvantages
    - Memory is bandwidth limited. Increase of processors without increase of bandwidth can cause severe bottlenecks
    - User is responsible for specifying synchronization, e.g., locks

### 12.4.2 Distributed Memory




Multiple processors operate independently but each has its own private memory

- Data is shared across a communications network using message passing
- User responsible for synchronization using message passing
- Advantages
    - Memory scalable to number of processors. Increase number of processors, size of memory and bandwidth increases.
    - Each processor can rapidly access its own memory without interference
- Disadvantages
    - Difficult to map existing data structures to this memory organization
    - User responsible for sending and receiving data among processors
    - To minimize overhead and latency, data should be blocked up in large chunks and shipped before receiving node needs it.

# 13. Control Management

Managing the flow of control in the execution of Prolog programs in parallel systems is another complex issue. It can be articulated in three major subtopics: generalization of sequential control, scheduling, and management of order-sensitive calls.

It is relatively straightforward to generalize sequential control in or-parallelism, since the local computations carried on by the or-agents are basically standard sequential executions. Only local changes are required during backtracking, in order to switch to the or-scheduler whenever no more work is locally available. In and-parallelism things are slightly more complex. Forward execution is complicated by the need of performing a ``join'' at the end of the parallel computation, in order to combine the partial solutions produced and continue with a sequential execution. Backward execution becomes extremely more complex, due to the need of unwinding a computation which is arbitrarily spread across the stacks of different agents.

Scheduling and management of order-sensitive executions present some challenging problems.

## 13.1 Scheduling:

While in sequential systems the order in which the execution proceeds is predetermined and unique (and it may characterize the operational semantics of the language), in parallel systems a new issue emerges, related to the choice of the order in which the execution will develop. Typically, the quantity of parallel work

generated by the computation is larger than the actual number of executing resources (i.e., processors). This means that, whenever one agent will complete its activity, it will be forced to look for a new piece of work to execute. This activity of searching for new work is generally named *Scheduling*.

In principle, scheduling should have as main objective the selection of work which allows an efficient usage of the resources--especially processors--and to complete the execution in a time close to the earliest possible time.

As we have mentioned before, execution in a parallel system can be viewed as the process of visiting/developing a computation tree (and-tree, or-tree, or and/or-tree). Considering that the nodes of the tree represent the different instances of (parallel) work available, scheduling can be described as an activity which selects nodes in such tree. In particular, in the case of and/or-parallelism, the selection of a node will also imply the selection of a specific form of parallelism (and- or or-) to be assigned to the considered computing entity.

It is custom to distinguish two categories of ``work''. *Speculative* work is work that lies within the scope of a pruning operator, and it may ultimately be removed--i.e., any effort spent in its execution will be wasted. *Mandatory* work is work that is not speculative.

Various criterias can be considered in developing a scheduling algorithm . A first distinction can be made depending on the approach in which work is made ``available'' to the rest of the system:

- *demand-driven:* idle agents will look for work generated by active agents;
- *work-driven:* the agent producing work will take care of interrupting its execution and make such work ``public''.

Most of the or-parallel schedulers developed are demand-driven (this is the case of the various schedulers developed for Muse and Aurora)--or-parallel work will be detected by searching for choice points with unexplored alternatives in the subtrees generated by other or-agents. The few and-parallel systems available are mainly using work-driven scheduling, in which each agent generating a parallel call will take care of pushing the subgoals associated to the parallel call in a work queue/stack, from where other agents will pick them up for remote execution.

Another distinction that can be made is based on the frequency of invocation of the scheduler:

- *preemptive:* in this case an execution may be interrupted before its completion and the scheduler called to verify whether there is ``better'' work available. If the answer is positive, then the current execution is abandoned and the agent is moved to the better source of work.
- *non-preemptive:* an execution will not be interrupted by the scheduler until it is completed.

Preemptive scheduling has been introduced in the most recent or-parallel schedulers in order to deal with *speculative work*. With a certain frequence the execution is interrupted and the or-agents moved towards less-speculative work, i.e. work that has less probability of being pruned.

All the schedulers developed for and-parallel systems have been non-preemptive. The only cases of preemption emerges in the implementation of dependent and-parallel systems, where and-agents are moved to different subgoals whenever the current execution gets suspended waiting for a binding for a shared variable (for which the current execution is not a producer). Only recently the design of more effective scheduling for and-parallel systems, taking into account also the issue of speculativeness of work, has been started .

A further issue emerges when dealing with and/or-parallel systems. In these systems, the scheduling is not only involved with the selection of the best piece of work, but it must also decide which *kind* of work should be taken (either and- or or-). It has been shown that maintaining a fixed configuration in the teams of agents may not lead to the best possible performance. This justifies the need for a *top-level* scheduler (also known as *reconfigurer* in Andorra-I), whose only task is to reconfigure the teams as a response to variations in the distribution of or- and and-work in the computation. Principles for designing reconfigurer have been studied by Dutra .The presence of a reconfigurer is definitely a key issue in the effectiveness of an and/or-parallel system.

## 13.2 Order-sensitive Executions:

We have frequently referred in the previous sections to the issue of maintaining Prolog semantics within the parallel computation. In simple words, the aim we are trying to achieve is to have the parallel implementation ``behaving" exactly like a sequential one, in terms of solutions obtained, order in which they are produced, and general external behaviour of the execution.

A sequential implementation of Prolog is characterized by a depth-first, left-to-right visit of the computation tree. This fixes the order in which each subgoal is processed. During a parallel execution, special actions are required in those cases in which a parallel computation may lead to a different behaviour. We have seen examples of this in the case of dependent and-parallelism, where an unrestricted parallel execution may lead to different solutions, since the order in which the variables get bound can be different--and this can affect the final outcome. Sequential behaviour can be restored by introducing synchronization points in the parallel execution, used to guarantee that all the *order-sensitive* operations are performed in the correct sequence.

Clearly, the presence of order-sensitive operations leads to a degradation of parallelism (since we are forced to reintroduce a sequential component in the execution), and the components of the system in charge of deciding which part of the program will be run in parallel (e.g., schedulers) should be designed in such a way to take this into account--e.g., avoiding parallelizing parts of the program which are heavily ``sequential''. On the other hand, the presence of order-sensitive operations does not require a sequentialization of the whole execution involved--only the order-sensitive operations need to be sequentialized, while the rest of the code in between can be run in parallel without any concern.

First of all it is quite clear that only certain kind of order-sensitive predicates will create problems when occurring in a parallel computation:

- in *or-parallelism*, meta-logical predicates will not create any problem, while both side-effects and control predicates require proper action;
- in *and-parallelism* all the above mentioned predicates require special action. On the other hand, if the system is dealing only with independent and-parallelism, then meta-logical predicates will not affect the execution.

The simplest approach is refusing parallelization of parts of code which contain this kind of predicates. This option has been adopted in some of the earlier proposals (e.g., ROPM, AO-WAM, etc.), but it is unacceptable, as it will lead to a considerable loss of parallelism.

A slightly improved solution has been proposed by other systems, like PEPSys ,where the program is statically subdivided into sequential and parallel modules, and side-effects are only allowed in the sequential ones. But, again, the loss of parallelism may be considerable.

Taken for granted that order-sensitive predicates need to be allowed also within the scope of parallel computations, the systems should be supplied with special rules that guarantee that Prolog semantics will be respected.

To maintain Prolog semantics in presence of or-parallelism, an order-sensitive predicate should be executed only after all the preceding ones have terminated. Determining a-priori this condition is impossible (it is akin to deciding the halting problem). Some approximations can be developed. The most commonly used one (e.g., Muse, Aurora, Andorra-I) is to suspend the execution of the order-sensitive predicate until its branch is leftmost in the whole tree--i.e., all the branches on its left have already been completed.

Only few techniques have been considered for handling order-sensitive predicates in presence of and-parallelism . The basic idea is analogous to what described in the case of or-parallelism: the execution of an order-sensitive predicate needs to be suspended until the branch (in the and-tree) to which it belongs is the leftmost active one (i.e., all the branches on its left have completed their executions). The approaches referenced above differ depending on the way in which detection of leftmostness in the tree is performed.

The conditions under which it is possible to safely execute an order-sensitive predicate in presence of and/or-parallelism can be obtained by combining the respective conditions for or- and and-parallelism. This has been studied by Gupta and Santos Costa .Some improvements can be obtained by taking advantage of the different computational models (e.g., ACE proposed a slightly better algorithm for detecting executability of order-sensitive predicates ).

# 14. Compile Time Analysis

The purpose of compile-time analysis is to statically (e.g., by simply considering the syntax of the program) derive information that can be used to produce a better compiled code and, more in general, to improve program execution. Both data and control information may be derived and used to increase speed and reduce code size. The issue is even more relevant when dealing with parallelism: compile-time analysis allows to increase the degree of parallelism exploited, reducing at the same time the run-time costs of performing the parallelization.

Various techniques have been adopted to extract knowledge about the program during compilation. It must be observed that logic programming languages, being based on clear and well-defined mathematical semantics, allow a relatively easy development of tools capable of performing a semantically correct transformation of the program (something which is not always guaranteed in the case of procedural languages ).The nicely designed semantics of these languages and their tight coupling with their operational behaviour makes the task of collecting information about the program behaviour simpler and theoretically well-founded.

Most of the analysis algorithms used so far are instances of a general method usually indicated with the name *Abstract Interpretation* . Abstract Interpretation is based on ``executing" the program on a domain different from the Herbrand Universe, carefully selected in order to extract information about the behaviour of the execution.

## 14.1 Compile-time Analysis for Parallelism:

We have already mentioned in a preceding section some types of information that can be useful to collect at compile time, and the way in which such information can be applied to improve sequential execution of logic programming languages. In this

section we will concentrate on the possibilities of compile-time analysis (and more specifically abstract interpretation) for helping parallel execution of logic programming.

### 14.1.1 Or-parallelism:

compile-time analysis have been scarcely used in the context of or-parallelism. Nevertheless there are considerable advantages that could be obtained by having static knowledge of various properties of the program:

1. or-parallelism is exploited by exploring in parallel different alternatives emanating from a given choice point. Abstract interpretation may be able to supply information regarding the probability of success of certain alternatives. An alternative that would surely fail with a certain arguments pattern may be avoided. In the same way knowledge regarding the determinacy of certain choice points may allow the scheduler to have a better view of the distribution of the work in the or-tree, allowing better scheduling choices.
2. as we have described earlier, independent or-parallelism is considerably easier to implement. Abstract interpretation could be used to detect occurrences of this form of parallelism, by guaranteeing that the subgoal to which the choice point is associated does not access any conditional variable (e.g., the subgoal has only ground arguments). Related to the point above, knowledge regarding which variables are accessed along every path of the computation may allow the scheduler to assign to the same or-agent all the alternatives which may access the same conditional variables.
3. in various models implementing or-parallelism, whenever an or-agent steals some work from another agent, a phase is required to remove/reinstall bindings in order to reconstruct the agent's state for the new computation. Various information extracted through compile-time analysis may help in

this phase (e.g., allowing to skip this phase). As an example, knowledge of the modes of the arguments may allow to detect the freeness status of a variable, removing the need of performing trailing and untrailing, and leading to avoidance of the bindings removal phase.

4. in the stack-copying approach to or-parallelism, compile-time analysis may allow to reduce the amount of information copied. Freeness and aliasing information may, as already observed, remove the need of dealing with binding de-installation and, partially, with binding installation.

### 14.1.2 And-parallelism:

compile-time analysis and, more specifically abstract interpretation, has been heavily used in the context of and-parallelism. In the following we sketch some of the main situations in which compile-time analysis can be used to improve and-parallel execution.

1. independent and-parallelism relies on the knowledge that the subgoals considered for parallel execution are mutually independent, i.e. they are not going to affect each other's execution. As we have seen in a previous section, one of the best approaches to independent and-parallelism is based on the use of Conditional Graph Expressions. At run-time these conditions need to be evaluated in order to verify whether a parallel computation is possible. Even if these tests are very simple, their execution represents an overhead. On the other hand, an abstract interpretation process focused on the detection of possible variable sharings may allow to statically detect various cases in which certain variables are independent, which in turn translates to the possibility of removing many of the tests in the CGEs.

2. non-strict independent and-parallelism is based on allowing and-parallel execution of subgoals that may share variables but at most the rightmost of them will try to bind them. The problem of detecting at compile-time cases of non-strict independence has been shown to be a very difficult problem .This is due to the fact that this is not an a-priori condition, i.e. it cannot be expressed directly in terms of tests on the status of certain variables (i.e., run-time detection is unfeasible). Hermenegildo and others have shown how to perform this analysis using sharing+freeness information.
3. compile-time analysis may allow to identify even more cases of possible independent parallelism. For example, knowledge that different subgoals sharing a variable will actually produce the same binding for the variable (in absence, in the case of Prolog, of order-sensitive predicates) guarantees the possibility of running them in parallel.
4. various forms of dependent and-parallelism require a certain amount of compile-time analysis. For example:
    - DDAS requires the program to be annotated at compile-time, identifying the existing shared variables. Since each shared variable requires a very particular treatment, abstract interpretation could be used to identify variables that are only apparently shared (e.g., they represent cases of non-strict independence) and do not require any special treatment.
    - various models of dependent and-parallelism could be developed with simple implementation schemes as long as certain information, like modes of all the predicates, are available. Abstract Interpretation has been shown to be extremely effective in computing modes for many Prolog programs .
5. instances of the Basic Andorra Model (like in the Andorra-I system) requires knowledge regarding the determinacy of the subgoals to be run in and-

parallel. This information is approximated in the Andorra-I system (by verifying that the subgoal will match at most one clause)--and compile-time analysis is used to generate decision trees capable of making this check very fast. Use of Abstract Interpretation may allow to relax this approximation and recognize more cases of determinacy--increasing the degree of parallelism exploited.

6. various information which can be obtained at compile-time can be used to optimize the and-parallel execution in various ways. For example, knowledge about the determinacy of certain subgoals may allow, in the case of independent and-parallelism, to arbitrarily change the order of the subgoals. This allows to ``merge'' the subgoals executed by the same agent, reducing the overhead (especially during backtracking) .
7. in various and-parallel systems (e.g. Andorra-I) compile-time analysis is used to detect the presence of order-sensitive predicates and to insert proper annotations (e.g. sequential conjunctions) in order to allow the system to execute the program respecting the Prolog semantics.

### 14.1.3 And/Or-parallelism:

combined systems exploiting and/or-parallelism may take considerable advantage from information collected at compile-time. Clearly, all the advantages regarding single parallelism, as described above, still apply. Regarding more specifically and/or-parallel systems:

1. compile-time analysis may allow to spot points in which the different forms of parallelism will more likely occur. This may allow the simplification of the mechanisms used in those parts of the execution. Knowing, for example, that no and-parallelism will occur in certain alternatives of an or-parallel

choice point, allows to use standard or-parallel mechanisms in exploiting parallelism from such choice-point, avoiding all the overhead associated to the additional layer of parallelism.

2. compile-time analysis may supply information to drive the scheduler during the execution; static knowledge about distribution of work will allow to simplify memory management. For example stack-copying (if used to support or-parallelism) may take advantage of static knowledge about distribution of work to identify the areas to be copied.
3. if binding arrays are used in supporting or-parallelism, static knowledge can be employed to have an estimate about the number of conditional variables and their distribution--which can be useful in establishing the size of the binding arrays and/or the size of the pages in which it will be partitioned (using Paged Binding Arrays scheme).

## 14.2 Granularity Control:

One of the major issues in compile-time analysis, which is valid for both or- and and-parallelism, is *Granularity Control*.

Intuitively, logic programming may offer a great deal of parallelism, but the question is whether it is worth to actually exploit every instance of parallelism present in the execution. Exploiting an instance of parallel execution (i.e., allowing remote execution of a certain piece of computation) introduces an unavoidable amount of overhead. If this overhead is larger than the actual speedup produced by having a parallel execution, then the resulting parallel execution may turn out to be slower than the sequential one. For example, if we consider having a parallel execution with $n$ instances of work, each of size $T_p$, while the remaining part of the execution requires $T_1$, then a sequential execution will take $t_{seq} = T_1 + n \times T_p$ while a perfect

parallel execution will require $t_{par} = T_s + T_p + n \times overhead$ where *overhead* is the overhead required for acquiring and starting one remote execution. What we want is $\frac{t_{seq}}{t_{par}} \geq 1$. By simplifying the formulas we can obtain

$$T_p > \frac{n}{n-1} \times overhead$$

which gives an idea about how large should be the parallel computation in order to allow a speedup to occur.

This introduces the need of granularity control, that is allowing parallel execution to take place exclusively if the size of the parallel task is sufficiently large to overcome the overhead introduced. For example, in the case of and-parallelism, we would like to modify the parallel annotation in order to include a test on the size of the parallel tasks before spawning them. A modified annotations for the Fibonacci recursive clause will look as follows:

fib(N,Res) ← if N > 5 then

fib(N-1, R1) & fib(N-2,R2)

else

fib(N-1,R1), fib(N-2,R2)

endif

R is R1+R2.

where parallel execution is not allowed if the argument is below 5.

Introducing granularity control involves two activities: *(i)* measuring the actual cost of a piece of computation; and *(ii)* selecting a proper threshold under which no parallel execution will be allowed.

The second activity requires measuring the overhead involved with spawning a parallel computation. This threshold can be either fixed by the programmer, and tuned it by evaluating the performance of the system with different thresholds, or computed automatically by the system, e.g. by running both a sequential and a parallel program and comparing the performances obtained.

The first activity is the more complex one. It requires being able to estimate the cost of computations. It would be desirable to have a compile-time support in doing this. No compiler will ever be able to actually compute the exact cost of a computation, being this an undecidable property. On the other hand, what we really need is not the exact cost, but a reasonable upper bound. And this should be computed efficiently.

Various works regarding the use of terms size to guide the partitioning of programs for parallel execution have been presented in the context of functional programming , where the solution is relatively easier.

# 15. Further Issues

Exploitation of parallelism from logic programming languages needs to take into account the current trends in the evolution of this programming paradigm. In particular, some of the most relevant topics are the followings.

*Constraint Logic Programming:* logic programming is extended with capabilities to handle constraints over different domains. Originally introduced by Lassez and successively applied in a wide variety of frameworks ,constraint logic programming is quickly becoming a fundamental feature of most implementations of logic programming. The interactions between parallelism and constraints handling are currently an important topic of research and many problems are still open.

*Concurrency:* concurrency can be defined as the ability to express actions which are logically simultaneous. This term is often confused with the concept of parallelism (i.e., gain of speed-up by executing actions simultaneously), although the two notions are inherently different--concurrent systems need not be parallel, and parallel systems need not be concurrent.

Concurrency has been shown to be a powerful programming instruments, allowing to write programs capables of coroutining and reactive behaviour.

*Data Parallelism:* Another important classification of the forms of parallelism is the one which distinguishes *Control Parallelism* from *Data Parallelism* .

Broadly speaking, Control Parallelism (often indicated also as *MIMD* parallelism) arises when different operations (or functions, procedures) are applied to different data items in parallel. No a-priori synchronization exists between the different parallel computations--i.e., any form of synchronization needs to be explicitly programmed.

Data Parallelism, also known as *SIMD/SPMD* parallelism, arises when the same operation is applied in parallel to more than one different data-items. This involves a sort of lock-step execution pattern where in one step the desired operation is concurrently applied to the different data items.

In many cases data parallelism can be seen as a special instance of control parallelism, but the chief advantage of data parallelism is its more specialized nature, which makes it suitable to more efficient implementations, incurring in a considerably reduced amount of overhead.

An ideal parallel system should provide features for efficiently supporting data parallelism, identifying its occurrences and reducing the overhead as much as possible.

# 16. Conclusion

Parallel logic programming systems seem to be the next step towards faster implementations of logic programming systems. It brings several new challenges to the system implementor, one of them being the problem of efficiently distributing processors between and-work and or-work which arise naturally during execution of the programs. This is an important and hard problem to solve in the scope of parallel logic programming systems.

The main objective of this work was to give a solution to the problem of efficiently distributing processors between both kinds of work, in the scope of Andorra-I. As Andorra-I aims at exploiting parallelism implicitly, we were challenged to make the system distribute and and or-work in a way transparent to the user. Our work was of great practical importance for Andorra-I, because we added a key missing component, the reconfigurer, responsible for doing automatic distribution of and- and or-work among the processors.

Before our work, Andorra-I depended upon a fixed configuration of workers into teams, which had to be selected by the user. This caused three main drawbacks:

- The user had to manually adjust workers into teams to run the programs. This is a difficult and inconvenient task, especially in the light of the Andorra-I aim to exploit parallelism implicitly.

- In most cases, the user would be unlikely to be able to choose the best fixed configuration, leading to performance far from optimal.

• Even if the user could (somehow) choose the best fixed configuration, that would often produce performance below the best possible (with dynamic reconfiguration). In our work, we studied the problem of automating the distribution of workers between and-work and or-work, and showed the first known solution for this problem. Because we wished to obtain speedups close to the best possible speedup, we investigated two different reconfiguring strategies, the work-guided strategy and the efficiency-guided strategy. The work-guided strategy relies on assessing the amounts of work existing in the teams or in the choicepoints to decide when to redeploy a worker. The heuristic behind this strategy is to attract workers to places with more work to be done. Assuming we can accurately estimate the amount of and-work and amount of or-work available during the execution, this seems to be a reasonable heuristic.

The other strategy, the efficiency-guided strategy, makes decisions by monitoring the efficiency of a worker, where efficiency is measured as percentage of time doing useful work over execution time. As the objective of every parallel system is to keep processors busy most of the execution time, the efficiency-guided strategy also seems to be a reasonable heuristic.

In order to keep the reconfigurer overheads low, we designed very simple decision algorithms.

In order to control the frequency of reconfiguring we designed our strategies around guideline parameters obtained from a performance analysis of Andorra-I.

We tested both strategies with an extensive set of experiments. The results support the strategies proposed by showing that both strategies can be implemented efficiently, with small overheads. They confirm that both reconfiguration strategies achieve a substantial overall performance improvement over any plausible fixed team version of Andorra-I.

As regards whether the two reconfiguring strategies achieve optimal performance, results show that for programs containing a varying balance of and- and or-parallelism, both strategies achieve good speedups, better than the speedup achieved with any fixed configuration. For most applications, the work-guided strategy works better than the best fixed configuration, but for some cases it does not achieve this target. The efficiency-guided strategy seems to be a general solution for dynamically scheduling and- and or-parallelism in parallel logic programming systems, since it produces similar or better results than the ones produced with the best fixed configuration, for the benchmark set tested. To the best of our knowledge, Andorra-I is the first parallel logic programming system that exploits both dependent and-parallelism and or-parallelism. With The help of our work it became a fully practical system that runs real applications efficiently.

We can summaries the main contributions of our work as being:

Systems namely how to distribute and-work and or-work automatically among the processors.

- For Andorra-I, in particular, we added a key component to the system, the reconfigurer, to automate the distribution of and- and or-work. This piece of software removed from the user the difficult business of choosing a fixed configuration of workers to run a particular computation.

- We proposed two reconfiguration strategies, both of which showed good performance (better overall than any plausible fixed configuration and close to our target performance).

# 17. References


**1**

S. Abramski and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis-Horwood, 1987.

**2**

H. Ait-Kaci. *Warren's Abstract Machine: a Tutorial Reconstruction*. MIT Press, 1991.

**3**

Uri Baron Et Al. The parallel ecrc prolog system pepsys: An overview and evaluation results. In *FGCS '88*, pages 841-850, 1988.

**4**

K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.* MIT Press, 1990.

**5**

K.A.M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1991. 19(6):445-475.

**6**

K.A.M. Ali and R. Karlsson. Scheduling Speculative Work in MUSE and Performance Results. Technical report, SICS, 1993.

**7**

G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 1994.

**8**

S. Haridi B. Hausman, A. Ciepielewski. OR-Parallel Prolog Made Efficient on Shared Memory Multiprocessors. In *IEEE Symposium on Logic Programming*, 1987.

**9**

R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In G.Levi and M.Martelli, editors, *Proc. of the 6th International Conference on Logic Porgramming*. MIT Press, 1990.

**10**

T. Beaumont and D.H.D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *10th Int'l Conference on Logic Programming*, pages 135-149. MIT Press, 1993.

**11**

Benhamou. *Constraint Logic Programming*. MIT Press, 1993.

**12**

M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *5th Int. Conf. and Symp. on Logic Prog.*, pages 669-683. MIT Press, August 1988.

**13**

D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. Technical report, Facultad de Informatica, Universidad Politecnica de Madrid, 1994.

**14**

S.-E. Chang and Y. P. Chiang. Restricted AND-Parallelism Execution Model with Side-Effects. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 350-368, 1989.

**15**

K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. Research Report Doc 83/5, Dept. of Computing, Imperial College of Science and Technology, May 1983. University of London.

**16**

A. Colmeraur. The birth of prolog. In *Second ACM-SIGPLAN History of Programming Languages Conference*, 1993.

**17**

J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

**18**

J. S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Norwell, Ma 02061, 1987.

**19**

V. Santos Costa. Improving the binding arrays model. private communication.

**20**

V. Santos Costa. *Compile-time Analysis for Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, University of Bristol, August 1993.

**21**

V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proc. 3rd ACM SIGPLAN PPoPP*, 1990.

**22**

P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, pages 238-252, 1977.

**23**

J. Crammond. The Abstract Machine and Implementation of Parallel Prolog. Research report, Dept. of Computing, Imperial College of Science and Technology, July 1990.

**24**

S. Debray D. Gudeman, K. De Bosschere. jc: An Efficient and Portable Sequential Implementation of Janus. In *Procs. of the Joint International Conference and Symposium on Logic Programming*. MIT Press, 1992.

**25**

S. Debray. The SB-Prolog System, Version 2.3.2: A User's Manual. Technical Report 87-15, Dept. of Computer Science, University of Arizona, March 1988.

**26**

S. Debray and M. Jain. A Simple Program Transformation for Parallelism. In *Proc. of the 1994 Symposium on Logic Programming*. MIT Press, 1994.

**27**

S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, pages 207-229, September 1988.

**28**

S.K. Debray and N.W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5), 1993.

**29**

D. DeGroot. Restricted AND-Parallelism. In *Int'l Conf. on 5th Generation Computer Systems*, pages 471-478. Tokyo, Nov. 1984.

**30**

B. Demoen and G. Maris. A comparison of some schemes for translating logic to C. In *ICLP94 Post-conference workshop on Parallel and Data Parallel Execution of Logic Programs*. Uppsala University, 1994.

**31**

N. Drakos. Unrestricted And-Parallel Execution of Logic Programs with Dependency Directed Backtracking. In *Proc. of the International Joint Conference on Artificial Intelligence*, 1989.

**32**

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

**33**

H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

**34**

B. S. Fagin. *A Parallel Execution Model for Prolog*. PhD thesis, The University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/380.

**35**

I. Foster and S. Tuecke. *Parallel Programming with PCN*. Argonne National Laboratory, January 1993.

**36**

M. Carlsson G. Gupta, K.M. Ali and M.V. Hermenegildo. Parallel execution of prolog programs: a survey. *Journal of Logic Programming*, 1995. to appear.

**37**

J. Gabriel, T. Lindholm, E. L. Lusk, and R. A. Overbeek. A tutorial on the warren abstract machine for computational logic. Research Paper ANL-84-84, Argonne National Laboratory, June 1985.

**38**

G. Gazdar and C. Mellish. *Natural Language Processing in Prolog*. Addison-Wesley, 1989.

**39**

D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Language Systems*, 7(1), 1985.

**40**

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

**41**

G. Gupta and V. Santos Costa. Cuts and Side-effects in And/Or Parallel Prolog. *Journal of Logic Programming*, 1995. to appear.

**42**

G. Gupta, V. Santos Costa, and E. Pontelli. A Systematic Approach to Exploiting Implicit Parallelism in Prolog Using Shared Paged Binding Arrays. Technical report, Dept. of Computer Science, New Mexico State University, 1994.

**43**

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

**44**

G. Gupta, V. Santos Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Intergrating Dependent-, Independent-, and Or-parallelism. Technical report, University of Bristol, March 1991.

**45**

G. Gupta, M. Hermenegildo, and V. Santos Costa. And-or parallel prolog: A recomputation based approach. *New Generation Computing*, 11(3,4):297-322, 1993.

**46**

G. Gupta, M. Hermenegildo, and E. Pontelli. &ACE: A High-performance Parallel Prolog System. In *IPPS 95*. IEEE Computer Society, Santa Barbara, CA, April 1995.

**47**

G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93-109. MIT Press, 1994.

**48**

G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659-680, 1993.

**49**

G. Gupta and B. Jayaraman. And-or parallelism on shared memory multiprocessors. *Journal of Logic Programming*, 17(1):59-89, 1993.

**50**

Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In *Proc. 7th Int'l Conf. on Logic Prog.* MIT Press, 1990.

**51**

B. Hausman. Turbo Erlang: approaching the speed of C. In G. Succi E. Tick, editor, *Implementations of Logic Programming Systems*. Kluwer, 1994.

**52**

P. Van Hentenryck. *Constraint Handling in Prolog*. MIT Press, 1988.

**53**

M. Hermenegildo. Abstract Interpretation and its Applications. Tutorial 2, Advanced School on Foundations of Logic Programming, University of Pisa, Italy, September 1990.

**54**

M. Hermenegildo. Towards Efficient Parallel Implementation of Concurrent Constraint Logic Programming. In T. Chikayama and E. Tick, editors, *Proc. of ICOT/NFS Workshop on Parallel Logic Programming and its Programming Environments*. University of Oregon, 1994.

**55**

M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 Int'l Conf. on Logic Prog.*, pages 253-268. MIT Press, June 1990.

**56**

M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. Technical Report ACA-ST-537-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, November 1989.

**57**

M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237-252. MIT Press, June 1990.

**58**

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

**59**

M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Proc. 3rd ICLP, LNCS 225*, pages 25-40. Springer-Verlag, 1986.

**60**

M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40-55. Imperial College, Springer-Verlag, July 1986.

**61**

P.M. Hill and J.W. Lloyd. *The Goedel Programming Language*. MIT Press, 1994.

**62**

L. Huelsbergen and J.R. Larus. Dynamic Program Parallelization. In *Lisp and Functional Programming*. Association for Computing Machinery, 1992.

**63**

S. Janson. *AKL: a Multiparadigm Programming Language*. PhD thesis, Swedish Institute of Computer Science, 1994.

**64**

Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. Technical Report PEPMA Project, SICS, Box 1263, S-164 28 KISTA, Sweden, November 1990. Forthcoming.

**65**

L. Kale. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616-632. Melbourne, Australia, May 1987.

**66**

P.H.J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. MIT Press, 1989.

**67**

Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468-477. San Francisco, IEEE Computer Society, August 1987.

**68**

F. Kluzniak. Developing Applications for Aurora Or-parallel System. Technical Report TR-90-17, Dept. of Computer Science, University of Bristol, 1990.

**69**

A. Krall and U. Neumerkel. The Vienna Abstract Machine. In *Proc. of Symposium on Programming Languages Implementation and Logic Programming*. Springer Verlag, 1990.

**70**

Jean Louis Lassez and Joxan Jaffar. Constraint logic programming. In *Proc. 14th ACM POPL*, 1987.

**71**

T.G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*. Prentice-Hall, 1992.

**72**

Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123-1141. University of Washington, MIT Press, August 1988.

**73**

J. W. Lloyd. *Logic Programming*. Springer-Verlag, 1984.

**74**

E. Lusk and al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3), '90.

**75**

K. Marriot M. García de la Banda, M. Hermenegildo. Independence in constraint logic programming. In *Proc. of the 1993 International Symposium on Logic Programming*. MIT Press, 1993.

**76**

M. Carro M. Hermenegildo, D. Cabeza. Using Attributed Variables in the Implementation of Parallel and Concurrent Logic Programming Systems. In *Proc. of the International Conference on Logic Programming*, 1995.

**77**

C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 463-475. Imperial College, Springer-Verlag, July 1986.

**78**

K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. Technical Report ACA-ST-031-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, January 1989.

**79**

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

**80**

S.L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

**81**

E. Pontelli and G. Gupta. Incremental Exploitation of Parallelism in Prolog. In *International Conference on Parallel and Distributed Computing Systems*. ISCA, 1995.

**82**

E. Pontelli and G. Gupta. On the Duality Between And-parallelism and Or-parallelism. In *Proc. of Euro-Par'95*. Springer Verlag, 1995.

**83**

E. Pontelli, G. Gupta, and D. Tang. Determinacy Driven Optimizations of Parallel Prolog Implementations. In *Proc. of the Int'l Conference on Logic Programming 95*. MIT Press, 1995.

**84**

E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Efficient Implementation of Independent And Parallelism. Technical report, New Mexico State University, 1994.

**85**

V. Santos Costa R. Yang, I. Dutra. Parallel Execution of Prolog on Multiprocessor Architectures: Design of the Andorra-I system. Technical report, University of Bristol, 1991.

**86**

B. Ramkumar. Distributed Last Call Optimization for Portable Parallel Prolog Systems. *ACM Letters on Prog. Languages and Systems*, 3(1), 1992.

**87**

B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proc. NACLP'89*, pages 313-331. MIT Press, 1989.

**88**

B. Ramkumar and L.V. Kale. Machine Independent AND and OR Parallel Execution of Logic Programs. Part i and ii. *IEEE Transactions on Parallel and Distributed Systems*, 2(5).

**89**

J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23-41, January 1965.

**90**

P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming ?* PhD thesis, U.C. Berkeley, 1990.

**91**

Vijay Saraswat. *Concurrent Constraint Logic Programming*. MIT Press, Cambridge MA, 1989.

**92**

E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers*. MIT Press, Cambridge MA, 1987.

**93**

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

**94**

K. Shen. *Studies in And/Or Parallelism in Prolog*. PhD thesis, U. of Cambridge, 1992.

**95**

K. Shen. Improving the Execution of the Dependent And-parallel Prolog DDAS. In *Proc. of PARLE 94*, 1994.

**96**

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

**97**

K. Shen and M. Hermenegildo. A Flexible Stack Memory Management Scheme for Non-Deterministic, And-parallel Execution of Logic Programs. Technical report, Facultad de Informatica, U. P. Madrid, 28660 Boadilla del Monte, Madrid, Spain, January 1993.

**98**

B. Shriver and P. Wegner. *Research Directions in Object-oriented Programming*. MIT Press, 1987.

**99**

R. Sindaha. The Dharma Scheduler -- Definitive Scheduling in Aurora on Multiprocessor Architecture. In *Proc. of ILPS'93*, 1993.

**100**

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

**101**

V.S. Sunderam. Pvm: a framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4), 1990.

**102**

P. Szeredi. Exploiting or-parallelism in Optimization Problems. In *Proc. of Joint International Conference and Symposium on Logic Programming*. MIT Press, 1992.

**103**

D. Sekita T. Chikayama, T. Fujise. A Portable and Efficient Implementation of KL1. In *ICOT/NSF Workshop on Parallel Logic Programming and its Programming Environments*. University of Oregon, 1994.

**104**

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

**105**

P. Tarau. The BinProlog Experience: Implementing a High-performance Continuation Passing Prolog Engine. Technical report, Département d'Informatique, Université de Moncton, 1995.

**106**

A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. Technical report, Association for Logic Programming, June 1990.

**107**

A. Taylor. *High-performance Prolog Implementation*. PhD thesis, Basser Dept. of COmputer Science, University of Sydney, 1991.

**108**

H. Tebra. Optimistic And-Parallelism in Prolog. In *PARLE 87*. Springer Verlag, 1987.

**109**

E. Tick and D. H. D. Warren. Towards a Pipelined Prolog Processor. *New Generation Computing*, 2(4):323-345, 1984.

**110**

E. Tick and X. Zhong. A Compile-time Granularity Analysis Algorithm and its Performance Evaluation. *New Generation Computing*, 11, 1993.

**111**

P.C. Treleaven. *Parallel Computers: Object-oriented, Functional, Logic*. J. Wiley & Sons, 1990.

**112**

K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140-156. MIT Press, Cambridge MA, 1987.

**113**

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

**114**

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

**115**

D. H. D. Warren. The Extended Andorra Model with Implicit Control. ICLP'90 Parallel Logic Programming workshop.

**116**

D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

**117**

D.A. Watt. *Programming Languages Concepts and Paradigms*. Prentice-Hall, 1990.

**118**

H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *IEEE Int'l Symp. on Logic Prog.*, pages 436-448, 1987.

**119**

M.J. Wise. *Prolog Multiprocessors*. Prentice-Hall, 1986.

**120**

O. Wolfson and A. Ozeri. A New Paradigm for Parallel and Distributed Rule Processing. In *SIGMOD Int'l Conf. on Manag. of Data*. ACM, 1990.

**121**

O. Wolfson and A. Silberschatz. Distributed Processing of Logic Programs. In *SIGMOD Int'l Conf. on Manag. of Data*. ACM, 1988.

**122**

T. Conway Z. Somogyi, F. Henderson. The Implementation of Mercury, an Efficient Purely Declarative Logic Programming Language. In P. Tarau K. De Bosschere, B. Demoen, editor, *ILPS'94 Post-conference Workshop on Implementation Techniques for Logic Programming Languages*, 1994.

**123**

H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.

**124**

J. Chassin de Kergommeaux and P. Codognet, Parallel Logic Programming Systems, ACM Computing Surveys, Vol. 26, No, 3, September 1994

**125**

Inês de Castro Dutra, February 1995, Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System,

**126**

Inês de Castro Dutra and Claudio F. R. Geyer, Parallelism in Logic Programming

**127**

http://cobweb.ecn.purdue.edu/~hankd/CARP/XPC/paper.html#hypertext_index

**128**

http://www.cs.wisc.edu/~dyer/cs540/notes/fopc.html

**129**

http://www.cs.nmsu.edu/~epontell/adventure/node8.html#SECTION00023000000000000000

**130**

http://www.cs.nmsu.edu/~epontell/adventure/paper.html

**131**

www.commscope.com/andrew/eng/partner.../andrew_inc/

**132**

en.wikipedia.org/wiki/Andrew_Project

**133**

www.springerlink.com/index/79081810531087N8.pdf

**134**

en.wikipedia.org/wiki/Andrew_File_System

**135**

http://www.linuxjournal.com/article/2913?page=0,0

**136**

http://asg.andrew.cmu.edu/andrew2/ANDREWII/LeongCursorMay92.html

**137**

http://asg.andrew.cmu.edu/andrew2/ANDREWII/requirements.html

Thanks