Last Alternative Optimization*

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

Abstract

We present a new optimization for or-parallel logic programming (Prolog) systems, called *Last Alternative Optimization (LAO)*. The LAO follows from the *Flattening Principle* and the *Principle of Duality of or-parallelism and and-parallelism.* Originally LAO was conceived as the dual of the *Last Parallel Call Optimization* [4] an optimization developed by us for and-parallel systems. LAO enables Prolog programs that have *data-or parallelism* to execute more efficiently. It also enables more efficient (parallel) execution of *Constraint Logic Programs over finite domains*. LAO is a fairly general optimization and can be readily applied to virtually any parallel system that exploits non-determinism (e.g., parallel search based artificial intelligence systems). The Last Alternative Optimization has been implemented in the ACE parallel Prolog system. The performance results presented in this paper indeed prove the effectiveness of LAO. We present a second optimization based on the flattening principle, called the *Balanced Nesting Optimization (BNO)*, that is related to LAO, and that also leads to reduction of parallel overhead.

1. Introduction

Logic programming is a popular computer programming paradigm that has been used in wide variety of applications. The most popular logic programming language is *Prolog*. An advantage of logic programming languages over traditional programming paradigms is that parallelism implicitly present in program constructs can be *automatically* exploited, resulting in faster execution. Thus, programs coded for sequential machines, can be run in parallel without much intervention from the user. This approach differs considerably from the approaches in which parallelism is explicitly programmed by the user. Three main types of (implicit) parallelism have been identified and successfully exploited in logic programs:

- *Or-parallelism:* arises when more than a single rule defines some relation and a procedure call unifies with more than one rule head—the corresponding bodies can then be executed in or-parallel fashion.
- *And-parallelism* arises when more than one goal is present in the query or in the body of a procedure, and these (multiple) goals are solved in parallel with each other.

In this paper we are mainly concerned with or-parallelism. An or-parallel execution can be depicted using a tree, called the *search tree* or the *or-tree*. The or-tree has a node for each subgoal solved during execution and a branch for each clause matching that subgoal. During parallel execution, the tree structure of the computation needs to be repeatedly traversed in order to search for multiple alternatives and/or cure failure of goals, and such traversal often requires synchronization between processors. These traversal and synchronization activities are absent from a sequential computation (in which the execution is reduced to a linear and fast scan of the branches in a predetermined order). If execution is to remain efficient, the tree structure developed should be as simple as possible. With this observation in mind we have developed the *Flattening Principle* to improve the efficiency of execution.

In this paper we present two optimizations called *Last Alternative Optimization* (LAO) and *Balanced Nesting Optimization* (BNO) that *simplify* the structure of the Prolog search tree allowing for more efficient traversal. These optimizations are based on the principle of flattening that we have successfully used for speeding up execution of and-parallel programs, and the *principle of duality*. The principle of flattening simply states that a reduction in the number of levels of nesting of an execution tree, generally results in improved performance. The principle of duality is based on our insight that, given a technique that has been developed for an and-parallel logic programming system, its dual can be successfully applied to an or-parallel system (and vice versa). We also show how LAO and BNO can be applied for: (i) exploiting *data or-parallelism* more efficiently, and (ii) exploiting parallelism from constraint logic

*This work is supported by NSF Grants CCR 92-11732, CCR 96-25358, INT 94-15256, and HRD 93-53271, and by NATO Grant CRG 921318.

programming (CLP) systems based on *finite domains* [8]. The results of implementing these optimizations in the ACE system, a high performance parallel Prolog system are also presented.

2. Last Alternative Optimization

The last alternative optimization is based on the principle of flattening. After we developed the LPCO [4], an optimization for and-parallel systems, the principle of duality led us to believe that the dual of LPCO should work for or-parallel systems. That is how the idea of LAO was born.

We illustrate the Last Alternative Optimization with an example. In a majority of or-parallel applications, or-parallelism arises because a variable can be given one of many possible values. This is typically coded as a call to `member` or `select` predicate. For example, given a variable `V`, we want it to assume values from the list `[1,2,3]`, and for each possible value of `V` we want to perform some computation (almost all non-deterministic search problems and constrained optimization problems are programmed in this manner). The query will look as follows:

```
?- member(V,[1,2,3]),compute(V,R).
```

where `R` will hold the result of the computation with a given value for `V`, and `member` is defined as follows.

```
member(X,[X|T]).
member(X,[Y|T]) :- member(X,T).
```

The search tree structure created is shown in figure 1(i). Notice that or-parallelism will be exploited when processors move to choice points (shown as dark circles in Figure 1) and pick the `compute` goals for execution. A possible scenario is shown in figure 1(i) itself where processors P1, P2, and P3 are exploring the three branches created for solving `compute(`i`,R)` where $1 \leq i \leq 3$. Note that the intent in the program is to fire off a `compute` goal for every element in the list `[1,2,3,4]`. However, because a repetitive action can be programmed in a logic programming language only via recursion, we are forced to use the recursive `member` predicate for firing off all the `compute` goals. Thus, the various calls to compute, one for each element of the list, can only be generated in a recursive manner. Ideally, we would like to have all calls to `compute` generated all at once as shown in Figure 1(ii).

The tree structure shown in figure 1(i) is nested, thus processors have to traverse larger distances to move around the tree, and obtain work (the various `compute` goals). In contrast, in Figure 1(ii) all `compute` goals are clubbed at one choice points from where they can all be picked by processors with less traversal.

The same discussion applies to the `select` predicate that, like the `member` predicate, is also traditionally used to spawn or-parallelism:

```
select(X,[X|T],T).
select(X,[Y|T],[Y|R]) :- select(X,T,R).
```

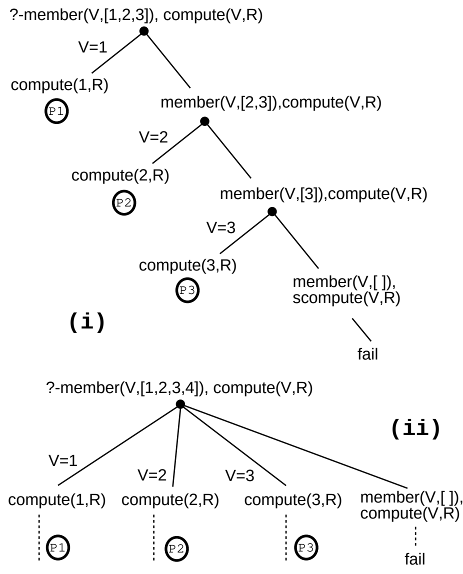

Figure 1. Search tree created for member

When the last alternative in a choice-point $\mathtt{B}_1$, further creates a choice-point $\mathtt{B}_2$, then there is no need to allocate this new choice-point, rather $\mathtt{B}_1$ can be updated with the information that will be stored in $\mathtt{B}_2$. The information in $\mathtt{B}_1$ is no longer needed as the last alternative has already been picked up for execution. (We do assume that when processors pick work in the form of untried alternatives from a choice-point, they do it in the Prolog order, i.e., left to right, so that the rightmost alternative is always the last alternative to be selected). Of course, while reusing the existing choice point, we have to make sure that we will not change the meaning of the program. Thus, we have to ensure that the all the alternatives have access to the correct environment, the correct arguments, etc. Note that while we have illustrated LAO with the `member` example, where the two choice points merged ($\mathtt{B}_1$ and $\mathtt{B}_2$ above) correspond to the same predicate, in the general case, they may correspond to different predicates (however, in such a case special action will be needed if predicates corresponding to the two choice points have different number of arguments; see [7] for details).

Note that LAO doesn' t bring much benefit to sequential implementations of Prolog [11], Unlike a sequential system, in an or-parallel implementation when the last alternative from a choice point is picked, that choice point cannot be deallocated as alternatives before the last one may still be active (i.e., they are still being explored by other processors). Thus, reuse of choice point in a parallel system can lead to considerable savings in memory and time. Also, because of the simplification of the tree structure due to flattening, the need for processors to synchronize with each other is reduced, resulting in not only a simple to implement model but also a more efficient one.

It is our conjecture that LAO will considerably benefit or-parallel systems such as Aurora [10] (at present we have only tried it on Muse). The `find_match` predicate used in the DNA application in [3] will also benefit from LAO. LAO will also improve the performance of programs that use failure driven loops based on Prolog's `repeat` builtin.

3. Implementation of LAO

ACE [6, 5] is a high performance implementation of parallel Prolog based on the SICStus WAM engine that exploits both and- and or-parallelism. We have incorporated the LAO in the ACE system and results are reported in section 3.1. In this section we describe the changes that had to be made to the ACE system in order to implement LAO. The ACE system uses the *stack copying* environment representation technique (originally developed for the MUSE or-parallel system [1]) for representing *multiple environments* that arise during or-parallelism. Stack copying is based on the idea of creating different copies of the existing environments when branching takes place (i.e., when an idle processor selects an alternative from a choice point to start a parallel computation).[1] The existence of local copies of the "shared" environments allows to overcome the binding management problem, since each processor will have a local space to store the locally generated binding. Connection between the different copies of the same choice point is maintained through a *shared frame*. The shared frame contains those parts of the choice point that are shared and that may be modified during the parallel execution

3.1. Implementation of LAO in ACE

The current implementation of LAO in ACE is quite simple. The opportunities for applying the optimization are automatically identified. A test is performed before creation of new choice points. If the current execution is the rightmost alternative of the previously created choice point, and both the previous choice point and the new one are "parallel"[2], then no new choice point will be created. Instead the previous choice point is reused.

The "reuse" of the choice point is simply realized by avoiding allocation of the new choice point and by using the fields of the old choice point to store the various state information. This gives the advantage that certain fields of the choice point may already contain the right value and their initialization may be avoided (e.g., pointer to the heap). Reuse of choice point also involves attaching the alternatives of the choice point being created to the existing choice point, and storing in the same choice point the relevant state information (e.g., value of the arguments of the associated subgoal). Some additional changes to the design of the data structures are required to accommodate the new features of LAO and its implementation in its most general form. Details are omitted due to lack of space and can be found elsewhere [7].

Table 1 describes the results obtained by running some benchmarks on the ACE systems with and without LAO. Each entry in the table indicates the unoptimized execution time (in milliseconds), the time obtained using LAO, and the corresponding percentage of improvement. All the figures have been obtained on a Sequent Symmetry multiprocessors[3].

4. Further Applications of LAO

The Last Alternative Optimization *transforms control or-parallel computations into data-or parallel ones*. This is especially obvious from the `member` example, where all the calls to `compute` (one for each element of the list) get clubbed into one choice-point as a result of the LAO. All these `compute` goals execute identical instructions but on different data. The effect of LAO is thus, similar to the *disj* operator in Smith's Multilog, except that the runtime system automatically exposes data or-parallelism from data or-parallel computations that have been programmed using a control parallel construct (recursion in the case of logic programming). A comparison with performance of Multilog is not possible because Multilog has not been implemented on a Sequent Symmetry. However, we believe that performance of data-or parallel programs on ACE augmented with LAO will be comparable to their performance on MultiLog, a dedicated data-or parallel system. The main advantage that LAO has over approaches such as Multilog is that it is applicable to more general parallel logic programming systems. No special machinery for data-parallelism needs to be built in.

LAO will be specially useful in parallel constraint logic programming systems, based on finite domains, such as CHIP [8]. CLP programs make extensive use of the `indomain` and `labeling` predicates that are very similar to the `member` predicate, and are thus amenable to LAO and data-parallelism.

5. Balanced Nesting Optimization

The LAO is the dual of LPCO. The dual of other optimizations devised for and-parallelism can similarly be con-

[1] Note that stack copying requires that each processor's address space be mapped in such a way that it is identical to all other processors' address space, so that on copying of stacks, no pointer relocation is required.

[2] Note that in most or-parallel systems whether a choice point is parallel or sequential is indicated by the user. Restricting the application of the LAO to parallel choice points is necessary in order to limit the influence of the optimization on the behaviour of the scheduler.

[3] Analogous improvements have been observed on a Sun Sparc multiprocessor machine.

Program	No. of Agents				
	1	2	4	8	10
Queen(naive)	3689/3889 (-5%)	2939/2129 (28%)	1959/1159 (41%)	1910/730 (62%)	1909/629 (67%)
Queen(smart)	799/850 (-6%)	510/450 (12%)	320/240 (25%)	229/150 (34%)	229/149 (35%)
Puzzle	2939/3001 (-2%)	1529/1589 (-4%)	890/809 ((9%)	540/429 (21%)	519/360 (31%)
Ancestors	2460/2706 (-10%)	1269/1370 (-8%)	669/629 (6%)	399/299 (25%)	340/201 (41%)
Members	8029/8450 (-5%)	4021/3731 (7%)	3733/2667 (29%)	3480/2080 (40%)	3400/2011 (41%)
Maps	35420/36240 (-2%)	21079/19879 (6%)	11620/12189 (-10%)	9290/8329 (10%)	6100/7100 (-16%)

Table 1. Improvements using LAO (unoptimized/optimized)

ceived of. It is well known that certain linearly recursive programs can be transformed into a doubly recursive divide-and-conquer programs that are more amenable to and-parallel execution [9, 2]. This transformation relies on the associativity and commutativity of operators used in the program. The same idea can be applied to or-parallel systems, and linearly recursive structured programs. Let's consider the member predicate again. The query `member(X, [1,2,3,4])` will produce a search tree shown in Figure 2(i). The objective of the member predicate is to act as the generator and launch the compute goal for every element of the list. A better way of doing this is to convert the list into a balanced binary tree, and then use tree-member as a generator. The effect of the program is the same except that tree-member produces a more balanced (flattened tree), so that the overhead of getting the work arising out of elements in the list that come later is not unnecessarily more. Figure 2 illustrates this situation.

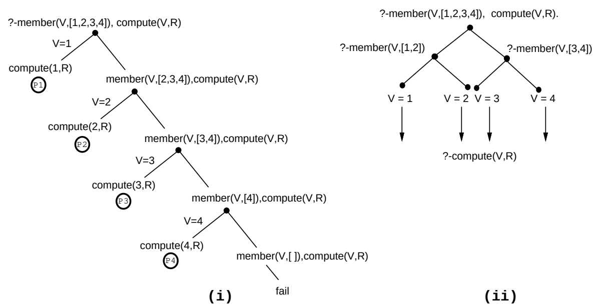

Figure 2. From Linear List to Binary Tree

Tree-member unfold the computation into a log-structured tree, traveling to one of whose nodes involves only a $O(\lg n)$ overhead. In contrast, member will involve $O(n)$ overhead. Performance improvement comes about because with a more balanced, flatter tree the overhead incurred in picking work (untried alternatives) is simplified. In case of MUSE, it also means less amount of copying, and in case of Aurora[4] less work at the time of task-switching. We have applied BNO to 2 benchmarks; performance figures are shown in Table 2. times obtained with and without the transformation.

[4] Aurora is an or-parallel system based on a different environment representation scheme.

Execution	No. of Agents				
	1	2	4	8	10
Unoptimized	4490	3919	1580	789	659
Optimized	4750	2990	1529	779	659
Improvement	-6%	24%	3%	1%	0%
Unoptimized	3490	3500	2340	2100	2040
Optimized	3590	3029	2019	1610	1570
Improvement	-3%	13%	14%	23%	23%

Table 2. Queens and Gen&test using BNO

References

[1] K. 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] S. Debray and M. Jain. A Simple Program Transformation for Parallelism. In *Proc. Symp. on Logic Programming*, MIT Press, 1994.

[3] E. Lusk et al. Applications of the Aurora Parallel Prolog System. In *Int'l Logic Progr. Symp.* MIT Press, 1993.

[4] E.Pontelli and G. Gupta. Data And-parallel Logic Programming in &ACE. In *Proc. Symp. on Parallel and Distr. Proc.* IEEE, 1995.

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

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

[7] G. Gupta and E. Pontelli. Last Alternative Optimization. Tech. Rep. 9606, New Mexico State University, 1995.

[8] P. V. Hentenryck. *Constraint Handling in Prolog*. MIT Press, 1988.

[9] M. Hermenegildo and M. Carro. Relating Data-Parallelism and (And-)Parallelism in Logic Programs. In *EuroPar'95*. Springer Verlag, 1995.

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

[11] M. Meier. Recursion vs. Iteration in Prolog. In *Int'l Conf. on Logic Progr.* MIT Press, 1991.