# On the Duality Between Or-parallelism and And-parallelism in Logic Programming

Enrico Pontelli & Gopal Gupta

Laboratory for Logic, Databases, and Advanced Programming
New Mexico State University
{epontell,gupta}@cs.nmsu.edu

**Abstract.** *Or-parallelism* and *And-parallelism* have often been considered as two distinct forms of parallelism with not much in common. The purpose of this paper is to highlight the inherently dual nature of the two forms of parallelism and the similarities that exist between them. The dualities and similarities observed are then exploited for gaining new insights into the design, implementation, and optimization of and- and or-parallel systems. The ideas developed in this paper are illustrated with the help of ACE system—a parallel Prolog system incorporating both and- and or-parallelism.
**Keywords:** Prolog, And-parallelism, Or-parallelism, Optimizations.

## 1 Introduction

Logic Programming is a declarative programming paradigm that is becoming increasingly popular. One of the distinguishing features of logic programming languages is that they allow considerable freedom in the way programs are executed. This latitude permits one to exploit parallelism implicitly (without the need for programmer intervention) during program execution. Indeed, three main types of parallelism—or-parallelism, independent and-parallelism, and dependent and-parallelism—have been identified and successfully exploited in logic programs. Parallel Execution of Logic Programs has been an active area of research since a decade [18, 8]. A great deal of effort has been spent in recent years in the design and development of parallel implementations of Logic Programming systems. However, parallel logic programming systems that can *efficiently exploit all* (profitable) forms of parallelism present in Logic Programs still elude us. This difficulty in combining the major forms of parallelism has been addressed in various ways—by introducing various forms of *restrictions* (e.g., allowing and-parallel execution of only deterministic subgoals as in the Andorra Model), by developing very complicated new mechanisms in which different forms of parallelism can co-exist to a certain extent, etc. The purpose of this work is to show that or- and and-parallelism are not two totally distinct forms, rather they *share* common traits. The utility of this observation is that mechanisms and methodologies that have been developed for one form of parallelism can be *re-used* for the other, suggesting new ways to implement/optimize individual implementations of or- and and-parallelism, and reducing the gap between the two forms of parallelism,thus simplifying the task of *efficiently* combining them in a single framework.

## 2 Parallelism in Logic Programming

Two forms of parallelism are typically identified in Logic Programming: or-parallelism and and-parallelism. Or-parallelism (ORP) arises when multiple rules define some relation and a call unifies with more than one rule head—the corresponding bodies can then be executed in *or-parallel* fashion. Or-parallelism is thus a way of efficiently searching for solution(s) to a goal. In principle, or-parallelism should be easy to implement, since the different branches of the parallel tree are actually computing different solutions *independently* of each other. This is only partially true, because multiple branches that are trying to compute different solutions to a goal share that goal's environment and may change it. The effect of one branch on the goal's environment must be shielded from the other branches. In a sequential implementation a single environment is sufficient for each clause, because the different bindings for a variable in a goal produced by different matching clauses are generated sequentially (through backtracking) and a single location for each variable is sufficient. In an or-parallel system this is not true anymore: different alternatives are concurrently explored and they may produce distinct bindings for the same variable (such variables are termed *conditional*). The problem of multiple environment management has to be solved by devising a mechanism where each parallel thread has a private area where it can store the locally produced bindings for the conditional variables. Typical solutions to this problem are: *(i)* creating distinct copies of the environment for each parallel thread (like in the Muse system); or *(ii)* maintaining private arrays/hash-tables in which the local bindings for conditional variables are stored (like in the Aurora system).

And-parallelism (ANDP) is generated by the parallel execution of a conjunction of subgoals. Two major classes of and-parallelism are typically identified, depending on whether and-parallel execution of subgoals accessing the same unbound variables is permitted or not: *(i) independent and-parallelism (IANDP)*: arises when conjunctive subgoals that do not share any unbound variables are executed in parallel. During IANDP no synchronization is required between parallel threads (as in &-Prolog [12] and ACE [9]); *(ii) dependent and-parallelism (DANDP)*: arises when conjunctive subgoals that share unbound variables are executed in parallel. DANDP can be further divided into two subclasses: *(a) restricted DANDP*: in which only subgoals satisfying certain conditions are allowed to execute in parallel (like in the Basic Andorra Model [2] and in non-strict IANDP [11]); *(b) general DANDP*: in which arbitrary subgoals can be executed in parallel (like the *Extended Andorra Model* [20], the *DDAS* [19], and in various committed-choice languages [18]). Whatever the model, and-parallel execution can be divided into 3 phases: **(i)** *ordering phase*: deals with the detection of dependencies among subgoals; **(ii)** *forward execution*: deals with the selection of subgoals and execution of the various resolution steps. The set of conjunctive subgoals that are executed in and-parallel is termed a *parallel call* (or a *parcall* for brevity); the corresponding descriptor that is placed in the stack when a parcall is made is called a *parcall frame*. Part of the forward execution phase is related to the scheduling of the subgoals among the active processors and, in the case of DANDP, the management of dependencies between subgoals; **(iii)** *backward execution*: deals with the sequence of steps taken on failure or when a new solution needs to be found. It requires extension of backtracking to the case of parallel execution.

## 3 Or-parallelism *vs* And-parallelism

In this section we draw a comparison between or-parallelism and and-parallelism. Table 1 compares and contrasts items of interest for the two forms of parallelism. The first two lines of the table relate and- and or-parallelism. Thus, while or-parallelism is obtained by parallel execution of *multiple alternatives in a choice point*, and-parallelism is obtained by executing *multiple subgoals in a parcall*. In both cases the parallel threads originate from a data structure containing a certain number of executions (choice points vs. parcall frames). Likewise, or-parallel execution produces multiple bindings for conditional variables, while and-parallel execution may do the same for shared variables. In both cases, during the execution of parallel threads, common variables can be concurrently accessed and/or bound. This creates possible *dependencies* that need to be resolved by adopting proper synchronization techniques (this issue is considered in section 4 and 5.3).

The duality between ORP and ANDP can be noted immediately in the way in which alternative executions are handled, since in ORP we are looking for *one* alternative to succeed (i.e., $\exists$), while in ANDP we are forced to verify that all the alternative executions are successful (i.e., $\forall \equiv \neg\exists$). The duality principle can also be seen in the way in which schedulers have been designed for the two kind of systems. In ORP a lot of effort has been spent in order to obtain *smart* schedulers, capable of selecting the alternative that is more likely to succeed as soon as possible; in ANDP the typical schedulers are quite simple (e.g., just push and pop and-parallel subgoals on goal stacks) due to the fact that ($\forall$) all the alternatives need to be tried. Nevertheless, as we will see in a later section, the duality noted above suggests ways to improve ANDP scheduling (section 5.2). This also suggests the possibility of managing ANDP work in an "ORP" way (section 5.1), leading to interesting improvements. Also, the dual nature of ORP and ANDP can be used to develop new approaches for dealing with dependent parallel executions (section 5.3).

| **Characteristic** | **ORP** | **ANDP** |
|---|---|---|
| *object of parall.* | alternatives in a choice point | subgoals in a goal |
| *dependency* | on conditional variables | on shared variables |
| *additional work* | on failure | on success |
| *speculativeness* | on success | on failure |
| *continuation* | independent computations | join of computations |

**Table 1.:** Comparison between ANDP and ORP

The duality of or-parallelism and and-parallelism has been only indirectly investigated in the past. For example, it has been employed for *translating* or-parallelism to and-parallelism and vice-versa [3, 4]. The idea is to exploit both forms of parallelism, while implementing only one. This approach has its own pros and cons. On one hand the burden of developing complex mechanisms to integrate different forms of parallelism can be avoided and the other form of parallelism obtained effortlessly. On the other hand, such compile-time translations produce poor performance due to the extreme overhead introduced by the presence of an additional interpretation layer.

## 4 Applying And-parallel Concepts to Or-parallelism

In this section we discuss how the concepts developed for and-parallelism can be applied to or-parallelism. As we have seen, for each form of parallelism we can identify various “degrees” of implementation, depending on how strong the restrictions imposed are on the parallelism allowed. This suggests a possible classification of parallelism (independently from whether it is or- or and-), into three main classes: *(i)* **Independent Parallelism**: it is characterized by the fact that parallelism is allowed exclusively between threads which do not require any kind of synchronization during their execution, i.e. threads which are *independent*; *(ii)* **Restricted Dependent Parallelism**: this class consists of systems that allow parallel execution of threads which may have dependencies but will never produce any *solvable* conflict on them (i.e. an actual conflict corresponds to a failure of the computation). Interactions between threads are limited but require some care at the implementation level (e.g., use of locks); *(iii)* **Dependent Parallelism**: arbitrary dependent parallel computations are allowed. The first thing that we can notice is that these classes are in increasing order of complexity and **Independent** $\subseteq$ **Restricted Dep.** $\subseteq$ **Dependent**. This increasing order of complexity of control parallelism, though quite general, is inspired from the different types and restrictions of and-parallelism that have been considered in the literature (as mentioned in section 2). It is quite interesting to consider how this hierarchy of control parallelism applies to *or-parallelism*: **(1) Independent Parallelism**: this instantantion of or-parallelism is characterized by the independence of its parallel threads—which translates to allowing the parallel execution of alternatives in a choice point only if they do not access any unbound conditional variable. This may occur in various situations (e.g., predicates performing tests on the validity of their inputs, etc.). **(2) Restricted Dependent Parallelism**: this represents another type of or-parallelism in which parallel execution is only allowed between alternatives which may access conditional variable, but for which conflicting bindings will not be produced. This model of parallelism requires implementation mechanisms slightly more complicated, but still considerably simpler than those required for full ORP. **(3) Dependent Parallelism**: exploitation of full ORP. This is the model of ORP typically taken into consideration and requires complex mechanisms to deal with environment management. Although the independent and restricted forms of or-parallelism may appear to be too strict, it is worth taking them into account. First of all their implementation is extremely simple and could be realized with almost no overhead; independent or-parallelism requires only the ability of identifying the choice-points with work available—no other support for parallelism is required (e.g., no new environment representation schemes). In addition, the ability of identifying (e.g., through static analysis) these forms of or-parallelism may offer the chance of optimizing existing implementations (e.g., removal of binding array's entries in Aurora or removing the binding installation process in Muse for parts of the or-parallel computation that are determined to be independent or restricted).

Actually, independent and restricted and-parallelism have found applications in deductive databases where programs that submit to independent or-parallel execution are called *uniformly decomposable* [6]. Considerable research has been done in deductive database community for identifying uniformly decomposable programs [21, 6, 7].

## 5 Applying Or-parallel Concepts to And-parallelism

### 5.1 Choice Points vs. Parcall Frames

In ORP parallel work is represented by alternative subgoals that can be derived by using the different matching clauses associated with a certain choice point. In ANDP parallel work is represented by subgoals associated to a parcall frame (or a similar data structure). Given this similarity ($ChoicePoint \cong ParcallFrame$), the mechanisms needed to exploit ORP differ from those of ANDP exclusively because of the duality between ORP and ANDP, since
*(1)* scheduling in ORP is driven by "failure" ($ORP \equiv \vee$) (i.e., if a branch fails, search must continue with the next branch, else the search may stop since a solution has been found) while in ANDP it is driven by "success" ($ANDP \equiv \wedge$) (i.e., if a goal succeeds, search must continue with the next goal, else the execution may stop since the conjunction has failed.
*(2)* since $ORP \equiv \vee$, then $Success_{ORP} \equiv \bigvee_{Alternatives} Alternative$ while $ANDP \equiv \wedge$ and this means $Success_{ANDP} \equiv \bigwedge_{Alternatives} Alternative$ (where alternatives in the context of ANDP mean subgoals of a parallel conjunction). This translates to the fact that in ORP, whenever one of the alternatives succeed there is no need to verify the success of the others, while in ANDP a final "join" is required in order to determine the global consistency of the computation[1].

Nevertheless, the two differences mentioned above do not affect the actual phase of generation of the parallel threads—and this opens various possibilities in terms of importing ORP's implementation techniques to ANDP systems. In particular, management of choicepoints in ORP systems is typically quite efficient, while management of parcall frames in ANDP systems is extremely overhead prone. The following three subsections present ways that have been inspired by ORP to make management of parcall frames more efficient and reduce the overhead of exploiting and-parallelism.

| *Goals executed* | ACE Agents 1 | | | 3 | | | 5 | | |
|---|---|---|---|---|---|---|---|---|---|
| | Unopt. | Opt. | % | Unopt. | Opt. | % | Unopt. | Opt. | % |
| fibonacci | 3692 | 3643 | 1% | 1353 | 1313 | 3% | 833 | 820 | 2% |
| takeuchi | 1880 | 1821 | 3% | 677 | 657 | 3% | 428 | 408 | 5% |
| hanoi | 2591 | 2514 | 3% | 1000 | 979 | 2% | 660 | 636 | 4% |

**Table 2.:** Improvements avoiding pushing subgoals

*(i) Elimination of Goal-stack:* In ORP the or-parallel work is represented by a choice point, and such work is "made public" by notifying the existence of the choice point to the other processors working in the system. Each idle worker will access the *public* choice points and extract work from there. This approach has

[1] Although in ORP some synchronization points may be required to implement side-effects.

various advantages; one of the most important is that the cost of extracting the work from the choice point and initializing the proper information for parallel execution is spread between the different processors.

In the design of many ANDP systems (e.g. &-Prolog, DDAS), instead, the processor generating the parcall frame is also in charge of creating a *goal frame* for each and-parallel subgoal, initializing it with all the information required to allow its remote execution, and pushing it on a *goal stack*. The first level of optimization that we suggest is to avoid generation of a goal frame for each and-parallel subgoal. Instead, we adopt a mechanism in which the parcall frames containing unexplored alternatives (i.e. untried conjunctive subgoals) are "made public" and visible to all the processors (as happens for choice points in ORP), allowing each idle processor to access one of these parcall frames and extract an unexecuted subgoal. This simple optimization, of treating parcall frames similarly to choicepoints, and not creating goal-stack entries has been implemented in the current version of the ACE system and the results are presented in table 2. The measured improvements vary on average from 1% to 5%, and are more pronounced when the computation generates a considerable number of and-parallel subgoals. This is the case of Takeuchi, which shows better results due to the larger number of parallel subgoals per parcall frame.

| *Goals executed* | ACE Agents | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | 3 | | | 5 | | |
| | Unopt. | Opt. | % | Unopt. | Opt. | % | Unopt. | Opt. | % |
| fibonacci | 3643 | 3643 | 0% | 1313 | 1284 | 2% | 820 | 807 | 2% |
| takeuchi | 1821 | 1821 | 0% | 657 | 637 | 3% | 408 | 386 | 5% |

**Table 3.:** Distribution of descriptors creation

*(ii) Distributed Allocation of Goal Slots:* In a typical ANDP system each goal being executed in and-parallel has a *descriptor* associated with it. This descriptor is termed a *slot* in the &-Prolog and in the ACE systems. Descriptors are part of the parcall frame to which the corresponding subgoals belong, and they are generated together with the parcall frame. Borrowing ideas from or-parallel implementation we can improve this model: the task of creating the descriptors can be distributed between the different processors, i.e., each processor creates the descriptors for the goals it executes. This allows a better distribution of the overhead and an improvement in execution time. Some results of this optimization applied to ACE are reported in table 3. Again an average improvement of 3–5% is observed.

*(iii) Parallelism on Demand:* As we mentioned earlier, one of the characteristics of ORP is that each processor takes care of performing all the operations necessary to support the execution of the selected alternative. This implies that no parallel overhead (i.e., overhead spent to support exploitation of parallelism) will be incurred if no parallelism is exploited. This is hardly true for ANDP systems; in models like &-Prolog, ACE, DDAS, etc. overhead will be incurred even if the whole parallel call is actually executed by a single processor (since the creation of the parcall frame and other bookkeeping operations are still present). The desire is to allow ANDP to work with minimal overhead, especially when a parallel conjunction is executed sequentially due to lack of processors. This is in fact what happens in ORP, where sequential execution of alternatives in a *parallel* choice point is realized through ordinary backtracking, with

almost no overhead. We term the phenomenon in which overhead is incurred only if parallelism is exploited, and no parallel overhead is incurred otherwise as *Parallelism on Demand*. For ANDP, part of this has been realized in the previous optimization, by distributing the task of creating a slot (i.e., and-parallel subgoal descriptor) to the processor which picks up the subgoal for execution. Nevertheless, this is far from being sufficient. In fact, even allowing distribution of this overhead, we still have not solved the problem of removing the overhead during sequential execution of parallel subgoals (because the slots will still be needlessly allocated even though the execution is sequential). There has recently been some work in this direction [16, 13]. The proposal that come closest to the ideal "parallelism on demand" behaviour mentioned above is an optimization implemented in the current version of the ACE system. This optimization, termed *Processor Determinacy Optimization* (and described in more details in [16]) by us, is inspired by the following principle: *"two (potentially) parallel pieces of computation that are executed sequentially and in the proper sequential order should incur no parallel overhead"*. In the current implemented version of this optimization, whenever a subgoal is selected for execution a check is performed to verify whether the new subgoal is the "sequential" successor of the one previously executed; if the test is successful then most of the parallel overhead is avoided. Figure 1 shows schematically the behaviour of the system during this optimization. The performance figures for this optimization are quite impressive, especially considering this only approximates *Parallelism on Demand*.

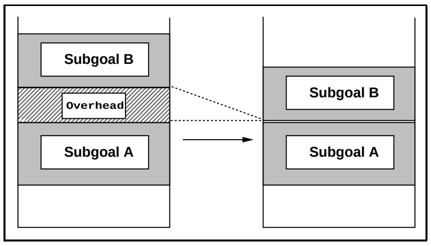


**Fig. 1.:** Det. Proc. Optimization

| Goals executed | ACE Execution Unopt. | Opt. |
|---|---|---|
| *Bt(0)* | 1461 | 1391 (5%) |
| *pmatrix(30)* | 5598 | 5336 (5%) |
| *listsum* | 2333 | 2054 (12%) |
| *hanoi* | 2183 | 1790 (18%) |
| *takeuchi* | 2366 | 1963 (17%) |

| Goals executed | ACE Execution Unopt. | Opt. |
|---|---|---|
| *Bt(0)* | 120 | 60 |
| *Deriv(0)* | 174 | 87 |
| *pmatrix(30)* | 1798 | 899 |
| *listsum* | 3000 | 1500 |
| *takeuchi* | 3558 | 2372 |
| *hanoi* | 4094 | 2047 |

**Table 4.:** Execution times (ms.) and memory usage (no. of structures)

The results obtained on some benchmarks in terms of improvement in sequential execution speed and in memory consumption (stack usage) are indicated in table 4. Work is in progress to obtain ideal implementation for parallelism on demand (avoiding allocation of parcall frames altogether in sequential executions).

### 5.2 Scheduling Related Issues

In most of the work on building and-parallel systems, the issue of scheduling is dealt with in a naive way. This is mainly due to the fact that ANDP systems are actually computing conjunctions of literals and, as such, they need in principle to execute *all* the subgoals associated to each parallel call. This is dual to the ORP case, where we are usually looking for *one* solution, and this requires success of at most one alternative out of each choice point built during the computation. This also relates to the duality between speculative computations in ORP and ANDP: in ORP success along one alternative of a choice point turns execution along other alternatives into speculative work[2], while in ANDP failure of a subgoal in a parallel call turns computation of subgoals to its right into speculative work. Nevertheless, the current approach taken in designing schedulers for ANDP systems seems to be too strictly oriented towards executions that are assumed to always succeed (e.g., CCL, &-Prolog, DDAS). The technique of trying failing subgoals first is never considered (e.g., in CCL the *right choices* are always assumed to be made). In ANDP systems, most of the time, the scheduling is based on simple goal-stacking, i.e., goals ready for parallel execution are pushed on a stack and from there picked up by idle processors. Only few variations have been suggested, like keeping track of the "best" sources of work [12] or using queuing instead of stacking [5].

Our claim, emerging from the duality between ORP and ANDP, is that even ANDP systems can take advantage of more sophisticated scheduling techniques, based on the same principles (or, better, their dual) that have been used for developing schedulers for ORP systems. In the case of ANDP, the ideal goal of a scheduler would be: *(i)* if the computation is going to succeed, we want to improve the speed-up and reduce the interactions between different processors; *(ii)* if the computation is going to fail (e.g., in an Independent ANDP computation one of the subgoals fails to find a solution), then the scheduler should ideally select first those subgoals that are more likely to fail, detecting the global failure as soon as possible (and reducing the amount of speculative work).

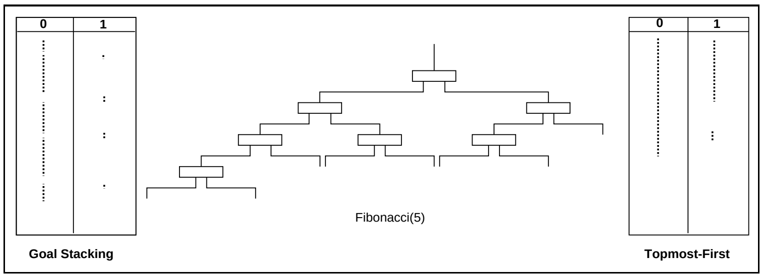


**Fig. 2.:** Comparison between Goal-stacking and Topmost-first

The issues that have to be considered in designing a scheduler for and ANDP system are essentially dual to those that are typically considered for ORP (e.g., see [1]). One possibility, in order to improve speed (cf. remark *(i)* above) during forward execution would be to get subgoals always from the parcall frames higher up in the computation tree, following the intuition that the granularity of these subgoals would be probably larger. This is essentially the same principle that has been used in the design of some of the schedulers adopted in ORP systems (like the *Manchester* scheduler for Aurora). A simplified version of this scheduler has been implemented in ACE and has resulted in improved performance.

Figure 2 shows the computation tree developed on the `fibonacci` benchmark. The distribution of ANDP work for the goal-stacking scheduler is shown

[2] This is not true if we want an *all-solutions* execution.

on the left, and for dispatching-on-topmost scheduler is shown on the right. Note that the one on the right is less fragmented, leading to higher granularity and better speedup. Table 5 contains the execution times on some benchmarks using the two schedulers; as we can see the behaviour is slightly worse for sequential execution (due to some additional costs incurred in the scheduler), but it improves when more processors are used, obtaining an average improvement of 4% using 5 processors. Similar techniques can be used to improve performance of the scheduler when dealing with failing computations (cf. remark 2 above). The ideal behaviour, the one in which failing computations are selected first, is clearly impossible to realize (since telling failing computations in advance is undecidable).

| *Goals executed* | ACE Agents | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | 3 | | | 5 | | |
| | Unopt. | Opt. | % | Unopt. | Opt. | % | Unopt. | Opt. | % |
| fibonacci | 3596 | 3701 | -3% | 1310 | 1296 | 1% | 810 | 778 | 4% |
| matrixmult | 5432 | 5545 | -2% | 1900 | 1877 | 1% | 1166 | 1120 | 4% |

**Table 5.:** Improvements using dispatching on topmost sched.

The typical solution in ORP systems for minimizing speculative work is to choose the parallel work that lies in the left part of the search tree so that parallel computation mimics sequential Prolog execution as much as possible. In ORP, if a processor is executing a branch that is potentially speculative, it will periodically check for the presence of "better" work (e.g., new alternatives in the leftmost part of the tree) and, if so, suspend the current computation and task-switch to one of the "better" alternatives [1]. The same technique is applicable to ANDP systems; focusing on the leftmost part of the computation tree seems to be a reasonable assumption, considering that with respect to failure the subgoals on the right are more speculative that those on the left (named elsewhere *first-fail principle*). A scheduler capable of this is currently under development as part of the ACE project. Table 6 gives the results obtained by simulating the effect of this type of scheduler on a benchmark with speculative work (a failing matrix multiplication). In the table, the lazy-left scheduler always picks leftmost work but only after it finishes its current task, while the eager-left scheduler will voluntarily suspend its current task immediately if less speculative and-parallel work appears to the left.

| *Goal* | Standard Scheduler | Lazy-Left Scheduler | Eager-Left Scheduler |
|---|---|---|---|
| failing matrix mult. | 616 | 365 | 91 |

**Table 6.:** Execution Time (ms.) with Speculative work

### 5.3 DANDP

Another interesting application that can be developed from the dual nature of ORP/ANDP is related to the design of systems exploiting *Dependent And-parallelism (DANDP).* As described earlier, a DANDP system allows and-parallel execution of two (or more) subgoals that share a common unbound variable (which may receive conflicting bindings from the two goals). The presence of these active dependencies requires the design of special mechanisms that ensure that the common variable is bound consistently, as in Prolog. To realize this, whenever a goal tries to bind a shared variable, it should perform a check in order to verify whether this binding may deviate the execution from the desired Prolog behaviour. This can be realized by introducing the notion of *producer* and *consumer* for each shared variable; at every point in the execution there should be only one producer for each shared variable (in Prolog semantics, the producer is the leftmost subgoal that has access to the variable). The producer subgoal is the only subgoal allowed to post a binding for the shared variable. All the other subgoals are designated as consumers and will suspend if they try to bind the shared variable. Suspended consumers will either wait for the variable to be bound by the producer or to become producers themselves (i.e., become leftmost in the computation tree). As discussed earlier, implementation of ORP has to deal with dependencies between parallel threads because different or-parallel threads may produce distinct bindings for the same (conditional) variable. The main difference is that in ORP the alternative threads are disjunct and consequently the bindings produced can be kept separate (since $\exists X(B_1(X) \vee B_2(X)) \equiv \exists X(B_1(X)) \vee \exists X(B_2(X))$), while in the case of DANDP the bindings need to be kept consistent (since $\exists X(p(X) \wedge q(X)) \not\equiv \exists X(p(X)) \wedge \exists X(q(X))$). Nevertheless we do not need to go to the extreme of keeping a unique location for these bindings, as done in most of the implementations (e.g., DDAS), since this requires complex synchronization mechanisms, locks, etc. An interesting alternative is exactly the one suggested by the duality principle above, by trying to split the existential quantification in two separate quantifications: $\exists X(p(X) \wedge q(X)) \equiv \exists X_1 X_2 (p(X_1) \wedge q(X_2) \wedge X_1 = X_2)$ where the equality $X_1 = X_2$ is used to maintain consistency between the bindings produced by $p$ and $q$. This gives rise to various degrees of possible implementations of DANDP, that differ from each other based upon the moment at which the equality $=$ is enforced. The simplest model is the one in which the parallel computations (e.g., $p(X_1)$ and $q(X_2)$) are carried on independently and '=' is applied only after they have both produced a binding for their variables. This essentially reduces DANDP to IANDP by delaying the dependencies after the end of the parallel execution (and has been adopted in some early works, e.g. [14]). Nevertheless, this is not too interesting and may lead to loss of Prolog semantics. Speculative work can appear (and, in extreme cases, it can be infinite—i.e., non-terminating computations). On the other hand, the most complex model is the one in which the $=$ is performed before starting the parallel execution, which leads to the various DANDP models implemented (like DDAS). We are interested in finding a solution between the two extremes.

Our solution is inspired from techniques for implementing or-parallelism. In or-parallelism different or-parallel threads may have dependencies between them, namely, through the conditional variables. The way this problem is solved there is that different or-parallel threads keep a different "view" of the variable with the help of the environment representation mechanism used. For example, in binding arrays [15], the common offset is stored in the stack location for that variable while the different bindings (that represent the different view of the variable for different or-parallel threads) are stored in the private binding arrays. In Muse, separate copies of the stack location for the variables are maintained by each thread. In DANDP, the problem is similar, the threads executing the producer and the consumer goals should have different "views" of the dependent variable.

That is, the thread executing the producer goal is allowed to bind the dependent variable, while the one executing the consumer goal is not (at least until the consumer goal becomes leftmost, in which case it becomes the producer). Thus, we can adapt a solution similar to the one used for or-parallelism. In particular, we use binding arrays.

Adapting BA to dependent and-parallelism requires some minor changes. Basically, a small BA is assigned to each subgoal, in which the "local view" of the shared variables associated to such subgoal will be stored. A *producer view* will be implemented as a direct pointer to the object. A *consumer view* will be realized as a pointer to the view of the immediately preceding subgoal. This is illustrated in figure 3. In this way the process of validating a variable access can be done concurrently while accessing its value—an indirection from one BA to another BA during dereferencing indicates that the subgoal has a consumer view of the shared entity. As in the BA model for or-parallelism, an *installation* phase is required upon scheduling of a computation to a computing agent. The installation step is executed by the agent to collect the BAs that represent the *shared environment* visible to the current computation. In figure 3, the shared environment for an agent executing `t` will be composed by the BA for `t` and the one for `q`.[3] Prolog semantics is respected by forcing any binding attempted through a consumer view to suspend. Furthermore, views may need to be modified upon completion of a subgoal (some consumer views may need to be transformed into producer views)—a simple bit associated to each BA and denoting the status of the corresponding subgoal (active or completed) is sufficient for this purpose. Our scheme is arguably superior to other implementation schemes for DANDP (like DDAS) [17]. Further details are not included here due to lack of space and can be found elsewhere [17].

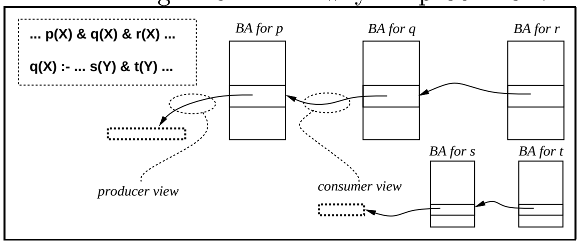


**Fig. 3.:** Use of BA for DAP

## 6 Conclusions

In this paper we presented an analysis of the dual nature of the two major forms of parallelism—or- and and-parallelism—exploited in Logic Programming. The existence of this dual nature allows one to use techniques developed for realizing/optimizing one form of parallelism for realizing/optimizing the other. Our observations regarding duality of or-parallelism and and-parallelism, we hope, will also simplify the development of combined and-or parallel systems, while giving rise to a whole new set of optimizations and offering new ways to efficiently exploiting parallelism. Most of the techniques discussed in this paper

[3] A BA may be needed for each subgoal in the parallel call. Newly developed data-structures such as *Paged Binding Arrays* (PBA) [10] can be used for this purpose. Each subgoal can be allocated a page in the PBA, making the management of BAs simpler.

have been implemented in the ACE system, an and-or parallel implementation of Prolog, and the results have been reported.

The research presented has benefited from discussions with M. Carro, M. Hermenegildo, K. Shen, and V. Santos Costa, all of whom we would like to thank. Ongoing work is supported by NSF Grants 9415256, CCR 92-11732, and HRD 93-53271, and by NATO Grant CRG 921318, and by a fellowship from Phillips Petroleum to Enrico Pontelli.

## References

1. K.A.M. Ali and R. Karlsson. Scheduling Speculative Work in MUSE and Performance Results. Technical report, SICS, 1993.
2. P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *Int'l Conf. on FGCS.* Tokyo, Nov. 1988.
3. M. Carlsson et al. A Simplified Approach to the Implementation of And-Parallelism in an Or-Parallel Environment. *ICLP*, 1988.
4. M. Codish, E. Shapiro. Compiling Or-Parallelism into And-Parallelism. *ICLP*, 1986.
5. J. Crammond. The Abstract Machine and Implementation of Parallel Prolog. Research report, Dept. of Computing, Imperial College, July 1990.
6. J. Du Pond and H.J. Hernandez. Parallelization of Right Uniform Datalog Programs and of Chain Sirups. *Journal of Computing and Information*, 1(1), 1994.
7. S. Tsur S. Ganguly, A. Silberschatz. A Framework for the Parallel Processing of Datalog Queries. In *Proc. of ACM SIGMOD Conf. on Manag. of Data*, 1990.
8. G. Gupta. *Multiprocessor Execution of Logic Programs.* Kluwer Press, 1994.
9. G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *ICLP*,MIT Press, 1994.
10. G. Gupta et al. Shared Paged Binding Arrays: Universal Data-structure for Parallel Logic Programming. NFS/ICOT workshop on Parallel Logic Progr., 1994.
11. M.Hermenegildo, F.Rossi. Non-Strict Independent And-Parallelism. In *ICLP*, 1990.
12. M.Hermenegildo. *Independent And-Parallel Prolog and its Architecture.* Kluwer, 1989.
13. M. Hermenegildo M. Carro. Backtracking families, unpublished manuscript, 1993.
14. L. Kale. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *ICLP*, 1987.
15. E. Lusk and al. The Aurora Or-parallel Prolog System. *NGC*, 7(2,3), '90.
16. E. Pontelli, G. Gupta, and D. Tang. Determinacy Driven Optimizations of Parallel Prolog Implementations. Proc. of ICLP'95, MIT Press, 1995.
17. E.Pontelli, G.Gupta. Dependent And-Parallelism in Logic Programming. Internal Report, LLDAP, 1995.
18. E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, 1987.
19. K. Shen. *Studies in And/Or Parallelism in Prolog.* PhD thesis, 1992.
20. D. H. D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, June 1990. SICS.
21. 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.