Optimal Implementation of And-Or Parallel Prolog†

Gopal Gupta
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA.
gupta@nmsu.edu

Vítor Santos Costa
Department of Computer Science
University of Bristol
Bristol BS8 1TR, UK.
costa@compsci.bristol.ac.uk

Abstract

Most models that have been proposed, or implemented, so far for exploiting both or-parallelism and independent and-parallelism have only considered pure logic programs (pure Prolog). We present an abstract model, called the Composition-Tree, for representing and-or parallelism in *full* Prolog. The Composition-Tree *recomputes* independent goals to ensure that Prolog semantics is preserved. We combine the idea of Composition-Tree with ideas developed earlier, to develop an abstract execution model that supports full Prolog semantics while at the same time avoiding redundant inferences when computing solutions to (purely) independent and-parallel goals. This is accomplished by *sharing solutions* of independent goals when they are pure (i.e. have no side-effects or cuts in them). The Binding Array scheme is extended for and-or parallel execution based on this abstract execution model. This extension enables the Binding Array scheme to support or-parallelism in the presence of independent and-parallelism, both when solutions to independent goals are recomputed as well as when they are shared. We show how extra-logical predicates, such as cuts and side-effects, are supported in this model.

1. Introduction

One of the main features of logic programming languages is that they allow implicit parallel execution of programs. There are three main forms of parallelism that are present in logic programs: Or-parallelism, Independent And-parallelism and Dependent and-parallelism. In this paper we restrict ourselves to Or-parallelism and Independent and-parallelism. There have been numerous proposals to exploit or-parallelism in logic programs [AK90, HCH87, LWH90, W84, W87, etc.]‡ and some to exploit independent and-parallelism [H86, LK88, etc.]. Models have also been proposed to exploit both or-parallelism and independent and-parallelism in a single framework [GJ89, RK89, BK88]. It is this aspect of combining independent and- and or-parallelism that this paper addresses.

Most models that have been proposed, or implemented, so far for combining or-parallelism and independent and-parallelism have either considered only pure logic programs (pure Prolog) [RK89, GJ89], or have modified the language to separate parts of the program that contain extra-logical predicates (such as cuts and side-effects) from those that contain purely logical predicates, allowing parallel execution only in parts containing purely logical predicates [RS87, BK88]. In the former case practical Prolog programs cannot be executed since most practical Prolog programs have

† This is an extended version of the paper "And-Or Parallel Implementation of Full Prolog with Paged Binding Arrays" that appeared in the "Proceedings of Parallel Architectures and Languages Europe 1992" (PARLE '92).

‡ See [GJ93] for a systematic analysis of the various or-parallel models.

extra-logical features. In the latter case programmers have to divide the program into sequential and parallel parts themselves, which requires some programmer intervention. This approach also rules out the possibility of taking "dusty decks" of existing Prolog programs and running them in parallel. In addition, some parallelism may also be lost since parts of the program that contain side-effects may also be the parts that contain parallelism. Systems such as Aurora, Muse [HCC88, LWH90, AK91], and &-Prolog [HG90, MH89, CC89] have followed a different approach—they exploit parallelism completely implicitly. Thus they can speed up existing Prolog applications with minimal or no rewriting. We contend that the same can be done for systems that combine independent and- and or-parallelism. (However, for &-Prolog, and-parallelism has to be annotated using Conditional Graph Expressions (CGEs) before execution commences. This can be done by the user or by automatic parallelization tools [MH89a] based on abstract interpretation. Although user-written annotations should perform better than these automatically generated annotations, experience has shown that the parallelization tools do uncover a significant amount of parallelism [MH89b].) In this paper we present a concrete model which facilitates the inclusion of side effects in an and-or parallel system. Our approach is two-pronged, first we present an abstract model of and-or parallelism for logic programs which mirrors sequential Prolog execution more closely when side-effects are present in and-parallel goals (essentially by recomputing independent goal rather than re-using them [GH92, SH91]); second, we introduce the concept of teams of processors to exploit and-or parallelism and present an efficient environment representation scheme based on Binding Arrays [W84,W87] to support it. For and-parallel goals that do not have any side-effects we use the well known techniques of *solution sharing* [RK89, GJ89, WR87] to optimally evaluate these goals in independent and-parallel†. Our approach can be contrasted with the PEPSys model [WR87, RS87] which also attempts to compute solutions to queries optimally, and supports side-effect and extralogical predicates in Prolog (but by using an approach that requires user's intervention). It should be pointed out that although our aim is to require the least amount of user intervention in the exploitation of parallelism, to obtain highly efficient execution some amount of user intervention may be necessary: for example, for obtaining high performance execution, the user may have to be given the ability to explicitly control the granularity of computation, at least until efficient tools for automatic granularity control are built. As a matter of fact, systems such as Aurora, Muse, and &-Prolog do give their users the ability to tell the system which parts of the program should and which parts should not be executed in parallel so that the user can control the granularity of computation.

Our environment representation technique is based on Binding Arrays and is an improvement of the technique used in the AO-WAM [GJ89]. We use Conditional Graph Expressions (CGEs) [H86] to express independent and-parallelism. Notice that our model subsumes the &-Prolog system and the Aurora system—applications that can be run in parallel in either of these systems can be run in parallel in our model as well.

† Our model is optimal in the sense that it avoids redundant computation of goals whenever possible. Note that our model is optimal only for or- and independent and-parallel execution. Optimal models of execution where dependent and-parallelism is also present have also been proposed; for instance, the Extended Andorra Model [HJ90, W90]. Also note that our model will not always use solutions sharing for computing and-parallel goals that do not have side-effects, because in certain cases (described later) the overhead incurred in sharing solutions is more than the benefit gained.

The rest of this paper is organized as follows: Section 2 presents the *Composition Tree* and the *Extended And-Or Tree* respectively for representing and-or parallel execution of goals that contain side-effects and that do not contain side-effects respectively; Section 3 presents the notion of teams of processors to efficiently execute logic programs in and-or parallel; Section 4 presents an efficient environment representation technique called the *Paged Binding Array* that is an extension of earlier techniques based on Binding Arrays [W84, W87]; Section 5 presents techniques that preserve sequential semantics during and-or parallel execution of logic programs containing side-effects and extralogical features. Finally, Section 6 presents our conclusions.

2. Or-parallelism, Independent And-parallelism

Or-parallelism arises when more than one rule defines some relation and a procedure call unifies with more than one rule head in that relation—the corresponding bodies can then be executed in or-parallel fashion. Or-parallelism is thus a way of efficiently searching for solutions to a goal, by exploring alternative solutions in parallel. It corresponds to the parallel exploration of the branches of the proof tree. Or-parallelism has successfully been exploited in full Prolog in the Aurora [LWH90] and the Muse [AK90] systems both of which have shown very good speed up results over a considerable range of applications.

Independent And-parallelism arises when more than one goal is present in the query or in the body of a procedure, and the run-time bindings for the variables in these goals are such that two or more goals are *independent* of one another. Independent and-parallelism is thus a way of speeding up a problem by executing its subproblems in parallel. One way for goals to be independent is that they don't share any variable at run-time. This can be ensured by checking that their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables. Independent and-parallelism has been successfully exploited in the &-Prolog system [HG90]. Independent and-parallelism is expressed in such systems through Conditional Graph Expressions (CGEs), which are of the form

$$(condition \;\Rightarrow\; goal_1 \;\&\; goal_2 \;\&\; \ldots \;\&\; goal_n\;)$$

which states that if *condition* is true, goals $goal_1 \ldots goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially. The *condition* can be any Prolog goal but is normally a conjunction of special builtins which include *ground*/1, which checks whether its argument has become a ground term at run-time, or *independent*/2, which checks whether its two arguments at runtime are such that they don't have any variables in common, or the constant *true* meaning that $goal_1 \ldots goal_n$ can be evaluated in parallel unconditionally. It is possible to generate CGEs automatically and quite successfully at compile-time using abstract interpretation [MH89a].

2.1. And-Or Composition Tree: And-Or parallelism with Recomputation

The most common way to express and- and or-parallelism in logic programs is through traditional and-or trees, which consist of *or-nodes* and *and-nodes*. Or-nodes represent multiple clause heads matching a goal while and-nodes represent multiple subgoals in the body of a clause being executed in and-parallel. Since in our model and-parallelism is permitted only via CGEs, we restrict and-nodes to represent only CGEs. For instance, given a clause `q :- (true => a & b)`, where `a` and `b` have 3 or-parallel solutions each, and the query `?- q`, then the resulting and-or tree would appear as shown in Figure 1. The original query `q` would therefore have nine solutions, one for each combination of a valid solution for `a` and of a valid solution for `b`.

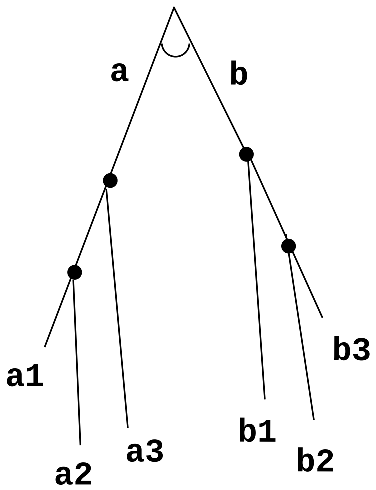


Figure 1: Conventional And-Or Tree

Notice that all of `q`'s solutions should be available to any goals in `q`'s continuation. In an and-or tree style execution, bindings made by the different alternatives to goal `a` are not visible to different alternatives to goal `b`, and vice versa, and hence the correct environment must be created before the continuation goal of the CGE can be executed. The problem with this style of parallel execution is that creation of the proper environments requires a global operation, for example, *Binding Array loading* in AO-WAM [GJ89], the complex dereferencing scheme of PEPSys [BK88], or the "global forking" operation of Extended Andorra Model [W90]. One can eliminate this problem, by extending the traditional and-or tree so that the various or-parallel environments that simultaneously exist *are always separate*.

The extension essentially uses the idea of recomputing independent goals of a CGE. Thus, for every alternative of `a`, the goal `b` is computed in its entirety. Each separate combination of `a` and `b` is represented by, what we term as a *composition* node (c-node for brevity)†. Thus the extended tree for the above query, which we term the *Composition-tree* (C-tree for brevity), would appear as shown in Figure 2—for each alternative of and-parallel goal `a`, goal `b` is computed in its entirety. Note that each composition node in the tree corresponds to a different solution for the CGE. To represent the fact that a CGE can have multiple solutions we add a branch point (choice point) before the different composition nodes. The C-tree can represent or- and independent and-parallelism quite naturally—execution of goals within a single c-node gives rise to independent and-parallelism while execution of different c-nodes, and of untried alternatives which are not below any c-node, gives rise to or-parallelism. The C-tree of a logic program is dynamic in nature, and its shape may depend on the number of processors and how they are scheduled, as will be explained shortly.

† The composition-node is very similar to the Parcall frame of RAP-WAM [H86].

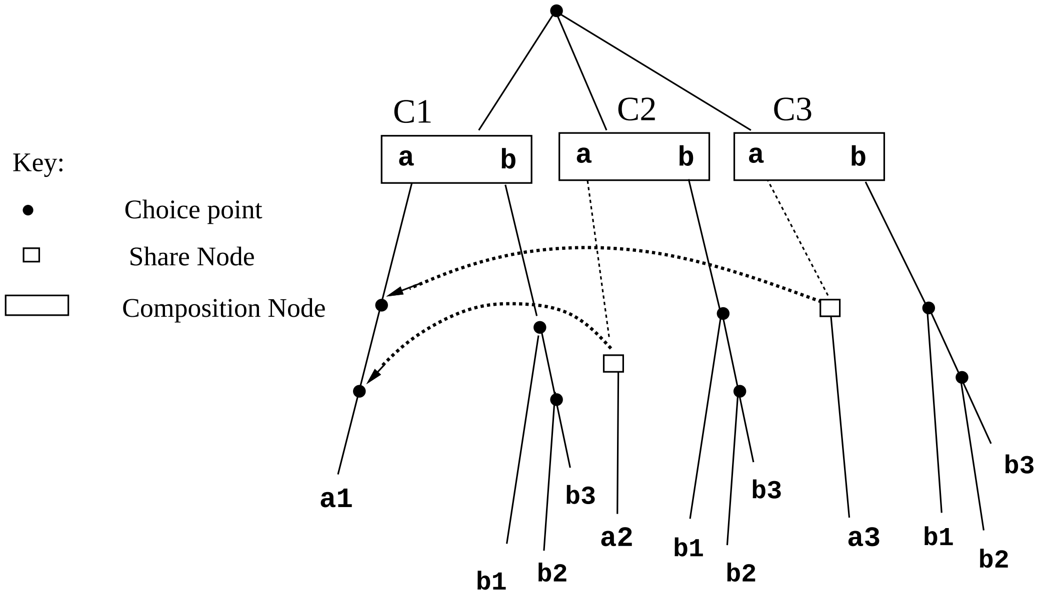


Figure 2: An Example Composition Tree

Notice the topological similarity of the C-tree with the purely or-parallel tree shown in Figure 3 for the program above. Essentially, branches that are shared in the purely or-parallel tree are also shared in the C-tree. This sharing is represented by means of a *share-node*, which has a pointer to the shared branch and a pointer to the composition node where that branch is needed (Figure 2). Due to sharing the subtrees of some independent and-parallel goals maybe spread out across different composition nodes. Thus, the subtree of goal `a` is spread out over c-nodes C1, C2 and C3 in the C-tree of Figure 2.

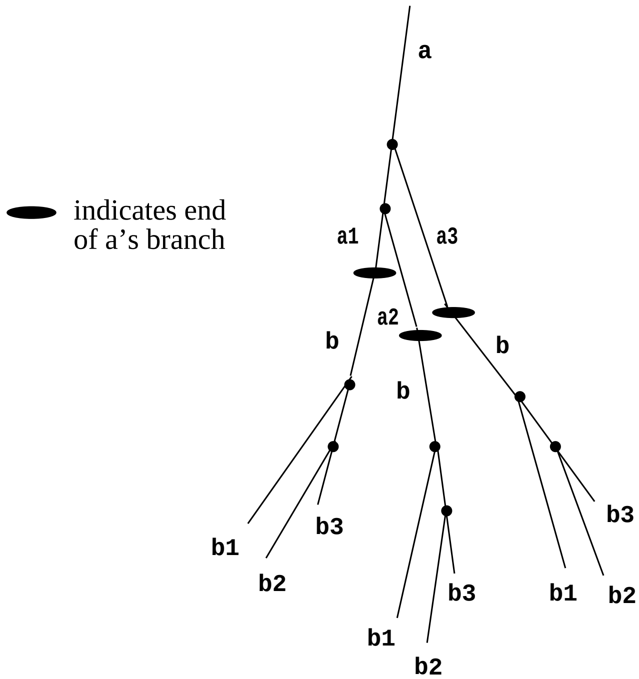


Figure 3: Equivalent Or-parallel Tree

2.2. Extended And-Or Tree: And-Or Parallelism and Solution Sharing

Consider the CGE `(true => a(X) & b(Y))`, where the goals `a` and `b` have m and n solutions respectively. In a system doing recomputation, such as Prolog or a composition-tree based system the goal `b` will be executed in its entirety for every solution found for goal `a`. Thus the cost of execution will be $O(m * n)$. This means that given that `a` and `b` are independent the solution for `b` will be redundantly recomputed many times. Classical and-or tree based models avoid this redundancy by computing the solutions for `a` and `b` only once, and then constructing the cross-product of these solutions to compute all solutions for the CGE. This way the computation cost will be $O(m + n)$, plus the cost of computing the cross-product. The technique of reducing the amount of computation is known as *solution sharing* or goal reusing, and was supported by the models proposed in the past for exploiting independent and- and or-parallelism [WR87, RK89, GJ89]. The solution sharing approach has some problems. Mainly, it only applies when goals `a` and `b` are pure, because the implicit assumption is that if goal `b` were to be recomputed, they will produce the same set of solutions. This assumption breaks in the presence of side-effects and extra-logical predicates, as the independent goals might produce a different set of solutions. For instance, when within goal `b` a value is read from standard input and then some processing is done based on the value read, the user might input a different value for every invocation of goal `b` (where each new invocation of `b` corresponds to a different solution of `a`), resulting in the set of solutions for `b` being different for each invocation. In such cases, clearly, solution sharing will not produce the correct result if we want Prolog's sequential semantics (where by sequential semantics we mean that the program, when run in parallel, shows the same external behaviour as it would if it were run under a sequential Prolog system). Being able to support sequential Prolog semantics transparently for a parallel execution is an important goal of our work.

The implementations of and-or parallel Logic Programming systems that employ solution sharing has been well researched [RK89, GJ89, WR87]. In particular, the Extended And-Or tree model [GJ89, GJ93] abstractly represents and-or parallel execution with solution sharing. In this paper we want to show how a solution sharing approach, in particular the one used in AO-WAM [GJ89, GJ93], can be combined with a recomputation based approach [GHS93, GS92a] to produce an efficient system that does optimal computation (i.e., avoids redundant re-computation of independent goals whenever possible) while supporting Prolog's sequential semantics. We will later show how multiple environments can be efficiently maintained in this combined approach. Such an approach combines the efficiency, and the support for full Prolog of recomputation-based systems, with the benefits of solution sharing.

To explain solution sharing we show an example Extended And-Or Tree in Figure 4. The example shows how multiple solutions to independent and-parallel goals are found and then combined into a cross-product. When the CGE `(true => b & c )` is encountered, its execution is initiated by adding a cross-product node (similar to Parcall frame of RAP-WAM [H86] and also to the C-node in the C-tree). Solutions to and-parallel goals are then found in parallel. When a solution to an and-parallel goal is found, a partial cross-product with existing solutions for other goals is constructed. For each tuple (i.e., a combination of solutions of independent and-parallel goals) in the cross-product set a node (called *sequential-node*) is created that corresponds to the continuation goal of the CGE. Once the sequential node has been created execution of the continuation goal can begin†. More details can be found elsewhere [GJ89, GJ93].

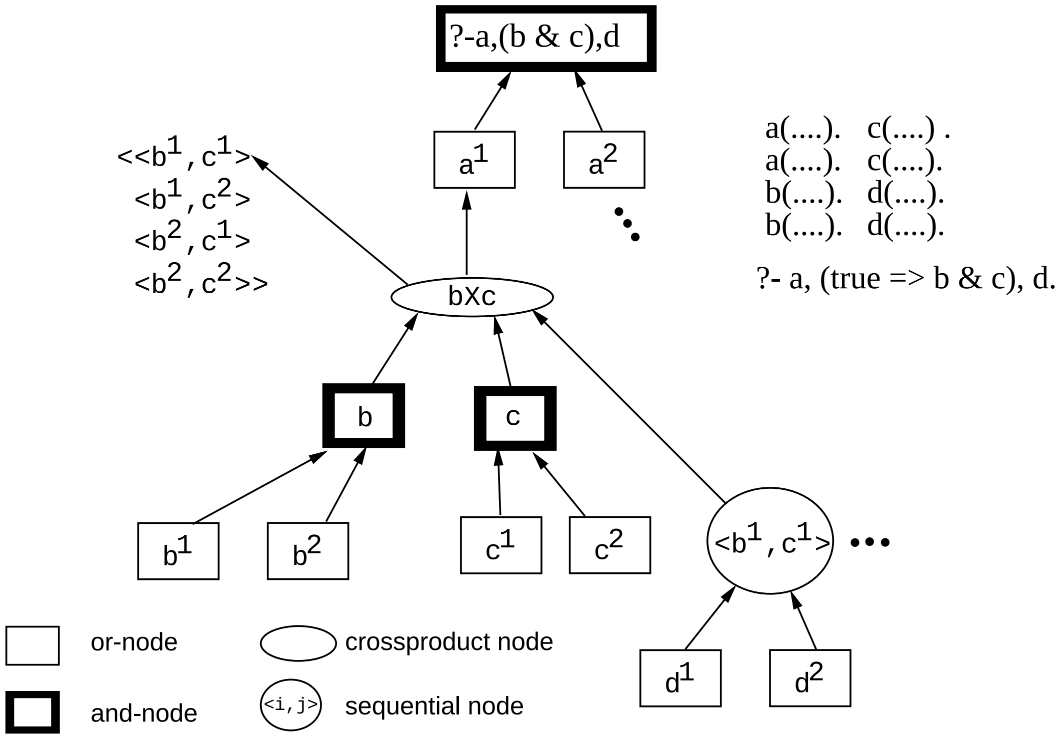


Figure 4: Extended And-Or Tree

2.3. Combining Goal Recomputation and Solution Sharing

The ideal parallel Prolog system would both support sequential Prolog semantics and keep the amount of computation optimal. Optimal computation is achieved when solutions of independent and-parallel goals are shared. However, solution sharing causes problems if independent goals have side-effect or extra-logical calls in them. One could solve this problem by sequentializing the execution of independent goals that have side-effects. Thus, if one of the goals in the CGE `(true => a(X) & b(Y))` has a side-effect, then the CGE will be executed sequentially. Following the same principles, if this CGE is nested inside another CGE then the outer CGE will have to be sequentialized too, and so on. It is obvious that the sequentialization process will have to be carried all the way to the CGE at the topmost level, resulting in too much parallelism being lost. If, however, such goals are recomputed then although some parallelism is lost in the presence of side effects and extra logical constructs, the loss is not so severe. Hence, there is a strong argument for recomputing non-deterministic and-parallel goals, if they are not pure.

The approach that we adopt is quite simple. If a goal in a CGE will lead to a side-effect or extra-logical predicate or if it will call a dynamic predicate (such CGEs can be identified by a simple examination of the program at compile time), then the CGE (we call such a CGE an impure CGE) will be executed in and-parallel using goal recomputation (Section 5 explains how side-effects and extra-logical predicates are supported in such a case to preserve sequential Prolog

† Note that the And-node in Figure 4 is very similar to Input Goal Marker and the Sequential-node to the Wait Marker of RAP-WAM [H86].

semantics). Thus, for such a CGE the code generated will construct a Composition subtree at runtime. However, if the goals in the CGE are determined to be pure (such a CGE is called a pure CGE), then the CGE is executed with solution sharing (as done in AO-WAM), and for such a CGE the code generated will construct an Extended And-Or subtree at runtime. Thus, goal reuse and goal recomputation are combined in one model so that the advantages of both are obtained. Figure 5 shows an example where a program has one pure CGE and an impure one. The execution tree for this program contains an extended and-or subtree for the pure CGE and a Composition subtree for the impure one. Note that a pure CGE cannot have another nested impure CGE inside it (it can, however, have pure ones nested inside). However, an impure CGE can have other pure CGEs nested inside it.

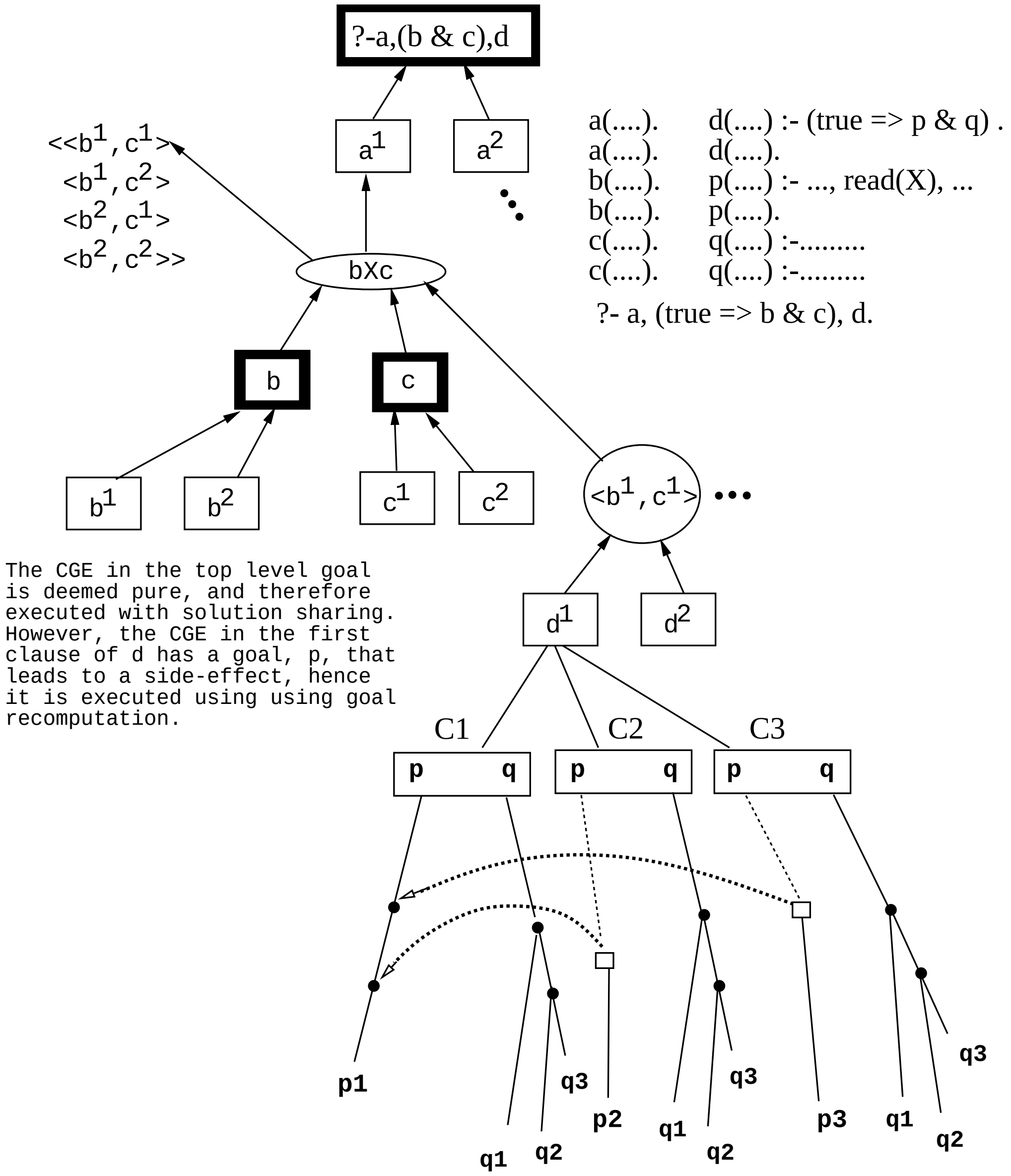


Figure 5: Combining Goal Reuse with Goal Recomputation

Goal reuse has a different problem that makes preservation of sequential Prolog semantics difficult. Or-parallelism makes it quite difficult to maintain the (Prolog) order of the different branches in the search tree without incurring performance penalties. This problem is a consequence of letting the cross-product of solutions be computed incrementally. Consider the Extended And-Or Tree shown in Figure 4. Assume now that two processors P1 and P2 are looking for solutions to goals b and c in parallel (other processors besides P1 and P2 may also be working on this tree). Also assume P1 finds a solution $\mathtt{b}_i$ for b before any other processor has found solutions for either b or c. At this point no tuples in the cross-product can be constructed since c doesn't have a solution. So P1 might elect to record this solution, and then attempt to find another solution for b. Meanwhile suppose the solution $\mathtt{c}_j$ is found by P2. The tuple $\langle \mathtt{b}_i, \mathtt{c}_j \rangle$ can be constructed by P2. P2 could now explore further solutions for b, but it can also start the continuation of the CGE for this combination of solutions $\langle \mathtt{b}_i, \mathtt{c}_j \rangle$. In the case where the execution of the continuation is begun as soon as a cross-product tuple is generated, we say that the cross-product set is generated *incrementally* (i.e., we don't wait for all solutions of independent goals to be reported to compute cross-product tuples).

Note that the first solution to be available for the continuation is not necessarily the first that Prolog would generate. Indeed, even if the solutions to each component of the CGE were generated in order, it would still be difficult to maintain Prolog order. Suppose that P1 has already generated the first solution, $\mathtt{b}_1$, and that P2 has generated $\mathtt{c}_1$. Suppose next that P1 finds the second solution for b, namely $\mathtt{b}_2$. The tuple $\langle \mathtt{b}_2, \mathtt{c}_1 \rangle$ will be constructed and the continuation of the CGE will be executed for this combination. Already at this point one can notice that the tuples of the cross-product set were not generated in Prolog order.

The Prolog order of cross-product tuples should have been $\langle \mathtt{b}_1, \mathtt{c}_1 \rangle$, $\langle \mathtt{b}_1, \mathtt{c}_2 \rangle$, $\langle \mathtt{b}_1, \mathtt{c}_3 \rangle$, $\langle \mathtt{b}_2, \mathtt{c}_1 \rangle$, $\langle \mathtt{b}_2, \mathtt{c}_2 \rangle$, $\langle \mathtt{b}_2, \mathtt{c}_3 \rangle$, $\langle \mathtt{b}_3, \mathtt{c}_1 \rangle$, $\langle \mathtt{b}_3, \mathtt{c}_2 \rangle$, $\langle \mathtt{b}_3, \mathtt{c}_3 \rangle$. Obviously, our goal would be to insert any tuple, say $\langle \mathtt{b}_i, \mathtt{c}_j \rangle$ in its correct position $(i-1)*n+j$, where $n = 3$ is the number of solutions for c. In general, this is not straightforward. First, in order to maximise or-parallelism we may not generate solutions for b and c sequentially, and thus we may not know the values of i and of j. Even if we generate solutions sequentially, and hence know the current value of i and j, we will only know n when we complete execution of c, or in other words, we would in general need to wait for all the solutions to c before starting execution of goals in the continuation.

This problem has several solutions. The user may decide whether he/she cares for the order in which the different solutions to a query are to be found. In this approach, before executing the query some annotations would indicate to the system the way that query should be evaluated, i.e., whether sequential Prolog semantics should be strictly followed or not, by setting appropriate flags. If the user does care about the ordering of solutions, then the necessary costs will be incurred to produce the cross-product set in Prolog order (this is discussed next). Otherwise, solutions are output in the order they are generated. An alternative option in such a case is to evaluate *all* CGEs, whether pure or impure, with goal recomputation so that the cross-product (and the problem of keeping its tuples in Prolog order) is avoided. This solutions may be advantageous for pure CGEs in quite a few cases anyway, as in general one should balance the cost of computing the cross-product set and keeping it in the Prolog order with the cost of performing redundant recomputation of goals. Recomputation is probably always a better solution, even for pure goals, if the grain-sizes of goals in a pure CGE are small, and if the left components of the CGE (b in this example) are deterministic.

We next discuss the two alternatives for goal reuse, first the cases where we do not care about solution order, and second the cases where we want to combine reuse with solution ordering.

The best case for goal reuse is when the user does not care for the order of solutions (or the order in which the side-effects will be performed). In this case, CGEs that are pure can be evaluated with goal reuse (without worrying about keeping the cross-product set in Prolog order) and those with side-effects and extra-logical predicates with goal recomputation. One such case where order is not relevant is if there are no cuts that may prune solutions to the CGE and the computation is in the scope of *setof*, that is the user is interested in all the solutions to a query. One similar case is again where there are no cuts that can prune the CGE and the computation is in the scope of a commit, that is, any correct solution will be acceptable. In general, we can be more ambitious and say that if the program does not have cuts the user is probably not very interested in a specific ordering of solutions.

It is important to remark that relaxing the order of solutions does not mean that the side-effects corresponding to different solutions will get arbitrarily mixed with each other, rather the order in which the execution tree will be searched will not be known in advance. The side-effects will be executed in whatever order the execution tree is searched but *side-effects in and-parallel goals will be performed from left to right.* In other words, the or-parallel branches may be executed in different order, but the different and-parallel branches within a single solution are still executed in the Prolog order.

Consider now the case where we want solution sharing (in pure CGEs) along with Prolog semantics. The cross-product set can be computed in much the same way as before but now the cross-product node needs to dynamically order the tuples in such a way that the system will be able to quickly find out if a tuple is "to the left" of a different tuple. Essentially, when a solution for a goal in the CGE is found, and its corresponding tuples constructed, the tuples must be inserted in the cross-product set of the CGE in a way that Prolog order is maintained. This will require that given a solution t of a goal g in a CGE, we must know which already generated solutions are to the left of t and which ones are to the right of t.

Consider the CGE `(true => a & b)`, where `a` has three solutions $\mathtt{a}_1$, $\mathtt{a}_2$, and $\mathtt{a}_3$ (in Prolog order) and likewise `b` has three solutions $\mathtt{b}_1$, $\mathtt{b}_2$, and $\mathtt{b}_3$ (in Prolog order). The Prolog order of tuples shall be: $\langle \mathtt{a}_1, \mathtt{b}_1 \rangle$, $\langle \mathtt{a}_1, \mathtt{b}_2 \rangle$, $\langle \mathtt{a}_1, \mathtt{b}_3 \rangle$, $\langle \mathtt{a}_2, \mathtt{b}_1 \rangle$, $\langle \mathtt{a}_2, \mathtt{b}_2 \rangle$, $\langle \mathtt{a}_2, \mathtt{b}_3 \rangle$, $\langle \mathtt{a}_3, \mathtt{b}_1 \rangle$, $\langle \mathtt{a}_3, \mathtt{b}_2 \rangle$, $\langle \mathtt{a}_3, \mathtt{b}_3 \rangle$. Consider the following scenario, solution $\mathtt{a}_1$ is found for `a`, and solution $\mathtt{b}_1$ is found for `b`. The tuple $\langle \mathtt{a}_1, \mathtt{b}_1 \rangle$ will be constructed. This is clearly the first tuple so it is inserted as such. Now suppose the solution $\mathtt{a}_2$ is found, the tuple $\langle \mathtt{a}_2, \mathtt{b}_1 \rangle$ will be added in the cross-product. Since $\mathtt{a}_1$ is to the left of $\mathtt{a}_2$, therefore $\langle \mathtt{a}_2, \mathtt{b}_1 \rangle$ must be inserted after the first tuple that was just found. The cross-product set at this moment is $\{\langle \mathtt{a}_1, \mathtt{b}_1 \rangle, \langle \mathtt{a}_2, \mathtt{b}_1 \rangle\}$). Suppose the solution $\mathtt{b}_3$ is found next. The tuple set of this solution $(\langle \mathtt{a}_1, \mathtt{b}_3 \rangle, \langle \mathtt{a}_2, \mathtt{b}_3 \rangle)$ will be constructed and its tuples inserted in the cross-product set in their appropriate position. As regards the first tuple in this set, its first element $(\mathtt{a}_1)$ agrees with the first element of the first tuple of the current cross-product set, so we need to compare their second elements ($\mathtt{b}_3$ and $\mathtt{b}_1$ respectively). Since solution $\mathtt{b}_3$ occurs to the right of solution $\mathtt{b}_1$, $\langle \mathtt{a}_1, \mathtt{b}_3 \rangle$ should be inserted to the right of the tuple $\langle \mathtt{a}_1, \mathtt{b}_1 \rangle$ in the cross-product set. We next need to compare the tuple $\langle \mathtt{a}_1, \mathtt{b}_3 \rangle$ to the second tuple ($\langle \mathtt{a}_2, \mathtt{b}_1 \rangle$) in the current cross-product set. Since the two tuples differ in their first element position, and solution $\mathtt{a}_1$ will be to the left of solution $\mathtt{a}_2$ in Prolog order, we must insert $\langle \mathtt{a}_1, \mathtt{b}_3 \rangle$ before $\langle \mathtt{a}_2, \mathtt{b}_1 \rangle$. The same process is repeated for $\langle \mathtt{a}_2, \mathtt{b}_3 \rangle$ and the cross-product set becomes: $\{\langle \mathtt{a}_1, \mathtt{b}_1 \rangle, \langle \mathtt{a}_1, \mathtt{b}_3 \rangle, \langle \mathtt{a}_2, \mathtt{b}_1 \rangle, \langle \mathtt{a}_2, \mathtt{b}_3 \rangle\}$.

Keeping the Prolog order of cross-product tuples therefore demands: (i) First, keeping track of the relative order of a solution to a goal among all the eventual solutions of that goal; and, (ii) Second, determining the appropriate position in the cross-product set at which a given tuple should be inserted (note that the cross-product now has to be maintained as a list rather than a true set, since in a set the position of an element is not important). Both these operations involve some costs. The first one involves maintaining information so that given two branches (solutions) we can tell which one is to the left of the other (i.e. which one will be discovered first in a sequential backtracking based execution). For this one could use the branch labeling techniques described elsewhere [RK89, H87]. The second one involves scanning the cross-product set while comparing the tuples, which will incur an overhead proportional to the size of the cross-product set. Again, techniques such as the notion of subroot node [S91] developed for or-parallel systems would help.

In the rest of the paper we assume that the cross-product set is always kept in Prolog order and that the user wants true sequential Prolog semantics.

3. Independent And-Parallelism and Teams of Processors

When a purely or-parallel model is extended to exploit independent and-parallelism then the following problem arises: at the end of independent and-parallel computation, all participating processors should see all the bindings created by each other. However, this is completely opposite to what is needed for or-parallelism where processors working in or-parallel should not see the (conditional) bindings created by each other. Thus, the requirements of or-parallelism and independent and-parallelism seem anti-thetical to each other. The solution that have been proposed range from updating the environment at the time independent and-parallel computations are combined [RK89, GJ89] to having a complex dereferencing scheme [BK88]. All of these operations have their cost.

We contend that this cost can be eliminated by organising the processors into teams. Independent and-parallelism is exploited between processors within a team while or-parallelism is exploited between teams. Thus a processor within a team will behave like a processor in a purely and-parallel system while all the processors in a given team will collectively behave like a processor in a purely or-parallel system. This entails that all processors within each team share the data structures that are used to maintain the separate or-parallel environments. For example, if binding arrays are being used to represent multiple or-parallel environments, then only one binding array should exist per team, so that the whole environment is visible to each member processor of the team. Note that in the worst case there will be only one processor per team. Also note that in a team setup a processor is free to migrate to another team as long as it is not the only one left in the team.

The concept of teams of processors has been successfully used in the Andorra-I system [SWY91], which extends an or-parallel system to accommodate dependent and-parallelism.

3.1. Teams of Processors and C-trees

The concept of organising processors into teams meshes very well with the notion of C-tree.

A team can work on a c-node in the C-tree—each of its member processors working on one of the independent and-parallel goal in that c-node. We illustrate this by means of an example. Consider the query corresponding to the and-or tree of Figure 2. Suppose we have 6 processors P1, P2, ..., P6, grouped into 3 teams of 2 processors each. Let us suppose P1 and P2 are in team 1, P3 and P4 in team 2, and P5 and P6 in team 3. We illustrate how the C-tree shown in Figure 2 would be created.

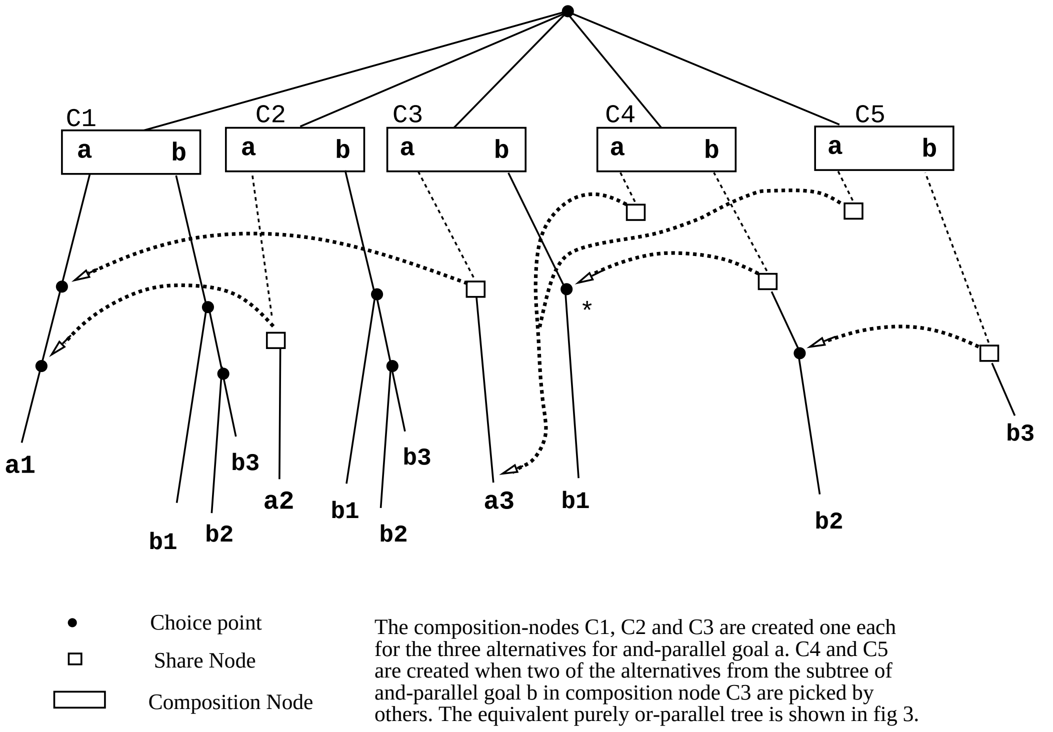


Figure 6: C-tree for 5 Teams

Execution commences by processor P1 of team T1 picking up the query `q` and executing it. Execution continues like normal sequential execution until the CGE is encountered, at which point a choice point node is created to keep track of the information about the different solutions that the CGE would generate. A c-node is then created (node C1 in Figure 2). The CGE consists of two and-parallel goals `a` and `b`, of which `a` is picked up by processor P1, while `b` is made available for and-parallel execution. The goal `b` is subsequently picked up by processor P2, teammate of processor P1. Processor P1 and P2 execute the CGE in and-parallel producing solutions `a1` and `b1` respectively. In the process they leave choice points behind. Since we allow or-parallelism below and-parallel goals, these untried alternatives can be processed in or-parallel by other teams. Thus the second team, consisting of P3 and P4 picks up the untried alternative corresponding to `a2`, and the third team, consisting of P5 and P6, picks up the untried alternative corresponding to `a3`. Both these teams create a new c-node, and restart the execution of and-parallel goal `b` (the goal to the right of goal `a`): the first processor in each team (P3 and P5, respectively) executes the alternative for `a`, while the second processor in each team (P4 and P6, respectively) executes the restarted goal `b`. Thus, there are 3 copies of `b` executing, one for each alternative of `a`. Note that the nodes in the subtree of `a`, between c-node C1 and the choice points from where untried alternatives were picked, are shared between different teams. (See Figure 2.)

Since there are only three teams, the untried alternatives of `b` have to be executed by backtracking. In the C-tree backtracking always takes place from the right to mimic Prolog's behaviour—goals to the right are completely explored before a processor can backtrack inside a goal to the left. Thus, if we had only one team with 2 processors, then only one composition node would be created, and all solutions would be found via backtracking, exactly as in &-Prolog [H86, HG90]. On the other hand if we had 5 teams of 2 processors each, then the C-tree could appear as shown in Figure 6. In Figure 6, the 2 extra teams steal the untried alternatives of goal `b` in c-node C3, This results in 2 new c-nodes being created, C4 and C5 and the subtree of goal `b` in c-node C3 being spread across c-nodes C3, C4 and C5. The topologically equivalent purely or-parallel tree of this C-tree is still the one shown in Figure 3. The most important point to note is that new c-nodes get created only if there are resources to execute that c-node in parallel. Thus, the number of c-nodes in a C-tree can vary depending on the availability of processors.

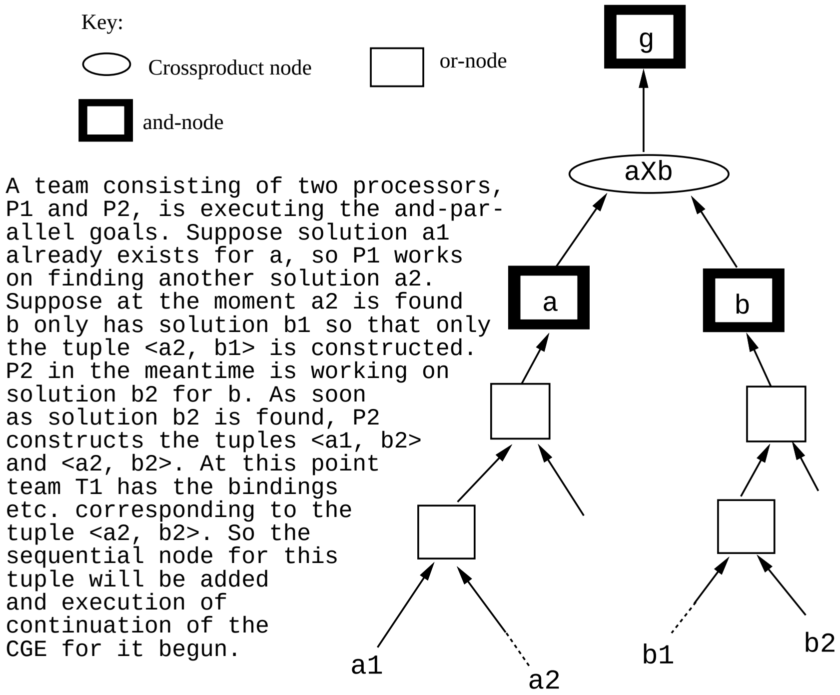


Figure 7: Teams of Processors and Extended And-Or Tree

3.2. Teams of Processors and Extended And-Or Trees

Teams of processors also mesh very well with the Extended And-Or trees†. On reaching a cross-product node, different processors in the team will work on different and-parallel branches in the Extended And-Or Tree. If a processor finds a solution to a goal, it constructs all tuples of the cross-product set that can be constructed at that moment and that contain this solution. It then checks if all its other team-mates have found a solution as well. If so, one of them will pick the tuple containing solutions that were just found, construct the sequential node for it, and continue with the continuation of the CGE. By having a team of processors, no updating of environment is

† Recall that in the AO-WAM the notion of teams of processors was not employed.

needed before continuing with the continuation (contrary to AO-WAM where a *loading* operation is needed before execution of the continuation goal can commence). This is further illustrated in Figure 7.

4. Environment Representation for And-Or parallelism

So far we have described and-or parallel execution with recomputation or reuse at an abstract level. We have not addressed the problem of environment representation in the presence of and- and or-parallelism. In this section we discuss how to extend the Binding Arrays (BA) method [W84, W87] to solve this problem. These extensions enable a team of processors to share a single BA without wasting too much space.

In an earlier paper [GJ93] we argued that environment representation schemes that have constant-time task creation and constant-time access to variable bindings, but non-constant time task-switching, are superior to those methods which have non-constant time task creation or non-constant time variable-access. The reason being that the number of task-creation operations and the number of variable-access operations are dependent on the program, while the number of task-switches can be controlled by the implementor by carefully designing the work-scheduler. This is the reason why we have adopted the Binding Arrays method [W84, W87] for environment representation. Another advantage of such schemes is that the execution tree is dynamically divided into *public* and *private* regions (such as done in MUSE [AK90] and Aurora [LWH90]), and parallel execution takes place in the public part while sequential execution in private parts, resulting in maximal efficiency. Whenever the processors run out of parallel work in the public region, a fraction of the private region of the tree is converted into a public region at a very low overhead and new parallel work produced. In contrast, in other approaches, such as hashing windows [WR87] and environment closing [C87] (i.e. environment representation schemes that either have non-constant time task creation or non-constant time variable access), the separation into private and public region is difficult to achieve, and even if one incorporates it in the implementation, the overhead of converting a public part into private part dynamically will be just too high.

4.1 The Fragmentation Problem in Binding Arrays

Recall that in the binding-array method [W84, W87] an offset-counter is maintained for each branch of the or-parallel tree for assigning offsets to conditional variables (CVs)† that arise in that branch. The 2 main properties of the BA method for or-parallelism are the following:

(i) The offset of a conditional variable is fixed for its entire life.

(ii) The offsets of two consecutive conditional variables in an or-branch are also consecutive.

The implication of these two properties is that conditional variables get allocated space consecutively in the binding array of a given processor, resulting in optimum space usage in the BA. This is important because large number of conditional variables can exist at runtime‡.

In the presence of independent and-parallel goals, each of which have multiple solutions, maintaining contiguity in the BA can be a problem, especially if processors are allowed (via backtracking

† Conditional variables are variables that receive different bindings in different environments [GJ93].

‡ For instance, in Aurora [LWH90] about 1Mb of space is allocated for each BA.

or or-parallelism) to search for these multiple solutions.

Consider a goal with a CGE: `a, (true => b & c), d`. A part of its tree§ is shown in Figure 8(i) (the Figure also shows the number of conditional variables that are created in different parts of the tree). If `b` and `c` are executed in independent and-parallel by two different processors P1 and P2, then assuming that both have private binding arrays of their own, all the conditional variables created in branch `b-b1` would be allocated space in BA of P1 and those created in branch of `c-c1` would be allocated space in BA of P2. Likewise conditional bindings created in `b` would be recorded in BA of P1 and those in `c` would be recorded in BA of P2. Before P1 or P2 can continue with `d` after finding solutions `b1` and `c1`, their binding arrays will have to be merged somehow. In AO-WAM [GJ89] the approach taken was that one of P1 or P2 would execute `d` after updating its Binding Array with conditional bindings made in the other branch (known as the BA loading operation). The problem with BA loading operation is that it acts as a sequential bottleneck which can delay the execution of `d`, and reduce speedups. To get rid of the BA loading overhead we can have a common binding array for P1 and P2, so that once P1 and P2 finish execution of `b` and `c`, one of them immediately begins execution of `d` since all conditional bindings needed would already be there in the common BA. This is consistent with our discussion in section 3 about having teams of processors where all processors in a team would share a common address space, including thus a binding array.

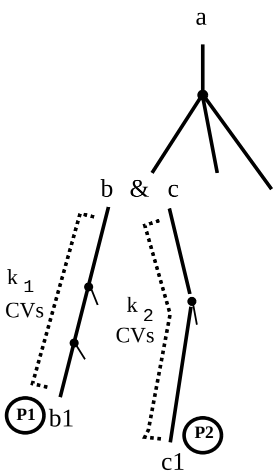


Fig (i): Part of a C-tree

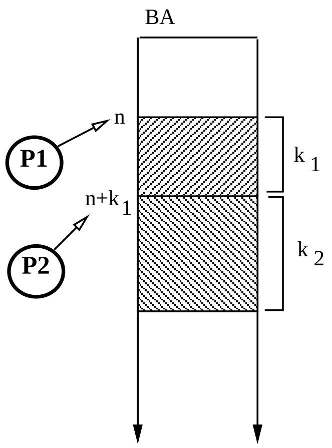


Figure (ii): Optimal Space Allocation in the BA

Figure 8: Binding Arrays and Independent And-Parallelism

However, if processors in a team share a binding array, then backtracking can cause inefficient usage of space, because it can create large unused holes in the BA. Referring again to the tree shown in Figure 8(i), when processors P1 and P2 start executing goals `b` and `c`, then to be able to efficiently utilise space in their common binding array they should be able to exactly tell the amount of space that should be allocated for conditional variables in branches `b-b1` and `c-c1`. Unfortunately, knowing the exact number of conditional variables that are going to arise in a branch is impossible since their number is only determined at runtime; the best one can do is to

§ Note that the tree could either be a Composition Tree or an Extended And-Or Tree depending on whether the CGE is impure or pure.

estimate them. Underestimating their number can lead to lot of small holes in the BA which will be used only once (Figure 9(i)), while overestimating them can lead to allocating space in the BA which will never be used (Figure 9(ii)). Thus, in either case space is wasted and a very large binding array may be needed to exploit independent and-parallelism with or-parallelism.

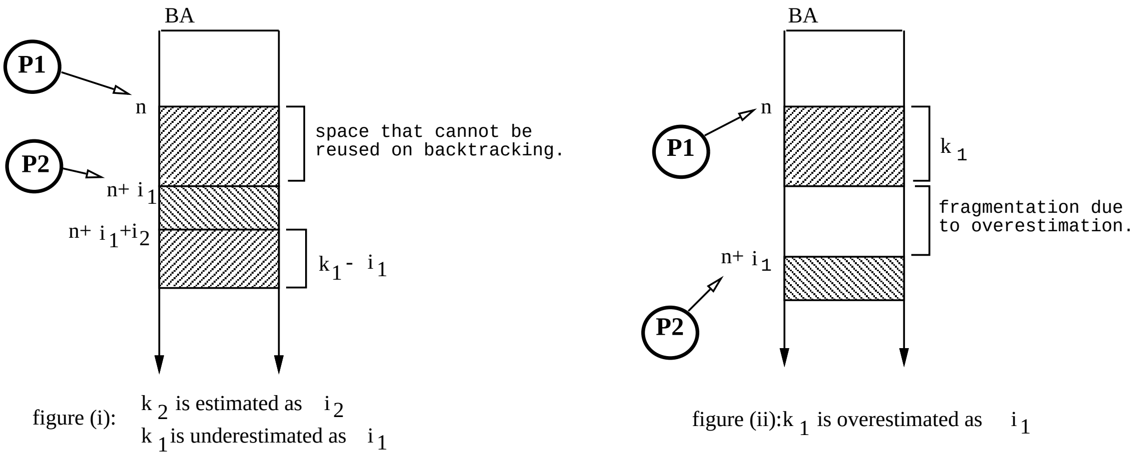


Figure 9: Problems with BA in presence of And-parallelism

4.2. Paged Binding Array

To solve the above problem we divide the binding array into *fixed sized segments*. Each conditional variable is bound to a pair consisting of a segment number and an offset within the segment. An auxiliary array keeps track of the mapping between the segment number and its starting location in the binding array. Dereferencing CVs now involves double indirection: given a conditional variable bound to $\langle i, o \rangle$, the starting address of its segment in the BA is first found from location i of the auxiliary array, and then the value at offset o from that address is accessed. A set of CVs that have been allocated space in the same logical segment (i.e. CVs which have common i) can reside in any physical page in the BA, as long as the starting address of that physical page is recorded in the ith slot in the auxiliary array. Note the similarity of this scheme to memory management using paging in Operating Systems, hence the name Paged Binding Array (PBA)†. Thus a segment is identical to a page and the auxiliary array is essentially the same as a page table. The auxiliary and the binding array are common to all the processors in a team. From now on we will refer to the BA as Paged Binding Array (PBA), and the auxiliary array as the Page Table (PT)‡.

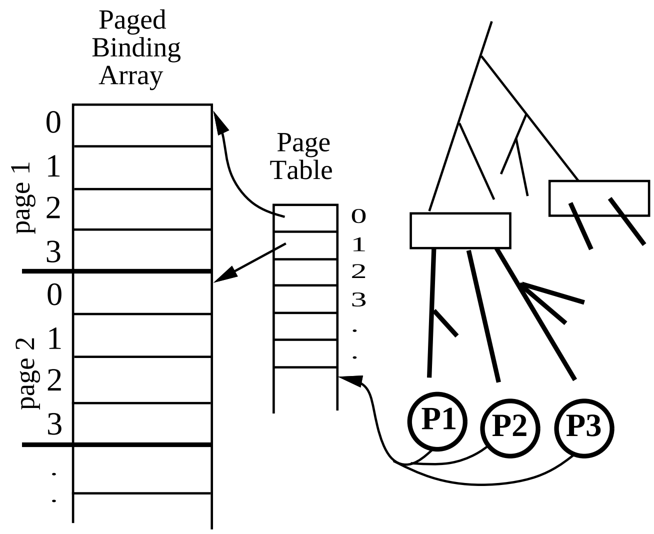
Every time execution of an and-parallel goal in a CGE is started by a processor, or the current page in the PBA being used by that processor for allocating CVs becomes full, a *page-marker node* containing a unique integer id i is pushed onto the trail-stack. The unique integer id is obtained from a shared counter (called `pt_counter`); there is one such counter per team. A new page is requested from the PBA, and the starting address of the new page is recorded in the ith location

† Thanks to David H. D. Warren for pointing out this similarity.

‡ A paged binding array has also been used in the ElipSys system of ECRC [VX91], but for entirely different reasons. In ElipSys, when a choice point is reached the BA is replicated for each new branch. To reduce the overhead of replication, the BA is paged. Pages of the BA are copied in the children branches on demand, by using "copy-on-write" strategy. In ElipSys, unlike our model, paging is not necessitated by independent and-parallelism.

of the Page Table. i is referred to as the page number of the new page. Each processor in a team maintains an offset-counter, which is used to assign offsets to CVs within a page. When a new page is obtained by a processor, the offset-counter is reset. Conditional variables are bound to the pair <i, o>, where i is the page number, and o is the value of the offset-counter, which indicates the offset at which the value of the CV would be recorded in the page. Every time a conditional variable is bound to such a pair, the offset counter o is incremented. If the value of o becomes greater than K, the fixed page size, a new page is requested and new page-marker node is pushed.

A list of free pages in the PBA is maintained separately (as a linked list). When a new page is requested, the page at the head of the list is returned. When a page is freed by a processor, it is inserted in the free-list. The free-list is kept ordered so that pages higher up in the PBA occur before those that are lower down. This way it is always guaranteed that space at the top of the PBA would be used first, resulting in optimum space usage of space in the PBA.



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

Figure 10: Paged Binding Array

Note that empty slots which will never be used again can be created in the page table. The main point, however, is that the amount of space rendered unusable in the PT would be negligible compared to the space wasted in the BA if it was not paged. Note that the PBA would allow backtracking without any space in it being used only once or never being used (the problems mentioned in section 4.1). However, some internal fragmentation can occur in an individual page, because when a CGE is encountered, conditional variables in each of its and-parallel goal are allocated space in a new page, and so if part of the current page is unused, it will be wasted (it may, however, be used when backtracking takes place and another alternative is tried). Given that the granularity of goals in CGEs is preferred to be large, we expect such wastage as a percentage of total space used to be marginal.

Algorithms for dereferencing and creation of a conditional variable are given below:

```
/*K is max. no. of slots in a page*/
create(V)
term *V {
    if (oc > K) ∨ (new and-parallel goal is begun) {
            oc = 0;
            p = head(free_page_list);
            i = pt_counter;     /*exclusive access to pt_counter is obtained.*/
            pt_counter++;
            PT[i] = p;      /*PT is the Page Table.*/
            push_page_marker(i) }
    oc = oc+1;
    V = <i, oc>}


deref(V)        /*unbound variables are bound to themselves*/
term *V {
    if V→tag == VAR
            if not V→value == V
                    deref(V→value)
            else V
    else if V→tag == NON-VAR
            V
        else {              /*conditional vars bound to <i, o>*/
            val = PBA[o + PT[i]];       /*PBA is the paged binding array.*/
            if val→value == val
                    V
            else deref(val) }}
```

Returning to the tree shown in Figure 8(i), when the c-node (or a cross-product node) is created, processor P1 will request a new page in the PBA and push a marker node in the trail-stack. The CVs created in branch `b-b1` would be allocated space in this new page. In the meantime P2 would request another page from the PBA where it will allocate CVs created in branch `c-c1`. P1 and P2 may request more pages, if the number of CVs created in respective branches is greater than K. If the branch `b-b1` fails, then as P1 backtracks, it frees the pages in the PBA that it allocated during execution. These pages are entered in the free-list of pages, and can now be used by other processors in the team, or by P1 itself when it tries another alternative of `b`.

Note that with the PBA, when a team picks an untried alternative from a choice point, it has to task-switch from the node where it is currently stationed to that choice point. In the process it has to install/deinstall the conditional bindings created along that path, so that the correct environment at that choice point is reflected (like in Aurora [LWH90], a conditional binding to a variable is trailed in the trail-stack, along with address of that variable). While installing conditional bindings during task-switching, if a team encounters a page-marker node in the trail-stack whose id is j, it requests a new page in the PBA from the free list, records the starting location of this page in jth location of the page table, and continues. Likewise, if it encounters a page-marker node in the trail-stack during backtracking whose id is k, it frees the page being pointed to by the kth location of the page table.

If the untried alternative that is selected is not in the scope of any CGE, then task-switching is more or less like in purely or-parallel system (such as Aurora [LWH90]), modulo allocation and deallocation of pages in the PBA. If, however, the untried alternative that is selected is in the and-parallel goal g of a CGE, then the team updates its PBA with all the conditional bindings created in the branches corresponding to goals which are to the left of g. Conditional bindings created in g above the choice point are also installed. Goals to the right of g are restarted and made available to other member processors in the team for and-parallel execution. Note that the technique of paged binding array is a generalisation of environment representation technique of AO-WAM [GJ89], hence some of the optimisations [GJ90a] developed for the AO-WAM, to reduce the number of conditional bindings to installed/deinstalled during task-switching, will also apply to the PBA model. Lastly, seniority of conditional variables, which needs to be known so that "older" variables do not point to "younger ones", can be easily determined with the help of the $<i, o>$ pair. Older variables will have a smaller value of i, and if i is the same, then a smaller value of o.

Note that the Paged Binding Arrays operations are independent of whether the execution of a CGE resulted in an Extended And-Or Tree or a Composition Tree. The only difference in the PBA operations *vis a vis* the C-trees and the Extended trees is that in case of the latter a team also allocates a new page when it begins on the continuation of the CGE. In the case of the C-tree this allocation of a new page for the continuation is not necessary. This difference has to do with the fact that solutions of goals in CGE in the Extended Tree are shared, while in the C-tree they are not.

5. Supporting Extra Logical Predicates

As mentioned, if side-effects and extra-logical predicates are present within an independent and-parallel goal of a CGE, then goal-recomputation will be used for executing this CGE. In this section we describe how sequential Prolog semantics is maintained by using the C-tree for solving CGEs. We first describe how side effects and cuts are supported in purely and-parallel models, such as &-Prolog, and purely or-parallel models, such as Aurora. We then present a technique, which is a synthesis of the two approaches, for supporting side-effects in an execution using the PBA.

5.1. Implementing Side-effects and Cuts in Purely Or-parallel Systems

To maintain sequential Prolog semantics, a side-effect predicate should be executed only after all others "preceding it" (preceding, in the sense of left-to-right top-to-bottom Prolog's sequential order of execution) have finished. In practice this condition is very hard to detect and a more conservative condition is used. In a purely or-parallel system, a side-effect predicate is executed only after the or-branch it is working in becomes the leftmost branch in the whole or-parallel search tree [HCC88]. Thus, if a side-effect predicate is encountered by a processor, the processor checks if the node containing that side-effect is in the leftmost branch of the or-parallel tree. If so, the side-effect is executed, otherwise the processor suspends until the branch becomes leftmost. After suspending, the processor may either busy-wait until that branch becomes leftmost, or it can search for work elsewhere, leaving this suspended side-effect for some other processor to execute when the side-effect becomes ready. In the case of cut the requirement that the branch become leftmost before the cut can be executed can be relaxed. The part of the tree that can be safely pruned (i.e. those branches that are to the right of branch b in the subtree in which branch b is leftmost, where b is the branch containing the cut), can be pruned immediately.

The central question now is how to keep track of the "leftmostness" property of or-branches, i.e., how to efficiently check if a branch containing a given node is leftmost. Various techniques have been proposed for this purpose. [A87, KPS88, AK91, BM91, S91]: These techniques essentially keep track of the root node of the subtree in which it is leftmost. This root node is called the *subroot node* [S91]. If the subroot node is identical to the root node of the or-tree, then the node is in a leftmost branch.

5.2. Implementing Side-effects and Cuts in Purely And-parallel Systems

In a purely and-parallel system with recomputation, such as &-Prolog, the problem of accommodating side-effects is somewhat simplified because at any given time only one solution exists on the stacks and the order in which solutions are explored is exactly like in sequential execution of Prolog. If an and-parallel goal in a CGE reaches a side-effect, it must make sure that all the sibling and-branches to its left have finished execution. This is because an and-branch to its left may fail, and in such a case, in the equivalent sequential Prolog execution the side-effect would never be reached and hence would never be executed. Therefore, to preserve sequential Prolog semantics, in a purely and-parallel system a side-effect should be executed only after all sibling and-branches to its left have finished execution. If the CGE containing the branch with side-effect is nested inside another CGE then all sibling and-branches of the inner CGE which are to its left must also have finished, and so on. If any of these and-parallel goals have not finished the execution of the side-effect is suspended until they do. Implementing the cut is simple in purely and-parallel systems because solutions are searched in the same order as Prolog. Like purely or-parallel systems, when a cut is encountered the parts of the tree that can be safely pruned are pruned immediately, and then the execution of goals to the right of the cut continues.

The central question is thus how to detect that all and-parallel goals to the left have finished. Two approaches have been proposed: (i) adding synchronisation code to CGEs using counting semaphores [D87, MH89]; (ii) Recursively checking if all sibling branches to the left have finished [CC89].

5.3. Side-effects in And-Or Parallel Systems with Goal Recomputation

In and-or parallel systems with goal recomputation, each or-parallel environment exists separately—they will only share those parts which will also be shared in equivalent purely or-parallel computation. There is no sharing of independent and-parallel goals (unlike the and-or parallel models with solution sharing). This simplifies the problem of supporting side-effects considerably. A side-effect in a given environment, can be executed when that environment becomes "leftmost" in the composition-tree, where by environment we mean the various and- and or-branches whose nodes participate in producing a solution for the top-level query.

Before we go any further, we define some terms. Given a node n in an and-branch b, the other and-branches of b which correspond to the same composition node (or c-node) as that of b are termed as *sibling and-branches of* n. The composition node is termed as *immediate ancestor c-node* of n, and the and-parallel goal corresponding to n is termed as the *and-parallel goal* of n. Thus, if we consider the node marked with an asterisk in Figure 6, then c-node `C3` would be its immediate ancestor c-node, the subgoal `b` in `C3` its and-parallel goal, and the branch `a3` of `C3` its sibling and-branch.

Recall that there is a 1-1 correspondence between the C-tree and the purely or-parallel tree of a query. Thus, when a side-effect is encountered in a C-tree, Prolog semantics can be supported by executing this side-effect only if its associated node is in the leftmost branch of the topologically equivalent purely or-parallel tree. Given a CGE (true => g_1 & ...& g_n) then the or-parallel subtree of the goal g_i would appear immediately below that of g_{i-1} in the topologically equivalent pure or-parallel tree (for every branch of g_{i-1} the tree for g_i would be duplicated. c.f., Figures 2 and 3). We define the notion of "locally leftmost" to study the relationship between the "leftmostness" property in a C-tree and in a purely or-parallel tree. A node is "locally leftmost" in a subtree if it is in the leftmost branch in that subtree. A subtree is "locally leftmost" in another subtree that contains it, if the root-node of the smaller subtree is "locally leftmost" in the bigger subtree.

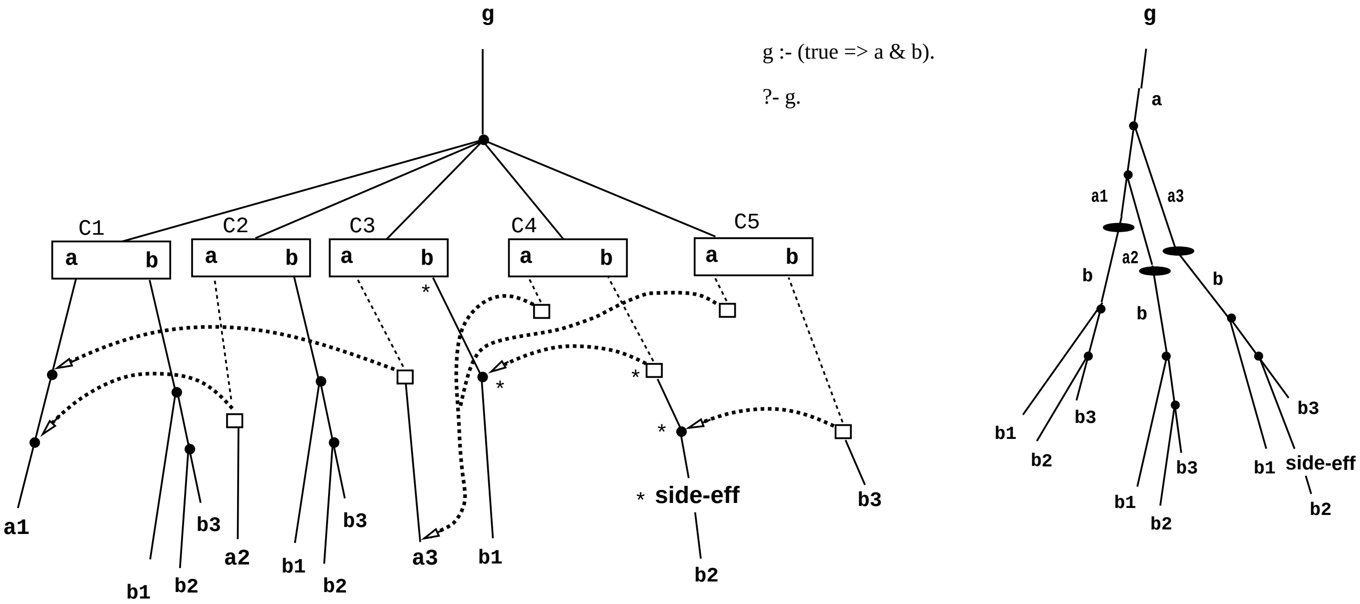


Figure (i): Composition Tree

Figure (ii): Equivalent Or-parallel Tree

Before the side-eff in the second branch of b corresponding to a3 can execute, it should become leftmost in the purely or-parallel tree (Figure ii). Note that in the C-tree (Figure i) this corresponds to the branch containing the side-effect becoming leftmost in the subtree of goal b spread out between the composition nodes C3 and C4 (consisting of nodes marked by asterisks). Also the branch a3 in the composition node should become leftmost in the subtree of a spread across composition nodes C1, C2, and C3, since composition node C4's subgoal a has a share node which points to the terminal node of branch a3.

Figure 11: Execution of Side-effects in C-tree

Consider a node n in the subtree of goal g_j in the CGE (true => g_1 & ... & g_n). Let b_1,b_2 ... b_{j-1} be the sibling and-branches to the left of node n. A node n is in the leftmost branch of the topologically equivalent pure or-parallel tree if and only if: (i) it is locally leftmost in the subtree of its and-parallel goal g_j; (ii) branches $b_1 \ldots b_{j-1}$ are locally leftmost in the subtree of their respective and-parallel goals $g_1 \ldots g_{j-1}$; and (iii) if the CGE containing n is nested inside another CGE then the c-node of n is locally leftmost in the subtree of its and-parallel goal, and this c-node's sibling and-branches to the left are locally leftmost in the subtrees of their respective and-goals, and so on. This follows from the topological equivalence of the C-tree to the corresponding purely or-parallel tree. Hence, if a node containing a side-effect satisfies conditions (i), (ii) and recursive condition (iii) above, this side-effect would be leftmost in the pure or-parallel execution tree, all side-effects before this side-effect would have finished, and hence this side-effect can be safely executed. Otherwise, the side-effect has to suspend, and wait until the condition is satisfied. Note that because we check for "local leftmostness" in the subtree of each and-parallel goal to the left, no ordering of c-nodes is necessary [GS92].

The Composition tree uses *share* nodes to represent computation shared between different c-nodes. Consider a processor trying to verify if its side-effect is leftmost. The subtrees corresponding to the sibling and-branches to the left of the node containing the side-effect may be spread over several composition nodes. To verify whether these nodes are locally leftmost, it is necessary to use the share nodes, follow their pointers to the corresponding choice points (dotted arrows in Figures 2 and 6), and check the local-leftmostness of the nodes that correspond to these choice points. This is further illustrated in Figure 11.

Finally, note that the property of a node being "leftmost" in g_j requires that execution of the sibling and-branches corresponding to goals $g_1 \ldots g_{j-1}$ in the C-tree is finished, otherwise "leftmostness" cannot be determined. Thus, the condition for supporting side-effects in purely and-parallel systems, described previously, is naturally included in the "leftmostness" property.

5.4. Side-effects in Optimal And-Or Parallel Systems

We have explained how side-effects can be implemented in the Composition-Tree. Consider now the case of pure CGEs that are executed through the Extended And-Or subtree model. As far as side-effects are concerned, the main point to note is that the ordering of solutions of a pure CGE is reflected in the left-to-right order of the sequential nodes in its corresponding Extended and-or subtree. Thus, if the continuation of a pure CGE leads to a side-effect (which it eventually will, because even printing the answer to the top-level query should be viewed as a side-effect), the order of the sequential nodes becomes significant (Figure 12).

The leftmostness property can be extended for CGEs that use solution sharing as follows: Consider a pure CGE and its associated Extended And-Or subtree. (This CGE may have other pure nested CGEs inside.) Suppose the continuation corresponding to one of the tuples, say t, of the outermost pure CGE leads to a side-effect (the continuation of nested CGEs cannot lead to a side-effect, otherwise the CGE that encloses them will not be pure). This side-effect can be safely executed if: (i) all tuples to the left of tuple t have been generated; (ii) the continuation of the CGE corresponding to all such tuples to the left of tuple t have completed; and, (iii) the cross-product node corresponding to the outermost CGE is in the leftmost branch of the search tree. In other words, the leftmost check simply boils down to checking if the continuation of the outermost pure CGE (sequential node) that leads to the side-effect is in the leftmost branch of the tree. Thus, if a node n, containing a side-effect, has an ancestor cross-product node then n should be locally leftmost in the subtree rooted at the cross-product node and this cross-product node should be in the leftmost branch of the whole tree. Note that the nested pure CGEs are not involved in the leftmostness check, only the outermost one is. This is because most of the work to help figure out leftmostness was done while constructing the cross-product sets (the requirement that the tuples be inserted in Prolog order). In this regard, the leftmostness check for a pure CGE is far more simpler than that for a impure CGE in which leftmostness has to be checked for all nested CGEs as well. Note that condition (i) above can be costly to verify, it can be dropped if the user tells the system that the order of solutions is not significant.

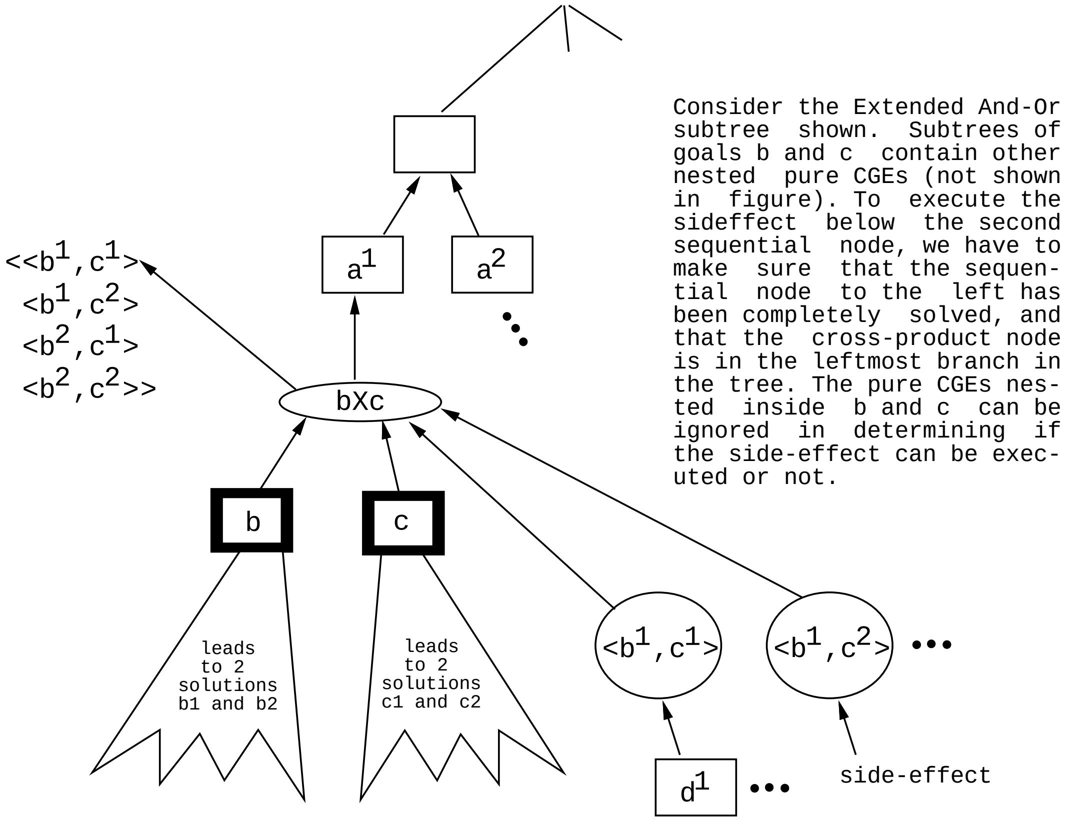


Figure 12: Side-effects and Solution Sharing

The conditions for executing a particular side-effect discussed for the Composition Tree and the Extended And-Or Subtree can now be summarized as follows [GS92]: Given a CGE in our and-or parallel model, a node can execute a particular side-effect: (i) if that node is locally leftmost in the subtrees of its and-parallel goal; (ii) if its sibling and-branches to its left are locally leftmost in subtrees of their respective and-parallel goals; (iii) if the node has an ancestor cross-product then it is locally leftmost in the subtree rooted at the cross-product node and the cross-product node is in the leftmost branch of the tree; and, (iv) if this CGE is nested inside another CGE, then the c-node corresponding to inner CGE must recursively satisfy rules (i), (ii), (iii), (iv).

6. Conclusions and Comparison with Other Work

Ideally, models that exploit both or- and independent and-parallelism from logic programs should attain the difficult goals of obtaining maximum parallelism while avoiding redundant computation and supporting the full Prolog language. Previous models that allow both or-parallelism and the parallel execution of independent goals have been able to either avoid redundant computation or to support full Prolog but not both. In this paper we presented a model that does support the execution of Prolog programs (containing extra-logical features of Prolog), and that does avoid wasteful recomputation.

Our model employs the Composition-tree, when side-effects make recomputation of goals necessary, and the Extended And-Or Tree, when preventing redundant computations is the main consideration. To unite the two different ideas, we use the notion of teams and of Paged Binding Arrays. Teams are groups of processors that work in and-parallel to together form an or-parallel environment. Division of available processors into teams considerably simplifies the implementation of a Composition-Tree model. More surprisingly, teams also reduce the overheads in the cross-product calculation for the Extended And-Or Tree. Finally, teams allow us to use the same data structure, *Paged Binding Arrays*, to represent or-parallel environments in both variants of and-or parallel computations.

The model can be viewed as directly building on previous work on the &-Prolog [HG90] system and on the Aurora system [LWH90], both of which have been very successful in exploiting independent and-parallelism and or-parallelism respectively. If an input program has only or-parallelism, then our model will behave like Aurora (except that another level of indirection would be introduced in binding array references). If a program has only independent and-parallelism then our model will behave like &-Prolog (except that conditional bindings would be allocated in the binding array). As Aurora and &-Prolog do, our model transparently supports the extra-logical features of Prolog, such as cuts and side-effects, which other and-or models [BK88, GJ89, RK89] cannot.

We firmly believe that our and-or parallel model can be implemented efficiently. The environment representation technique developed in this paper is quite general and can be used for even those models that have dependent and-parallelism, such as Prometheus [SK91], and IDIOM (with recomputation) [GSY91]. It can also be extended to implement the Extended Andorra Model [W90, HJ90].

Acknowledgements

We are grateful to David H.D. Warren for his support, and to Raéd Sindaha and Tony Beaumont for answering many questions about Aurora and its schedulers. Thanks are due to Manuel Hermenegildo, Enrico Pontelli, and Kish Shen for many discussions. Thanks are also due to anonymous referees of this journal and of PARLE'92 whose comments led to tremendous improvements. We would also like to thank Professors Daniel Etiemble and J-C Syre, co-editors of this special issue, for considering our work worthy of inclusion, and for their patience while we were revising the paper. This research was initially supported by U.K. Science and Engineering Research Council grant GR/F 27420. Continued support is provided by NSF Grants CCR 92-11732 and HRD 93-53271, Grant AE-1680 from Sandia National Labs, by an Oak Ridge Associated Universities Faculty Development Award, and by NATO Grant CRG 921318. Vítor Santos Costa would like to thank the Gulbenkian Foundation and the European Commission for supporting his stay at Bristol.

References

[A87] K. Ali, "A Method for Implementing Cut in Parallel Execution of Prolog," In *International Symposium on Logic Programming*, 1987, pp. 449-456.

[AK90] K. Ali, R. Karlsson, "The Muse Or-parallel Prolog Model and its performance," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press.

[AK91] K. Ali, R. Karlsson, "Full Prolog and Scheduling Or-parallelism in Muse." In *International Journal of Parallel Programming*, 1991, Vol 19, No. 6, pp. 445–475.

[BK88] Uri Baron, et. al., “The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results,” In *Proceedings of International Conference on Fifth Generation Computer Systems ’88*, Tokyo, pp. 841-850.

[BM91] T. Beaumont, S. Muthu Raman, et al. “Flexible Scheduling or Or-parallelism in Aurora: The Bristol Scheduler,” In *Proceedings of PARLE ’91*, Springer Verlag, LNCS 506, pp. 403-420.

[C87] J. S. Conery, “Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors,” *1987 IEEE International Symposium in Logic Programming*, San Francisco, pp. 457-467

[CC89] S-E. Chang and Y.P. Chiang, “Restrict And-Parallelism Model with Side Effects,” Proceedings of North American Conference on Logic Programming, 1989, MIT Press, pp. 350-368.

[D87] D. DeGroot, “Restricted And-Parallelism and Side-effects,” In *International Symposium on Logic Programming*, San Francisco, 1987, pp. 80-89.

[GJ89] G. Gupta and B. Jayaraman, “Compiled And-Or Parallel Execution of Logic Programs,” In *Proceedings of the North American Conference on Logic Programming ’89*, MIT Press, pp. 332-349.

[GJ93] G. Gupta and B. Jayaraman, “Analysis of Or-parallel Execution Models of Logic Programs,” In *ACM Transactions on Programming Languages and Systems*, 15(4):659-680.

[GJ90a] G. Gupta and B. Jayaraman, “Optimizing And-Or Parallel Implementations,” In *Proceedings of the North American Conference on Logic Programming ’90*, MIT Press, pp. 737-756.

[GH92] G. Gupta and M. Hermenegildo, “Recomputation Based And-Or Parallel Implementations of Prolog,” In *Proceedings of International Conference on Fifth Generation Computer Systems ’92*, Tokyo, 1992.

[GS92] G. Gupta, V. Santos-Costa, “Complete and Efficient Methods for supporting Cuts and Side Effects in And-Or Parallel Prolog,” In *Proc. 4th IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, IEEE Press, 1992.

[GS92a] G. Gupta, V. Santos-Costa, “And-Or Parallel Prolog based on the Paged Binding Array,” In *Proceedings of Parallel Architectures and Languages Europe ’92 (PARLE92)*, Springer Verlag, Lecture Notes in Computer Science 605, 1992.

[GHS93] G. Gupta, M. Hermenegildo, V. Santos-Costa, “And-Or Parallel Prolog: A Recomputation based Approach,” In *Journal of New Generation Computing*, Vol 11(3-4):298-321.

[GSY91] G. Gupta, V. Santos Costa, R. Yang, M. Hermenegildo, “IDIOM: A Model for Integrating Dependent-and, Independent-and and Or-parallelism,” In *Proceedings of International Logic Programming Symposium*, MIT Press, Oct. 1991.

[GJ93] G. Gupta and B. Jayaraman, "And-Or Parallelism on Shared-Memory Multiprocessors," In *Journal of Logic Programming*, (in press).

[H86] M. V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs". *3rd International Conference on Logic Programming*, London, 1986.

[H87] M. V. Hermenegildo, "Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs," In *Proc. 4th ICLP*, MIT Press, 1987, pp. 556–575.

[HCC88] B. Hausman, A. Ciepielewski, and A. Calderwood, "Cut and Side-Effects in Or-Parallel Prolog," In *International Conference on Fifth Generation Computer Systems,* Tokyo, Nov. 88, pp. 831-840.

[HCH87] B. Hausman, et. al., "Or-Parallel Prolog Made Efficient on Shared Memory Multiprocessors," in *IEEE International Symposium in Logic Programming*, San Francisco, 1987.

[HG90] M. V. Hermenegildo, K.J. Green, "&-Prolog and its performance: Exploiting Independent And-Parallelism," In *Proceedings of the 7th International Conference on Logic Programming*, 1990, pp. 253-268.

[HJ90] S. Haridi, S. Janson, "Kernel Andorra Prolog and its Computation Model," In *Proceedings of ICLP*, MIT Press, June, 1990, pp. 31-46.

[KPS88] L. V. Kalé, D. A. Padua, D. C. Sehr, "Parallel Execution of Prolog with Side-Effects," In *J. of Supercomputing*, **2**(2), 1988, pp. 209-223.

[LK88] Y-J. Lin and V. Kumar, "AND-parallel execution of Logic Programs on a Shared Memory Multiprocessor : A Summary of Results", in *Fifth International Logic Programming Conference,* Seattle, WA.

[LWH90] E. Lusk, D.H.D. Warren, S. Haridi et. al. "The Aurora Or-Prolog System", In *New Generation Computing*, Vol. 7, No. 2,3, 1990 pp. 243-273.

[MH89] K. Muthukumar and M. Hermenegildo, "Complete and Efficient Methods for Supporting Side-effects in Independent/Restricted And-Parallelism," In *Proceedings of International Conference Logic Programming*, MIT Press, 1989.

[MH89a] K. Muthukumar and M. Hermenegildo, "Compile-time Derivation of Variable Dependency Using Abstract Interpretation," In *Journal of Logic Programming*, 13(2,3):315–347, July 1992.

[MH89b] K. Muthukumar and M. Hermenegildo, "The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism," In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH89a] K. Muthukumar, M. V. Hermenegildo, "Determination of Variable Dependence Information through Abstract Interpretation." In *Proceedings of the 1989 North American Conference on Logic Programming*, MIT Press.

[RK89] B. Ramkumar and L. V. Kalé, "Compiled Execution of the REDUCE-OR Process Model," In *Proc. of North American Conference on Logic Program '89*, MIT Press, pp. 313-331.

[RS87] M. Ratcliffe, J-C Syre, "A Parallel Logic Programming Language for PEPSys" In *Proceedings of IJCAI '87*, Milan, pp. 48-55.

[S91] R. Sindaha, "The Dharma Scheduler — Definitive Scheduling in Aurora on Multiprocessor Architecture," In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing,* Dallas, TX, 1992.

[SH91] K. Shen and M. Hermenegildo, "A Simulation Study of Or- and Independent And-Parallelism," In *Proceedings of International Logic Programming Symposium,* MIT Press, 1991.

[SK91] K. Shen, "Studies of And-Or Parallelism in Prolog," Ph.D. thesis, Cambridge University, 1992.

[SWY91] V. Santos Costa, D. H. D. Warren, R. Yang, "Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism," In *Proceedings of Principles & Practice of Parallel Programming*, Apr. '91, pp. 83-93.

[VX91] A. Véron, J. Xu, et. al., "Virtual Memory Support for Parallel Logic Programming Systems," In *Parallel Architectures and Languages Europe '91 (PARLE)*, Springer Verlag, LNCS 506, 1991.

[W84] D. S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategies," In *The 1984 International Symposium on Logic Programming,* Atlantic City, pp. 198-202.

[W87] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog – Abstract Design and Implementation Issues," *1987 IEEE International Symposium in Logic Programming*, San Francisco.

[W90] D.H.D. Warren, "Extended Andorra Model with Implicit Control" Talk given at Workshop on Parallel Logic Programming, 7th ICLP, Eilat, Israel.

[WR87] H. Westphal, P. Robert, J. Chassin and J. Syre, "The PEPSys Model: Combining Backtracking, AND- and OR-parallelism," In *1987 IEEE Int'l Symp. in Logic Prog.*, San Francisco, pp. 436-448.