An Overview of the ACE Project

Gopal Gupta & Enrico Pontelli
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: The goal of the ACE project is to fruitfully exploit *all* forms of parallelism from Prolog programs in a single framework. Three principal forms of parallelism, namely, or-parallelism, independent and-parallelism, and dependent and parallelism are found in logic programs. Our goal is to not only exploit the control-parallel incarnations of these forms of parallelism but also their data-parallel ones. Our aim is to exploit all forms of parallelism without any intervention by the user, thus all parallelism is either automatically detected at compile time or implicitly exploited at run-time. We also propose a number of optimizations that greatly improve the runtime performance of our parallel system. These optimizations are applicable at runtime, and lead to exploitation of data-parallel versions of and- and or-parallelism. Our system employs the notions of conditional graph expressions, RAPWAM, and stack-copying and thus subsumes both Muse (an or-parallel system) and &-Prolog (an independent and-parallel system). The final ACE system will also subsume DDAS (a dependent and-parallel system), Reform Prolog (a data-and parallel system), and Multilog and Firebird (data or-parallel systems).

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

1 Introduction

Parallel processing of numerical problems has been an active area of research for almost two decades now [20]. 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 [9], 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.

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* (i.e., *implicit* parallelism). 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. 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[1]—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.

Three principal kinds of (implicitly exploitable) control parallelism can be identified in logic programs: *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[2] concurrently working along different or-alternatives are named *or-agents*. *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*. *Dependent and-parallelism* arises when two or more goals in the body of a clause have a common variable and are executed in parallel. Various models have been proposed in the literature to exploit the different forms of parallelism. 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 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 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.

2 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 [15] or hand-coded by the programmer. Execution of all goals in a parallel conjunction is started in parallel when control reaches that parallel conjunction—a proper descriptor is allocated and the subgoals are pushed in a work queue. Backtracking becomes complicated in and-parallel systems because more than one goal may be executing in 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

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

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

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 [15], and it is based on a direct generalization of the right-to-left backtracking behaviour of standard Prolog. This backtracking semantics has been fully implemented for the first time in the &ACE [6] system. The implementation of &ACE itself is inspired by the RAPWAM [15][3]. &ACE is an extension to the sequential WAM (Warren Abstract Machine [19]) 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 are quite impressive. The tests that have been run on &ACE range from simple toy programs to actual applications (see table 1), and not only 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 (like matrix multiplication and hanoi). 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 1 indicates, for each benchmark, the execution time (in ms.) and the speed-up obtained.

3 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. 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. 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. The solution we propose consists of generating at compile time different sequences 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 (which means that our parallel system should be able to run a Prolog program written for a sequential system 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). In general, violating the left-to-right ordering of bindings production may lead to a different behaviour. 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. The (few) models presented in the literature dealing with dependent and-parallelism [16] can be simply distinguished depending on the way in which the role of producer/consumers is computed and handled.

[3] Other implementations based on the principles of RAPWAM have also been proposed in the past, like &-Prolog [15] and DDAS [16].

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)
hanoi(11)	1671	550 (3.04)	336 (4.97)	180 (9.28)
boyer(0)	9290	3829 (2.43)	3199 (2.90)	2687 (3.46)
compiler	29902	12522 (2.39)	6437 (4.65)	4801 (6.23)
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)

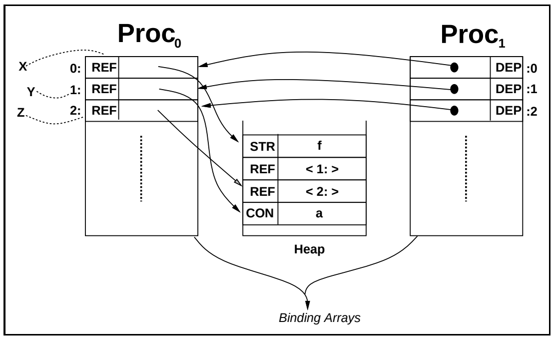


Figure 1: Environment Represent. for Dep. And-P.

We have designed a new implementation scheme for dependent and-parallelism, based on a generalization of the concept of the Binding Arrays [10] 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 1[4]. 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). This model 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]The figure illustrates the case of two parallel subgoals sharing a variable `X` which has been bound to the term `f(a,Z)`.

4 And/Or-parallelism

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. The problem becomes more complicated once we consider concurrently

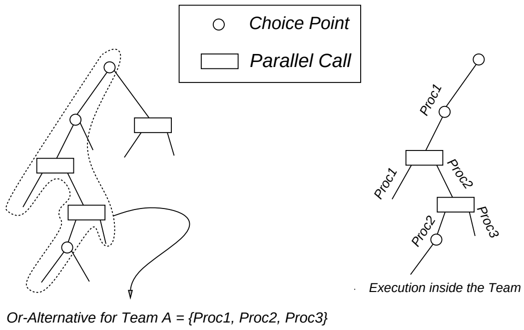


Figure 2: Layered Exploitation of Parallelism

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 requirements of and- and or-parallelism, in terms of environment representation, are antithetical. The solution 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 2 illustrates this principle. This principle has been used in the models presented in the next two sections.

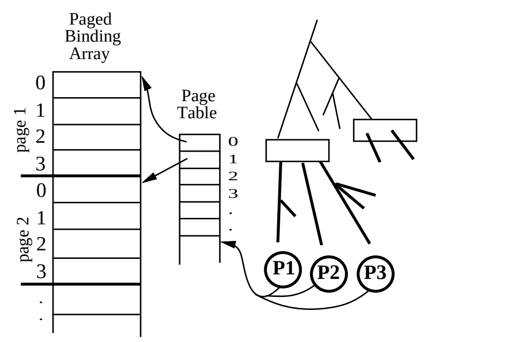
4.1 Stack-Copying

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. 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 stack and logical sequence of execution. A solution to these problems has been proposed in the ACE system [8]. ACE is based on the layered model sketched before. 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. 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) . Even though ACE is not completely implemented, the system is already showing improvements in execution times over the solely and-parallel system &ACE; on a *generate & test* benchmark (generation is non-deterministic and test contains 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.

4.2 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 [10]. 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, and this mapping is unique in the system. 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. 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



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

Figure 3: Paged binding array for And/or Parall.

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[5]. 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 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 3 illustrates this concept.

5 Runtime Optimizations and the Duality Principle

We have developed a number of runtime optimizations [14, 7], based on well-known optimization principles that help us considerably in improving the efficiency of our engine. The optimization principles employed are listed below:

Flattening Principle

> *Flatten the tree structure, reducing the levels of nesting whenever possible, preserving the operational semantics.*

The use of flattening principle is illustrated in the Section 6.

Procrastination Principle

> *The execution of an operation should be delayed until its effects are needed by the rest of the computation.*

The procrastination principle is used during the allocation of markers used to demarcate the stack segments corresponding to and-parallel goals. If an and-parallel goal is deterministic, we do not need to know its boundaries during backtracking. Thus, the marker for an and-parallel goal is allocated only when it is clear that that goal is non-deterministic. Thus, the allocation of a marker is procrastinated until the time the first choice point is allocated. If the end of the and-parallel goal is reached without allocating any choice point then the marker is never allocated. Avoiding allocation of markers leads to considerable savings in time and memory during execution (as much as 25% in execution time and 50% in control stack memory consumed).

Sequentiality Principle The sequentiality principle can be stated as follows:

> *Two consecutive branches of the same node of the computation tree executed by the same computing agent should produce minimal overhead.*

The sequentiality principle is also employed during and-parallel execution. If the same processor picks up consecutive goals in a parallel conjunction, then the two consecutive goals are treated as one sequential goal, thus reducing the parallel overhead. Given the parallel conjunction `(a & b & c)`, if processor P1 executes `a` and `b`, and processor P2 executes `c`, then at runtime, when `b` is picked up, we pretend as if the parallel conjunction is `((a, b) & c)`. This idea has also been incorporated

[5] Other approaches in which each processor has a private binding array are possible, but additional operations (copying of bindings from one array to the other) would be required with unacceptable overheads.

into the ACE implementation, and has resulted in considerable savings in time and space (similar to those for the procrastination principle).

Reuse Principle

> *Reuse a data-structure or computation whenever possible.*

The reuse principle is used in the last parallel call optimization principle described in section 6.

Duality Principle

> *And- and or-parallelism are inherently the same, and can be realized using similar implementation techniques.*

The duality principle has been exploited in improving both the and-parallel component and the or-parallel component of ACE [13]. Scheduling techniques from or-parallel implementations have been incorporated into scheduler of and-parallelism, resulting in better performance. Likewise, optimization techniques developed for and-parallelism have been applied to or-parallelism resulting in improved performance [13].

6 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, SPMD parallelism also arises naturally, and often, in Prolog programs. This is especially true of logic programs that manipulate lists.

Both or-parallelism and and-parallelism can be cast into data-parallel versions. Our idea behind exploiting data-parallelism is very simple: a repetitive task that is inherently data-parallel is normally programmed using recursive unfolding (a control-parallel structure) in Prolog. At runtime we recognize such structures and transform them to more efficient data-parallel structures. This is illustrated further below for both and-parallelism and or-parallelism.

6.1 Data And-parallelism

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.

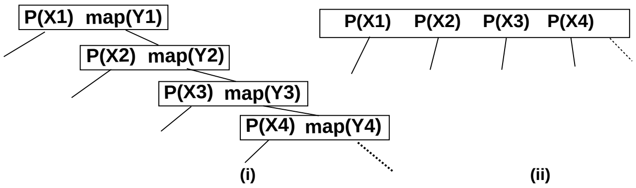


Figure 4: Control and Data Parallelism

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 4(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 4(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 and implemented [2] with very good results. 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)*. 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.

This optimization allows 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[6]. On the other hand the system shows speedups very close to those obtained with dedicated systems. Figure 5 illustrates the speedup curves obtained using LPCO (and compares with the analogous curves for control parallel execution) on some benchmarks containing a mix 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).

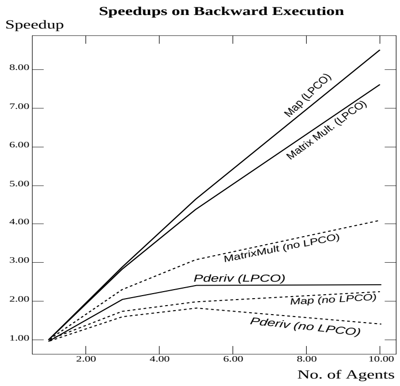


Figure 5: Speedup curves with and without LPCO

6.2 Data Or-parallelism

In the previous subsection we saw that data and-parallelism can be successfully exploited from logic programs, and as was demonstrated, it's much more cost-effective to exploit and-parallelism as data-parallelism rather than as control-parallelism. According to the duality principle, the same should be true for or-parallelism. This is indeed the case. In many prolog programs, or-parallelism is exploited via the member or the select predicate (e.g. in the well-known N-queens problem). The member (or select) acts as a generator of values taken from a list and then a test predicate filters out the unacceptable values. Most constraint logic programming (CLP) applications also work in a similar way, except that the generator generates values listed in the finite domain (of course, in CLP part of the test may be executed early before the generation process starts, to prune the finite domain and make execution more efficient). The generation process in constraint logic programming languages (e.g. CHIP [21]) is very similar to generation using member or select.

Consider the following very simple program:

```
member(X,[X|T]).
member(X,[Y|T]) :- member(X,T).
:- member(X, [1, 2, 3, 4]), even(X).
```

Where `even(X)` succeeds only if `X` is an even number. The role of `member` is essentially to go down the list creating all the necessary choicepoints, so that `X` gets successively bound to the elements in the list (See Figure 6.(i)). The main aim of the member predicate in the goal above is to produce a 4 way choicepoint. This can be achieved by flattening the tree structure created by member (See Figure 6.(ii)). This flattening, like in LPCO, can be done at runtime if certain conditions are met [12]. This flattening results in a simpler tree structure and thus in less overhead in exploiting or-parallelism. This optimization, termed Last Alternative Optimization (LAO), has been implemented in the ACE system and has shown good results. Preliminary results are shown in Table 2. On applying some simplifications,[7] the performance becomes slightly better (see Table 3). The performance can be further improved by developing a dual of Debray's transformation for or-parallelism [3].

Dedicated data-or parallel systems, such as Firebird [18] and Multilog [17], have been built, with good results. Both Firebird and Multilog are based on ideas similar to each other, though Firebird is designed with CLP systems in mind, while Multilog is designed for Prolog (also, they are specifically targeted towards SIMD machines such as the Connection Machine). LAO gives us the capability of incorporating data or-parallelism into a general purpose or-parallel system. Thus, the final ACE system will have the capability to subsume systems like Firebird and Multilog. This demonstrates that ACE can be just as efficient as a special purpose system when the

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

[7]These simplifications have do with getting around the restrictions imposed by the SICStus Prolog implementation; the ACE engine is based on SICStus Prolog.

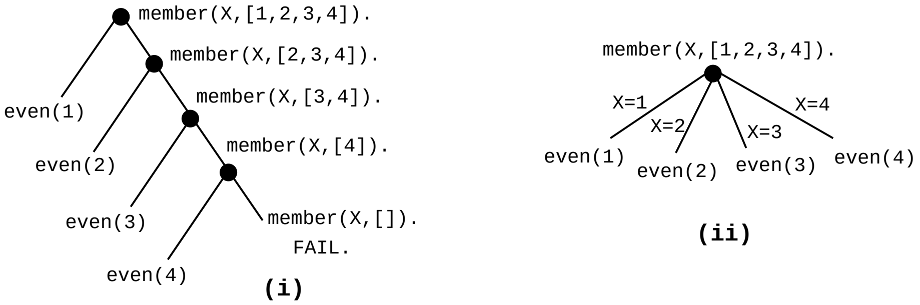


Figure 6: Data Or-parallelism

Goals executed	ACE agents 2	5	10
member	40 / 36	36 / 22	34 / 20
queens	77 / 70	56 / 51	32 / 29
ancestor	40 / 50	36 / 34	34 / 20

Table 2: Unoptimized/Optimized Execution times for LAO (in sec.)

special case arises, while still being usable in the general case.

7 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. The ACE system is still under development, and we hope should be ready for distribution in the near future. Our experience tells us that considerable research still needs to be done in devising runtime optimizations so that efficient runtime systems can be developed for parallel logic programming. The ACE project is a step in that direction.

Acknowledgements

The research presented in this paper has benefited from collaboration with Universities of Bristol, Madrid, and Oporto. Discussions with K. Ali, M. Carro, M. Hermenegildo, R. Karlsson, K. Shen, V. Santos Costa, and D.H.D. Warren have greatly helped, all of whom we would like to thank. Ongoing work is supported by NSF Grants CCR 92-11732, HRD 93-53271, and INT 95-15256 and by NATO Grant CRG 921318.

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] 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.

[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

Query	optimized	unoptimized
queens	50	35
member	54	35

Table 3: Top-level only application of LAO (time in sec.)

for Logic, DB, and Advanced Programming, 1995. Internal Report.

[6] 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.

[7] G. Gupta and E. Pontelli. Optimization Principles for Parallel Non-deterministic Implementations. Internal Report. NMSU. Jun '95.

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

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

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

[11] E. Pontelli and G. Gupta, Data-parallel Logic Programming in &ACE. In *IEEE Symposium on Parallel and Distributed Programming '95*. IEEE Computer Society, San Antonio, TX, Oct. 1995.

[12] E. Pontelli and G. Gupta. Last Alternative Optimization. Internal Report. NMSU. 1995.

[13] E. Pontelli and G. Gupta. On the Duality between And-parallelism and Or-parallelism in Logic Programming. In *European Conf. on Parallel Processing (EUROPAR) '95*. Lecture Notes in Computer Science, Springer Verlag, pp. 43-54.

[14] E. Pontelli, D. Tang, G. Gupta. Determinacy Driven Optimization of And-parallel Logic Programming Systems. In *International Conference on Logic Programming*, MIT Press. June 1995.

[15] 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.

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

[17] D. A. Smith. MultiLog: Data Or-parallel Logic Programming. In *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, June 1993, pages 314–332.

[18] B-M Tong and H-F Leung. Massively Parallel Concurrent Constraint Programming. In *Proceedings of International Logic Programming Symposium '93*, Vancouver, Canada.

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

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

[21] P. Van Hentenryck. Constraint Handling in Prolog. MIT Press, 1987.