Parallel Symbolic Computation in ACE

Enrico Pontelli and Gopal Gupta
Laboratory for Logic, Databases, and Advanced Programming
New Mexico State University
Box 30001, Dept. CS
Las Cruces, NM 88003
{gupta,epontell}@cs.nmsu.edu
http://www.cs.nmsu.edu/lldap

Abstract

We present an overview of the ACE system, a sound and complete parallel implementation of Prolog that exploits parallelism transparently (i.e., without any user intervention) from AI programs and symbolic applications coded in Prolog. ACE simultaneously exploits all the major forms of parallelism—*Or-parallelism*, *Independent And-parallelism*, and *Dependent And-parallelism*—found in Prolog programs. These three varieties of parallelism are discussed in detail, along with the problems encountered in their practical exploitation. Our solutions to these problems, incorporated in the ACE system, are presented. The ACE system has been implemented on Sequent Symmetry and Sun Sparc Multiprocessors; performance results from this implementation for several AI programs are presented, that confirm the effectiveness of the choices made.
Keywords: Prolog, Parallel AI, Or-Parallelism, And-Parallelism.

1 Introduction

An important property of logic programming languages is that they are single assignment languages. Unlike conventional programming languages they disallow destructive assignments and explicit control information. Not only does this allow cleaner (declarative) semantics for logic programs, and hence a better understanding of them by their users, it also permits an evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can be executed in any order without affecting the meaning of the program [51]. In particular, these operations can be performed by the evaluator *in parallel* [27].

Parallel execution of the different operations in a logic program results in *implicit* exploitation of parallelism, i.e., no input is required from the user to exploit parallelism; rather, parallelism is exploited by the evaluator at run-time itself. Clearly, the approach to exploit parallelism implicitly has significant advantages over the "explicitly parallel programming" approach, i.e. the approach based on extensions of logic programming languages with explicit constructs to deal with parallel execution (fork operations, synchronization constructs, etc.), and total *user control* on their use in the exploitation of parallelism[1]. Thus, implicit exploitation of parallelism makes the parallelization process transparent to the programmer.

[1]Note that many systems adopting implicit exploitation of parallelism also extend the language with new constructs, but the introduction of these constructs in the source program is automatically performed by the compiler.

To achieve this goal of exploiting parallelism implicitly from Prolog programs, the ACE model [30] was proposed. The ACE system is a realization of the ACE model, designed and implemented by the authors. The ACE systems aims to (fruitfully) exploit maximum possible parallelism from Prolog. It does so by exploiting all forms of parallelism present in Prolog programs. The ACE parallel Prolog system has recently been completed, and preliminary findings are reported in this paper. The purpose of this paper is twofold:

1. The paper illustrates the general issues related to building systems that automatically exploit the different levels of and- and or-parallelism present in logic programs. The specific solutions adopted in the ACE system are described, with emphasis on its innovative aspects. ACE is arguably the first implemented system capable of efficiently and simultaneously exploiting or-parallelism, independent and-parallelism, and a very general form of dependent and-parallelism. This paper highlights the achievements of ACE as the first system to exploit all forms of parallelism. In particular, performance results in exploiting dependent and-parallelism and combined and/or-parallelism are reported here for the first time.

2. Many previous proposals in the area of parallel logic programming have not paid sufficient attention to establishing the *practicality* of the approach of implicitly exploiting parallelism. Exploitation of parallelism must guarantee good speedups coupled with good overall efficiency. This also means that the parallel system should be capable of exploiting parallelism from large, complex, real-life applications. In the context of this paper we illustrate the effectiveness of ACE on a variety of AI applications (independently developed by other researchers). The results in terms of ease of parallelization and speedups achieved are illustrated throughout the paper, and they confirm the validity of the design choices made in the ACE system.

1.1 Logic Programming languages and Prolog

Logic Programming [51] is a well-known programming paradigm based on a subset of First Order Logic—called *Horn Clause Logic*. A program is composed of a set of clauses (i.e., implications) of the form $A \leftarrow B_1, \ldots, B_n$ and execution is driven by a *query* (or *goal*) of the form $\leftarrow B_1, \ldots, B_n$.

Given a program P and a query $\leftarrow B_1, \ldots, B_n$, its execution entails verifying if the conjunction $B_1 \wedge \ldots \wedge B_n$ is a logical consequence of the program P—i.e., if $P \models B_1 \wedge \ldots \wedge B_n$. Once the verification process is complete, the solution to the query is represented by the bindings generated for variables that occur in the query. The execution process is based on two basic mechanisms: *Resolution* and *Unification* [67]. Unification of two atoms A and B is the process of computing a substitution[2] σ (the *unifier* of A and B) such that $A^\sigma = B^\sigma$.

The resolution process works as follows. Given a query $\leftarrow B_1, \ldots, B_{i-1}, B_i, B_{i+1}, \ldots, B_n$, the resolution process selects a literal (also called a subgoal) B_i from the query, and a clause $H \leftarrow \bar{S}$ from the program, such that H and B_i unify. It then constructs a new *resolvent*

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

where σ is the (most general) unifier of H and B_i. The new resolvent is treated as the next query, and this process continues until the resolvent becomes empty.

In general, no restrictions are imposed on the order in which a logic programming interpreter selects the subgoals and the clauses used. At any given step, there may be several subgoals in the query, as

[2] The notation A^σ indicates the application of the substitution σ to the atom A.

well as several clauses, that may potentially be chosen for resolution. Thus, the process of selecting a subgoal or a clause at each resolution step is *non-deterministic.* Since each subgoal present in a query is to be solved in order to arrive at an answer, the non-determinism present in the subgoal-selection process is termed *don't care* non-determinism. On the other hand, since each clause chosen at a given step potentially leads to a distinct solution, the non-determinism present in the clause-selection process is termed *don't know* non-determinism. Depending on the strategy *(computation rule*) chosen to resolve away this non-determinism, different languages and different semantics can be obtained [51].

Prolog [73] is the most popular logic programming language. Prolog's operational semantics selects subgoals and matching clauses as follows:

- The query is treated as a *list* of subgoals and the leftmost subgoal is always selected.
- The program is treated as a sequence of clauses and the clause that comes textually first in the program is always selected.

Prolog extends pure Horn clause logic with extra-logical features, with the aim of making programs more efficient and the language more expressive. These extensions consist of:

- **Meta-logical Predicates:** that are used to manipulate terms and to query the state of the computation (e.g., finding the instantiation state of the variables).
- **Cut:** that allows programmers to prune parts of the search tree explored during execution.
- **Side-effect and Database predicates:** that are used to perform I/O and to dynamically modify the structure of the program (e.g., dynamically add new clauses to the program).

Note that most of the programs containing extra-logical predicates are sensitive to the order of execution, i.e. the outcome of the execution depends on the selection strategies adopted.

Logic programming systems attempt to find all possible solutions to a user's query by the process of *backtracking.* At any given step, if a clause is chosen that leads to a failure (i.e., a situation where no clauses can be used to continue the execution), then the system backtracks to that step, and selects the textually next clause to find a solution. Thus, more solutions can be obtained by forcing an artificial failure after each solution (in most system this is done by the user, who types a ';' after a solution is reported).

1.2 Prolog Application Areas

Experience shows that Prolog can be successfully employed in virtually any application area. Logic programming has been actively used as a tool for the development of deductive database applications, in which the traditional *retrieval* capability of the database is integrated with powerful *inference mechanisms* [77]. Intelligent compilers also have been realized in Prolog. Extensions of logic programming such as *constraint logic programming* have been successfully applied to efficiently solve practical operation research problems, like resource allocation and scheduling [68]. Nevertheless, AI and symbolic computation remain the application areas that have benefited the most from the logic programming paradigm. Prolog has been used with extreme success in developing applications in a wide variety of AI-related areas, such as natural language processing [26], planning [65], expert systems [5], etc. Not only can AI and symbolic computation problems be efficiently coded in Prolog, our aim in this paper is to show that such problems can be effectively parallelized and their execution speeded up on parallel Prolog systems.

1.3 Parallelism in Logic Programming

Logic Programming offers excellent opportunities for *implicit* exploitation of parallelism (i.e. exploitation of parallelism without any—or very limited—help from the user). This follows from the fact that both don't know non-determinism and don't care non-determinism can be translated into parallelism. Since any subgoal from the current resolvent can be picked up for execution without affecting the final outcome, many of them can be solved at once to speed up the execution. Likewise, since any clause can be chosen for resolution from the set of candidate clauses, many of them can be tried simultaneously. Thus, there are different forms of parallelism that can be exploited from logic programs. Two major forms are customarily defined [11]:

1. **Or-parallelism:** Or-parallelism arises when multiple clauses match a subgoal, and the bodies of these clauses are tried in parallel. Each clause gives rise to an execution path that may potentially lead to a solution. This can be depicted as a tree (called the *or-tree*). The or-tree has a node for each selected subgoal and a branch for each clause matching that subgoal. Figure 1(i) shows an example or-tree.

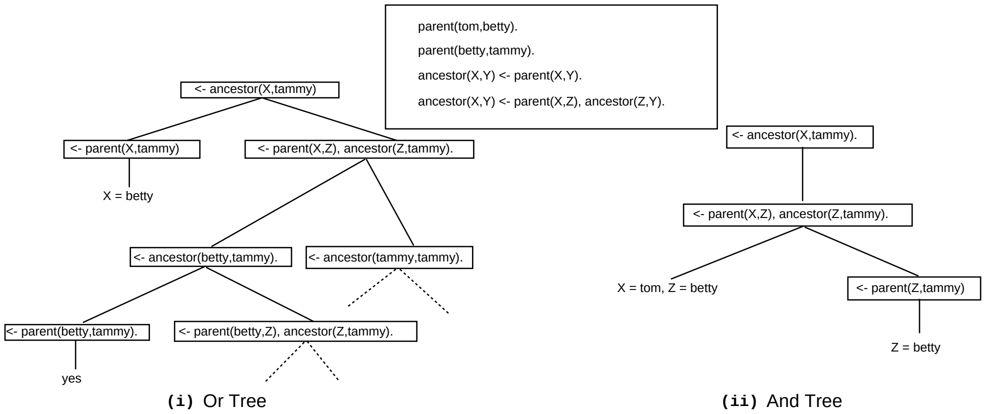

Figure 1: And-tree and Or-tree

In a sequential system the or-tree is visited in a predetermined order (for Prolog, a depth-first order of traversal is used). Whenever a leaf of the tree is reached (corresponding to a success—a solution has been detected—or a failure—no further resolution steps are possible) and a new solution is desired, a node with unexplored branches is selected, backtracked to, and traversal of the new branch started. Nodes in an or-tree with multiple branches are termed *choice-points*. In an or-parallel system different computing agents concurrently explore different branches of the or-tree.

2. **And-parallelism:** And-parallelism arises when multiple subgoals in the resolvent are tried in parallel. And-parallelism can be depicted with the help of an *and-tree*: each node represents (part of) the resolvent and each branch corresponds to one subgoal in it. Figure 1(ii) shows an example and-tree.

In a sequential execution subgoals are selected one at a time from the current query; the computation stops when the query is empty or when unification fails. This is equivalent to a traversal of the and-tree in a predefined order (again, in Prolog the tree is traversed in depth-first, left-to-right order). And-parallelism is obtained by allowing concurrent resolution of different subgoals of a given query—i.e., allowing different computing agents (processors) to explore different branches of the and-tree.

It is common to distinguish two levels of exploitation of and-parallelism:

(a) *Independent and-parallelism (IAP)* arises when more than one goal is present in the query or in the body of a clause, and the runtime bindings for the variables in these goals are such that two or more goals are *independent* of one another. Two subgoals $\mathcal{A}$ and $\mathcal{B}$ are *independent* if either the two subgoals do not share any unbound variables, or the bindings produced by one for the common variables will not affect the computation of the other subgoal [41] (i.e., they either agree on the bindings for the common variables or the second subgoal simply does not access such variables during its execution [39, 40]). Parallel execution of such goals gives rise to (independent) and-parallelism.

(b) *Dependent and-parallelism (DAP)* arises when two or more goals in the body of a clause have at least a common variable and, during the parallel execution, the different subgoals actually compete in creating bindings for such variable. Dependent and-parallelism can be exploited in two ways:

    i. The dependent goals are executed in a fairly independent fashion, eventually supplying them with separate copies of the environments to guarantee independence. An additional step is required a posteriori to verify consistency of the bindings produced for the shared variables (*back-unification*). This can be either realized at the end of the execution of the subgoals (*goal-level* synchronization, adopted for example in Epilog [80] and ROPM [48]), or immediately after a conflicting binding (*binding-level* synchronization, adopted for example in the *Optimistic* schemes proposed by Drakos [19] and Tebra [75]).

    ii. for each variable a unique subgoal is identified (the *generator* or *producer*) and allowed to produce bindings for such variable, while the other subgoals accessing such variable (*consumers*) are limited to read-only accesses. Within this scheme, there is a large variety of proposals on how to manage the producers and consumers (e.g., static [69] vs. dynamic [70], consumers started a-priori [70, 75, 54] or only after the producer has bound the variable [11, 47]).

    Case 2(b)i is very similar to independent and-parallelism. Case 2(b)ii includes one of the most popular forms of dependent and-parallelism, the *stream-parallelism*, where producer and consumers incrementally write and read parts of a large structure (typically a list). This is useful for speeding up producer-consumer interactions (e.g., those found in system programs, and in list-processing programs). Stream-parallelism forms the basis for Committed Choice Languages (e.g., Parlog [10], GHC [76], and Concurrent Prolog [69]). More general models for dependent and-parallelism (like DDAS/DASWAM [70] and ACE) can subsume the most common cases of stream-parallelism.

In theory, the different kinds of parallelism should be orthogonal to each other, i.e., one could be exploited without affecting the exploitation of the other. Thus, it should be feasible to exploit all of them simultaneously. However, in practice this does not hold true. The different forms of parallelism impose conflicting requirements on the underlying engine, making the task of concurrently supporting their exploitation extremely challenging. Thus, no *efficient* parallel system has been built yet that achieves this. A system that exploits maximal parallelism from logic programs while achieving the best possible performance is the ultimate goal of researchers in parallel logic programming. The ACE system, described in this paper, is a step towards that goal.

Although parallel logic programming systems are still being researched, very mature and robust systems such as MUSE [1], Aurora [52], Eclipse [20], Andorra-I [12], &-Prolog/CIAO [38, 37], and various implementations of committed choice languages [25, 74], are available today (however, most of these systems only exploit a single form of parallelism). Parallel logic programming got a big boost when it was selected by the Japanese for their Fifth Generation project as a vehicle for obtaining fast and intelligent systems based on parallel processing [55] (one should, however, note that the Fifth Generation Project focused on exploiting parallelism through committed choice logic languages rather than through traditional logic languages such as Prolog). The ACE system [30, 61] is the first system developed that exploits or-parallelism, independent and-parallelism, and dependent and-parallelism in a single framework. An efficient implementation of ACE exists, which has shown good performance over a large range of problems (with an average overhead of 5-15% over sequential executions).

2 And- and Or-parallel Execution: An Abstract View

And-Or parallel execution of a logic programming system can be viewed as parallel traversal of an and-or tree. An and-or tree has or-nodes and and-nodes. Or-nodes are created when there are multiple ways to solve a goal. An and-node is created when a goal invokes several conjunctive subgoals. Sequential execution is shown in the and-or tree as a linear branch. The or- and and-nodes represent *sources of parallelism*, i.e., points of the execution where it is possible to fork parallel computations. Figure 2 illustrates an and-or tree for a logic program ('&' in the program indicates a parallel conjunction, i.e. a conjunction of subgoals that can be executed in parallel). Note that the and-nodes and the or-nodes may be descendents of each other during parallel execution of the and-or tree. This results in nesting of and-parallelism and or-parallelism within each other, making the management and implementation of parallel execution enormously complex. The complexity is exacerbated by the fact that non-deterministic systems admit backtracking, and incorporation of backtracking implies that a computation should be restorable to every point where a choice was made. In the context of and-or parallelism this means saving extra information and allocation of extra data-structures, so that in the event of backtracking the state of the computation can be restored.

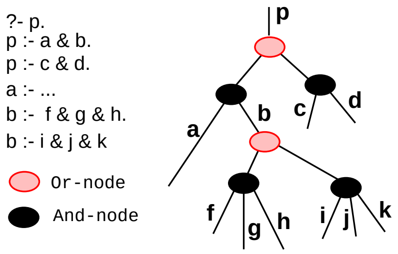

Figure 2: And-Or parallel Tree

Given the and-or tree for a program, its sequential execution amounts to traversing the and-or tree in a pre-determined order. Parallel execution is realized by having different computing agents (processors) concurrently traversing different parts of the and-or tree in a way consistent with the operational semantics of the programming language. By operational semantics we mean that data-flow and control-flow dependencies should be respected during parallel execution (similar to loop parallelization of Fortran programs, where flow dependencies have to be preserved [81]). Typically, this amounts to guaranteeing that

- the order in which bindings to variables are performed during the parallel computation should not affect neither the soundness, the completeness, nor the order in which solutions are computed w.r.t. the corresponding sequential execution;
- the *order-sensitive* operations present in the execution (e.g., input/output operations) are performed in the same order as in a sequential execution.

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

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

These reasons makes research in design of *efficient* implementation models for parallel logic programming systems extremely challenging. These models need to take into account the existence of overheads and try to reduce their impact on the overall performance of the parallel execution.

Having given the reader some idea of the difficulties encountered in designing an efficient parallel logic programming system, in the rest of the paper, we discuss or-parallelism, independent and-parallelism, and dependent and-parallelism in greater detail, describing the problems that arise in exploiting them individually, and illustrating the specific solutions adopted within the ACE system for solving these problems. Performance figures of the ACE system for some AI programs are also presented. Conclusions and directions for future research close our presentation.

3 Or-parallelism

Or-parallelism arises when a subgoal can unify with heads of more than one clause. In such a case the bodies of these clauses can be executed in parallel with each other giving rise to or-parallelism.

Example 3.1 *A breadth-first planner for an Action Calculus (developed by M. Gelfond and A. Provetti [65]) generates plans using the following top-level procedure*[3]*:*

```
makeAplan(F,PlansIn,[H|T]) :-
    addAstratum(PlansIn,NewPlans),
    collectValidPlans(F,NewPlans,[H|T]).

makeAplan(F,PlansIn,ValidPlans) :-
    addAstratum(PlansIn,NewPlans),
    makeAplan(F,NewPlans,ValidPlans).
```

[3]The actual code is about 400 lines long, only a fragment of the code is shown here.

where the first clause attempts to find a solution to the problem within one more step of the previously generated partial plans, while the second clause assumes that the solution will be in plans of greater length. Both the clauses can lead to a solution of the problem, and it is very likely that they will both generate very large computations. Thus, the two clauses are excellent candidates for or-parallelism, i.e., they can be tried in parallel in response to a query.

Or-parallelism manifests itself in a number of applications [49]. It arises while exercising rules of an expert systems where multiple rules can be fired simultaneously to achieve a goal. It also arises in applications that involve natural language sentence parsing: the various grammar rules can be applied in or-parallel to arrive at a parse for a sentence (e.g., non-deterministic parsing [26]). Or-parallelism frequently arises in database applications where there are large numbers of clauses. Or-parallelism also arises in generate and test kind of problems—the various alternatives to be tested can be generated in or-parallel.

3.1 Implementation of Or-parallelism

In principle, or-parallelism should be easy to implement since the various alternative clauses which unify a given literal are independent of each other, thus requiring little communication between processors. However, in practice, implementation of or-parallelism is difficult because the computation performed before an or-node with multiple alternatives is *shared* among the processors which are exploring such alternatives (Figure 3). During execution of a resolution step, new variables may be created (the variables are local to the clause used for the resolution). Later on the computation may split, due to the presence of or-parallelism, and the different parallel executions may clash in trying to read and/or write the variables created in the shared part of computation. Figure 3 illustrates this situation by visualizing the computation as a tree, where different branches represent alternatives explored in parallel. In the figure we have two branches generating independent bindings for a variable created in the shared part of the computation. In sequential execution, only one branch exists at a time, so bindings for `X` can be stored in-place and later removed when `X` is unbound on backtracking. In an or-parallel execution, in contrast, multiple bindings are produced simultaneously, and they cannot be stored in the (single) memory location allocated to `X`. Instead they have to be stored in a location associated with the branch that created the binding.

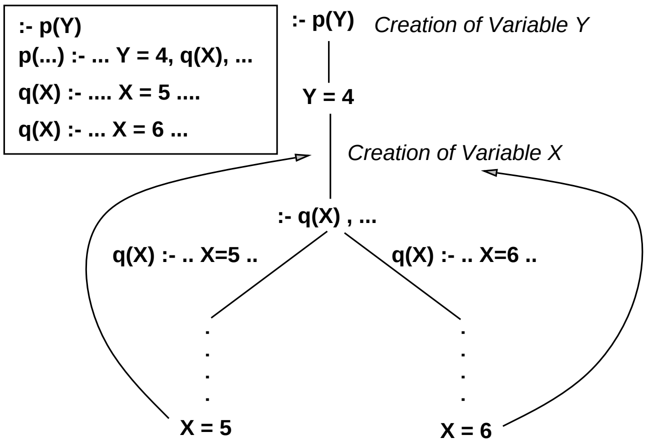

Figure 3: Environment Problem in or-parallelism

To summarize, given two nodes in two different branches of or-tree, all nodes above (and including) the least common ancestor node of these two nodes are shared between the two branches. A variable created in one of these ancestor nodes might be bound differently in the two branches. The environments of the two branches have to be organized in a fashion that, in spite of the ancestor nodes being shared, the correct bindings applicable to each of the two branches are easily discernible. Basically we can distinguish between two classes of variable bindings:

Unconditional Bindings: a binding is called unconditional if there are no intervening choice points between the node where the variable was created and the node where the binding is generated. (e.g., variable `Y` in Figure 3). An unconditional binding can be stored in the memory location allocated for that variable, as the variable is guaranteed to have a single, unique binding.

Conditional Bindings: a binding created in an or-parallel branch for a variable created before the branching took place is said to be conditional. Different branches may generate different conditional bindings for the same variable (e.g. variable `X` in Figure 3). Conditional bindings have to be associated with the or-parallel branch that created them.

The main issue in implementing or-parallelism is the efficient representation of multiple environments that co-exist simultaneously in the or-parallel tree corresponding to a program. Note that the main problem in management of multiple environments is that of efficiently representing and accessing the conditional bindings; the unconditional bindings can be treated as in normal sequential execution of logic programs (i.e., they can be stored in-place in the shared environment as mentioned earlier). Essentially, the problem of multiple environment management has to be solved by devising a mechanism where each branch has some private area where it stores all the conditional bindings applicable to itself. A number of execution models have been proposed in the literature for exploiting or-parallelism (a listing of about 20 of them can be found in [31]). These models differ in the technique they employ for solving the problem of environment representation. The ACE system tackles the or-parallelism implementation issues using a model called *stack-copying*, originally developed for the Muse or-parallel system [1, 2].

The *stack-copying* model assumes a number of extended sequential Prolog engines, called *workers*, each with its own local address space, and some global space shared by all workers. A worker consists of two components: the *engine*, which performs the actual Prolog work, and the *scheduler*. The schedulers, working together, schedule the parallel work between engines and support the sequential semantics of Prolog. In the stack-copying method a *separate* environment is maintained by each processor in which it can write without causing any binding conflicts. Each agent stores all the conditional bindings generated in its local environments. When an idle agent P_2 picks an untried alternative to resolve a certain subgoal, left by another agent P_1, it copies all the stacks of P_1. By doing this we are guaranteed that each agent will have a local copy of all the environments—thus avoiding any conflicts during the binding process. Furthermore, the process of performing the copying operation can be made extremely efficient by introducing various optimizations (e.g., incremental stack copying [2]). Figure 4 shows an example of using stack-copying during the execution of the previously mentioned planner program.

The ACE system controls the granularity of jobs at run-time by avoiding sharing very small tasks. The idea is that when a busy processor reaches a situation in which it has only one *local* choice-point (i.e., the other processors have not been notified yet of the existence of such choice-point), it will make it visible to the other workers only when that node is still alive after a certain number of Prolog procedure calls. Without such a mechanism the gains due to parallel execution can be lost as the number of workers is increased—as the computation of one worker can get trashed by the continuous requests for copying even the smallest computations.

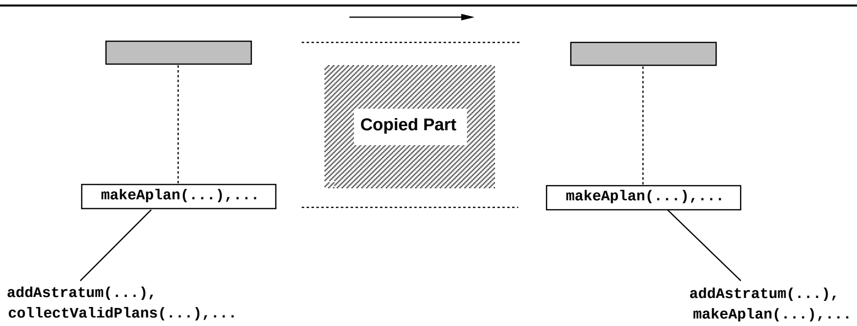

Figure 4: Example of Stack-copying Computation

The speed-ups obtained are near linear for programs with large amounts of or-parallelism. For programs that do not have enough or-parallelism to keep all available workers busy, the speed-ups are (near) linear up to the point where all parallelism is exploited. The speed-up does not increase or decrease thereafter with increase in the number of workers. For programs with no or very low or-parallelism, the speed-ups obtained are close to 1 due to very low parallel overheads. Table 1 shows the result of running two benchmarks, namely, the 8-queens problem and Gelfond and Provetti's Planner, on ACE. The speed-ups obtained for these two problems are substantial. Note that the planner program is quite large, about 450 lines of Prolog code. Thus, programs of non-trivial size can be speeded up with or-parallelism.

Goals	ACE workers			
executed	1	5	10	15
queens	6.70	1.37 (4.89)	0.71 (9.44)	0.50 (13.4)
plan	3.31	0.83 (4.0)	0.43 (7.70)	0.26 (12.73)

Table 1: Runtimes (in seconds) and speedups (in parentheses) of ACE.

The idea of using stack-copying to support or-parallelism is not new and has been adopted in other or-parallel implementations [1, 20]. Nevertheless, ACE contributes in various ways in improving technology in this area:

1. stack-copying mechanisms have been generalized to encompass the presence of and-parallelism. In turn, this lead to the development of more precise and effective methods for keeping track of distribution of work in the or-tree;

2. the scheme for exploiting or-parallelism has been refined, allowing to take full advantage of the most common patterns of or-parallel execution—especially in the case of data-parallelism [32]. Figure 5 illustrates some instances of improvement in execution speed achieved by ACE with respect to the Muse or-parallel system (which is, to date, the fastest or-parallel system available); in particular figure 5 takes advantage of an optimization scheme called *Last Alternative Optimization (LAO)*, which generalizes the concept of tail-recursion optimization to the case of or-parallel computations [32, 33].

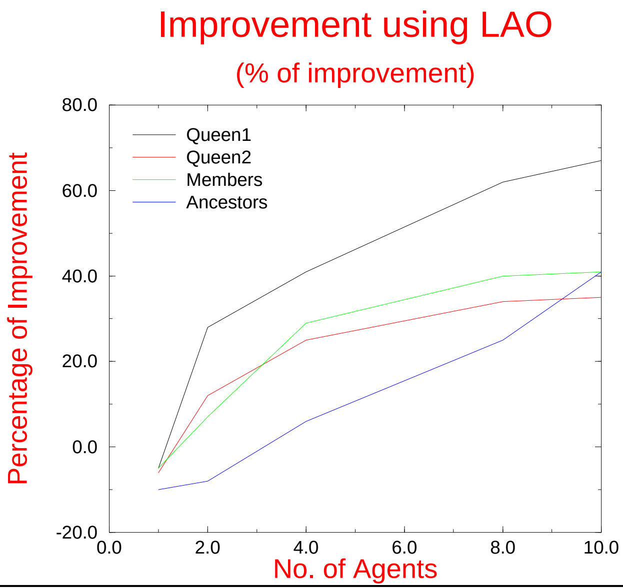

Figure 5: Improvement in Execution speed in Or-parallel Executions

4 Independent And-parallelism

Independent and-parallelism arises when two or more *independent* subgoals can be executed in parallel. Given two subgoals $\mathcal{A}$ and $\mathcal{B}$, these are defined to be independent with respect to a program P and a variable substitution σ if

- for each computed substitution θ obtained from the resolution of $\mathcal{A}^\sigma$ (w.r.t. P), the computation tree generated by resolving $\mathcal{B}^\sigma$ is structurally equivalent to the one obtained from $\mathcal{B}^{\sigma\circ\theta}$; and,
- for each computed substitution θ' obtained from the resolution of $\mathcal{B}^\sigma$ (w.r.t. P), the computation tree generated by resolving $\mathcal{A}^\sigma$ is structurally equivalent to the one obtained from $\mathcal{A}^{\sigma\circ\theta'}$.

Intuitively, the previous definition states that the bindings produced by one subgoal do not affect the computation of the other subgoal—i.e., either the two subgoals do not share any variable (*strict* independence), or the only one them uses each of the common variables (*non-strict* independence)[4].

Example 4.1 *We illustrate independent and-parallelism with a natural language processing application (developed by J. Wiebe's group at New Mexico State University) for translating Spanish dialogs into English. The whole process involves applying six different analysis rules to a frame-based representation of the dialog, and combining the results obtained from these rules. This is expressed by a clause of the form (once again, only a fragment of the program is shown here, the actual program is about 5000 lines long):*

```
collect_int(PrevAILT,PrevUtter,CurILT,T_Utter,PartialList) :-
     int1(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P1),
     int2(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P2),
```

[4]Technical discussions about the notions of strict and non-strict independent can be found elsewhere [53]. A detailed analysis of these concepts is beyond the scope of this paper.

```
int3(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P3),
int4(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P4),
int5(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P5),
int6(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P6),
combine_output([P1,P2,P3,P4,P5,P6],[],PartialList).
```

As we can observe, the execution of goals `int1` *through* `int6` *is independent from the each other (the common arguments* `PrevAILT`, `PrevUtter`, `CurILT`, *and* `T_Utter` *can be easily shown to be ground whenever this clause is executed). Also, the combination of the outcomes is performed only once at the end by the* `combine_output` *predicate. This suggests the possibility of executing each* `int`*i subgoal in parallel with the others. This is an example of independent and-parallelism.*

And-parallelism manifests itself in a number of applications—those in which a given problem can be divided into a number of independent sub-problems. For example, it appears in divide and conquer algorithms, where the independent recursive calls can be executed in parallel (e.g., matrix multiplication, quicksort, etc.). Or-parallelism arises only in non-deterministic programs, whereas and-parallelism arises in non-deterministic as well as deterministic programs. Thus, exploitation of and-parallelism results fruitful in the majority of the applications, and is necessary in order to speed up highly deterministic applications.

4.1 Dependency Detection for Independent And-parallelism

In this section we examine the problems associated with implementing and-parallelism.

The main issue is represented by the *dependency detection* problem, i.e., detecting when two or more subgoals can be computed in and-parallel without affecting each other's execution. If and-parallel execution is performed without caring for data-dependencies, then it may lead to erroneous or wasteful computations. If we allow unrestricted parallel execution of a goal like `:- r(X), s(X)`, we may encounter a situation where `s` produces a huge number of possible bindings for X while `r` accepts only a very few of them—in such a case

1. wasted work is performed, since a sequential execution would have pruned the space of solutions of `s`; this is because, in sequential execution, `s` would have been called with `X` bound to some value.

2. if `r` and `s` are allowed to executed independently, then every value produced by `s` for `X` needs to be unified with the values produced by `r` for verifying consistency (this process is called *back-unification*, and can introduce considerable overhead).

Data dependencies cannot be detected completely at compile time (though compile-time analysis has been shown to be quite effective in most of the cases [24, 6, 15]): a clause like `p(X,Y) :- r(X),s(Y)` may appear to allow independent parallel execution of its body, but at run-time the two variables `X, Y` may be aliased (e.g., by a query `:-p(Z,Z)`). On the other hand a continuous, complete run-time check would introduce an excessive overhead.

A number of approaches have been proposed for dealing with data dependencies. They range from purely compile-time techniques (in which the compiler attempts to detect data dependencies and annotate the safe sources of parallelism [8]), to purely runtime ones (where all variable dependencies are detected at run-time [11]).

There is a trade-off between the amount of and-parallelism exploited and the data-dependency analysis overhead incurred at runtime. Purely compile time techniques may lose significant portion of and-parallelism (due to the limited amount of information available at compile-time) but incur very little run-time overhead; on the other hand, purely run time techniques capture maximal and-parallelism at the expense of high overhead.

The solution adopted in the ACE system (and in its direct predecessor &-Prolog [38]) encapsulates the dependency information in the code generated by the compiler. In this way simple and inexpensive runtime checks can be done to identify sources of dependency. This technique was first devised by DeGroot and is called Restricted (or Fork/Join) And-Parallelism (RAP) [17], and was further enhanced by Hermenegildo [42, 43, 45]. It does not capture all the and-parallelism, although it does manage to exploit a substantial portion of it (see later). Moreover, research has shown that generation of these annotations through compile-time analysis [15, 24, 41, 7, 46] based on abstract interpretation [13] is extremely effective for a large variety of applications.

It is also interesting to observe that the amount of information that is collected during the data-dependency analysis can be used not only to exploit parallelism but also to optimize other phases of execution. In particular, knowledge of dependencies between subgoals allows the implementation of *intelligent backtracking*, a backtracking mechanism which uses dependencies information to select alternatives that are more likely to lead to a success [50, 45].

ACE is capable of exploiting independent and-parallelism automatically by means of a parallelizing compiler. Explicit parallelization of programs by the user is also supported through some language extensions, which allow expressing arbitrary and-parallelism (i.e., not only independent). The same language is used to make the result of the automatic parallelization visible to the user if so desired.

In this system, program clauses are compiled into *Conditional Graph Expressions (CGEs)* (originally developed by DeGroot [17] and successively refined by Hermenegildo [43] and Shen [70]). Conditional Graph Expressions are expressions of the form

$$(\texttt{condition} \;\Rightarrow\; \texttt{goal}_1 \;\&\; \texttt{goal}_2 \;\&\; \ldots \;\&\; \texttt{goal}_\texttt{n}),$$

meaning that, if *condition* is true, goals $goal_1 \ldots goal_n$ should be evaluated in parallel, otherwise they should be evaluated sequentially. No restrictions are imposed on the content of *condition*. In the work by DeGroot [17], the *condition* contains only tests of the form $ground(v_1, \ldots, v_n)$, which checks whether all of the variables $v_1, \ldots, v_n$ are bound to ground terms, and $independent(v_1, \ldots, v_n)$, which checks whether the set of variables reachable from each of $v_1 \ldots v_n$ are mutually exclusive of one another. The *condition* can also be the constant *true*, which means the goals can be executed in parallel unconditionally. These two kinds of tests have been shown to be sufficient to efficiently tackle dependence detection. Generalization of the conditional graph expressions have been proposed to deal with different forms of parallelism [71, 59] and with the problem of controlling granularity of the parallel computations [16].

The groundness and independence conditions are evaluated at runtime (a simple technique which keeps track of groundness and independence properties of variables through tags is presented in [17]). The method is conservative in that it may type a term as non-ground even when it is ground—one reason why this method is regarded as “restricted”. CGEs generated at compile-time through abstract interpretation manage to capture a substantial amount of and-parallelism [57].

4.2 The ACE Independent And-parallel Runtime System

The ACE IAP run-time system (also named &ACE [61]) is based on an extension of the Warren Abstract Machine (WAM) [78]. The actual implementation has been performed by extending the SICStus Prolog abstract machine.

The philosophy behind the design of the &ACE system is to achieve efficiency similar to a standard Prolog implementation for sequential execution while minimizing the overhead of running parallel code. The &ACE system runs on a number of processors (also known as *and-agents*) which collaborate in exploiting the and-parallelism detected and annotated by the parallelizing compiler.

Each agent is similar to a standard Prolog engine, extended with a few additional data structures which are used to support:

- execution of a CGE, i.e., activation of an and-parallel execution;
- scheduling activities, i.e., searching for parallel work in the system;
- backtracking, i.e., detecting and propagating failure conditions, and selecting unexplored alternatives.

Goals which are ready to be executed in parallel are pushed on to a *goal stack*[5]. The goals are then available to be executed by any agent (including the agent which pushed them).

A data-structure, called *Parcall frame*, is used for coordinating and synchronizing the parallel execution of the goals inside a parallel call, both during forward execution and during backtracking. *Markers* are used to delimit *stack sections* (corresponding to the execution of different *parallel* goals on different processors) and they implement the storage recovery mechanisms during backtracking of parallel goals in a manner similar to choice-points for sequential goals [63, 64, 44, 72].

The &ACE run-time system architecture comprises a collection of WAM data areas, a collection of agents, and a shared code area. The agents (Unix processes/threads) run programs from the code area on the stack sets. All agents are identical (there is no "master" agent). Also, agents are started and put to "sleep" as needed in order to not overload the system when no parallel work is available. For more details the reader is referred to [61, 63].

Example 4.2 *The example clause from the natural language processing program will be translated by the parallelizing compiler as follows:*

```
collect_int(PrevAILT,PrevUtter,CurILT,T_Utter,PartialList) :-
   (  int1(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P1) &
      int2(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P2) &
      int3(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P3) &
      int4(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P4) &
      int5(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P5) &
      int6(PrevAILT,PrevUtter,CurILT,T_Utter,_,_,P6)   ),
    combine_output([P1,P2,P3,P4,P5,P6],[],PartialList).
```

expressing the fact that the subgoals `int1` *through* `int6` *can be run in and-parallel (the compiler has already statically determined that the various variables* `PrevAILT`*,* `PrevUtter`*,* `CurILT`*, and* `T_Utter` *will be ground when the clause is entered). Figure 6 shows a possible pattern of execution for this clause.*

[5] The term *goal stack* is inherited from the previous implementation models; the structure used to maintain the local work produced by a worker is actually very similar to a queue [61].

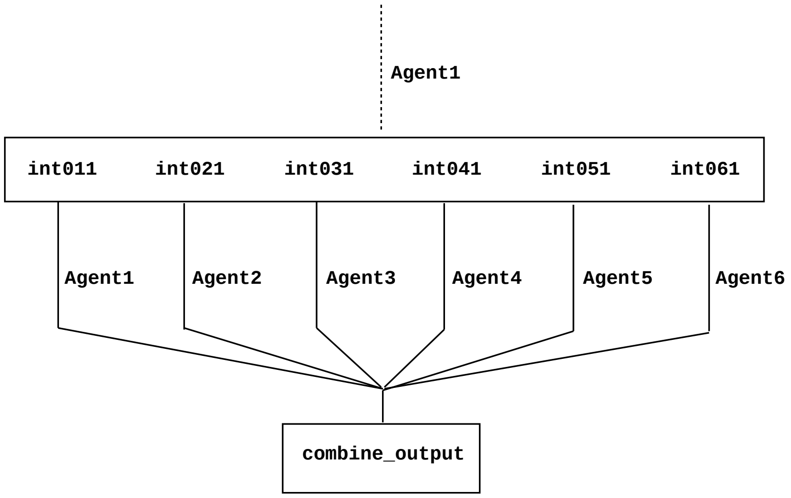

Figure 6: Example of execution in Independent And-Parallel

4.3 Issues in Implementing And-Parallelism

Experience in implementation of and-parallel logic programming systems has demonstrated that the main issue to achieve effective exploitation of parallelism and efficient executions is *memory management* and, more specifically, management of the *trail stack*. In simple words, the trail stack is a data structure used to store the addresses of the all the variables that have received conditional bindings during the execution. This structure is required in order to be able to remove these bindings during backtracking.

One of the most difficult problems to solve in the design of an and-parallel system is the management of backtracking. Backtracking in an and-parallel system involves traversing, in a fairly sequential manner, a computation that is spread across different stacks. This traversal is required in order to:

- recover memory;
- restart forward execution after a new point in the computation tree that contains an untried alternative has been located. In particular all the bindings generated in the part of the computation that has been discarded need to be removed—this is realized by scanning the appropriate part of the trail stack.

In a sequential implementation it is relatively easy to perform both activities during backtracking [78]. In an and-parallel system things become more complex, as the computation is arbitrarily spread on the stacks of the different processors participating in the execution. In particular, two approaches to backtracking can be taken:

1. *Private Backtracking*: each agent is allowed to backtrack only on the parts of the computation that lie in its own stacks. This is a straightforward generalization of the sequential model and allows on-line garbage collection with ease. On the other hand it requires synchronization between the different agents, in order to transfer control from one agent to the other during backtracking. Our experience has shown that this may lead to severe slow-downs on programs with fine-grained parallelism and intense backtracking activity. This approach has been used in the &-Prolog system and in a preliminary version of &ACE.

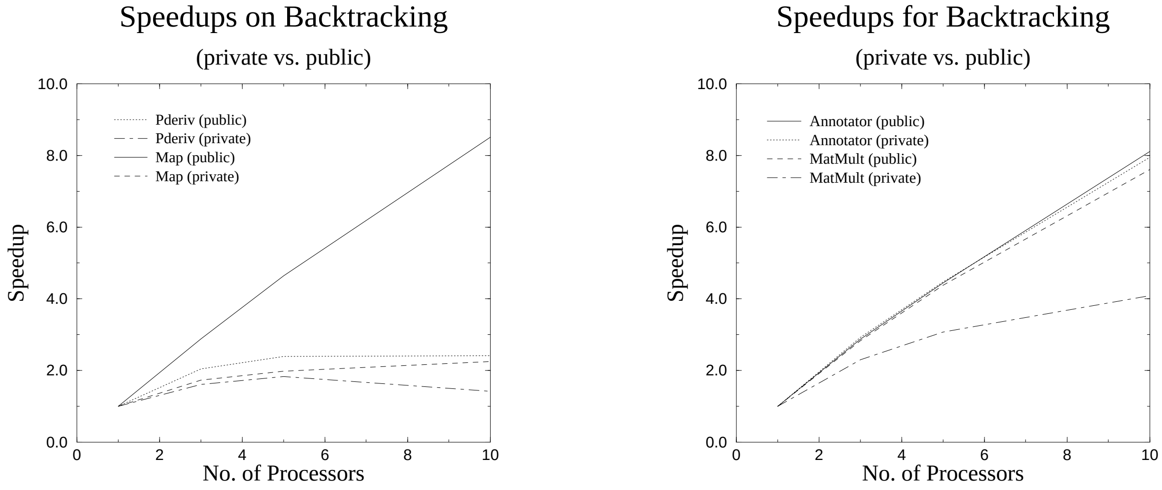

Figure 7: Comparison of speedups - Private vs. Public Backtracking

2. *Public Backtracking*: any agent can backtrack on the stack of any other agent as well as its own. This allows the system to keep the backtracking activity within one and-agent (no transfer of control is required), but this makes concurrent garbage collection impossible.

The current version of &ACE implements public backtracking, with off-line garbage collection. Experiments have shown that private backtracking is considerably slower than public backtracking because of the excessive interactions between different computing agents. Considering benchmarks involving heavy backtracking (e.g., programs using the generate&test approach to solve a problem), the overhead in private backtracking is very high (often close to 60-80%), and it tends to increase when the number of processors used in the computation is increased. Public backtracking maintains the overhead due to backtracking within 15%. Figure 7 summarizes the results achieved using the two forms of backtracking in four different benchmarks. As we can observe, public backtracking allows to maintain the speedups achieved from parallel execution of the program, while private backtracking tends to loose the advantages of parallel execution due to the intense overhead during backtracking. In the future, when ACE will be completely built using the thread Library, variable priority threads will be used for concurrent garbage collection.

The remaining open problem is finding a simple way of restarting forward execution after backtracking: the main issue is efficiently unwinding the trail before restarting execution. Effectively detecting the parts of trail that need to be unwound, especially when using public backtracking, can be complex and time-consuming. At present &ACE performs a *stack linearization* operation upon completion of and-parallel subgoals and associates trail stack segments to each choice point created.

4.4 Performance Results

&ACE is arguably the most complete implementation of an and-parallel Prolog system reported to date. It can support a much richer parallel backtracking behaviour than the other systems proposed in the literature (like &-Prolog/CIAO [38], to which &ACE owes a lot of its original inspiration). &ACE went through various revisions before reaching the current stage. Originally based on a direct implementation of the RAP-WAM [42] engine, it has been successively redesigned in order to incorporate better memory management, various optimization schemes, and exploitation of and-parallelism from non-deterministic programs.

Experimental results regarding the run-time system can be found in [61]. Actual speedups obtained on the Sequent Symmetry and the Sun Sparc Multiprocessors systems were reported for the parallelized programs for different numbers of processors. We have invested considerable effort in testing our system on real-life programs, i.e., existing large Prolog programs developed independently by other people. These programs have been originally written for sequential implementations of Prolog. Example of this is *NLP*, a Spanish-to-English discourse analyzer (mentioned in a previous section). As we can observe from table 2, the speedups achieved on these programs are extremely good. The benchmarks that have been run on &ACE range from simple toy programs to actual applications (see table 2), and are not just limited to the area of symbolic computing—see, for example, *compiler*, a full-blown Prolog compiler written in Prolog—over 2200 lines of code. This attests to the versatility of Prolog as a language and of the &ACE system as a parallel implementation. The figures clearly indicate that the current implementation, even though not completely optimized, is quite effective. On many benchmarks, containing a sufficient amount of parallelism, the system manages to obtain linear speedups. Our implementation incurs an average parallel overhead of about 10-15% over SICStus Prolog. Some of this parallel overhead can be avoided through proper run-time optimizations. Table 2 indicates, for each benchmark, the execution time (in ms.) and the speed-up obtained.

Goals		**ACE agents**					
Query	*Size*	1	3	5	10	15	18
hanoi(12)	30	3758	1251 (3.00)	775 (4.85)	390 (9.6)	288 (13.0)	230 (16.3)
compiler	2200	41052	14804 (2.77)	9726 (4.22)	5100 (8.05)	3302 (12.43)	2700 (15.20)
matrix_mult(30)	40	4970	1700 (2.93)	1009 (4.93)	519 (9.58)	350 (14.2)	305 (16.3)
NLP	5000	45830	15860 (2.89)	9799 (4.68)	6110 (7.5)	4700 (9.75)	4350 (10.54)

Table 2: Execution times in msec (Speedups are shown in parenthesis)

&ACE takes also advantage of a great variety of optimization schemes, which attempt to reduce the parallel overhead to the minimum [62] and to take advantage of frequent execution patterns (e.g., data-parallel programs [21]).

5 Dependent And-parallelism

As mentioned earlier, given two subgoals with data dependencies between them, they can be executed in parallel. However, this parallel execution has to be carried out under certain constraints since unrestricted parallel execution of two dependent subgoals can result in either unnecessary computation being performed, or in erroneous results. This form of parallelism is known as *dependent and-parallelism*.

The presence of data dependencies implies the existence of variables shared by the parallel subgoals, for which the subgoals are competing in the creation of bindings. Such variables are typically called *shared* (or *dependent*) variables.

The constraint that is usually laid down is that only one of the subgoals can bind each dependent variable (the *producer* subgoal), while the others can only read its value (the *consumer* subgoals)[6]. Thus,

[6]Alternative approaches to DAP, like those based on the previously mentioned *back-unification*, are not considered here, because of their inherent high overhead and the difficulty of supporting Prolog sequential semantics [58].

the dependent variable becomes a one way communication channel (or a *stream*). Hence, dependent and-parallelism is also known as *Stream parallelism.*

Example 5.1 *We use* Metallel *(by Eric Iverson), a Prolog application used to recognize cases of metaphors and/or metonymy from a set of English sentences, to illustrate dependent and-parallelism. The top-level clause of the program has the following structure:*

```
do0(Sentence,Best) :-
   dcg(Senses,Sentence,[]),
   verb_first(Senses,Senses1),
   multi(Senses1,[empty,empty,_],[_,AvlNew,LowVal]),
   find_best(AvlNew,Best).
```

The different subgoals present in the body of the clause for `do0` *are processing the input in a pipelined fashion—*`dcg` *is incrementally building the list* `Senses`*,* `verb_first` *is incrementally reading* `Senses` *and building* `Senses1`*, etc. This represents a good example where DAP can be exploited: overlapping generation of the next element of a structure with consumption of the previously created one.*

ACE attempts to mimic Prolog semantics as much as possible during dependent and-parallel execution. It *dynamically* keeps track of the *producer goal*—the leftmost executing goal for a variable is the producer. The producer is the only subgoal allowed to create bindings for the variable. All the other subgoals accessing the same object (*consumers*) will be only granted read access to the variable.

This approach to dealing with DAP is a relatively natural generalization of sequential semantics, and in various forms have been included in most of the proposals for DAP (e.g., the mode declarations of Parlog [10], the dynamic reordering of literals of the And/Or-model [11], the determinate-first behaviour in the Basic Andorra Model [12]). The dynamic management of producer/consumer has been studied in depth within the DDAS project [70] and refined and formalized within the ACE project [59].

5.1 Problems in Implementation of Dependent And-parallelism

The main problem in implementing dependent and-parallelism is controlling the parallel execution of the consumer goals. That is, given two subgoals and a dependent variable `X`, one a producer of `X` and another a consumer, the execution of both can be initiated in parallel. In general, no restrictions are imposed on the execution of the producer. However, one must make sure that the consumer goal computes only as far as it does not instantiate the dependent variable `X`. That is, consumer subgoals are granted read-only access to the dependent variable. Any attempt of a consumer to bind the dependent variable will lead to its suspension. The consumer will be waken up only after a non-variable binding has been produced for `X` by the producer goal. Alternatively, the producer may terminate without generating any binding for the dependent variable. In this case the producer "role" have to be reassigned to a different (and active) subgoal—as a consequence, a suspended consumer may get awakened due to this reassignment of the producer position.

Thus, the two main concerns in implementing dependent and-parallelism are:

1. determining which instance of a given dependent variable is a producer instance and which instances are consumer instances;
2. efficient mechanism for waking up suspended subgoals when the variables on which they were suspended are instantiated—or when they, in turn, become producers.

We have formally demonstrated [58] that the following results hold:

- no implementation scheme supporting the most general form of DAP can be implemented without incurring a non-constant time overhead;
- it is possible to maintain constant-time overhead as long as no aliasing between dependent variables are allowed.

These results have been used in the design of dependent and-parallel engine of ACE, as illustrated in the next section.

5.2 DAP in ACE

In ACE, the automatic parallelization mechanisms described for the case of IAP have been generalized to produce annotations for dependent and-parallelism. The purpose of these annotations is twofold:

1. identify the conjunctions of subgoals which are more likely to produce speedups if executed in parallel. This is mainly based on comparing granularity of the computations generated by the different subgoals;
2. identify the dependent variables present in the conjunctions (i.e., variables that are accessed by different "parallel" subgoals) and produce appropriate code for handling them.

Example 5.2 *The code below is an example of the annotations produced for the clause previously described.*

```
do0(Sentence,Best) :-
   '$mark_var'([Senses,Senses1,AvlNew]),
   (
     '$and_goal'([[Senses → A_Senses]],dcg(A_Senses,Sentence,[])) &
     '$and_goal'([[Senses → B_Senses],[Senses1 → A_Senses1]],
                           verb_first(B_Senses,A_Senses1)) &
     '$and_goal'([[Senses1 → B_Senses1],[AvlNew → A_AvlNew]],
                           multi(B_Senses1,[empty,empty,_],[_,A_AvlNew,LowVal])) &
     '$and_goal'([[AvlNew → B_AvlNew]],find_best(B_AvlNew,Best))
   )
```

The notation "&" is used to indicate parallel execution of subgoals. Shared variables are identified at compile time and specially marked, by the '\$mark_var' predicate. Each instance of a shared variable in the parallel conjunction is explicitly renamed (through the "→" annotation), in order to give to each subgoal a different "access path" to the shared object.

At the implementation level, DAP is realized by identifying each shared variable and its producer and consumer goals. Each subgoal that has access to a shared variable maintains an independent access path to the shared object. A *filter* is attached to each path to indicate if the subgoal accessing the variable through that path is a producer subgoal or a consumer goal. Thus, all shared variables are viewed by a subgoal through a filter. This idea has been concretized in the *Filtered Bindings Model* [59, 60].

Figure 8 presents an intuitive picture of this model. Each subgoal has a local path to access the shared object (in this case heap location allocated to hold the value of the shared variable) and the path contains a filter. In the figure the filter is linked to information stored in the subgoal descriptor—this common information will be used to verify whether the subgoal is a producer (i.e., it is the leftmost active subgoal in the parallel call) or a consumer.

```
?- (p(X) & q(X)). is compiled into
?- mark(X), ('$and_goal'([[X,X1]], p(X1))
            & '$and_goal'([[X,X2]], q(X2))).
```

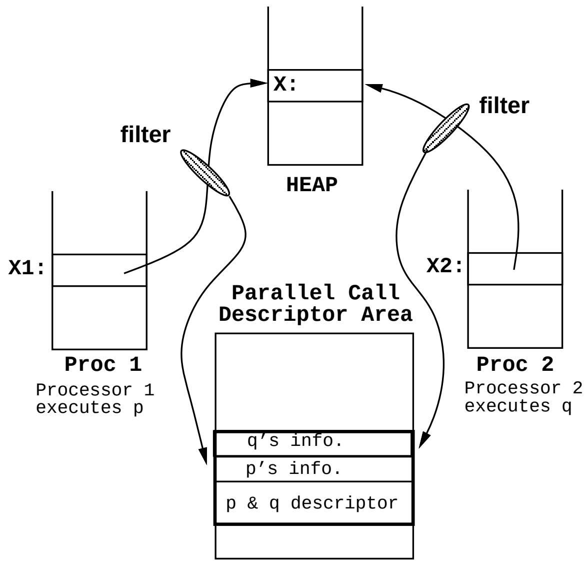

Figure 8: The Filtered Bindings Model

Every access to a shared variable by a subgoal will go through the filter corresponding to that subgoal, which will allow it to determine the "type" of the access (producer or consumer). By properly organizing the unification process, as long as there is guarantee that no aliasing between shared variables occurs (unless they are both producer accesses), it is possible to prove that, at any time a variable access will require traversal of at most one filter—which means constant-time validation of the access. The setup of a parallel call and the detection of continuation also do not require any non constant-time operation (cost is always bounded by the number of dependent variables detected by the compiler in that parallel call[7]). Details of the various possible cases that can occur during unification (e.g., binding of two shared variables, binding of a shared variable to a standard term, etc.) have been studied and are not reported here for lack of space (see [58]). An additional step is required when a subgoal terminates: if it is a producer goal and the variable is still unbound, then it should transfer the producer status to the next active subgoal in the parallel call by changing its filter. This is also a constant-time operation as the next goal to the right can be found by looking at the descriptor of the parallel call.

Thus, the Filtered Bindings model is a model that exploits a very general form of dependent and-parallelism and performs all the basic operations in constant-time. This is (to our knowledge) the first model ever proposed to meet these criteria.

[7]We are also working under the assumption that the compiler produces "mark" annotation for the program conservatively, i.e., during execution if a shared variable `X` is bound to a structure containing an unbound variable `Y` before the parallel conjunction corresponding to `X` is reached then both `X` and `Y` are marked as shared. Otherwise, for correctness, the structure `X` is bound to will have to be traversed to find all unbound variables occurring in it and mark them as shared.

The main restriction to exploitation of parallelism is that unbound shared variables are not allowed to be bound to each other, unless both accessed in producer position. If this restriction is relaxed then a non-constant overhead will be produced in the variable access operation. In the ACE implementation, in fact, this restriction is not imposed, since, in practice, the pathological case that leads to the non-constant overhead case seldom arises.

The filtered bindings model can be implemented in various ways. The current implementation, realized in the ACE system, implements filters as a word in the subgoal descriptor, and paths as a pair of words, one pointing to the actual variable and one pointing to the filter. Local paths related to shared variables introduced in the same parallel call share the same filter.

Consumer accesses suspend in presence of unbound variables. The current implementation supports both busy-waiting and goal suspension. During consumer access, if an unbound variable is reached, the processor busy-waits for a small fixed amount of time, and if no bindings have been produced it releases the current subgoal (to a queue of suspended computations) and reenters scheduling. Though quite naive, this approach has produced good results. For small granularity computations the short busy waiting allows the processor to receive the binding without paying the overhead of going through a sequence of suspension/awakening. For computations with a large number of goals and a considerable delay between starting of the consumer execution and completion of the corresponding producer, the short busy waiting does not seriously affect the efficiency of exploitation of parallelism.

Variable suspensions have been implemented using the traditional *suspension lists* [14]. Upon suspension, a state-record for the subgoal is created and linked to both the variable on which suspension occurred and the filter of the corresponding subgoal. When either of the two information changes status, the associated lists of state-records are awakened and moved to the processors run queues.

5.3 Performance Results

Table 3 illustrates the results obtained by running in dependent and-parallel some benchmarks using the ACE system.

Goals	**ACE agents**				
Query	1	2	4	6	8
Genetic	17779	9419 (1.89)	4810 (3.70)	3269 (5.44)	2540 (7.0)
TSP	23059	11519 (2.0)	5770 (4.0)	3850 (5.99)	3009 (7.66)

Table 3: Execution times in msec (Speedups are shown in parenthesis)

Genetic is a Genetic algorithm solution to a search problem, while *TSP* is a solution to the TSP problem developed using branch&bound techniques. As one can observe from these results, the system is capable of producing excellent speedups, at the price of a relatively low overhead (at most 20%).

The initial (and highly unoptimized) prototype has been compared with Andorra-I [12], Reform Prolog [4], and DASWAM [70], which are the most efficient and robust implementations of DAP available to date. The results [58] can be summarized as follows:

- Reform exploits a restricted form of parallelism (recursion-parallelism). On the ACE benchmarks where Reform is capable of extracting parallelism, the two systems show similar speed and speedups.

- Andorra-I is considerably slower than ACE, due to its high overhead. In most of the benchmarks ACE has also been able to exploit better speedups than Andorra-I.
- DASWAM is the only system exploiting the same general form of DAP. Speedups produced by the two systems are very close, but ACE is considerably faster and, in particular, capable of maintaining overhead within very reasonable levels (never more than 20%, and in most of the cases below 10%).

6 Combination of different forms of Parallelism

6.1 Introduction

As one can gather, parallel systems that exploit only one form of parallelism from logic programs have been efficiently implemented and have reached a mature stage (except, probably, for dependent and-parallelism). Programs with a single form of parallelism can be very efficiently executed in parallel within the ACE system. The next challenge is to be able to exploit multiple forms of parallelism from the same program simultaneously. Simulations have shown that indeed better speed-ups will be achieved if more than one source of parallelism are exploited [71] (i.e., practical programs, indeed, have multiple forms of parallelism).

The current research trend is split along two (though not independent) tracks:

1. **Combination of or- and and-parallelism:** Prolog is maintained as target language, and its full sequential semantics is implemented. The implementation of the Basic Andorra Model, namely, Andorra-I [12] is a good example in this class since it combines determinate dependent and-parallelism, implemented using techniques from JAM [14], with or-parallelism, implemented using Binding Arrays technique [52, 79]. Some other models have been designed and are currently under development, like the PBA model [29] which attempts to combine independent and-parallelism with or-parallelism. Older proposals (e.g. And/Or-model [11], ROPM [66], PEPSys [23], AO-WAM [27]), have either not been implemented, or their implementations are far too complex and inefficient.
2. **New languages for exploiting and-or parallelism:** In the other path that researchers have taken, the focus is more on the design of new languages and/or new computational models, which do not necessarily respect the sequential semantics of Prolog, and which offer a broader scope for the exploitation of parallelism. Some of them, like AKL [34], have been shown to subsume (under certain restrictions) many of the other execution models (e.g. dependent and-parallelism with coroutining, or-parallelism).

In the ACE project we aimed at developing a parallel implementation of Prolog. Thus, ACE falls in the first category.

The problems faced in implementing combined and- and or-parallel systems are essentially the sum of problems faced in implementing and-parallelism and or-parallelism individually. However, often in the combined system the problems faced in one may worsen those faced in the other, especially those regarding control of execution, representation of the environments, and memory management.

6.2 Combined And/Or-Parallelism in ACE

The main problem to be solved in any implementation of or-parallelism is the *Environment Representation* problem. As illustrated in section 3.1, two (or-parallel) threads originating from the same choice point may bind the same variable to different values. Thus, there is a need for having an environment representation mechanism that is capable of keeping the distinct bindings produced by or-parallel computations separate and associating the correct set of bindings with each or-parallel thread.

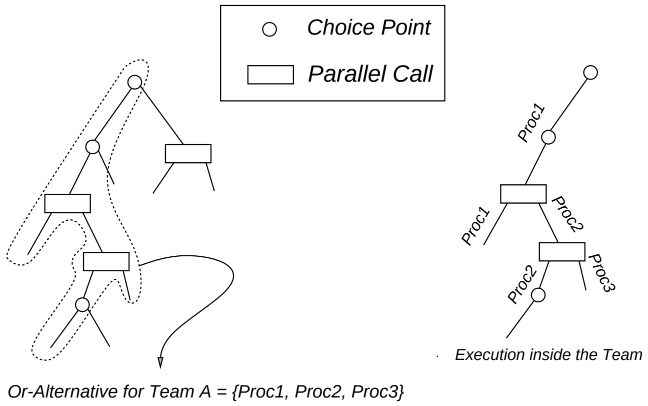

Figure 9: Layered Exploitation of Parallelism

The problem becomes more complicated once we consider concurrent exploitation of both or- and and-parallelism. This is not unexpected: in and-parallelism the different parallel computing agents are working on different parts of *the same* solution to the problem at hand—they need to be able to freely access each other's environment, and a binding produced by one of them needs to be visible to the other agents—while in or-parallelism the different computing agents are actually computing *different* solutions—the parallel computations need to be kept as much separate as possible. Thus, the requirements of and- and or-parallelism, in terms of environment representation, are antithetical.

The solution employed in the ACE system (first proposed in the Andorra-I system, and also shared by other approaches, like SPBA [29]) is to create a *layering* in the exploitation of the different forms of parallelism. Computing agents are organized into *Teams* (whose structure can dynamically change in order to respond to variations in the distribution of work). Each team acts as a unique entity in the exploitation of or-parallelism (i.e., a separate or-alternative is assigned to each team), while the agents inside the team will take care of exploiting the and-parallelism offered by the or-alternative assigned to their team. In this way the exploitation of and-parallelism is realized at a lower layer, nested inside the exploitation of or-parallelism. Figure 9 illustrates this principle. This principle has been used in the model presented in the next section.

6.3 Generalized Stack-Copying

As described before, in stack-copying, whenever a computing agent $\mathcal{A}$ steals one alternative from a choice point (created by $\mathcal{B}$) to explore it independently, it creates a complete copy of the environments allocated by $\mathcal{B}$. In this way every variable that can be in danger of receiving conflicting bindings from different or-parallel threads, is copied and allocated a different memory location in every or-parallel thread. Note that to implement stack copying the address space has to be mapped in a special way so that pointers in the stacks being copied can be relocated without changing their meaning [1, 30].

The generalization of the stack copying mechanism to the case of and- and or-parallelism presents some new problems. In particular, the presence of and-parallelism makes the process of identifying the parts of computation that need to be copied extremely complex, since the computation is not anymore sequentially stored in the stack of a single agent (as in the pure or-parallel case), but is distributed across different agents (the agents in the team). Furthermore, and-parallelism leads to the loss of the correspondence between physical position of the computations on the stacks and logical sequence of execution [42]. A solution to these problems has been proposed in the ACE system [30].

Each processor in the ACE system is basically an and-parallel engine, capable of carrying on its own computation and interacting with a certain number of other processors (those belonging to the same team). ACE [30] is obtained from &ACE by adding: (a) a mechanism to keep track of the amount of or-parallel work produced by the computation of a team; (b) a mechanism to allow interaction of one team with another in order to guarantee synchronization and execution of *sharing operations*; (c) extended backtracking, allowing calls to the or-scheduler whenever a team backtracks over a shared choice point.

Or-parallel work is obtained by one team from another through the operation of *sharing*. A sharing operation involves copying (part of) the computation tree generated by a team to another team. The operation is not straightforward since the computation tree is spread over the address spaces of the different processors belonging to the team. During a copying operation, the processors of the two teams involved need to synchronize and to communicate in order to allow identification of the parts of computation to be copied.

Furthermore, we want to perform copying incrementally, i.e., transfer only what is strictly necessary, and this selective operation is complicated by the arbitrary ordering of the different sections of computation on the stacks of the various team members. To make the whole process more effective, we have introduced in ACE some *sharing conditions* that a choice point should satisfy in order to allow its sharing with other teams (i.e., giving away its untried alternatives for or-parallel processing to other teams). Intuitively, a choice point p satisfies the sharing conditions if

- p is a parallel node (i.e., it offers viable alternatives for parallel execution);
- the and-parallel branches to the left of p have been completed (i.e., they are not going to be subject of further extensions).

In ACE, during a sharing operation, only the choice points satisfying the sharing condition are actually taken into consideration. This simplifies both the copying operation (it is easier to detect which parts of the computation need to be transferred), the scheduling activity (we are guaranteed that everything on the left is terminated and successful and we do not have arbitrary intermixing of shared and private parts in the computation tree), and the management of side-effects and extra-logical predicates. Figure 10 illustrates an example of how copying is performed: we have two active teams, $\mathcal{T}_A$ and $\mathcal{T}_B$, and $\mathcal{T}_B$ steals work from the choice point A in the computation created by $\mathcal{T}_A$ (figure 10(i)). Choice point A instead of choice point B is selected as source of work because B does not satisfy the sharing conditions (since the branch denote by (*) is still active). The activity of taking work consists of copying the *difference* in the computation tree between the two teams (i.e., whatever $\mathcal{T}_A$ has done between the choice point in which team $\mathcal{T}_B$ is presently positioned—choice point C—and the choice point A from which work is taken) and is illustrated in figure 10(ii).

Different approaches to incremental copying have been studied and heuristics to choose the most appropriate in each situation have been developed. The interested reader is referred to [30] for a detailed discussion of this topic.

At the moment ACE supports or-parallelism, independent and-parallelism, and dependent and-parallelism individually. Work is in progress to allow for mixed exploitation of these three forms of parallelism. Even though this has not been completely implemented yet, the system is already showing improvements in execution times when or-parallelism is combined with and-parallelism, compared to solely and-parallel execution. On a *generate & test* benchmark (the generation is non-deterministic and

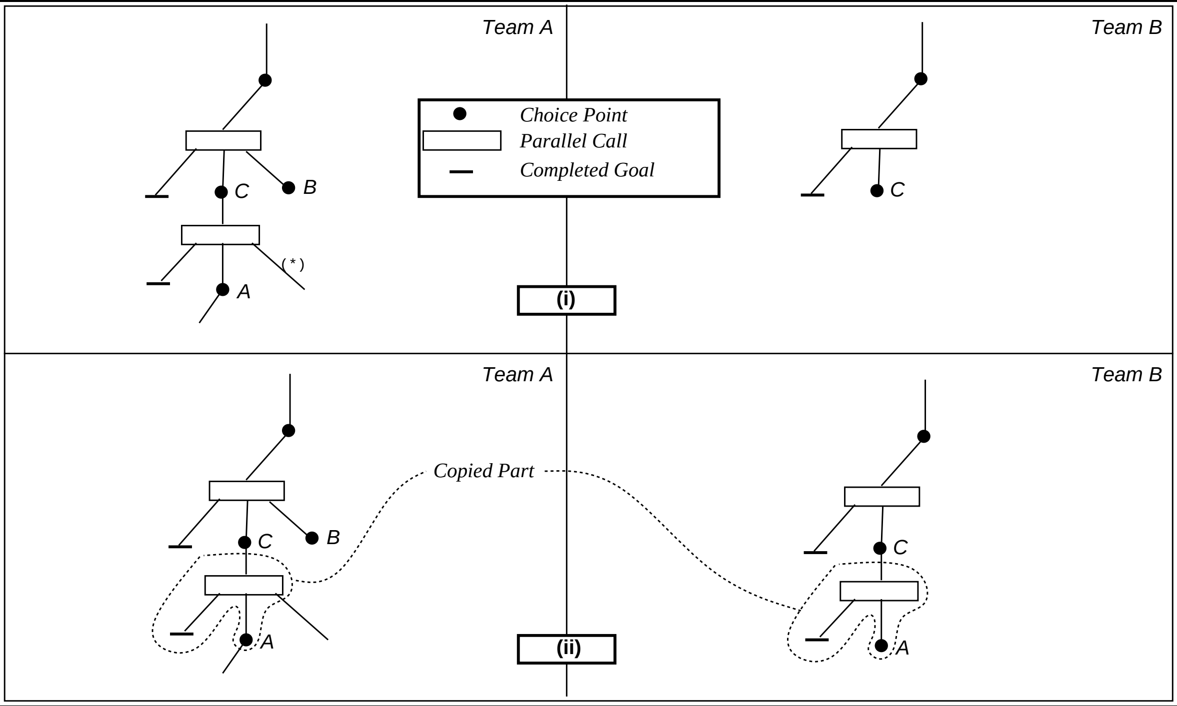

Figure 10: Copying Process

the test contains independent and-parallelism) the prototype showed an improvement in execution time from 1211ms for the and-parallel execution to 917ms for the and/or-parallel one.

6.4 Performance

Current preliminary results for and/or parallel executions are reported in Table 4.

Query Executed	*Sequential*	*And-Parallel*			*Or-Parallel*		*And/Or-Parallel*	
		2	3	5	2	3	2×2	3×2
Naive Queens	259	249	230	250	139	129	109	99
Generate&Test	6139	4092	3330	3070	3120	3120	1679	-
Crypto	7840	5649	3389	2739	4330	840	3120	610
Planner	3299	3299	3299	3299	1670	1230	1670	1230

Table 4: And/Or-parallel Executions (time in ms.)

The numbers indicate the execution times obtained varying the number of agents used. In the And/Or-parallel case, the notation $m \times n$ indicates that m teams of n and-agents each were used[8].

For the *Queens* and the *Planner* problems, execution times are relative to an all-solutions execution. In the other programs the times are relative to a single-solution execution—these programs are deterministic and a "cavalier" annotation has been used to complete execution as soon as the solution is detected (and this explains the presence of superlinear speedups).

[8] At present ACE adopts a static team organization. This will be changed in the near future.

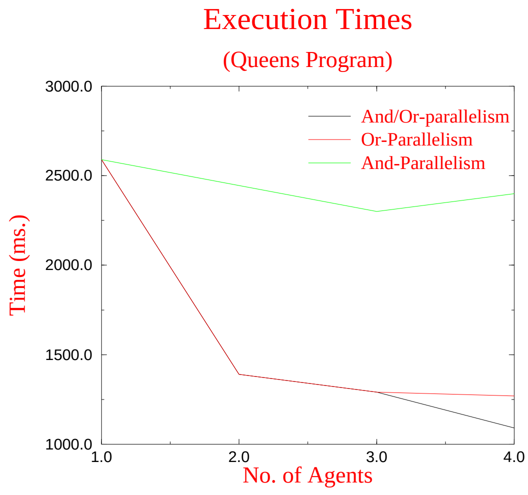

Figure 11: Queens Program

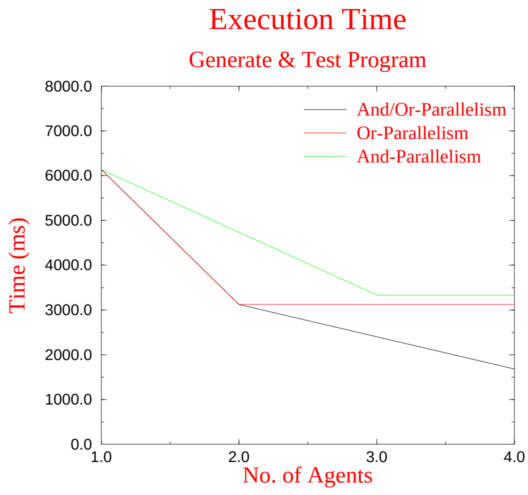

Figure 12: Generate&Test Program

The *Planner* system does not offer any and-parallelism. The results have been reported to point out how the exploitation of both and- and or-parallelism does not produce slow-downs with respect to exploitation of a single form of parallelism.

6.5 Improving And/Or-Parallelism

The development of the or-parallel component of ACE, based on stack-copying, has raised many questions regarding how suitable are the standard implementation techniques developed for pure or-parallel systems (e.g. stack-copying, binding arrays, version vectors) to support the presence of and-parallelism.

Stack-copying and Binding Arrays (as well as Version Vectors [3]) represent two extremes in the management of or-parallelism. In stack-copying the amount of information shared between the or-agents is minimized, by creating separate copies the "common" computation. On the other hand in Binding Arrays the amount of shared information is maximized, keeping only the binding areas separate. Both the approaches have their advantages and disadvantages when used in presence of and-parallelism. Stack copying becomes quite involved, due to the lack of correspondence between logical structure of the computation and its physical distribution. Binding Arrays are complex to share between different and-agents.

It is an open topic of research, currently being explored within the ACE project itself, to investigate the characteristics of computational models laying in between these two extremes, and the eventual relationships between these different models and the characteristics of the target architecture.

7 Support for Full Prolog

To make logic programming practical many extra-logical, metalogical, and side-effect predicates have been incorporated in Prolog, the most commonly used logic programming language. Some researchers have taken the view that a parallel logic programming system should transparently execute Prolog programs in parallel [52, 35]. That is, the same effect should be seen by a user during parallel execution of a Prolog programs, as far as side-effects etc. are concerned (including printing of the final solutions), as in its sequential execution with left-to-right, top-to-bottom selection rule. Such a system is said to support *sequential Prolog semantics*. The advantage of such an approach is that existing Prolog programs can be taken and executed in parallel without any modifications.

Essentially, an order sensitive predicates should be executed only after all the "preceding" ones have been completed (preceding is intended with respect to their sequential order of execution). If the preceding order sensitive predicates have not been executed yet, the current one should be suspended, and resumed at the right moment. However, given an order sensitive predicate, determining which are the preceding order sensitive calls is an undecidable property; therefore the knowledge that the preceding sensitive predicates have been completed can only be approximated.

In an or-parallel system [36] the preceding order sensitive predicates are assumed to be finished if the or-branches containing them have finished execution. In other words, an order sensitive predicate is executed only when the branch containing it becomes the leftmost in the or-parallel tree. Likewise, in an and-parallel system, such as &-Prolog and &ACE, an order-dependent predicate occurring in an and-parallel goal p_i is going to be executed only once all the branches of the and-tree to the left of p_i have finished execution [56, 18, 9].

Our belief is that and-or parallel systems should also support full Prolog semantics. Thus, our aim is to support the full Prolog language in ACE [28]. Thus, in the full ACE system the sequential semantics of Prolog is guaranteed to be preserved even though the program is being executed in parallel. Thus, a user can develop and debug his/her program on a sequential machine and then migrate it to the ACE system to obtain better performance by exploiting and-or parallelism.

The conditions for execution of order-sensitive predicates in an and/or-parallel system can be obtained as combinations of the conditions described before for the single forms of parallelism. This boils down to verifying that the order sensitive predicate is

- leftmost in the and-tree of the computation;
- all the branches of the and-tree to the left of the order-sensitive predicate represent solutions to the leftmost active alternatives of all the choice points in the or-tree;

General techniques for handling order-sensitive executions in presence of and/or-parallelism have been studied [28]. On the other hand, the particular structure of the execution in ACE (and, in particular, the presence of the sharing conditions) allows to simplify this management [30] (especially regarding the first of the two tests mentioned above), making execution of programs rich of order-sensitive operations arguably faster than in other and/or-parallel models.

8 Conclusions

In this article we described the different sources of implicit parallelism—or-parallelism, independent and-parallelism, and dependent and-parallelism—found in logic programming languages, and the many problems encountered in exploiting them. We presented the ACE and-or parallel system that exploits all three forms in a single framework. Parallel logic programming is an exciting area of research and will continue to be so, until the dream of *efficiently* exploiting all sources of parallelism present in logic programs in the most cost effective way is realized. The ACE system is a step towards that dream. The current state-of-the-art is that there are efficiently engineered systems that exploit only a single form of parallelism, e.g., MUSE, Aurora (or-parallelism), &ACE [61] and &-Prolog (independent and-parallelism), and many efficient implementations of committed choice languages [69, 22]. However, there are none that exploit *all* sources of parallelism present in logic programs efficiently (with the exception of Andorra-I system that exploits two of them). ACE is the first system that exploits all three forms of parallelism in single framework. The ACE system supports full Prolog semantics. Thus, Prolog programs developed for sequential machines can be migrated to the ACE system without any change. The efficiency of the ACE system in exploiting parallelism was illustrated with the help of several real-life benchmarks. On most of the benchmarks, ACE is able to achieve near perfect speedups.

The main contributions of the ACE project, namely, the development of an efficient and complete and/or-parallel system, were described in this paper along with a thorough analysis of the practicality of the ACE system—i.e. its ability to execute large and complex applications (AI applications in this specific case) with ease, and extract significant amounts of parallelism with very low overhead.

Acknowledgements

Ongoing work is supported by NSF grants CCR 92-11732, HRD 93-53271, and INT 95-15256, by NATO Grant CRG 921318, and a fellowship to Enrico Pontelli by Phillips Petroleum. The research presented has benefited from discussions with K. Ali, M. Carro, the CLIP group at UPM, M. Hermenegildo, R. Karlsson, V. Santos Costa, and K. Shen. We wish to thank M. Gelfond, E. Iverson, A. Provetti, and J. Wiebe for allowing us to work with the programs developed by their research teams. We would also like to thank the anonymous referees for their comments and suggestions.

References

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

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

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

[4] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the Language and its Implementation. In *Proc. of the 10th Int'l Conference on Logic Programming.* MIT Press, 1993.

[5] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley Ltd., Wokingham, England, 1986.

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

[7] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *International Static Analysis Symposium*, LNCS. Springer Verlag, 1994.

[8] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

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

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

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

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

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

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

[15] M. Garcia de la Banda. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming.* PhD thesis, Universidad Politecnica de Madrid, 1994.

[16] S. K. Debray, N.-W. Lin, and M. Hermenegilo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation.* ACM Press, June 1990.

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

[18] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

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

[20] ECRC. *ECLiPSe User Manual*, 1993.

[21] E.Pontelli and G. Gupta. Data And-parallel Logic Programming in &ACE. In *Proc. of 7th IEEE Symposium on Parallel and Distributed Processing*, 1995.

[22] K. Hirata et al. Parallel and Distributed Implementation of Logic Programming Language KL1. In *Int.'l Conference on Fifth Generation Computer Systems*, 1992.

[23] Uri Baron et al. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *FGCS '88*, pages 841–850, 1988.

[24] M. Garcia de la Banda F. Bueno and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-based Automatic Program Parallelization. In M. Bruynooghe, editor, *International Logic Programming Symposium.* MIT Press, 1994.

[25] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

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

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

[28] G. Gupta and V. Santos Costa. Cuts and Side-effects in And/Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, 1996.

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

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

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

[32] G. Gupta and E. Pontelli. Last Alternative Optimization and Parallel Implementation of CLP(FD). Technical Report NMSU-CSTR-9606, New Mexico State University, 1995.

[33] G. Gupta and E. Pontelli. Last Alternative Optimization for Or-parallel Logic Programming Systems. In *Eight International Symposium on Parallel and Distributed Processing.* IEEE Computer Society, 1996.

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

[35] B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and Side-Effects in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems.* Tokyo, November 1988.

[36] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[37] M. Hermenegildo. CIAO, a Generic Parallel Concurrent Constraint Logic Programming System. In *Principle and Practice of Constraint Programming.* Springer Verlag, 1994.

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

[39] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

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

[41] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1), 1995.

[42] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, U. of Texas at Austin, August 1986.

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

[44] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, May 1987.

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

[46] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[47] P. Kacsuk. A Highly Parallel Prolog Interpreter based on the Generalized Data Flow Model. In *Proc. of the 2nd International Logic Programming Conference*, Uppsala, July 1984.

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

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

[50] Y.-J. Lin. *A Parallel Implementation of Logic Programs*. PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[51] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

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

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

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

[55] T. Moto-Oka, editor. *Fifth Generation Computer Systems*. North Holland, 1982.

[56] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming*. MIT Press, June 1989.

[57] 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.

[58] E. Pontelli and G. Gupta. Dependent And Parallelism in Logic Programming. Technical report, Laboratory for Logic, DB, and Advanced Programming, 1995. Internal Report.

[59] E. Pontelli and G. Gupta. Non-determinate Dependent And-Parallelism Revisited. Technical Report NMSU-CSTR-9601, Dept. of Computer Science, New Mexico State University, 1996.

[60] E. Pontelli and G. Gupta. Non-determinated Dependent And-Parallelism Revisited. In *Proc. of the Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996. (extended abstract).

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

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

[63] E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Efficient Implementation of Independent And Parallelism. *Computer Languages*, 1996. To appear.

[64] E. Pontelli, D. Tang, G. Gupta, and M. Carro. Last Parallel Call Optimization and Fast Backtracking in And-parallel Logic Programming Systems. In *International Logic Programming Symposium*. MIT Press, 1994. (Extended Abstract).

[65] A. Provetti. *Towards the Formalization of Causal Aspects of Legal Reasoning*. PhD thesis, University of Bologna, Italy, 1996.

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

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

[68] V. Saraswat and P. Van Hentenryck. *Principle and Practice of Constraint Programming*. MIT Press, 1995.

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

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

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

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

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

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

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

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

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

[78] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

[79] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *Proc. IEEE Int'l Symp. on Logic Prog.*, pages 92–102, 1987.

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

[81] M. Wolfe. *High Performance Compiler for Parallel Computing*. Addison Wesley, 1996.