Incremental Exploitation of Parallelism in Prolog

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

Abstract: In this paper we discuss the problems encountered in exploiting all forms of parallelism—or-parallelism, independent and-parallelism, and dependent and parallelism—from Prolog programs. We also present our solutions to these problems. We describe a parallel Prolog engine that we have developed, called &ACE, that exploits independent and-parallelism. We discuss how or-parallelism and dependent and-parallelism can be incrementally incorporated in &ACE. We also present an efficient technique for exploiting data-parallelism in Prolog. We present performance figures for our parallel implementation done on a Sequent Symmetry and Sun Sparc multiprocessors.

Keywords: Prolog, And-parallelism, Or-parallelism, Optimizations

1 Introduction

Parallel processing of numerical problems has been an active area of research for almost two decades now [15, 22]. There are parallelizing compilers and other tools available for parallel numerical processing that programmers can use without being experts on parallel processing. However, the same is not true of parallel processing of Symbolic and Artificial Intelligence (AI) applications. There has been some work [14, 20], but there are still no systems that are general enough and that programmers can use without being experts on parallel processing.

In this paper we deal with systems that represent a step towards the goal of building a general automatic tool for parallelizing symbolic and AI applications. Our approach is based on the *logic programming* paradigm and the *Prolog* technology. A programmer wishing to parallelize his/her symbolic or AI program will code it in Prolog on a uniprocessor. Our compiler and runtime system will then run this Prolog program automatically in parallel. Our parallelizing system supports the full Prolog language, complete with all its extra-logical (the cut), metalogical (var, assert, retract, etc.), and side-effect (read, write, etc.) predicates that are sensitive to the order of execution of the Prolog program.

Logic programming is a paradigm of programming where programs are expressed as logical implications. Logic programming languages are suited for a wide range of applications, from compilers to databases to symbolic applications, as well as for general purpose programming. The most popular logic programming language is Prolog. The key property of logic programming languages, with respect to the issue of parallelization, is that they are *single assignment languages*. Unlike conventional programming languages they disallow destructive assignment and explicit control information. Not only does this allow cleaner (declarative) semantics for programs, and hence a better understanding of them by their users, it also makes it easier for 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 (declarative) meaning of the program. In particular, these operations can be performed by the evaluator *in parallel*.

An important characteristic of logic programming languages, thus, is that parallelism can be exploited in an *implicit* way. This can be done directly by the program evaluator (the runtime system) as suggested above, or, alternatively, it can also be done by a parallelizing compiler. The task of the parallelizing compiler is essentially to unburden the evaluator from making run-time decisions regarding which parts of the program to run in parallel. Note that the program can also be parallelized by the user (through suitable annotations). In all cases, the advantage offered by logic programming is that the process is easier because of the more declarative and high level nature of the language. Furthermore, the exploitation of parallelism at run-time or parallelization at compile-time can be done quite successfully in an automatic way and without requiring any input from the user. Clearly, implicit exploitation of parallelism can in many cases have significant advantages over explicit parallelization[1]. In that sense, one can look in some ways towards Prolog for solving a new form of "(parallel) software crisis" that is posed to arise with the new wider availability of multiprocessors[2]—given systems such as the one described in this paper, one can run Prolog programs written for sequential machines in parallel with no or minimal effort. For the rest of the paper we assume that the reader is familiar with Prolog and its execution model.

It must be pointed out that while the target application areas are Symbolic and AI applications, in fact, our system can also be used for parallel execution of general purpose programs. This stems from the fact that Prolog is an excellent language for writing problem solving (non-numerical) programs. This is borne out from some of the benchmarks we have used in the Performance Section (Section 2.2) of this paper.

Three principal kinds of (implicitly exploitable) control parallelism can be identified in logic programs:

1. *Or-parallelism* arises when more than one clause defines some predicate and a literal unifies with more than one clause head—the corresponding bodies can then be executed in parallel with each other, giving rise to or-parallelism. Or-parallelism is thus a way of efficiently searching for solutions to the query, by exploring alternative solutions in parallel. The computing agents[3] concurrently working along different or-alternatives are named *or-agents*.

2. *Independent and-parallelism* 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, i.e., their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables. Parallel execution of such goals gives rise to (independent) and-parallelism. The computing agents in charge of executing in parallel different subgoals are named *and-agents*.

3. *Dependent and-parallelism* arises when two or more goals in the body of a clause have a common variable and are executed in parallel.

Another distinction that is often made is between *Control* (or *MIMD*) parallelism and *Data* (or *SPMD*) parallelism. The first one extract parallelism by parallelizing the control structure of the execution, while the second one parallelizes operations on aggregate data structures.

The aim of our research is to (profitably) exploit all these forms of parallelism present in Prolog.

Various models have been proposed in the literature to exploit the different forms of parallelism (although only few efficient implementations have been actually developed). Nevertheless only very few researchers have tackled the more complex problem of integrating the different forms of parallelism in a unique framework. Some recent proposals (e.g., [10]) have attempted to reach this goal, but the results are still not well-established and the approaches are based on the development of a *new* language, with a less intuitive operational semantics than

[1] This does not mean, of course, that a knowledgeable user should be prevented from parallelizing programs manually or even programming sequentially but in a particular way that makes it possible for the system to uncover more parallelism.

[2] For example, affordable (shared memory) multiprocessor workstations are already being marketed by venders such as Sun (Sun Sparc 10), SGI (Challenge), etc.

[3] A computing agent is basically a Prolog engine extended with mechanisms required for inter-agent synchronization and scheduling.

that of Prolog. Our intent is to develop an implementation model in which various forms of parallelism are *concurrently* and *efficiently* exploited, and which supports the full Prolog language. That is, the system should allow the programmer to maintain the same view of the operational semantics of the language as during sequential execution, thus exploiting parallelism in a completely transparent way.

The novelty of our approach is its *modularity* and *incrementality*: starting from a basic (and very efficient) independent and-parallel engine (named &ACE) we successively add simple mechanisms capable of supporting more complex forms of parallelism. This approach allows to adapt the mechanisms developed to almost any existing parallel logic programming system, easily extending their ability of dealing with all the major forms of parallelism

2 Independent And-parallelism: &ACE

In this section we briefly describe independent and parallelism and how it is implemented. Foundations of much of the work described in this section were laid down in [4], however, the specific implementation described is that of the &ACE system [8].

2.1 Independent And-parallelism

Conventionally, an and-parallel Prolog system works by executing a program that has been annotated with parallel conjunctions. These parallel conjunction annotations are either inserted by a parallelizing compiler [11] or hand-coded by the programmer. Execution of all goals in a parallel conjunction is started in parallel when control reaches that parallel conjunction. Parallel conjunctions may also be conditional, which means that the goals in the conjunction are executed in parallel only if the condition, i.e., the expression upon which the conjunction is conditioned, evaluates to true (e.g., Conditional Graph Expressions [4, 11]).

The typical form of an and-parallel annotation is

$$\cdots(\langle conditions\rangle \Rightarrow G_1 \& \ldots \& G_n)\cdots$$

where $\langle conditions\rangle$ identifies the eventual tests required to allow the parallel execution of the subgoals $G_1, \ldots, G_n$ (e.g., the *conditions* may be simple run-time tests to guarantee the independency of the successive subgoals).

Whenever a parallel conjunction is encountered during execution, a data structure, the *parcall frame*, describing the parallel conjunction is allocated on the (control) stack. It contains various bookkeeping information (like number of subgoals in the conjunction), together with a descriptor, called *slot*, for each subgoal in the conjunction. At the same time appropriate data structures (e.g. work queue, termed goal stack in the logic programming literature) are initialized to allow remote execution of the newly generated subgoals (see Figure 1).

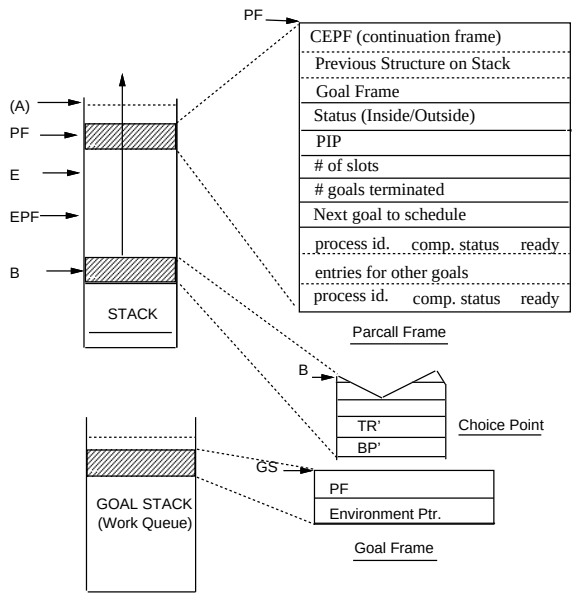

Figure 1: Datastructures needed for and-parallelism

A processor taking a subgoal from a parallel conjunction will initially allocate an *input marker* on its stack, to identify the beginning of the section of computation dedicated to this subgoal, and start the corresponding computation. At the completion of the subgoal another marker, the *end marker*, will be allocated to close the section[4].

Backtracking becomes complicated in and-parallel systems because more than one goal may be executing in

[4] Pointers to the input and end markers are stored in the slot of the corresponding subgoal in the parcall frame for backtracking purposes.

parallel, one or more of which may encounter failure and backtrack at the same time. Unlike a sequential system, there is no unique backtracking point, and the distributed nature of the execution may require a considerable synchronization activity between the different computing agents. In an and-parallel system we must ensure that the backtracking semantics is such that all solutions are reported. One such backtracking semantics has been proposed by Hermenegildo and Nasr [11]: consider the subgoals shown below, where ',' is used between sequential subgoals (because of data-dependencies) and '&' for parallel subgoals (no data-dependencies): `a, b, (c & d & e), g, h`. Assuming that all subgoals can unify with more than one rule, there are several possible cases depending upon which subgoal fails. If subgoal `a` or `b` fails, sequential backtracking ("Prolog-like") occurs, as usual. Since `c`, `d`, and `e` are mutually independent, if either one of them fails before finding any solution, then a limited form of intelligent backtracking can be applied and backtracking can be forced to continue directly from `b`—but see further below. If `g` fails, backtracking must proceed to the right-most choice point within the parallel subgoals `c & d & e`, and recompute all goals to the right of this choice point (following Prolog semantics). If `e` were the rightmost choice point and `e` should subsequently fail, backtracking would proceed to `d`, and, if necessary, to `c`. Thus, backtracking within a set of and-parallel subgoals occurs only if initiated by a failure from outside these goals, i.e., "from the right" (also known as *outside backtracking*). If initiated from within, backtracking proceeds outside all these goals, i.e., "to the left" (also known as *inside backtracking*). As mentioned before, this latter behavior is a form of "intelligent" backtracking, which takes advantage of the knowledge about independence of subgoals. When backtracking is initiated from outside, once a choicepoint is found in a subgoal p, an untried alternative is picked from it and then all the subgoals to the right of p in the parallel conjunction are restarted (in parallel).

2.2 &ACE

Independent and-parallelism with the backtracking semantics described above has been implemented quite efficiently by the authors in the &ACE system [8]. &ACE implementation itself is inspired by the RAPWAM [12][5]. &ACE is an extension to the sequential WAM (Warren Abstract Machine [21]) for and-parallel execution of Prolog programs with and-parallel annotation (such as CGEs [4]).

The current implementation of &ACE is running on both Sequent Symmetry and Sun Sparc multiprocessors. The results obtained with this system, even though the system is still in its early stages of development, are quite impressive. The tests that have been run on &ACE range from simple toy programs to actual applications. The results for the following benchmarks are initially reported: Matrix Multiplication, Quicksort, Takeuchi, Tower of Hanoi, Boyer (a reduced version of the Boyer-Moore theorem prover), Compiler (the Aquarius Prolog compiler from UC Berkeley that is approximately 2,200 lines of Prolog code), Poccur (a list processing program), BT_cluster (a clustering program from British Telecom, UK), Annotator (the annotator part of the &-Prolog parallelizing compiler that is about 1,000 lines) and, Simulator (a simulator for simulating parallel Prolog execution that is about 1,100 lines). Note that our suite of benchmarks does not just contain problems from the area of symbolic computing or AI. Rather, it also contains programs that are very "general" in nature, such as a compiler. This attests to the versatility of Prolog as a language and of the &ACE system that it is able to run in parallel a wide variety of programs.

Table 1 indicates, for each benchmark, the execution time (in ms.) and the speed-up obtained. Some of the speedup curves are plotted in Figures 2(i) and 2(ii). 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 (like matrix multiplication and hanoi). With more processors in the multiprocessor systems we believe we should be able to obtain higher speedups, provided the program contains that much parallel work.

The largest benchmark is the PLM Prolog Compiler (approximately 2,200 lines of Prolog code). Note that for the *compiler*, *quicksort*, and *boyer* benchmarks, the speedup curve flattens out because at some point all available parallelism is exhausted.(e.g., in the case of

[5] Other implementations based on the principles of RAPWAM have also been proposed in the past, like &-Prolog [11] and DDAS [19].

Goals executed	&ACE agents 1	3	5	10
matrix_mult(30)	5214	1768 (2.95)	1059 (4.92)	534 (9.76)
quick_sort(10)	1536	632 (2.43)	455 (3.38)	373 (4.12)
takeuchi(14)	1811	586 (3.09)	368 (4.92)	200 (9.06)
hanoi(11)	1671	550 (3.04)	336 (4.97)	180 (9.28)
pderiv	5375	1840 (2.92)	1100 (4.89)	550 (9.77)
boyer(0)	9290	3829 (2.43)	3199 (2.90)	2687 (3.46)
compiler	29902	12522 (2.39)	6437 (4.65)	4801 (6.23)
poccur(5)	3197	1079 (2.96)	662 (4.83)	371 (8.62)
bt_cluster	1343	480 (2.8)	312 (4.30)	189 (7.11)
annotator(5)	1422	475 (2.99)	322 (4.42)	187 (7.60)
simulator	3295	1232 (2.67)	827 (3.98)	536 (6.15)

Table 1: Unoptimized/Optimized Execution times in msec (Speedups are shown in parenthesis)

compiler, the speed-up was measured while the compiler was compiling a program of 8 clauses, thus the maximum speedup that can be obtained will be 8; if a larger program was compiled higher speedups will be obtained). Our implementation incurs an average parallel overhead of about 10-15% over SICStus Prolog. Some of this parallel overhead is avoided by triggering optimizations mentioned earlier that are based on recognizing determinacy of goals. These optimizations yield, depending on the program, an improvement of 5% to 25% over the unoptimized version. Some improvement data is shown in Table 1 (each entry in Table 1 shows the time in milliseconds; the number in parenthesis gives the speed-up obtained).

3 From Independent to Dependent And-parallelism

Dependent and-parallelism arises when two conjunctive subgoals that share a common variable are executed in parallel. For example, given the goal `?- p(X), q(X)` executing `p(X)` and `q(X)` in parallel will lead to dependent and-parallelism. Dependent and-parallelism can be viewed as a generalization of independent-and parallelism. The presence of unbound variables (also termed *shared variables*) leads to several synchronization and execution ordering issues (discussed later) that make exploitation of dependent and-parallelism more complex than that of independent and-parallelism.

3.1 Issues in Exploiting Dependent And-parallelism

The presence of dependencies between and-parallel computation (because of shared variables) results in a number of new issues that need to be solved before dependent and-parallelism can be exploited. The two main problems that need to be tackled are:

Atomic Access: the presence of multiple goals executing in parallel that may access/bind the same variable simultaneously introduces the problem of ensuring that the integrity of the shared variable is maintained. That is, the operation of accessing or binding a shared variable should become atomic. Ensuring atomicity can be quite difficult because typically the operation of binding or accessing a variable, at the level of WAM, is compiled into a number of instructions. This is particularly true when an input variable is bound to a complex term. An arbitrary intermixing between two parallel computations may lead to a time-dependent and incorrect behaviour.

In the literature very few solutions have been proposed to this problem. The solutions that have been proposed range from the use of locks during binding creation [2] to the introduction of "invalid" entries on the heap to avoid ahead of time accesses of variables that are being bound [19]. The solution we propose is arguably superior to all these, in terms of efficiency and overhead. It consists of generating at compile time different sequences

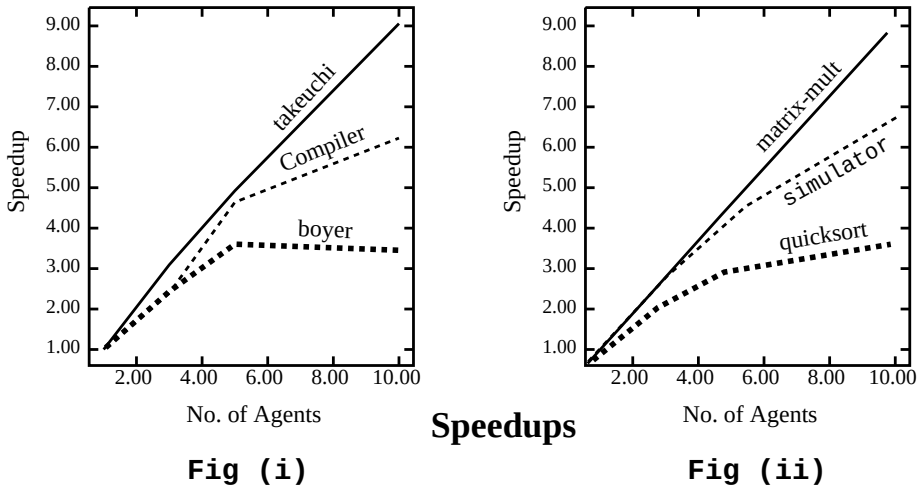

Figure 2: Speedups Curves for Selected Benchmarks on the Sequent Symmetry

of instruction for the same binding operation, and adding a simple run-time test which allows to select the sequence that most appropriately will avoid time-dependent behaviour [5].

Ordering: since we desire to support the Prolog language with its sequential semantics in our parallel system (which means that our parallel system should be able to run a Prolog program written for a sequential system without requiring any modification and produce the same behavior), a model for dependent and-parallelism needs to guarantee that bindings to variables are produced in the *correct order* (i.e., in sequential Prolog order). This can be seen in the following example. Let's consider the program

```
p(1).
q(2).        q(1).
```

and the query `:- p(X) & q(X)` (where, as before, `&` is used to represent a conjunction that can be executed in and-parallel). In a sequential execution `p(X)` is executed first, producing a binding $\{X \mapsto 1\}$, and successively the subgoal `q(1)` will be successfully completed. On the other hand, if `q(X)` is executed first, then it will produce a binding $\{X \mapsto 2\}$ and successively the subgoal `p(2)` will fail, leading to an unsuccessful execution. To tackle this problem, the notions of *producer* and *consumer* of a variable have been introduced. At every step of the execution, for each shared variable we identify exactly one producer (i.e., the subgoal which has the right to bind the variable). All the other subgoals accessing the shared variable are only allowed to read the value—i.e., they are consumers. A consumer accessing an unbound variable will suspend its execution (since it is not allowed to bind it), waiting either for a binding to be produced or for its turn to become itself a producer for such variable. The producer and consumer status is determined according to Prolog's operational semantics. Thus the goal containing the leftmost occurrence of the variable in the goal chain is designated as the producer for that variable. All other goals containing the same variable are designated as consumers. A consumer becomes a producer for a variable when the designated producer finishes execution without binding the shared variable and the consumer in question is the leftmost among all potential consumers of the shared variable. The (few) models presented in the literature dealing with dependent and-parallelism [19, 17] can be simply distinguished depending on the way in which the role of producer/consumers is computed and handled.

A careful study of the characteristics of a dependent and-parallel model leads to the identification of some basic operations that need to be implemented [5]: *(i)* creation of a parallel conjunction; *(ii)* remote set-up of a parallel subgoal; *(iii)* access/binding of a shared variable; and, *(iv)* producer/consumer detection. Our intuition, similar to the one for or-parallelism [6], is that at least one of this operation *cannot* be performed in constant time [5].

3.2 A New Model for Dependent And-parallelism

From the discussion in the previous section, we can formulate the following criteria for an ideal dependent and-parallel model: operations *(i)* through *(iv)* above should all be realized in constant time. Our intuition (as mentioned above) tells us that an ideal model is impossible to realize, because at least one operation has to be non-constant time. Given that for any model exploiting dependent and-parallelism, non-constant cost is going to be incurred in one of the four operations, we should prefer those models that move the non-constant time in operations whose frequency can be controlled by the runtime system. For example, having non-constant time variable access seems to be less appealing than having non-constant time setup of remote subgoals (since the frequency of this operation can be controlled by building a smart scheduler).

Using this criteria we have designed a new implementation scheme for dependent and-parallelism, based on a generalization of the concept of the Binding Arrays [16] used for or-parallelism. In this model, a table is associated with each subgoal, which represents that subgoal's "view" of the shared variables during the execution. Every shared variable will have one location in each of these tables, as shown in figure 3. The figure illustrates the case of two parallel subgoals sharing a variable `X` which has been bound to the term `f(a,Z)`.

The entries in the binding tables are linked, in order to allow retrieval of bindings produced by other subgoals. Furthermore, the traversal of these links during variable access allows one to distinguish between producer/consumer views of the variable—going through a link and reaching a location in a table associated to an active subgoal is equivalent to performing a *consumer* access to such variable (i.e., no binding activity will be allowed during such access).

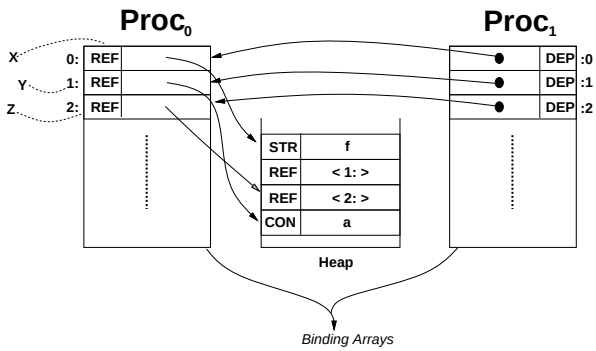

Figure 3: Environment Representation for Dep. And-parallelism

This model, which is currently in its implementation stage, is arguably better than those presented in the past, mainly because it guarantees a constant time access to shared variables (while most of the other proposals have a non-constant time access operation), and makes the process of determining producer/consumer status very efficient.

4 Or-parallelism

4.1 Introduction

The main problem to be solved in any implementation of or-parallelism is the *Environment Representation* problem. Two (or-parallel) threads originating from the same choice point may lead to binding 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.

In the literature a wide range of environment representation schemes to support implicit exploitation of or-parallelism have been proposed. The proliferation in the number of schemes is due to the considerable degree of freedom available in the design of models for or-parallelism. In [6] three criteria for classifying or-parallel models are proposed, based on analyzing the main steps that each model needs to implement (environment creation, variable access, and task switching). In [6] it is shown that at least one of the three operations will be a

non constant-time operation; furthermore, the fact that task switching (i.e., moving from one point of the computation tree to the other) is the only operation whose frequency can be controlled by the run-time system (e.g., by building a smart scheduler) makes models in which task-switching is the only non-constant time operation more suitable for efficient implementation.

4.2 And-Or Parallelism

The problem becomes more complicated once we consider concurrently exploiting 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. The requirement of and- and or-parallelism, in terms of environment representation, are antithetical.

This problem has been tackled in the ACE system [9] and in the SPBA model [7]—although the solution proposed can be virtually applied to almost any system combining and- and or-Parallelism. The idea 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 unbalanced 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 the team. In this way the exploitation of and-parallelism is realized at a lower layer, nested inside the exploitation of or-parallelism. Figure 4 illustrates this principle.

Nevertheless, the generalization of the environment representation techniques developed for or-parallelism to the case of and- and or-parallel system is not straightforward and must be appropriately modified in order to take into account this "layered" approach to parallelism.

4.2.1 Stack-Copying

One of the most successful approaches adopted for implementing or-parallelism is the *stack-copying* approach [1], adopted in the Muse system. The idea is relatively simple: whenever a computing agent $\mathcal{A}$ steals one alternative from a choice point (created by $\mathcal{B}$) to explore it independently, then it creates a complete copy of the environments allocated in $\mathcal{B}$. 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, 9].

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 needs 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 stack and logical sequence of execution.

A solution to these problems has been proposed in the ACE system [9].

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 its own team). The only new features that need to be added are the followings: (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 be-

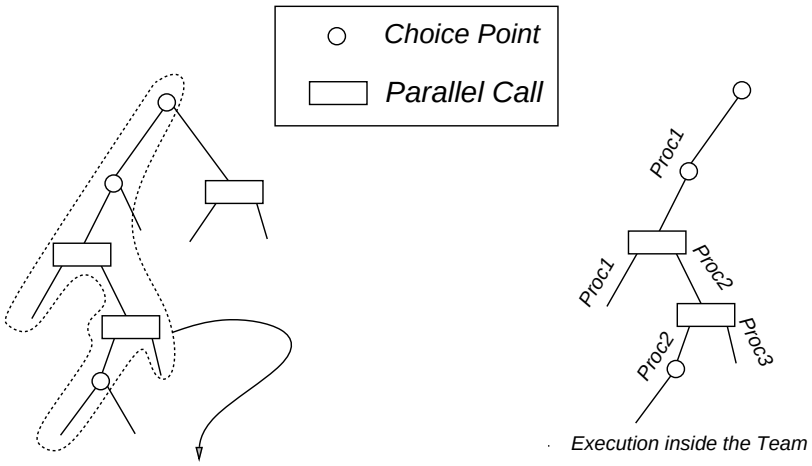

Figure 4: Layered exploitation of Parallelism

longing to the team. 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) . A choice point p satisfies the *sharing conditions* iff the whole computation on its left (in the and-tree exploited by the team) has already been completed. Figure 5 shows an example of this: choice point p satisfies the conditions, while the choice point q does not, since there is a branch on its left which has not completed yet (in the figure, a rectangular box represents an and-parallel call, each branch emanating from such a box corresponds to an and-parallel goal in the and-parallel call; darkened circles represent choicepoints).

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 6 illustrates an example of how copying is

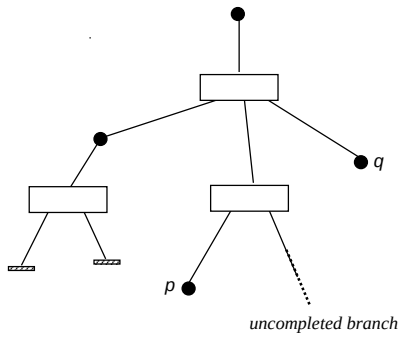

Figure 5: Sharing assumptions in ACE

performed: we have two active teams, $\mathcal{T}_1$ and $\mathcal{T}_2$, and $\mathcal{T}_2$ steals work from the choice point A in the computation created by $\mathcal{T}_1$ (figure 6(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 $\mathcal{T}_1$ has done between the choice point in which team $\mathcal{T}_2$ is presently positioned—choice point C—and the choice point from which work is taken) and is illustrated in figure 6(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 [9] for a detailed discussion of this topic.

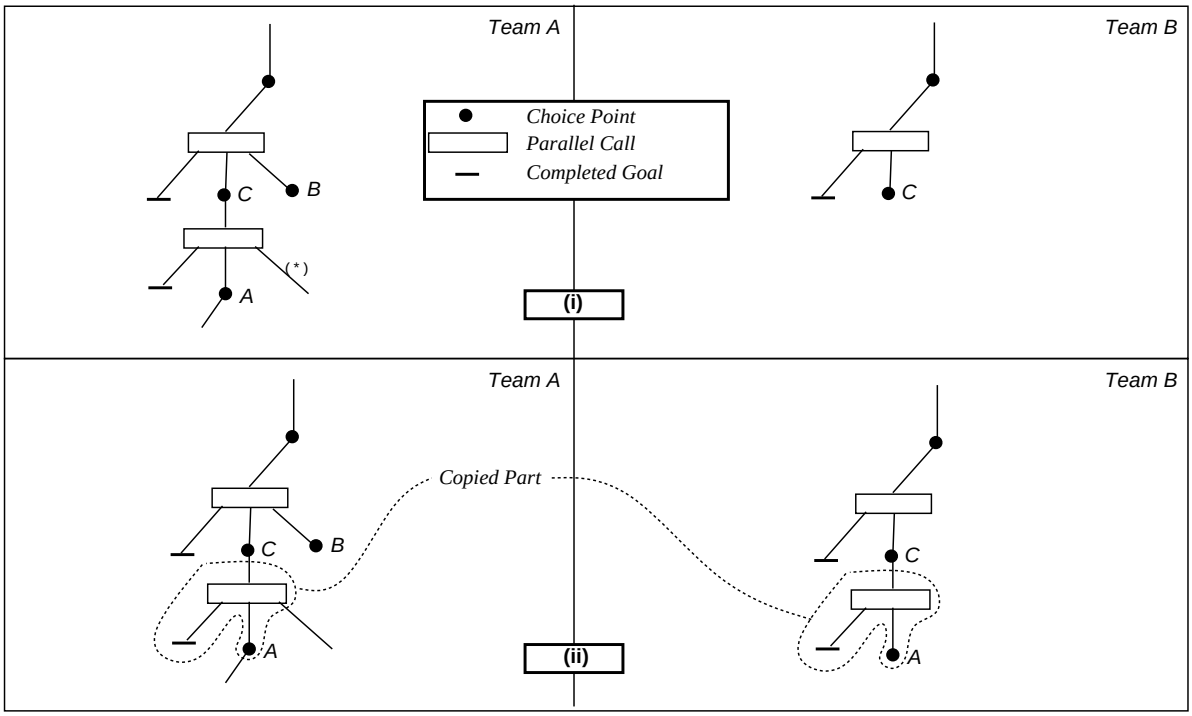

Figure 6: Copying Process

ACE is currently under development as a collaborative project between New Mexico State University and the Universidad Politecnica de Madrid. Even though it is not complete, the system is already showing improvements in execution times over the solely and-parallel system &ACE; table 2 shows some preliminary results obtained on a *generate & test* benchmark (generation is non-deterministic and test contains and-parallelism).

4.2.2 Binding Arrays

The other most significant approach taken to solve the environment representation problem in or-parallelism is the *Binding Arrays* method adopted in the Aurora system [16]. The idea is to associate with each agent (processor) a private array in which bindings for variables can be stored. Each variable for which binding conflicts between parallel threads are possible is translated into an index in the binding array. Each agent producing a binding for such a variable will store the binding in its private array at the location indicated by the index corresponding to that variable (the index is stored in the memory location corresponding to the variable in the control stacks and thus is accessible to all the processors; a tag indicates that the value is an index and not an actual binding). In this way the bindings produced by each agent for variables for which conflicting bindings can potentially be produced are stored locally in the binding arrays of the agents, avoiding any conflict. Whenever an agent takes work from a selected choice point, it needs to load its binding array with all the bindings produced up to that point of the execution.

The idea of binding array can also be generalized to the case of and- and or-parallel system. In this case the major problem to be tackled is the management of the binding arrays. A binding array should be associated with every 'computing agent' in charge of exploiting or-parallelism—now in an and+or parallel system such 'agent' is actually a team of processors. Thus the processors in the team need to share a unique binding array[6].

[6] Other approaches in which each processor has a private binding array are possible, but additional operations (copying of bind-

Benchmark	*Sequential*	*And-parallel*	*And/Or-parallel*
`generate&test`	1975	1211	917

Table 2: Generate&Test Benchmark

Nevertheless, a straightforward usage of a central binding array by a team of processor would lead to a far from optimal management of the space (garbage collection becomes very complex) and would make the operations of loading the binding array during stealing of work from a choice point a very expensive operation.

The solution developed by us for this problem is named *Paged Binding Array (PBA)*. The idea is to split the binding array of a team into fixed size pages, maintaining a page table for each team. During and-parallel computation in a team, each and-parallel subgoal will receive a different page (requiring a new one when it runs out of space in the current one). Figure 7 illustrates this concept.

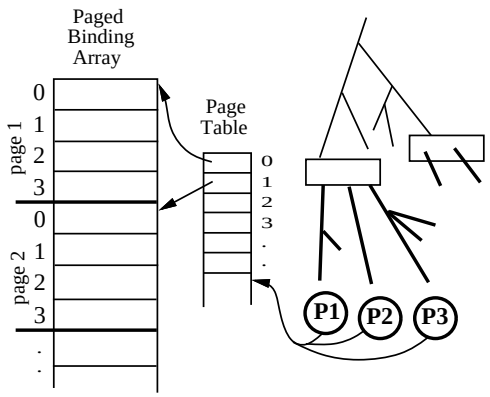

Figure 7: Paged Binding Arrays for And/Or Parallelism

This makes memory management and binding array loading straightforward and efficient.

5 Control *vs* Data Parallelism

In general, parallelism in any framework can be exploited in one of the two forms: MIMD or SIMD/SPMD. In MIMD different operations are concurrently applied to different data items (control parallelism). In SIMD (and SPMD) the same instruction/procedure is applied in parallel to different data items (data-parallelism). In the previous discussion we focused essentially on models exploiting the most general form of control parallelism. Nevertheless, Prolog SPMD parallelism arises naturally, and often, in Prolog programs. This is especially true of logic programs that manipulate lists. As an example, consider the program:

```
map(_,[],[]).
map(P,[H|T],[H0|T0]) :-
    call(P,H,H0),
    map(P,T,T0).
```

in which a predicate `P` is applied to all the elements of a given list.

A control parallel execution of this kind of programs (obtained by running the `call` in and-parallel with the successive recursive call) would not lead to an optimal exploitation of parallelism. This can be easily understood by observing the structure of the computation developed: in figure 8(i) we can see that the data structures required to support and manage the execution of the parallel conjunction (required especially to support backtracking) are nested one inside the other, forming a possibly very deeply tree. The cost of constructing and, in particular, visiting this tree represent a pure overhead that may affect the performance of parallel execution.

On the other hand, by identifying the data parallel nature of computation and taking advantage of it, we could simplify the structure of the computation as shown in figure 8(ii). Visiting and managing this structure is definitely simpler and more efficient.

Systems targeted to the exploitation of this form of data parallelism in Prolog programs have been designed

---

ings from one array to the other) would be required with unacceptable overheads.

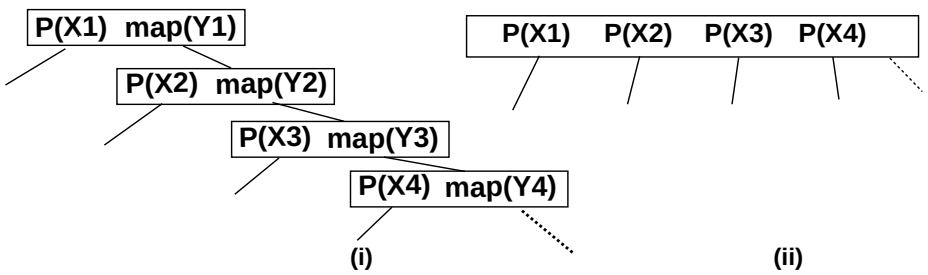

Figure 8: Structure of Control and Data Parallel Computations

and implemented [13] with very good results. Nevertheless, these systems are capable only of this specific form of parallelism (i.e., anything that does not fit in the mold of data-parallelism cannot be executed in parallel; so from the point of view of their usability as general parallel systems they have severe limitations [18]).

Comparable results can be obtained by applying some optimization techniques to an ordinary and-parallel system. We have developed one such optimization scheme, named *Last Parallel Call Optimization (LPCO)*[7]. The effect of LPCO can be simply described as follows: whenever a parallel call is nested inside another one, and the nested one does not have any continuation (i.e., it is the *last* parallel call), then it is possible to "promote" the nested parallel call and merge it with the outermost one (figure 9).

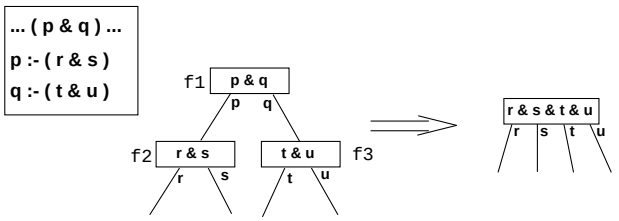

Figure 9: Last Parallel Call Optimization

This optimization allows, as we can see in figure 9, to obtain a computation structure that matches the one obtained from a SPMD style of execution. LPCO has been introduced in the &ACE system and the results are encouraging. The performance is slightly worse than the one produced by dedicated SPMD systems: this is due to the fact that in &ACE the branches are still generated one at the time, taking time linear in the number of branches, while dedicated systems have more effective ways of unrolling the recursion[8]. On the other hand the system shows speedups very close to those obtained with dedicated systems. Figure 10 illustrates the speedup curves obtained using LPCO (and compares with the analogous curves for control parallel execution) on some benchmarks containing a relevant combination of both data-parallelism and backtracking. As we can see the use of LPCO guarantees speedups also in cases where a control parallel system would not find any speedup at all (due to the complexity of traversing the deeply nested control structures).

6 Conclusion

In this paper, we discussed the major forms of parallelism found in logic (Prolog) programs. We discussed how each form can be efficiently exploited. We also discussed techniques that allow MIMD parallelism to be more efficiently realized as SPMD parallelism. We also presented performance data for the system that we have developed and that realizes some of the techniques discussed in this paper.

[7] The name is justified by the similarity between this optimization and the *Last Call Optimization* typically adopted in sequential implementations of Prolog, which is in turn is a generalization of the more commonly known tail recursion optimization.

[8] But compile-time transformation have been suggested [3] that would easily improve our results and bring them at par to those of dedicated SPMD systems.

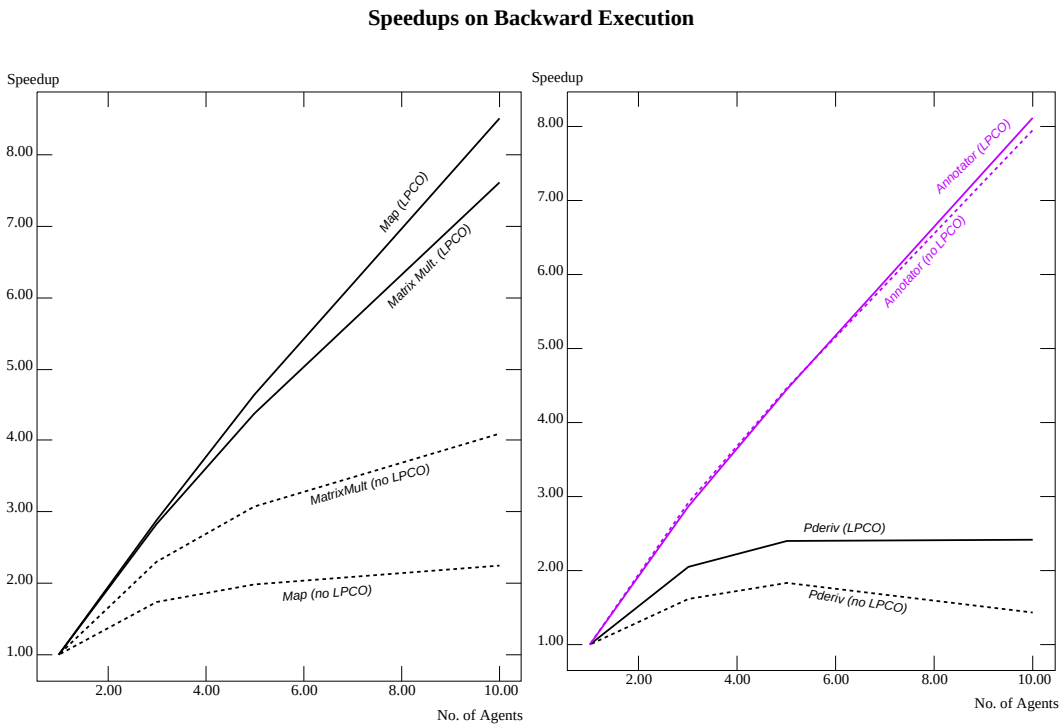

Figure 10: Speedup curves with and without LPCO

Acknowledgements

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

References

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

[2] 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.

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

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

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

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

[7] 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.

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

[9] 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.

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

[11] 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.

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

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

[14] L. Kanal and C.B. Suttner, editors. *Proceedings of the Workshop on Parallel Processing for AI.* Technische Universität München, Institut für Informatik, 1992.

[15] L. Lamport. The Parallel Execution of DO-loops. *Communications of the ACM*, 17:83–93, 1974.

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

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

[18] 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.

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

[20] L.N. Kanal V. Kumar, P.S. Gopalakrishnan, editor. *Parallel Algorithms for Machine Intelligence and Vision.* Springer Verlag, 1990.

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

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