Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems *

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

Naive parallel implementation of nondeterministic systems (such as a theorem proving system) and languages (such as a logic, constraint, or a concurrent constraint language) can result in poor performance. We present three optimization schemas based on flattening of the computation tree, procrastination of overheads, and sequentialization of computations that can be systematically applied to parallel implementations of non-deterministic systems/languages to reduce the parallel overhead and to obtain improved efficiency of parallel execution. The effectiveness of these schemas is illustrated by applying them to the ACE parallel logic programming system. Performance data presented shows that considerable improvement in performance can result.

1 Introduction

Non-determinism arises in many areas of computer science. Artificial intelligence and constrained optimization are two such areas where non-determinism is commonly found. By non-determinism we mean the existence of multiple (potential) solutions to a problem. Search problems, generate-and-test problems, constrained optimization problems, etc. fall in this class. Non-determinism has also been incorporated in many programming languages: logic programming languages (e.g., Prolog), constraint programming languages (e.g., Chip) [7], concurrent constraint languages (e.g., AKL), and rule based languages (e.g., OPS5) being some of the salient examples.

Non-determinism present in a problem offers a rich source of parallelism. A problem is usually expressed as a *goal* to be achieved/proved/solved together with *rules* (or clauses) that specify how a given goal can be reduced into smaller subgoals. Given a (sub-)goal, there may be multiple ways of reducing it (non-determinism). On applying a rule, a (sub-)goal may reduce to a number of smaller (conjunctive) subgoals, each of which need to be solved in order to prove the original (sub-)goal. Two principal forms of parallelism can be exploited in problems that admit non-determinism:

1. *Or-parallelism:* the different potential solutions can be explored in parallel (i.e., given a subgoal, there may be more rules that can be used to reduce it).
2. *And-parallelism*: while looking for a specific solution, the different operations involved can be executed in parallel (i.e., conjunctive subgoals may be reduced in parallel).

Or-parallelism is a direct result of nondeterminism, while and-parallelism is the "traditional" form of parallelism, also found in standard (deterministic) programming languages.

Solution of problems with non-determinism has been abstractly represented using *and/or trees*. Or-parallelism can be exploited by exploring the branches emanating from an *or-node* of this tree in parallel. Likewise, and-parallelism is exploited by exploring the branches emanating from *and-nodes* in parallel. A system that builds an and-or tree to solve a problem with non-determinism may look trivial to implement at first, but experience shows that it is quite a difficult task. A naive parallel implementation may lead to a slow down, or, may incur a severe overhead compared to a corresponding sequential system. Excessive parallel overhead may cause a naive parallel system to run many times slower on one processor compared to a similar sequential system.

This paper presents a number of general optimization schemas that can be employed by implementors of parallel non-deterministic systems to keep the overhead incurred for exploiting parallelism low. A system builder can examine his/her design to come up with concrete optimizations based on the schemas proposed in this paper. It is very likely that these optimizations will help in considerably reducing the parallel overhead, and in obtaining a more efficient system. This is indeed how we improved the efficiency of the ACE parallel logic programming system. Reduction of parallel overhead results in improved *sequential efficiency* (performance of the parallel system on one processor) of the system. It should be noted that the objective of parallel execution is to obtain better *absolute performance* and not just better speed-ups. This implies that it is absolutely imperative that the sequential performance of a parallel system be very close to that of the state-of-the-art purely sequential systems. Application of the optimizations based on schemas presented in this paper can take one close to this goal. These optimization schemas are mainly meant for non-deterministic systems and languages (such as parallel

*This research is supported by NSF Grants CCR 96-25358, HRD 96-28450, INT 9415256, and NATO Grant CRG 921318.

AI systems, theorem proving systems and implementation of logic, constraint, and concurrent constraint languages). Our optimization schemas are in the spirit of Lampson's ideas on software system design [5], where he offers general hints for designing efficient application programs. We would like to adopt the same perspective in this presentation. We have developed various parallel systems, and the techniques we are describing have been distilled out of our experiences. We are not claiming these techniques to be

- novel;
- appropriate to every situation;
- formally defined; or,
- guaranteed to always work.

They can be treated as suggestions that a system designer can employ to obtain a more efficient implementation. In most cases we believe (and empirical evidence suggests) that these techniques will lead to greater efficiency, but there may be cases where the cost of applying an optimization may be more than the benefits that accrue from it. It's up to the system implementor to judge whether or not a particular optimization is going to benefit his/her system.

Our aim behind presenting these optimization schemas is to organize the area of parallel implementation optimization by developing a suite of simple and general optimization schemas that implementors can use to develop optimizations specialized to their own implementations. We would like to stress the distinction that we make in this paper between the optimization schemas and the actual optimization techniques themselves. An optimization schema can be viewed as providing general guidelines that form the underpinning of a class of specific optimizations, where each such specific optimization is an implementation technique for improving execution performance. The main contribution of this paper is to present several optimization schemas, and to show how they can be employed to develop new actual optimizations.

The optimization schemas we present have been used for devising actual optimizations for the *ACE system* [13], a high performance parallel Prolog system developed by us. These optimizations have resulted in vast improvement in performance of the ACE system. The concrete performance improvement figures for the ACE system are presented as a testimony to the effectiveness of these schemas. As a result of these optimizations, in many applications the parallel overhead in ACE was reduced to less than 2%, a remarkably small overhead given the enormous complexity of and-or parallel systems The optimizations based on our schemas are intended to be applied at runtime (rather than at compile-time), therefore, the benefits accrued have to be balanced against the cost of applying the optimization. We believe that, in general, the benefits accrued will be more than the cost of applying these optimizations; at least that's what is borne out by our experiments with the ACE system, where the cost of applying the optimization developed is limited to very simple runtime checks. However, for each specific optimization that can be devised from a schema, its designer will have to decide whether or not his/her implementation can benefit from that optimization. It should be noted that because our schemas are for developing runtime optimizations, these optimizations can be triggered every time a situation where they can be applied arises. This is in contrast to applying them at compile-time, where the conditions under which they are to be triggered can only be approximated (e.g., determinacy of goals can be detected at compile-time only in some of the cases, however, at runtime, determinacy of goals is completely known).

In the rest of the paper, we take logic programming systems as representatives of non-deterministic systems. Thus, we present our optimizations schemes and their applications in the context of logic programming, though they apply equally well to parallel implementations of arbitrary non-deterministic systems (such as parallel constraint systems, concurrent constraint systems, parallel tree-search based AI systems, parallel theorem provers, etc.).

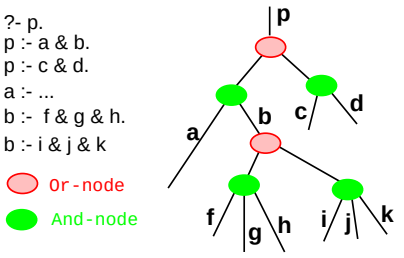


Figure 1. And-Or parallel Tree

2 And-Or Tree

Parallel execution of a logic programming system can be viewed as parallel traversal of an and-or tree. An and-or tree has or-nodes (choice points) and and-nodes. Or-nodes are created when there are multiple ways to solve a goal. An and-node is created when a goal invokes several conjunctive subgoals. The or- and and-nodes represent *sources of parallelism*, i.e. points of the execution where it is possible to fork parallel computations. Figure 1 illustrates an and-or tree for a logic program (`&' in the program stands for parallel conjunction). Note that the and-nodes and the or-nodes may be descendents of each other during parallel execution of the and-or tree. This results in nesting of and-parallelism and or-parallelism within each other, making the management and implementation of parallel execution enormously complex[1]. The complexity is exacerbated by the fact that non-deterministic systems admit backtracking[2], and incorporation of backtracking implies that a computation should be restorable to every point where a choice was made. In the context of and-or parallelism this means saving extra information and allocation of extra data-structures, so that in the event of backtracking, the state of the computation can be restored. Several such extra data-structures are employed in logic programming systems for parallel execution (see Figure 2):

- *Choice point*: allocated whenever a non-deterministic goal is called; it also serves as a source of or-parallel work.
- *Parcall frame*: allocated when a parallel conjunction is called; it serves as a source of and-parallel work.
- *Marker nodes*: allocated to delimit (or mark) the segments of stacks corresponding to goals taken from a parallel conjunction. There are two types of marker nodes, those that

[1] Only very few efficient and/or-parallel implementations of Prolog have been realized.

[2] Note that even in or-parallel systems, backtracking is present, because typically there are far more alternatives than processors available. Thus, multiple branches may be explored by a processor via backtracking.

mark the beginning of a goal (called *input marker*), and those that mark the end (called *end markers*).

• *Sentinel nodes*: allocated to delimit segments of stacks corresponding to an alternative taken from a choicepoint.

These extra data-structures can be quite heavy, and can add considerable overhead to execution. It should be emphasized that not all these data-structures are needed in parallel implementation of deterministic systems; this is because in non-deterministic systems they are required purely for the purpose of recording state so as to restore it in the event of backtracking, and deterministic systems do not backtrack.

2.1 Overheads of Parallel Exploration

Given the and-or tree for a program, its sequential execution amounts to traversing the and-or tree in a pre-determined order. Parallel execution is realized by having different processors concurrently traversing different parts of the and-or tree in a way consistent with the operational semantics of the programming language. By operational semantics we mean that data-flow (e.g., variables' bindings) and control-flow (e.g., input/output operations) dependencies should be respected during parallel execution (similar to loop parallelization of Fortran programs, where flow dependencies have to be preserved). Parallelism allows overlapping of exploration of different parts of the and-or tree. Nevertheless, as mentioned earlier, this does not always translate to an improvement in performance. This happens mainly because of the following reasons:

(1) the tree structure developed during the parallel computation needs to be explicitly maintained, in order to allow for proper management of nondeterminism and backtracking—this requires the use of additional data structures, not needed in sequential execution. Allocation and management of these data structures represents an overhead.

(2) the tree structure of the computation needs to be repeatedly traversed in order to search for multiple alternatives and/or cure eventual failure of goals, and such traversal often requires synchronization between the processors. The tree structure may be traversed more than once because of backtracking, and because idle processors may have to find nodes that have work after a failure takes place or a solution is reported.

2.2 Reducing Overheads in And-Or Tree

So far we have intuitively identified the main sources of overhead in a parallel computation: *(i)* management of additional data structures; and, *(ii)* need to traverse a complex tree structure. Based on this we can now try to identify ways of reducing these overheads.

Traversal of Tree Structure: there are various ways in which we can improve the activity of traversing the complex structure of a parallel computation: *(1)* simplification of the computation's structure so that it can be traversed more efficiently; *(2)* use of the knowledge about the computation (e.g., determinacy) in order to reduce the amount of traversal of the computation tree.

Data Structure Management: since allocating data structures is generally an expensive operation, our aim should be to reduce the number of new data structures created. This can be achieved by: *(1)* reusing existing data structures whenever possible (as long as desired execution behaviour is preserved). *(2)* avoiding allocation of unnecessary structures.

This suggests possible conditions under which we can avoid creation of additional data structures: *(i)* no additional data structures are required for parts of the computation tree which are *potentially* parallel but are actually explored by the same computing agent (i.e., potentially parallel but practically sequential); *(ii)* no additional data structures are required for parts of the computation that will not contribute to the nondeterministic nature of the computation (e.g., deterministic parts of the computation).

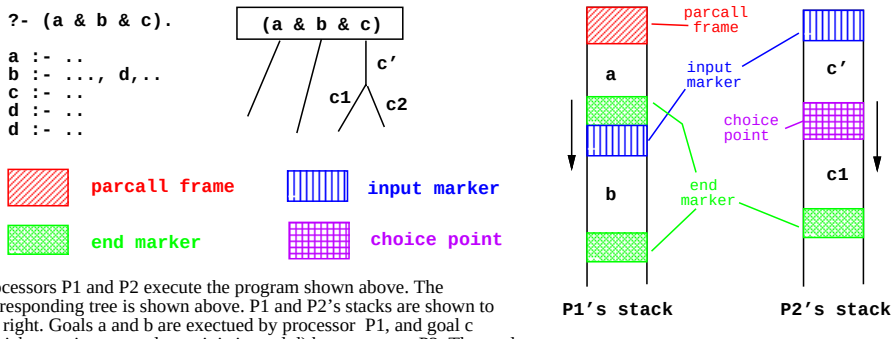
These observations can be concretized into three optimization schemes that are presented in the following sections. Note that the optimization schemes we present are not generally applicable to deterministic languages and systems because deterministic systems do not have to perform the two operations that we seek to eliminate: (i) repeated traversal of (parts of) an and-or tree during backtracking and search for work; and, (ii) allocation of state-saving data-structures to facilitate this traversal. Before describing our optimization schemes we give a brief introduction to the ACE system next, since it is used as the testbed for our optimization techniques.

2.3 The ACE System

ACE is an and-or parallel system for full Prolog developed by the authors [13, 11, 12]. The ACE system has been implemented by extending the SICStus Prolog system. For the purpose of illustrating the optimization schemas we only need to understand the and-parallel component of ACE (called &ACE). &ACE executes *independent* goals in and-parallel, i.e., conjunctive goals that are determined to not influence each other's execution *at runtime* are run in and-parallel. Programs supplied to the systems are annotated in order to make parts of the program that can be safely run in and-parallel explicit (through the use of the "&" parallel conjunction operator; the ',' indicates the usual sequential conjunction). Note that independence is a runtime property. This annotation has to be either done manually by the programmer, or automatically by an abstract-interpretation based parallelizing compiler (the &ACE system uses the latter [9]). Thus, given a clause *p :- q, r, s, t.* in which *r* and *s* are determined to be independent of each other, it would be annotated as *p :- q, (r & s), t.* Because not all conjunctive goals in ACE are executed in parallel, the computation structure created is not exactly similar to the conventional and-or tree. Rather, a parallel conjunction is analogous to the and-node of an and-or tree, while a sequential conjunction is represented as a linear branch (though for the rest of this paper, this distinction is not of great significance).

At runtime, each time a parallel conjunction (also termed *parallel call* or *parcall*) is reached, a descriptor (*parcall frame*) for the conjunction is allocated on the stack (the parcall frame contains one *slot* for each subgoal belonging to the parallel call, where each slot contains information about that specific goal) and the subgoals are made available to the other processors for parallel execution. Each agent that selects a subgoal for execution will initially allocate an *input marker* on its control stack to indicate the beginning of a new execution; analogously, an *end marker* will be allocated to mark the end of the subgoal's execution. These marker are required in order to guide backtracking activity on the parallel call in the case of failure (See Figure 2).

The &ACE system is fully operational, has very high se-



Processors P1 and P2 execute the program shown above. The corresponding tree is shown above. P1 and P2's stacks are shown to the right. Goals a and b are exectued by processor P1, and goal c (which contains a non-deterministic goal d) by processor P2. The goal a does not need an input marker as the parcall frame marks its beginning.

Figure 2. Data-structures for Saving Computation State

quential efficiency, and has shown excellent speed-ups on a variety of benchmarks including some a couple of thousand lines long [13]. The unoptimized &ACE engine incurs an average parallel overhead of 10% to 25% compared to sequential SICStus Prolog.

3 Simplification of the Computation

As mentioned before, one of the main sources of overhead introduced during and-or parallel computation is due to the need to repeatedly traverse a deeply nested tree structure. This traversal is related to both the management of parallel work (e.g., scheduling) and the management of nondeterminism (e.g., searching for multiple solutions). One way to improve the execution performance is to simplify the structure of the computation tree, in order to make its traversal cheaper and faster.

Flattening The Computation Tree the first optimization schema we present is based on "flattening" the computation tree, and can be stated as follows: "*Flatten the tree structure, reducing the levels of nesting whenever possible, preserving the operational semantics.*"

Recall that during parallel execution and-parallelism and or-parallelism may be nested within each other. This complicates the management of parallelism and increases the parallel overhead. The flattening of the tree helps in reducing the levels of nesting of and- and or-parallelism thereby reducing overhead. Of course, the flattening of the computation tree should be done in a way such that program semantics are unaltered (in case of logic programming systems this means that the order in which backtrack literals are chosen is preserved).

The flattening scheme manifests itself in many situations, both in non-deterministic as well as deterministic systems: the tail recursion optimization (and its logic programming counterpart, last call optimization [16]), *unfold* transformation [10], *distributed last call optimization* [15], flattening of nested data-parallel calls [3]. merging of nested *barriers* [14], etc. We discuss two optimizations that result from the application of flattening in the ACE system. These optimizations are termed *last parallel call optimization* [12] and *last alternative optimization* [6] respectively.

3.1 Flattening: Last Parallel Call Optimization

The Last Parallel Call Optimization (LPCO) is very similar in spirit to the last call optimization found in sequential implementations of Prolog [16]. The intent of the last parallel call optimization is to merge, whenever possible, distinct parallel conjunctions, which are nested one inside the other. The main aim of this optimization is to reduce the depth of nesting of parallel calls.

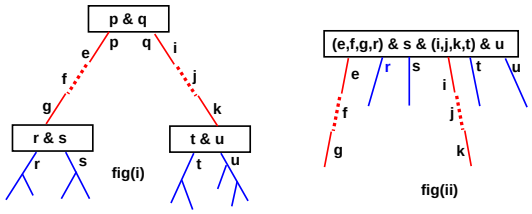


Figure 3. Reusing Parcall Frames

LPCO cannot be applied every time a nested and-parallel call arises inside another and-parallel call, since doing so in an unrestricted way may change the operational semantics when backtracking over parcalls. However, under certain restrictions, the operational semantics is preserved. To illustrate LPCO in the most general case, consider a parallel conjunction of the form *(p & q)*, where (See Figure 3(i)): *p :- e, f, g, (r & s)* and *q :- i, j, k, (t & u)*. LPCO will apply to *p* (resp. *q*) if: *(i)* there is only one (remaining) matching clause for *p* (resp. *q*), i.e., *p* (resp. *q*) is determinate; *(ii)* all goals preceding the parallel conjunction in the clause for *p* (resp. *q*) are determinate; *(iii)* the parallel call *(r & s)* (resp. *(t & u)*) is at the end of the clause for *p* (resp. *q*). The benefits of LPCO become even more evident when a failure causes backtracking over a parallel call: the expensive traversal of the tree structure in search of a new alternative is replaced by a simple linear scan of the subgoals descriptors inside a single parcall frame. One could argue that the improved scheme described above can be accomplished simply through compile time transformations. However, in many cases this may not be possible. For example, if *p* and *q* are dynamic predicates or, more simply, if there is insufficient static information to detect the determinacy of *p* and *q*, then the compile-time analysis will not be able to trigger the application of the optimization. Also, for many programs the number of parallel conjunctions that can be combined into one will only be determined at run-time. For example, consider the following recursive clause:

```
process_list([H|T],[Hout|Tout]) :-
   process(H,Hout) &
process_list(T,Tout).
```

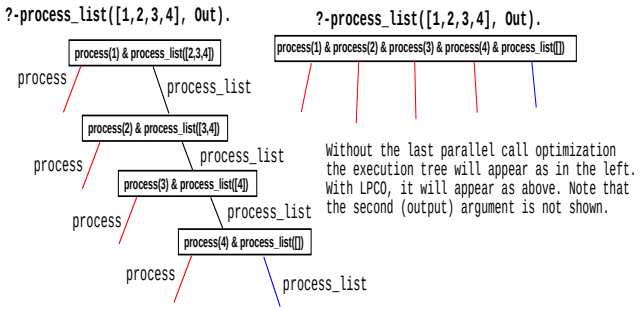


Figure 4. Reuse for Recursive Programs

`process_list([],[]).`

In this case, compile time transformations cannot unfold the program to eliminate nesting of parcall frames because it will depend on the length of the input list. However, using our runtime technique, given that the goal *process_list* is determinate, nesting of parcall frames can be completely eliminated (Figure 4). As a result of the absence of nesting of parcall frames, if the *process* goal fails for some element of the list, then the whole conjunction will fail in one single step (in unoptimized execution, failure has to be sequentially propagated from the bottommost parcall frames to the ones higher up, which incurs a greater overhead).

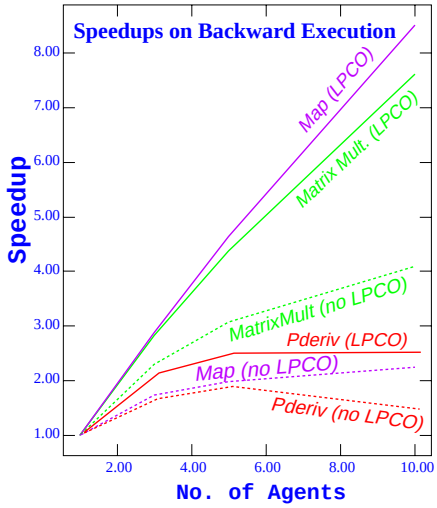


Figure 5. Backtracking: Speedup Curves

On applying the LPCO in a situation such as the one shown in Figure 4, control parallelism that is deeply nested transforms itself into data-parallelism resulting in improved efficiency. In fact, our data-parallel version of Prolog [11], based on LPCO is comparable in performance to dedicated data-and parallel Prolog systems such as Reform Prolog [2].

The results[3] of applying LPCO to the ACE system are shown in Tables 1 and 2 (times are in sec.).[4] Table 1 shows the performance improvement during forward execution (where LPCO results in only marginal improvement if at all), while Table 2 shows performance improvement during backtracking, where the performance due to reduction in levels of nesting are significant. The performance improvement in terms of speed-up curves are shown in Figure 5. The most noticeable is the map benchmark: without the optimization almost no speedup can be observed (due to the heavy overhead of backtracking), while using the optimization an almost linear speedup can be obtained. Use of LPCO produces also a considerable improvement in memory consumption: experiments have shown that the usage of control stack can be decreased by almost 50% [12].

[3] Execution times are on the Sequent Symmetry Multiprocessor.

[4] It should be noted that different sets of benchmarks have been used for illustrating the gains obtained from the different optimizations. This is because, in general, results for only those benchmarks have been presented for a particular optimization that have shown substantial improvement with that optimization. Improvements have been observed almost for all benchmarks for each optimization presented.

3.2 Flattening: Last Alternative Optimization

The Last Alternative Optimization (LAO) is the or-parallel dual of LPCO. We illustrate the LAO 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 `[1,2,3,4]`, 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,[1,2,3,4]),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).
```

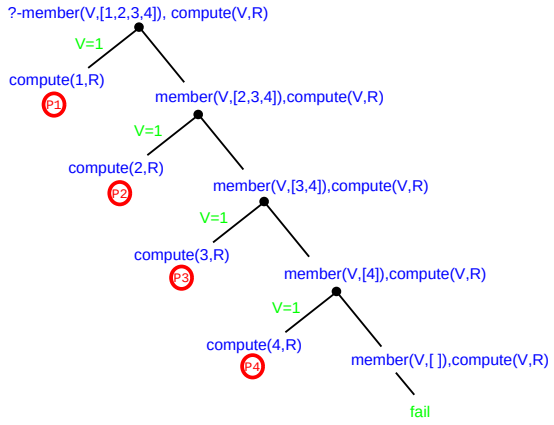


Figure 6. Search tree for member

The or-parallel tree structure created is shown in figure 6. Note that the intent in the program is to fire off a `compute` goal for every element in the list `[1,2,3,4]`. LAO simplifies the tree structure as follows: when the last alternative in a choice-point B_1, further creates a choice-point B_2, then there is no need to allocate this new choice-point, rather B_1 can be *updated* with the information that will be stored in B_2. As a result, in the member example all alternatives are clubbed (Figure 7) at one choice point from where they can all be picked by processors with less traversal.

Table 3 presents the performance improvements due to LAO. On one processor, LAO suffers performance degradation, which is mainly due to certain characteristics of

Benchmark executed	Number of Processors 1	3	5	10
map[2]	7.14/ 6.39 (11%)	2.51 / 2.32 (8%)	1.99 / 1.48 (26%)	1.91 / 1.48 (23%)
occur(5)	3.65/3.15 (14%)	1.25/1.02 (18%)	.75/.64 (15%)	.43/.35 (19%)

Table 1. Savings in Execution time (forward execution only)

Benchmark executed	Number of Processors 1	3	5	10
matrix	6.30/5.36 (15%)	2.73/ 1.90 (30%)	2.05/ 1.22 (40%)	1.54/ .70 (54%)
pderiv	9.49/ 5.61 (41%)	5.88/ 2.75 (53%)	5.19/ 2.34 (55%)	6.67/ 2.342 (65%)
map[1]	24.21/ 14.98 (38%)	14.01/ 5.20 (63%)	12.24/ 3.23 (74%)	10.73/ 1.76 (84%)
annotator	3.94/ 3.86 (2%)	1.35/1.34 (1%)	.88/ .87 (1%)	.49 / .47 (4%)

Table 2. Exec. Time in sec. (LPCO with Backward Exec.); % improvement in parentheses

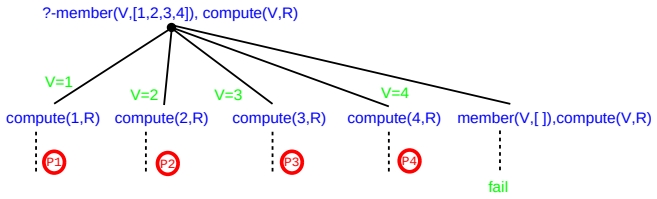


Figure 7. Search tree for member w/ LAO

the MUSE implementation on which the implementation of ACE is based, rather than due to LAO itself. A more careful implementation of MUSE should solve this problem.

The LAO can also be used for parallelizing and optimizing constraint languages (e.g., CHIP).

4 Avoidance of Unnecessary Operations

A general model to support parallel execution is usually designed to tackle the worst case situations. This may result in overheads even when the worst case is not encountered. This can occur in two situations: paying the price for supporting nondeterminism in presence of deterministic computations, and paying the price for supporting parallelism in presence of sequentially executed pieces of computation. Avoiding overhead in these situations is not a straightforward task, because knowledge about properties of the computation (e.g., determinacy, sequentiality) can only be obtained *a posteriori*, after execution. There are two ways to avoid these worst-case induced overheads, and they can be concretized as optimization schemas, one based on *procrastination* of overheads, and one based on *sequentialization* of computations.

Procrastination of Overheads The optimization schema can be stated as follows: "*The execution of an operation that constitutes an overhead should be delayed until its effects are needed by the rest of the computation.*" Here again we assume that delaying will not alter the operational semantics of the language/system. The intuition is that certain operations may be delayed indefinitely, i.e., their effect may never be needed by the computation. The idea of procrastination has been repeatedly utilized in the design of Warren Abstract Machine (WAM) [17] for efficient execution of Prolog programs (e.g., space for caller goal's environment is allocated by the callee clause). Another example of the procrastination schema is the *shallow backtracking* optimization of M. Carlsson [4]. Lazy evaluation of functional languages can also be regarded as an application of the procrastination schema.

Sequentialization of Parallel Operations the idea of sequentializing parallel operations 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.*" This schema has been implicitly used in various parallel systems. For example, many or-parallel implementations of Prolog (like Muse [1] and Aurora [8]) allow the sequential exploitation of consecutive alternatives from a parallel choice point with minimal overhead (using almost standard backtracking). The optimization in Aurora and Muse used to accomplish this, divides the or-parallel tree into *public* and *private* parts, and can be seen as an instance of the sequentialization scheme. When a processor is in the private part of the search tree, execution is exactly as in a sequential Prolog system. Granularity control in parallel systems can also be seen as an application of this idea. Next we illustrate the application of procrastination and sequentialization schemas to the ACE system.

4.1 Procrastination: Shallow Parallelism Opt.

During and-parallel execution a processor can pick up a goal for execution from other processors once it becomes idle. The general model of &ACE requires at this stage the allocation of a data structure on the stack, called a *(input) marker*. This marker is used to indicate the point at which the execution of a subgoal was initiated, partition the stack in *sections* (one for each parallel subgoal), and maintain the logical connections between a parallel subgoal and its parent computation. The same considerations need to be applied when a subgoal is completed—a marker (*end marker*) needs to be allocated to indicate the completion of a subgoal and the end of the corresponding stack section. The presence of these section markers is fundamental in order to enforce the proper behaviour during backtracking—encountering an input marker indicates that a subgoal has been completely backtracked over and backtracking needs to proceed in the logically preceding subgoal, while encountering an end marker indicates that we are backtracking into a parallel call. The expense incurred in allocating these markers is considerable, since each marker stores various

Program	Number of Processors				
	1	2	4	8	10
Queen[1]	3689/3889 (-5%)	2939/2129 (28%)	1959/1159 (41%)	1910/730 (62%)	1909/629 (67%)
Queen[2]	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 3. Improvements using LAO (unoptimized/optimized); % improvement is shown in parenthesis

Benchmark executed	Number of Processors			
	1	3	5	10
matrix_mult	5.59/5.2 (13%)	1.9/1.7 (11%)	1.1/1.0 (9%)	.57/.53 (7%)
takeuchi	2.4/1.8 (25%)	.83/.58 (30%)	.52/.36 (31%)	.25/.20 (20%)
hanoi	2.2/1.6 (27%)	.76/.55 (28%)	.47/.33 (30%)	.23/.18 (22%)
occur	3.6/3.1 (14%)	1.2/1.0 (17%)	.75/.66 (12%)	.43/.37 (14%)
bt_cluster	1.4/1.3 (7%)	.52/.48 (8%)	.34/.31 (9%)	.20/.18 (10%)
annotator	1.6/1.4 (13%)	.55/.47 (15%)	.39/.32 (18%)	.21/.18 (14%)

Table 4. Shallow Parallelism: Unoptim./Optim. Exec. times (sec.) (% improv. shown in parenthesis)

information and the number of markers ($O(2 \times n)$ where n is the number of parallel subgoals) can be very high. However, if it is known that a subgoal is deterministic, then the presence of markers is *unnecessary* since there is no need to backtrack inside these subgoals, as they do not have any untried alternatives. Thus, backtracking over deterministic goals is a pure overhead. Given a parallel conjunction

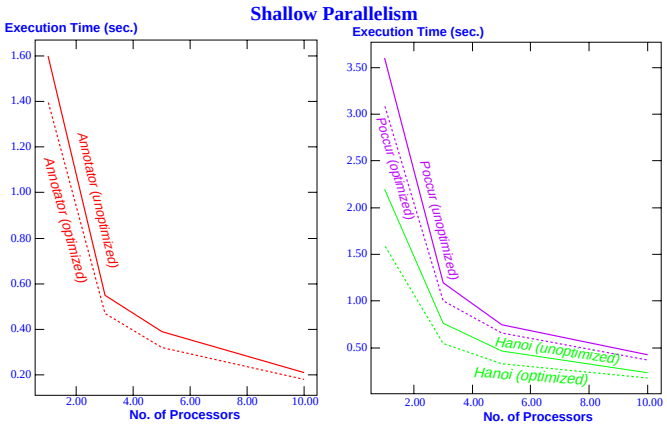


Figure 8. Exec. time w/ Shallow Parallelism

(g_1 & g_2 & g_3) if it is known that g_2 is deterministic then backtracking should proceed directly from g_3 to g_1. We only have to make sure that every binding trailed during the execution of g_2 is untrailed (undone) before backtracking moves into g_1. This implies that actually there is no need to allocate the input marker node and the end marker node for goal g_2 during forward execution—and this applies to every determinate and-parallel subgoal. Allocation of marker, as described above, can be avoided only if the deterministic nature of the subgoal is known. Determinacy information is not available *a priori* (unless some form of compile-time analysis is applied, which will discover some of the cases), therefore the question is how do we avoid the allocation of the input marker. Here the idea of procrastination of overheads comes to the rescue: the allocation of an input marker is procrastinated until a choice point is to be created or until the whole subgoal reaches termination. In the first case the input marker is created before allocating the choice point; in the second case, since the end of the computation is reached without any choice point (i.e., it is a deterministic subgoal), the allocation of both input and end markers can be simply avoided. The only additional operation required is to keep track (by using the slot) of the trail section used during the execution of the deterministic subgoal. This information will be needed for untrailing later during backtracking. We term this optimization, where we avoid allocation of input and end marker nodes, *shallow parallelism optimization*.

The shallow parallelism optimization has been incorporated in the &ACE system. The results obtained are extremely good. An improvement of 5% to 25% over unoptimized ACE implementation is obtained due to this optimization alone. In table 4 the execution times and percentage improvement obtained on some common benchmarks, all creating quite large and-or computation trees, are listed. The optimized and unoptimized speedup curves are shown in figure 8.

4.2 Sequentiality: Processor Determinacy Opt.

The aim of a parallel system is to exploit the highest possible amount of parallelism that will lead to improved execution performance. However, the amount of parallelism present is usually greater than the actual amount of computing resources available—which results in the same processor successively executing *multiple units* of parallel work, e.g. different subgoals of the same parallel execution. Thus, we get to a situation in which two potentially parallel pieces of computation are executed sequentially. The interesting situation occurs when the two units of work are actually executed in the same order in which they would be executed

Goals executed	Number of Processors			
	1	3	5	10
matrix_mult(30)	5598/5207 (8%)	1954/1765 (11%)	1145/1067 (7%)	573/536 (7%)
quick_sort(10)	1882/1503 (25%)	778/621 (25%)	548/443 (23%)	442/367 (20%)
takeuchi(14)	2366/1632 (45%)	832/600 (39%)	521/388 (34%)	252/200 (26%)
poccur(5)	3651/3104 (15%)	1255/1061 (18%)	759/649 (17%)	430/353 (22%)
bt_cluster	1461/1330 (10%)	528/482 (10%)	345/294 (17%)	202/165 (22%)
annotator(5)	1615/1298 (24%)	556/454 (23%)	392/302 (30%)	213/171 (25%)

Table 5. Unoptimized/Optimized Execution times in msec (% improvement is shown in parenthesis)

during a purely sequential computation; in such a case all the additional operations performed, related to the management of parallel execution, represent pure overhead. The sequentialization idea can be applied in such a situation to reduce this overhead. We call the resulting optimization the *Processor Determinacy Optimization (PDO)* because it comes into affect after the processor that is going to execute a given goal is determined. The application of the PDO is also *a posteriori*. Once a situation in which the optimization can be applied is detected, i.e. the scheduler returns (or, it manages to select) a subgoal which, considering sequential semantics, immediately follows the one just completed, we can avoid most of the overhead associated with the second subgoal. This saving is obtained by simply avoiding allocating any marker between the two subgoals and—in general—treating them as a single, contiguous piece of computation. Thus, given *(a & b & c)* if processor executing *a* picks up *b* after finishing then the effect of the PDO will be as if the original parallel conjunction was *((a, b) & c)*. There are several advantages in applying PDO: (i) memory consumption as well as execution time is reduced since allocation of markers between consecutive subgoals executed on the same processor is avoided; and, (ii) execution time during backward execution is also reduced, since backtracking on the two subgoals flows directly without the need of performing the various actions associated with backtracking over markers. Implementation of PDO is quite simple and requires minimal changes to the architecture: an additional check on exit from the scheduler is needed to verify if the new subgoal can be merged with the one previously executed. Table 5 shows the improvements in time obtained for ACE.

5 Conclusions

In this paper, we presented three general optimization schemas that can be used for designing optimizations specific to a parallel non-deterministic system. While the resulting optimizations may not *always* produce an improvement, empirical evidence suggests that they can help in improving execution efficiency in most cases. The optimization schemas, based on *flattening of the computation tree*, the *procrastination of overheads*, and the *sequentialization of computation* were illustrated by applying them to ACE, a high-performance parallel logic programming system. Application of optimizations based on these schemas to ACE resulted in the parallel overhead being reduced, on average, to less than 5% (less than 2% for many programs). The improvement in parallel execution efficiency was concretely demonstrated by presenting performance figures collected on a Sequent Symmetry Multiprocessor.[5] While these schemas have been illustrated on the ACE system, they are fairly general, and can be readily applied to other nondeterministic systems such as parallel theorem proving systems, parallel rule based and AI systems, and parallel implementations of constraint and concurrent constraint languages.

References

[1] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model. In *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] G.E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, 1990.

[4] M. Carlsson. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In *Int. Conf. on Logic Progr.*, MIT Press, 1989.

[5] B. Lampson et al. Hints for Computer System Design. In *ACM Symp. on Operating Systems Principle*. ACM, 1983.

[6] G. Gupta and E. Pontelli. Last Alternative Optimization and Parallel Implementation of CLP(FD). Tech. Rep. 9606, New Mexico State Univ., 1996.

[7] P. Van Hentenryck. *Constraint Handling in Prolog*. MIT Press, 1988.

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

[9] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables. In *Int. Conf. on Logic Progr.*, MIT Press, 1991.

[10] A. Pettorossi and M. Proietti. Transformation of Logic Programs *Journal of Logic Programming*, 19/20, 1994.

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

[12] 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.

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

[14] V. Ramakrishnan, I.D. Scherson, and R. Subramanian. Efficient Techniques for Fast Nested Barrier Synchronization. In *ACM Symp. on Parallel Algorithms and Architectures*, 1995.

[15] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proc. NACLP'89*, MIT Press, 1989.

[16] D. H. D. Warren. An Improved Prolog Implementation Which Optimises Tail Recursion. Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, 1980.

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

[5] Similar improvements were observed on a Sun Sparc 10.