Practical Optimization Schemas for Parallel Implementation of Non-deterministic Languages and Systems

Gopal Gupta and Enrico Pontelli

Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
{gupta,epontell}@cs.nmsu.edu

Naive parallel implementation of non-deterministic systems (such as a theorem proving system) and languages (such as logic, constraint, or concurrent constraint languages) 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 different parallel logic programming systems. Performance data presented shows that considerable improvement in execution efficiency can be achieved.

Additional Key Words and Phrases: Non-deterministic languages, parallel processing, optimization principles

1. INTRODUCTION

Non-determinism arises in many areas of computer science. By non-determinism we intend 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 [Van Hentenryck 1988]), concurrent constraint languages (e.g., AKL [Saraswat 1989]), and rule based languages (e.g., OPS5 [Brownston 1985]) 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; given a subgoal, there may be more rules that can be used to reduce it, and these are explored in parallel.

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 non-determinism, while and-parallelism is the more "tradi-

This research is supported by NSF grants CCR 96-25358, CDA-9729848, EIA 98-10732, HRD 98-00209, HRD 96-28450, and INT 95-15256, and by a grant from the Fullbright Foundation.
This paper is an extended version of the paper "Optimization Principles for Parallel Implementation of Non-deterministic Languages and Systems" appeared in the proceedings of the 1997 International Parallel Processing Symposium, IEEE Computer Society.

tional" form of parallelism, commonly exploited in standard (deterministic) programming languages (e.g., Fortran [Zima and Chapman 1991]).

Solution of problems with non-determinism has been abstractly represented using *and/or trees*. *Or-nodes* express the existence of alternative ways of proceeding with the computation, while *and-nodes* denotes the decomposition of the current computation in sub-computations. 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. The parallelism present in these frameworks is typically very irregular and unpredictable; for this reason, parallel implementations of non-deterministic languages typically rely on *dynamic scheduling*. Thus, most of the work for partitioning and managing parallel tasks is performed during run-time. These duties are absent from a sequential execution and represent *parallel overhead*. 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 as well as other parallel logic programming implementations (e.g., MUSE and Aurora). Reduction of parallel overhead results in improved *sequential efficiency*—performance of the parallel system on one processor—and *absolute efficiency*—overall execution time—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 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 [Lampson 1983], where general hints for designing efficient application programs are offered. 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.

The optimization schemas we present are quite intuitive, and can be found scattered in the literature as they have been applied in the past in implementations of non-deterministic systems to optimize execution. However, ours is the first attempt at formulating them in their general form in a single framework, and showing through an actual example how they can be *systematically* applied to improve execution performance. The optimizations inspired by these general schemas are also applicable to deterministic systems and languages. However, the benefits brought about by them are not as high as those for non-deterministic systems, as demonstrated later. Non-deterministic systems and languages incur a much higher overhead in their implementations and many of these overheads do not even exist in implementations of deterministic systems.

Our aim behind presenting these optimization schemas is to develop 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. We also show how many of the optimizations proposed in the past for various other parallel non-deterministic systems can be explained in terms of these schemas. We hope that this work will induce other researchers to discover more such optimization schemas and techniques.

The optimization schemas we present have been used for devising actual optimizations for the *ACE system* [Gupta et al. 1994; Pontelli et al. 1995; Pontelli and Gupta 1997b], a high performance parallel Prolog system. 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. The same optimizations have been applied to other parallel logic programming implementations (e.g., &-Prolog [Hermenegildo and Greene 1990; Hermenegildo 1986], Aurora [Lusk et. al. 1990], and MUSE [Ali and Karlsson 1990]) with results consistent to those observed for ACE. As a result of these optimizations, in many applications the parallel overhead was reduced to less than 2%, a remarkably small overhead given the enormous complexity of and-or parallel systems—overhead in parallel Prolog systems can be as much as 200% to 300%. The optimizations based on our schemas are mostly 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 obtained 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 and inexpensive 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 triggering conditions can only be approximated. For example, determinacy of goals can be detected at compile-time only in some of the cases, while at runtime more precise information about the determinacy of a goal is typically available. Availability of statically determined information [Warren et al. 1988] regarding the execution (e.g., types, determinacy, granularity, variable instantiations [Pfenning 1992; Debray and Warren 1986; Pontelli et al. 1997; Debray et al. 1990; Debray and Warren 1988]), however, can lead to other instances of our optimization schemas, further extending their applicability [Pontelli et al. 1996].

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 other non-deterministic systems (such as parallel constraint systems, concurrent constraint systems, parallel tree-search based AI systems, parallel theorem provers, etc.).

The paper is organized as follows. In section 2 we provide a brief introduction to parallel logic programming, with particular focus on the models used in the ACE systems (used for the benchmarks presented in this work). In section 6 the concept of and-or tree is explored and sources of potential overhead are analyzed. Section 4 and 5 present the three optimization schemas we propose and their application to parallel logic programming.

2. AND- AND OR-PARALLELISM IN ACE

ACE is an and-or parallel system for full Prolog developed by the authors [Gupta et al. 1994; Pontelli et al. 1995; Pontelli and Gupta 1995; Pontelli et al. 1995]. The ACE system has been implemented by extending the SICStus Prolog system [Carlsson 1988]. The ACE system is capable of extracting both and-parallelism and or-parallelism from executions of standard Prolog programs.

In this section we present a brief introduction to logic programming and analyze the issues in exploiting and-parallelism and or-parallelism. The solutions proposed in the ACE systems are described—which represent the current state-of-the-art in the field.

2.1 Logic Programming

Programs in logic programming are composed by collections of rules, called *clauses*. We are assuming the following notation:

Head :- Body.

is a program rule; Head is an atomic formula while Body is a conjunction of atomic formulae. The rule, intuitively, describes the logical implication

$$\forall(Body \rightarrow Head)$$

Program execution originates from a *goal*, i.e., a conjunction of atom formulae. A goal is represented as

?- Goal.

and the purpose of the execution is to show that the conjunction Goal is a logical consequence of the rules in the program.

Execution in logic programming is based on the notion of *resolution*. During resolution, a component of the goal is selected and reduced using one of the program rules. The computation terminates when no elements to be proved are left in the goal.

We will use the following notation to describe a computation based on resolution (let us denote with $\bar{G}$ a conjunction of formulae $G_1, \ldots, G_n$ and with $L : L'$ the concatenation of the two lists L and L'):

- the program is composed by a sequence of rules $r_1, r_2, \ldots, r_m$
- the state of the computation is represented by

$$\langle \mathsf{Goal}; Rules; Solution\rangle$$

  where Goal is the current goal (conjunction of atomic formulae), *Rules* is a list of lists of program rules, describing the unexplored alternatives for each step performed, and *Solution* is the solution produced so far. The solution is expressed as a substitution which assigns values to the variables involved in the computation. The *initial state* for a goal ?- G is

$$\langle \mathsf{G}; [\,]; \epsilon\rangle$$

  where [ ] is the empty list and ϵ is the empty substitution.
- the computation can be decribed as a state transformation process; i.e., the computation can be seen as a sequence of states $S_0 \vdash S_1 \vdash \ldots$, where S_0 is the initial state and each state is obtained from the preceding one using a derivation step ($\vdash$). Given a state S_z of the form

$$\langle p, \bar{G}_1, \bar{G}_2; L; \theta\rangle$$

  the operational semantics of the language determines the next state as follows:

—*Standard Derivation Step:* if the (ordered sequence of) clauses $r_1, \ldots, r_k$ contains all the clauses H_i :- Body_i such that the equation $\mathsf{H}_i = p^\theta$ admits a solution (σ_i), then

$$\langle p, \bar{G}_1, \bar{G}_2; L; \theta \rangle \ \vdash \ \langle \mathsf{Body}_1, \bar{G}_1, \bar{G}_2; L : [[r_2, \ldots, r_k]]; \theta \circ \sigma_1 \rangle$$

where $\circ$ is the usual function composition operation.

—*Backtracking Step:* if there are no rules in the program which allow to reduce p, and if there exists $1 \leq b < z$ such that

- L is the list $L' \,:\, [[r_2, \ldots, r_k]]$
- S_b has the form $\langle q, \bar{G}_3, \bar{G}_2; L'; \theta_1 \rangle$
- S_{b+1} has the form $\langle \mathsf{Body}_1, \bar{G}_3, \bar{G}_2; L; \theta_1 \circ \theta_2 \rangle$
- for $b+1 \leq i \leq z$ the second component of S_i is always of the form $L : L''$ (with L'' possibly empty)
- r_2 is of the form Head_2 :- Body_2 and the equation $\mathsf{Head}_2 = q^{\theta_1}$ has solution θ_2

Then the derivation step is defined as:

$$\langle p, \bar{G}_1, \bar{G}_2; L; \theta \rangle \ \vdash \ \langle \mathsf{Body}_2, \bar{G}_3, \bar{G}_2; L' \,:\, [[r_3, \ldots, r_k]]; \theta_1 \circ \theta_3 \rangle$$

—the notation $\vdash^*$ will be used to denote the reflexive and transitive closure of $\vdash$

2.2 And-Parallelism

The and-parallel component [Pontelli et al. 1995], 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. The engine has also been extended to support efficient execution of *dependent and-parallelism* [Pontelli and Gupta 1997a]. Programs supplied to the systems are automatically annotated [Hermenegildo et al. 1991; Muthukumar et al. 1997; Pontelli et al. 1997] 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 [DeGroot 1984; Hermenegildo and Greene 1990]; the ',' indicates the usual sequential conjunction). Note that independence is a runtime property. This annotation has to be done either manually by the programmer, or automatically by an abstract-interpretation based parallelizing compiler (the ACE system uses the latter, using an extension of the CIAO parallelizing compiler [Hermenegildo 1994; Muthukumar and Hermenegildo 1991; Bueno et al. 1994a; Bueno et al. 1994b; Pontelli et al. 1997]). 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, and the subgoals are made available to the other processors for parallel execution. The parcall frame contains one *slot* for each subgoal belonging to the parallel call, where each slot contains information about that specific goal—i.e., the slot is the local descriptor of each goal belonging to a parallel call. 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 the backtracking activity on the parallel call in the case of failure (see Figure 7). The various data structures are shown in Figure 1.

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 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

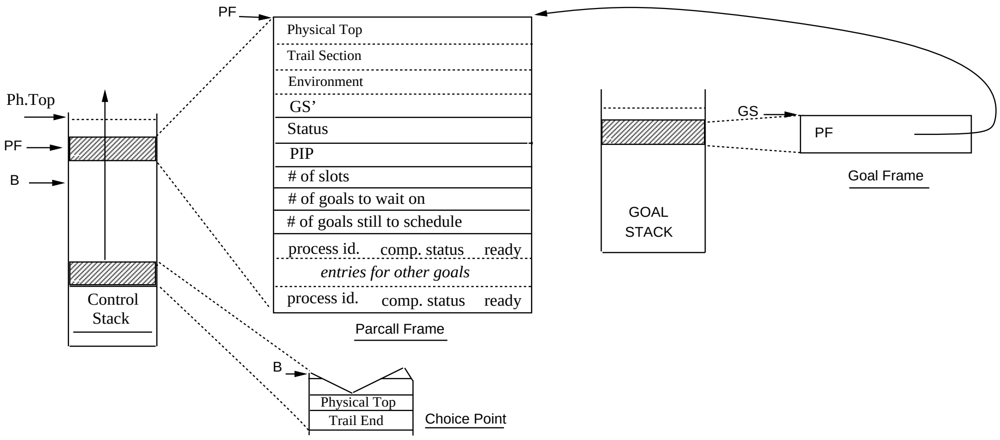


Fig. 1. Data-structures needed for and-parallelism

is such that all solutions are reported. One such backtracking semantics has been proposed by Hermenegildo and Nasr [Hermenegildo and Nasr 1986]: consider the subgoals shown below, where ',' is used between sequential subgoals (because of data-dependencies) and '&' for parallel subgoals (no data-dependencies):

```
a, b, (c & d & e), g, h.
```

Assuming that all subgoals can unify with more than one rule, there are several possible cases depending on which subgoal fails: If subgoal `a` or `b` fails, sequential backtracking ("Prolog-like") occurs, as usual. Since `c`, `d`, and `e` are mutually independent, if either one of them fails before finding any solution, then a limited form of intelligent backtracking can be applied and backtracking can be forced to continue directly from `b`—but see further below. If `g` fails, backtracking must proceed to the right-most choice point within the parallel subgoals `c & d & e`, and recompute all goals to the right of this choice point (following Prolog semantics). If `e` were the rightmost choice point and `e` should subsequently fail, backtracking would proceed to `d`, and, if necessary, to `c`. Thus, backtracking within a set of and-parallel subgoals occurs only if initiated by a failure from outside these goals, i.e., "from the right" (also known as *outside backtracking*). If initiated from within, backtracking proceeds outside all these goals, i.e., "to the left" (also known as *inside backtracking*). When backtracking is initiated from outside, once a choice-point is found in a subgoal p, an untried alternative is picked from it and then all the subgoals to the right of p in the parallel conjunction are restarted (in parallel).
Independent and-parallelism with the backtracking semantics described above has been implemented quite efficiently by the authors in the ACE system [Pontelli et al. 1995]. The ACE system is built upon the SICStus (sequential) Prolog Engine which ensures good performance in the single processor case. ACE implementation itself is inspired by the RAPWAM [Hermenegildo 1986; Hermenegildo and Greene 1990][1]. The RAPWAM model has been generalized in ACE to provide a more efficient memory management scheme [Pontelli et al. 1995] and faster implementation of backtracking [Pontelli et al. 1996; Pontelli and Gupta 1998]. The ACE system has shown remarkable results on a variety of benchmarks. On most of the commonly used benchmarks the system has shown excellent execution times and speedups—figure 2 shows the speedups obtained on some of these benchmarks. ACE has been successfully applied to execute in parallel very large real-life Prolog applications (programs over 35,000 lines long have been successfully parallelized and executed). For further details regarding the structure of ACE and its performance the reader

[1]Other implementations based on the principles of RAPWAM have also been proposed in the past, like &-Prolog [Hermenegildo and Greene 1990] and DDAS [Shen 1992].

is referred to [Gupta et al. 1994; Pontelli et al. 1995; Pontelli et al. 1996].

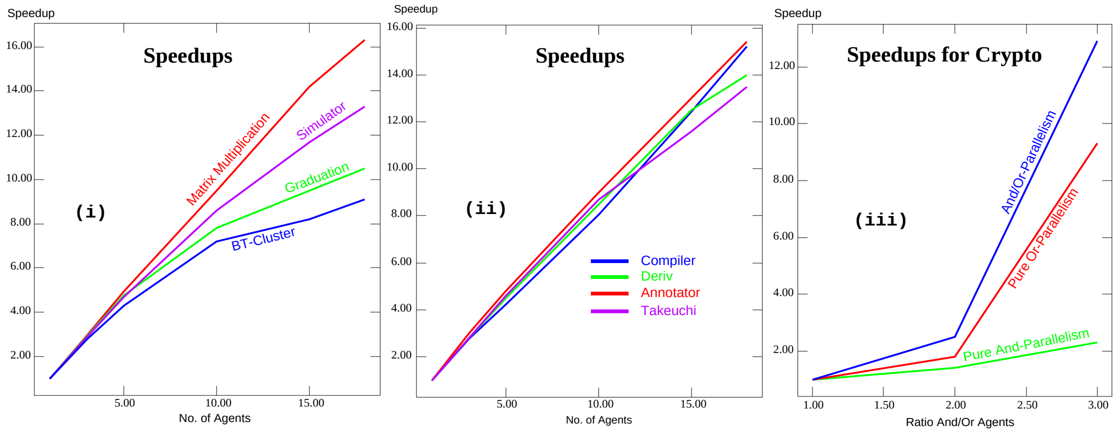


Fig. 2. And-parallel speedups of ACE

2.3 Or-Parallelism

The or-parallel component of ACE is based on a generalization of the *stack-copying* approach, (originally developed for the MUSE or-parallel system [Ali and Karlsson 1990]) for representing *multiple environments* that arise during or-parallelism. In traditional sequential implementations only one branch of the or-parallel tree resides on the control stack at any given time. As a result, bindings for conditional variables can be stored in-place. These bindings are undone on backtracking with the help of the *trail stack*, so that if the next branch also produces a binding for the conditional variable then it can be safely stored. As indicated in figure 3, in the presence of or-parallelism alternative executions arising from a choice point may be active simultaneously and may potentially generate different bindings for variable allocated in environments created before the branching point (the choice point where or-parallelism originated). Thus, multiple environments can exist at runtime, and these have to be carefully maintained so that each branch sees only the bindings that it created.

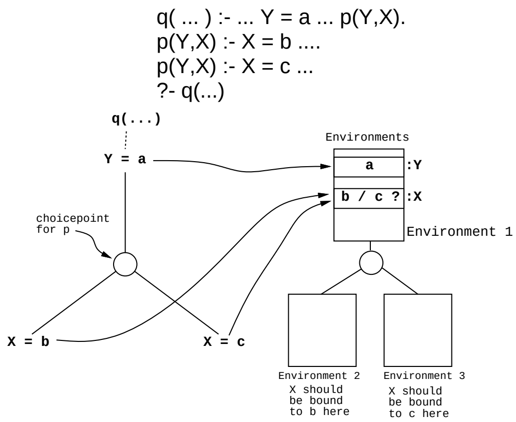


Fig. 3. Environment Representation Problem

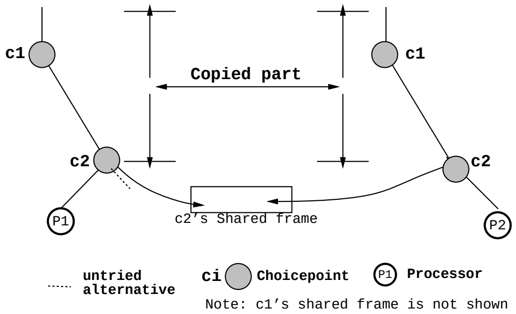


Fig. 4. Stack Copying management of choice points

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). Thus, in Figure 4, if processor `P2` selects the second alternative from choice point `c2`, then before starting execution it will copy all choice points and environments from the root node up to choice point `c2` on its local stack[2]. The existence of local copies of the "shared" environments allows to overcome the previously described binding management problem, since each processor will have a local space to store the locally generated binding.

In the stack copying model the whole state of computation is copied; this requires copying not only the environments created before the choice point, but also the choice points themselves (and other data areas, e.g. the trail stack), in order to allow each processor to perform its execution as independently as possible from the others. However, choice points hold control information and, thus, need to be shared data-structures. 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 (see figure 4).

The process of selecting a choice point and copying the computation state up to such choice point (and creating the needed shared frames for each copied choice point) is termed a *sharing operation* [Ali and Karlsson 1990].

Figure 5 shows the speedups achieved by ACE on some or-parallel benchmarks.

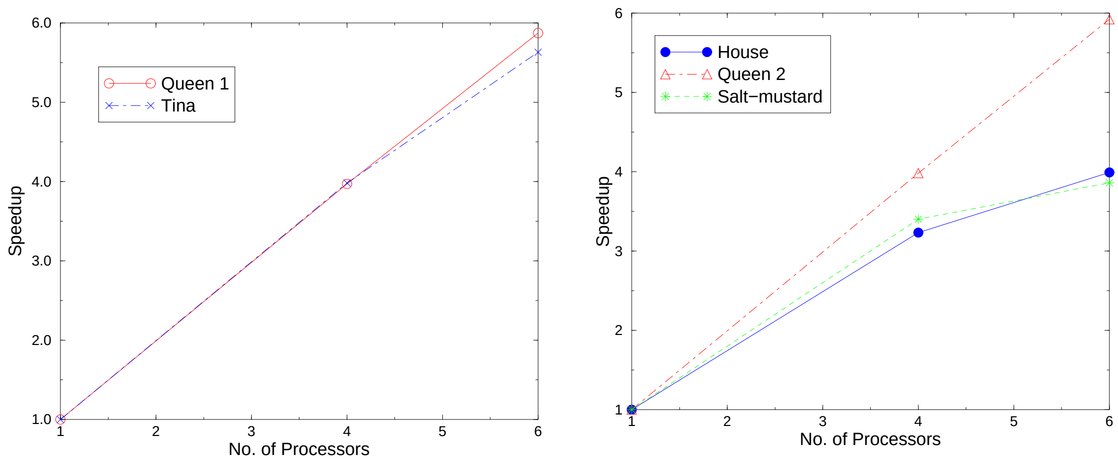


Fig. 5. Or-parallel speedups of ACE

3. AND-OR TREE

Parallel execution of a logic programming system can be viewed as the *parallel traversal/construction* of an *and-or tree*. An and-or tree is a tree structure containing two kind of nodes, or-nodes and and-nodes. *Or-nodes* are created when there are multiple ways to solve a goal. In logic programming terminology or-nodes are termed choice points. 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 6 illustrates an and-or tree for a logic program.

[2] 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.

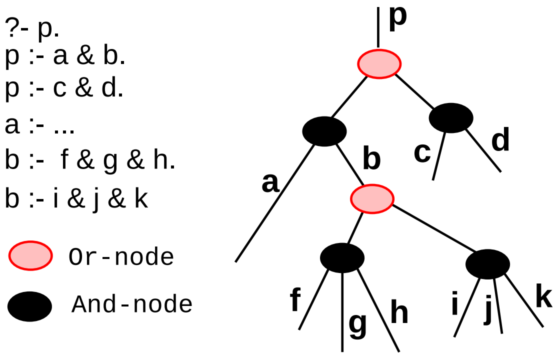


Fig. 6. And-Or parallel Tree

Note that the and-nodes and the or-nodes may be descendent of each others 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[3]. This complexity originates from the fact that the different kind of nodes have to be treated differently in order to guarantee the correct semantics of the execution. Furthermore, the complexity is exacerbated by the fact that non-deterministic systems admit backtracking[4], 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 [Hermenegildo 1986; Hermenegildo and Greene 1990; Pontelli et al. 1995], 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 7):

—*Choice point*: allocated whenever a non-deterministic goal is called; it serves as a source of or-parallel work, as well as a description of the non-determinism present in the computation.

—*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 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 choice-point [Lusk et. al. 1990].

These extra data-structures can be quite large, 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 because in non-deterministic systems they are required mainly for the purpose of recording state so as to restore it in the event of backtracking, and deterministic systems do not backtrack. Additionally, some of these data structures (e.g., the Parcall Frame) are needed in order to allow dynamic scheduling of parallel tasks, a basic feature in most of these systems, and commonly unnecessary in parallelization of imperative programming languages (e.g., Fortran [Zima and Chapman 1991]).

3.1 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 con-

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

[4]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.

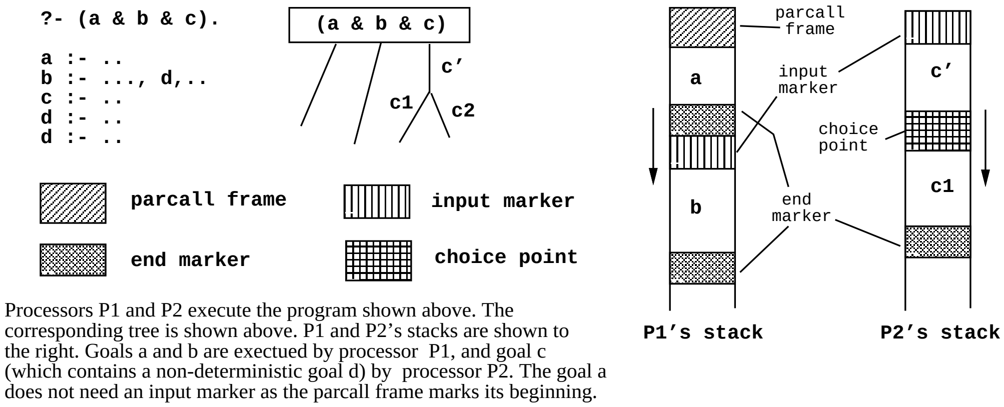


Fig. 7. Data-structures for Saving Computation State

currently 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 are 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:

—the tree structure developed during the parallel computation needs to be explicitly maintained, in order to allow for proper management of non-determinism and backtracking—this requires the use of additional data structures, not needed in sequential execution. Allocation and management of these data structures represent overhead during parallel computation with respect to sequential execution;

—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 (dynamic scheduling). This traversal is much simpler in a sequential computation, where the management of non-determinism is reduced to a linear and fast scan of the branches in a predetermined order.

3.2 Reducing Overheads in And-Or Tree

So far we have intuitively identified the main sources of overhead in a parallel computation—management of additional data structures, and 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 the process of traversing the complex structure of a parallel computation can be made more efficient:

1. simplification of the computation's structure: by reducing the complexity of the structure to be traversed it should be possible to achieve improvement in performance;

2. use of the knowledge about the computation (e.g. determinacy) in order to guide the traversal of the computation tree: information collected from the computation may suggest the possibility of avoiding traversing certain parts 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 this does preserve the desired execution behaviour).

2. avoiding allocation of unnecessary structures: most of the new data structures introduced in a parallel computation serve two purposes: *(i)* support the management of the parallel parts of the computation; *(ii)* support the management of non-determinism.

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 non-deterministic 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, since it is used as the testbed for our optimization techniques.

4. SIMPLIFICATION OF THE COMPUTATION'S STRUCTURE

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 non-determinism (e.g., searching for multiple solutions). This traversal is absent in sequential systems, where the whole computation can be encoded as a simple linear structure (e.g., sequence of choice points in a control stack). In parallel systems, by contrast, the computation structure gets arbitrarily distributed across the stacks of multiple processors (Figure 7).

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 [Warren 1980]) is an instance of the flattening scheme, where nested recursive calls get flattened into one single call.

—The *unfold* [Pettorossi and Proietti 1994] transformation, frequently used in both sequential and parallel systems, is another instance of the flattening scheme. In the context of logic programming unfolding leads to merging of choice-points reducing levels of their nesting. This has been applied to optimize or-parallel systems [Gupta and Pontelli 1996] and implementations of constraint languages [Van Hentenryck 1988].

—In parallel implementation of flat committed choice languages [Tick 1994] the nested goals created are held in a single large pool resulting in faster implementation. In fact, flat versions of committed choice languages evolved so that levels of nesting of processes could be eliminated to obtain faster implementations.

—Another limited form of flattening has been used in the ROPM [Ramkumar and Kale 1989] and goes under the name of *distributed last call optimization*. The objective of the optimization is to simplify the structure of the computation to reduce the message flow during the execution. If a goal g calls a goal h which in turns calls p, then if p is the last call in the execution of h and h is the last call in g, the solutions produced by p can be directly communicated to g, without going through h.

—An instance of flattening has also been used for efficient implementation of nested data-parallel operations in languages such as Paralation Lisp [Sabot 1988], C*, etc. Guy Blelloch was the first one to advocate the use of flattening in implementing data-parallel languages [Blelloch 1990]. Recently it has also been incorporated in other data-parallel systems [Palmer et al. 1994].

—The flattening schema is also employed in optimizing parallel code that has nested *barriers*. Parallel programs with nested barriers are transformed so as to merge the multiple barriers into one, resulting in improved performance [Ramakrishnan et al. 1995].

It should be noted, however, that though the flattening schema can be applied to both deterministic and non-deterministic systems, it brings about greater improvement in efficiency in non-deterministic systems. This is because in non-deterministic systems the tree structure may get simplified during the first traversal (i.e. during creation); on subsequent traversal (say, during backtracking) the tree structure is much simpler, resulting in considerably reduced overhead. For example, in the last parallel call optimization (described next), an optimization based on the flattening schema, much larger efficiency gains due to flattening are observed during backward execution (backtracking) rather than during forward execution when the tree is first created.

We discuss two optimizations that result from the application of flattening in the ACE system. These optimizations are termed *Last Parallel Call optimization* and *Last Alternative optimization* respectively.

4.1 Flattening: Last Parallel Call Optimization

4.1.1 *Introduction.* The Last Parallel Call Optimization (LPCO) is very similar in spirit to the last call optimization found in sequential implementations of Prolog [Warren 1980]. 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.

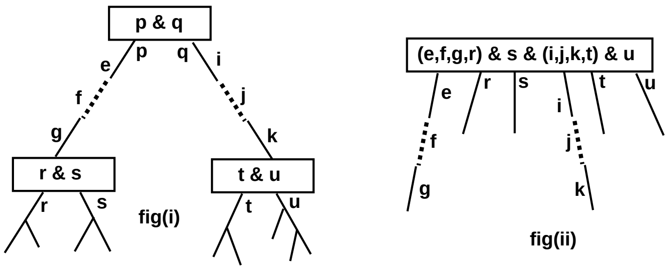


Fig. 8. Reusing Parcall Frames

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, consider a parallel conjunction of the form *(p & q)*, where (See Figure 8(i)):

p :- e, f, g, (r & s). *q :- i, j, k, (t & u).*

LPCO will apply to p (resp. q) if:

(1) there is only one (remaining) matching clause for p (resp. q), i.e., p (resp. q) is determinate;
(2) all goals preceding the parallel conjunction in the clause for p (resp. q) are determinate;
(3) the parallel call *(r & s)* (resp. *(t & u)*) is at the end of the clause for p (resp. q).

These conditions[5] can be more easily understood by considering the layout of the information on the stacks: conditions 1 and 2 are equivalent to the absence of choice points between the current point of execution and the logically preceding parallel call. It is important to observe that these conditions are met in a large variety of programs (e.g., programs possessing an iterative structure), making the optimization applicable very frequently. If these conditions are satisfied then a new parcall frame is not needed for the parallel conjunction in the clause. Rather the parcall frame for *(p & q)* can be extended with an appropriate number of slots and execution continues as if clause for p was defined as *p :- ((e,f,g,r) & s)*[6], if we determine at the time of the parallel call *(r & s)* that *e, f,* and g are determinate. This is illustrated in Figure 8(ii). Note also that the conditions for LPCO do not place any restrictions on the nature of the parallel subgoals in the clause for p (resp. q). Clearly, the goals r, s, etc. can be non-deterministic.

More formally, LPCO can be triggered whenever the following conditions hold:

—during the computation we have encountered a state of the form

$$\langle(\ldots\&p_{i-1}\&p_i\&p_{i+1}\&\ldots),\bar{G};L;\theta\rangle$$

—there exists a derivation

$$\langle p_i;[\,];\theta\rangle \ \vdash^* \ \langle(q_1\&\ldots\&q_m);[\,];\psi\rangle$$

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.

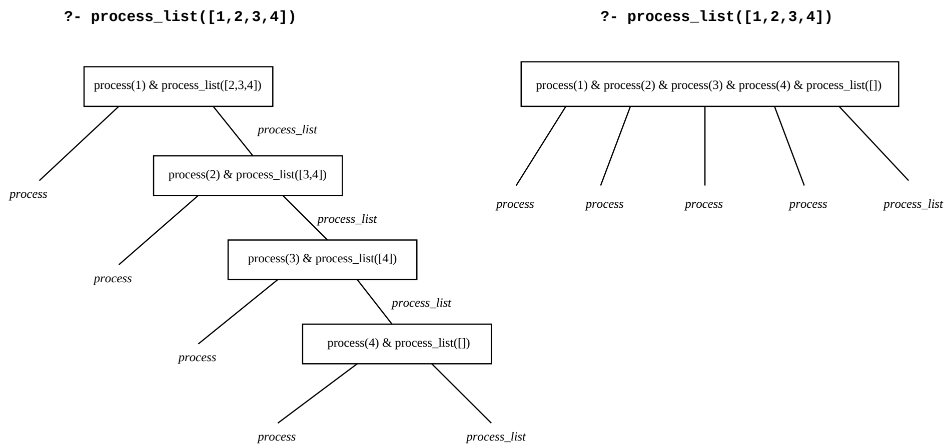


Fig. 9. Reuse of Parcall Frames for Recursive Programs

[5]These conditions can be relaxed [Pontelli and Gupta 1996] but this is not analyzed any further for the sake of simplicity.

[6]Although, in our case, the bindings finally generated by *e, f, g* should be produced *before* starting the execution of s.

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 (that arises very frequently in real-life programs):

```
process_list([H|T],[Hout|Tout]) :-
   process(H,Hout) & process_list(T,Tout).
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 run-time technique, given that the goal *process_list* is determinate, nesting of parcall frames can be completely eliminated (Figure 9). 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 much greater overhead).

On applying the LPCO in a situation such as the one shown in Figure 9, control parallelism that is deeply nested transforms itself into data-parallelism resulting in improved efficiency. In fact, our data-parallel version of Prolog [Pontelli and Gupta 1995], based on LPCO is comparable in performance to dedicated data-and parallel Prolog systems such as Reform Prolog [Bevemyr et al. 1993].

4.1.2 *Implementation of LPCO.* To implement LPCO, the compiler will generate a different instruction when it sees a parallel conjunction at the end of a clause. This instruction, named `opt_alloc_parcall`, behaves the same as the `alloc_parcall` instruction (the instruction used to create parallel conjunctions) of ACE, except that if the conditions for LPCO are fulfilled the last parallel call optimization will be applied.

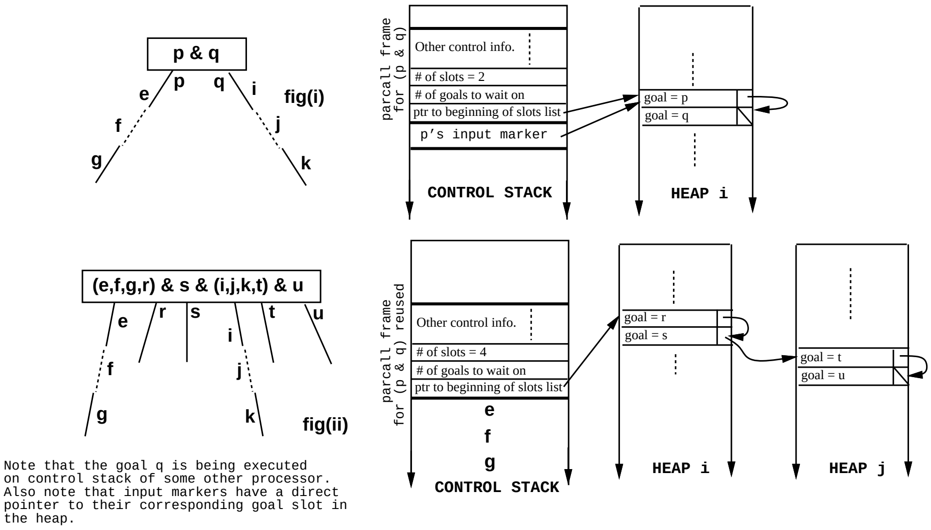


Fig. 10. Allocating Goal Slots on the Heap

The behaviour of `opt_alloc_parcall` is quite straightforward: once it is reached, it will perform a check to verify whether the conditions for the application of LPCO are met—i.e. no choice points are present between the current point of execution and the immediate ancestor parcall frame. If the check succeeds, then LPCO will be applied, otherwise the instruction will proceed creating a new parcall frame—i.e., it behaves like a normal `alloc_parcall` instruction. The above mentioned check is immediate and does not introduce any significant overhead—it is sufficient to check that the data structure currently on top of the stack is a parcall frame or an input marker (and not a choice-point).

To apply LPCO, the immediate ancestor parcall frame (or immediately enclosing parcall frame) will be accessed and if the current parallel conjunction has n and-parallel goals, then n new slots corresponding to these n goals will be added to it. The number of slots should be atomically incremented by n in the enclosing parcall frame.

Introducing the LPCO in the ACE system requires only one small change in the architecture. In the original ACE (as in all the and-parallel systems based on the RAPWAM model) the slots which are used to describe the subgoals of a parallel call are stored on the stack as part of the parcall frame itself. Given that the enclosing parcall may be allocated somewhere below in the stack, adding more slots to it may not be feasible. To enable more slots to be added later, the slots will have to be allocated on the heap and a pointer to the beginning of the slot list stored in the parcall frame (Figure 10). The slot list can be maintained as a double linked list, simplifying the insertion/removal operations. Also, each input marker of an and-parallel goal has a pointer to its slot in the slot list for quick access (this is already part of the original ACE design). With the linked list organization, adding new slots becomes quite simple as shown in Figure 10. Note that the modification of the slot list will have to be an atomic operation. Furthermore, the modification of the slot list has to be a *backtrackable* operation, i.e., its effect should be removed upon backtracking.

The enclosing parcall frame becomes the parcall frame for the last parallel call, and rest of the execution will be similar to that in standard ACE. The garbage collection mechanisms used on the heap guarantees that as soon as we have completely backtracked over a nested parallel call (optimized by LPCO) the space taken by the slots will be immediately recovered.

Note that changing the representation of slots from an array recorded on the stack (inside a parcall frame) to a linked list on the heap will not add any inefficiency because an and-parallel goal can access its corresponding slot in constant time via its input marker, and any other operations on the slots requires anyway a linear scan of all the slots in the parallel call.

It is obvious that LPCO indeed leads to saving in space as well as time during parallel execution. In fact:

- space is saved by avoiding allocation of the nested parcall frames;
- occasionally some time may be saved during forward execution, although the time complexity of applying LPCO is usually comparable to the time complexity of allocating a parcall frame;
- time is saved during backtracking, since the number of control structure to traverse is considerably reduced.

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	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 – time in sec.)

4.1.3 *Performance Results.* The results of applying LPCO to the ACE system are shown in Tables 1 and 2 (times are in sec.). The performance figures have been obtained on a Sequent Multiprocessor; it must be noted that *equivalent* results have been observed on different architectures (i.e., Sun Sparc Multiprocessors and Pentium based Multiprocessors). In this paper we

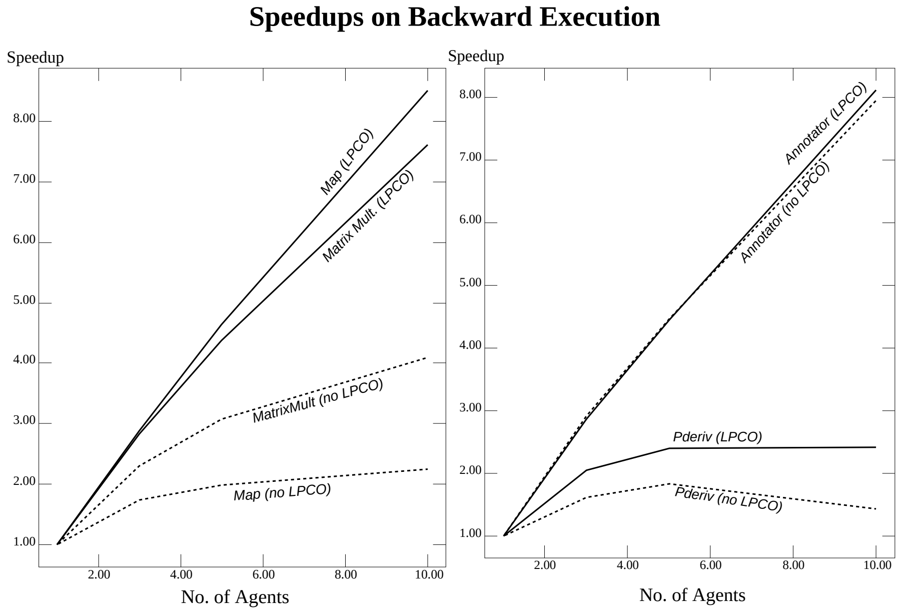


Fig. 11. Speedup Curves for exec. with backtracking

Benchmark executed	Number of Processors 1	3	5	10
MatrixMult	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

report the results from Sequent as this is the machine with the larger number of processors accessible for our experiments. It should also be noted that different sets of benchmarks have been used for illustrating the performance gains obtained from different optimizations presented in the paper. All the benchmarks have produced improvements in performance using the different optimizations—we have deemed more interesting to report those situations in which the effects are more notable, as this will allow us to relate the performance of the optimized systems with the characteristics of the benchmarks executed. All benchmarks considered have been drawn from the pool of benchmarks commonly used in the parallel logic programming community [Santos Costa et al. 1991; Ali and Karlsson 1990; Lusk et. al. 1990; Bevemyr et al. 1993; Ramkumar and Kale 1989; Roy 1993].

The benchmarks considered for LPCO are:

—*map*[1]*:* an application involving repeated large granularity data-parallel computations deeply nested one inside the other;

—*map*[2]*:* another instance of the problem of *map*[1] but with the use a fine-grained computation in the data-parallel part of the execution;

—*MatrixMult:* program for matrix multiplication;

—*Occur:* symbolic search of occurrences of elements in a deeply nested list structure;

—*Pderiv:* program for symbolic differentiation;

—*Annotator:* the complete abstract interpreter used by ACE for automatic program parallelization;

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 11. For forward execution the improvements have been generally very modest if not negligible. The only cases where a more visible improvement has been registered are those where the parallel computations are deeply nested and very fine grained (e.g., map^2. In these cases the cost of creating parcall frames tend to overcome the "real" computation during unoptimized execution.

Most benchmarks have instead presented noticeable improvements for benchmarks with backtracking. As expected the improvements are more evident in presence of backtracking over deeply nested parallel computations—as the LPCO allows to flatten the computation and translate backtracking into a fast scan of a list of subgoals. The most noticeable is the map^1 benchmark (see Figure 1): 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% [Pontelli et al. 1995]. This is illustrated in figure 12, which contrasts the combined usage of the control stack and heap in the unoptimized and optimized cases.

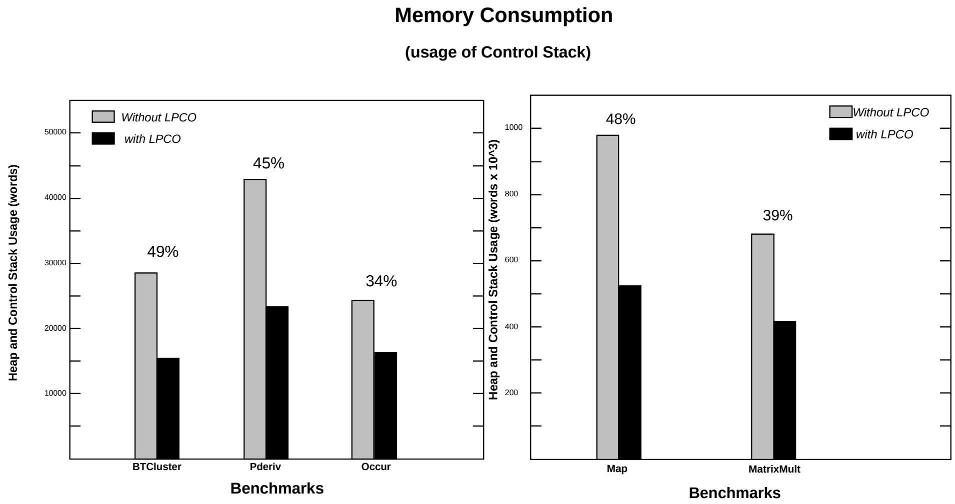


Fig. 12. LPCO: Improvement in Control Stack and Heap usage

A dual optimization can be developed to improve efficiency of or-parallel executions, by merging choice points generated along the last alternative of each or-node. This optimization, named *Last Alternative Optimization (LAO)*, is described next.

4.2 Flattening: Last Alternative Optimization

4.2.1 *Introduction.* We illustrate the Last Alternative Optimization (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).
```

The or-parallel tree structure created is shown in figure 13.

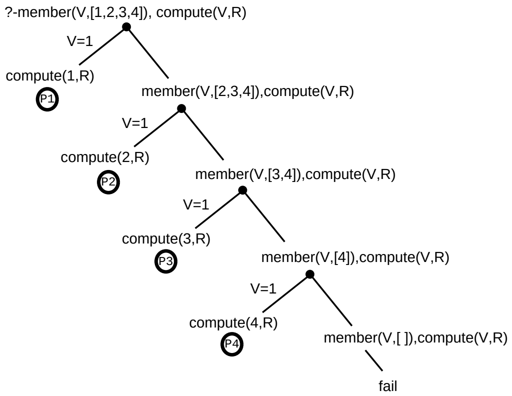


Fig. 13. Search tree for member

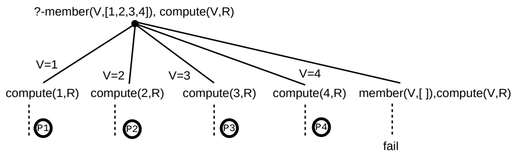


Fig. 14. Search tree for member w/ LAO

Notice that or-parallelism will be exploited when processors move to choice points (shown as dark circles in Figure 13 and 14) and pick the compute goals for execution. A possible scenario is shown in figure 13 itself where processors P1, P2, P3, and P4 are exploring the four branches created for solving compute(i,R) where $1 \leq i \leq 4$.

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'd like to have all calls to compute generated all at once as shown in Figure 14.

The tree structure shown in figure 13 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 14 all compute goals are clubbed at one choice point 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).
```

From a more formal perspective, the LAO optimization will be triggered whenever the following conditions hold:

—p and q are "parallel" goals, i.e., goals on which we allow exploitation of parallelism;

—the computation encounters a transition

$$\langle p, \bar{G}; L; \theta \rangle \ \vdash \ \langle \bar{G}', \bar{G}; L : [[r_2, \ldots, r_k]]; \theta \circ \sigma \rangle$$

—r_k has the form Head_k :- Body_k and the equation $\mathsf{Head}_k = p^\theta$ has the solution σ_k

—the following derivation is possible

$$\langle \mathsf{Body}_k, \bar{G}; L; \theta \circ \sigma_k \rangle \;\; \vdash^* \;\; \langle q, \bar{G}'; L; \theta' \rangle$$

Several researchers have proposed ways for making the work distribution operation more efficient. For example, Smith, in his Multilog system [Smith 1994; Smith 1993], has proposed the addition of a *disj* primitive to Prolog, that the programmer should place before call to member. Thus, a typical Multilog query has the form:

`?- ` *disj* `member(X,[1,2,3,4]), compute(X,R).`

The *disj* is handled by the Multilog system in a special way: the `compute` predicate is iteratively called for each element in the list. Thus, intuitively, the list structure is converted into an array like structure, on which a data-parallel `compute` call is made.

Our goal is to avoid the overhead of having to traverse the extra nestings created due to recursive unfolding of the member function, but without requiring any help from the programmer. The solution thus is to recognize the case in Figure 13 and somehow transform it at runtime to the situation shown in 14. This is indeed what the Last Alternative Optimization accomplishes.

This flattening can be accomplished at runtime as follows: 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 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.

Again, also LAO can be thought of as a generalization of the last call optimization (LCO) found in sequential Prolog system for or-parallel systems. The last call optimization itself is a generalization of the familiar tail recursion optimization for sequential Prolog systems. The last alternative optimization not only saves execution time but also space. Execution time is saved in two ways:

—the search tree is simplified (its depth is reduced); as a consequence, the processors have shorter paths to traverse to get to choice-points that contain work in the form of untried alternatives.

—since an existing choice-point is used instead of allocating a new one, time is saved—various fields of the older choice points can be re-used without any update.

An interesting point to note about LAO is that, although it is applicable to sequential implementations of Prolog [Meier 1991], it does not bring much benefit to sequential execution. This is because in sequential execution, when the last alternative is picked, the corresponding choice point is already deallocated. However, for parallel implementations of Prolog, the efficiency gains can be enormous as our performance figures indicate. 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 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.

Parallel logic programming systems that incorporate or-parallelism system will considerably benefit from LAO, as the simplification of the structure of the tree leads to fewer constraints on the scheduling algorithm, and thus to more efficient scheduling. For instance, in the DNA sequencing program of Lusk et al. [Lusk et al. 1993], one of the largest or-parallel program that has been ever tried, or-parallelism is obtained from the following two rules

```
find_match([C|Right],Left) :-
   find_a_matching_pattern(C,Right,Left).
find_match([C|Right],Left) :-
   find_match(Right,[C|Left).
```

A `find_match` query will create a right-recursive tree that is explored in parallel. Lusk et al switch the order of rules, due to the scheduling policy used in their Aurora or-parallel system, in order to generate a left recursive tree, so that work is more evenly distributed. With LAO such considerations become moot, since all the untried alternatives become available in a single choice-point. (Lusk et al extensively discuss how to make the scheduler work the way they want it to in their paper [Lusk et al. 1993]). It is our conjecture that incorporation of LAO in Aurora will considerably improve the performance of Lusk et al's program (LAO provides exactly the effect that Lusk et al are seeking) as well as of other or-parallel programs run on the Aurora system.

LAO will also improve the performance of programs that use failure driven loops based on Prolog's `repeat` builtin. The `repeat` builtin (defined below) creates an infinite number of choice points from where work gets picked up (a cut is usually necessary later to destroy the infinite choice-points and terminate the computation). Applying LAO will merge the multiple choice points created by the recursive `repeat` into one single choice point.

```
repeat :- repeat.
repeat.
```

An optimization for sequential systems based on flattening, and that is somewhat similar to LAO, has been proposed by Meier [Meier 1991].

This optimization is also complementary to the work on compile-time analysis and transformation to improve scheduling of parallel tasks [Debray and Jain 1994; Hermenegildo and Carro 1995].

4.2.2 *Implementation of LAO.* 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"[7], then no new choice point will be created. Instead the previous choice point is reused. For the sake of simplicity, we have restricted the implementation to apply the optimization only to choice points which are associated to the same predicate.

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).

It is easy to include LAO in any or-parallel implementation. In the text below we sketch the changes we had to make to accommodate LAO in the ACE and in the MUSE systems. As discussed earlier, a parallel choice point that has been shared in ACE actually replicates itself into

[7]Note that in most or-parallel systems whether a choice point is parallel or sequential is indicated by the user. Only a parallel choice point can give or-parallel work to other processors. 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.

(1) a number of *private* copies of the choice point, one for each processor that took work from that choice point, and
(2) the shared frame, with each private copy having a pointer to it.

Intuitively, the private copies are meant to store the "static" part of the state (the information that will not change between execution of the different alternatives), while the shared frame is used to maintain the part of the state that will be modified during execution (and modifications performed by one processor need to be visible to the others) [Ali and Karlsson 1990].

Given that implementation of LAO requires modification of certain fields of an existing choice point, these fields need to be transferred from the choice point to the shared frame, in order to allow communication of the changes to the other processors. In particular, two relevant parts of the choice point that need to be *"globalized"* or transferred to the shared frame are:

—the arguments of the subgoal;
—the pointer to the trail stack[8].

The general structure of the choice point and of the shared frame is depicted in figure 15.

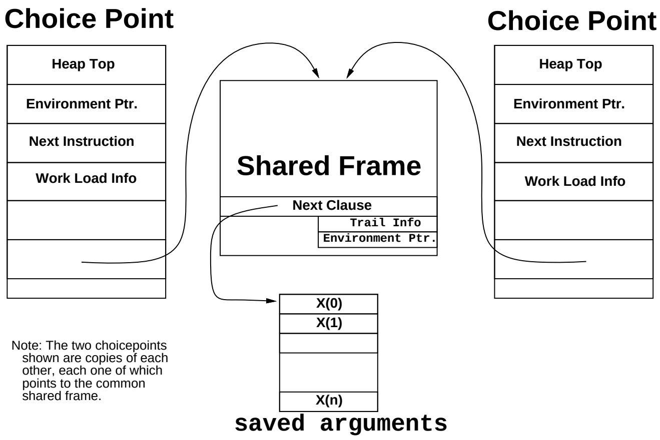


Fig. 15. Choice Points and Shared Frames

The current prototypical implementation of LAO is mainly aimed at taking advantage of recursive generators like those described in the previous examples (`member`, `select`, etc.). LAO can be easily generalized to situations where the choice point created along the last alternative of another choice point corresponds to a different predicate. The price of this generalization is the requirement of further globalizing some more fields of the choice point (in particular the environment pointer needs to be moved to the shared frame). Furthermore, the more general scheme should take care of the following two situations:

(1) the different procedures may have different number of arguments (see figure 15)—this requires the use of *Argument Frames* associated to the alternatives stored in the choice point;
(2) various environments may have been allocated between the two choice points being merged. If the alternatives of the latter choice point are promoted to the former one, then these environments need to be "communicated" to the other processors (otherwise they will not be able to correctly execute the new alternatives). This means that a further copying phase may be needed upon backtracking over an optimized choice point.

Nevertheless, the savings guaranteed by LAO easily overcome the additional expenses for the above overheads.

[8]The trail stack keeps track of the bindings that need to be undone on backtracking.

4.2.3 *Performance Results.* Table 3 describes the results obtained by running some benchmarks on the ACE systems with and without LAO. The results have been recorded on a Sequent multiprocessor, but comparable results (in terms of speedups and improvements with respect to the unoptimized case) have been observed on Sparc-based and Pentium-based Multiprocessors. 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[9]. Figure 16 compares the speed curves with and without optimization, while figure 17 represents the improvement in execution time obtained using LAO.

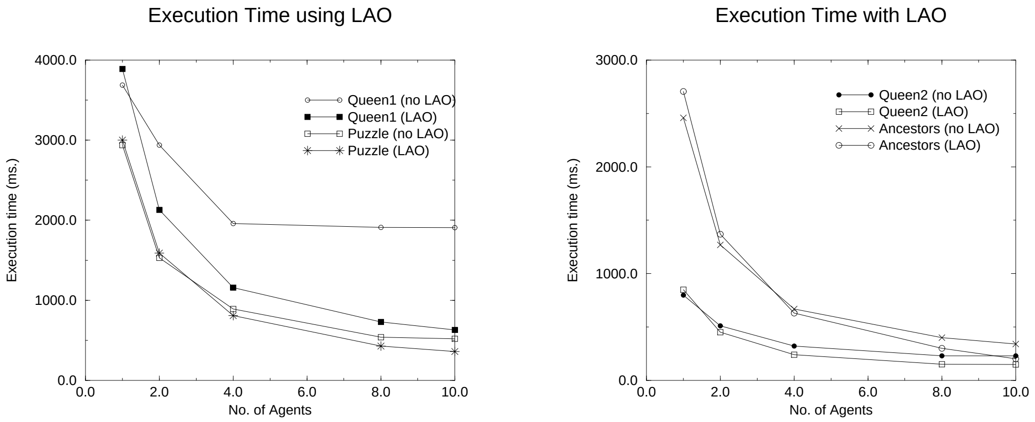


Fig. 16. Speed Curves for LAO Benchmarks

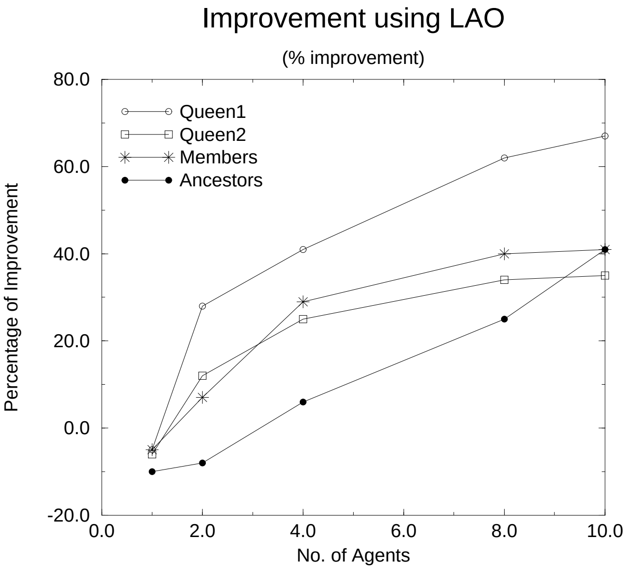


Fig. 17. Improvement in Execution Time

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

Even though the current implementation of LAO is still a prototype, the numbers produced are quite impressive. Especially with larger number of processors, the reduction in execution time is frequently over 30%. These good results are consequence of the particularly good interaction between LAO and stack copying. The application of LAO allows to "promote" new alternatives created by a processor, storing them directly in a choice point that has already been copied by other processors (i.e., it has already a shared frame associated with it). In this way the new alternatives become immediately visible to the other processors, without the need of performing further sharing activities. As a consequence the number of sharing operations performed during execution is considerably reduced (in some benchmarks this number decreased from over 100 to less than 10). Considering that each sharing operation may be quite expensive, a good reduction in execution time can be expected.

Briefly, the benchmarks analyzed are

—*Queen*^{1}, a generate&test version of the queens problem;

—*Queen*^{2}, a smarter solution to the queens problem;

—*Puzzle*, a solver for a crypto-arithmetic puzzle;

—*Ancestors*, a program performing a search in a genealogical tree;

—*Members*, a generate&test solution to an optimization problem;

—*Maps*, another map coloring problem applied to a map of West Europe.

All these programs use either the `member` or the `select` predicate to generate alternative paths to the solution(s).

It must be noted that, in the current implementation, the presence of LAO tends to produce a slight degradation in execution time on a single processor (as can be observed from the execution times on one processor). This is due to the additional cost of testing, at each choice point creation, whether LAO can be applied or not. We expect to be able to considerably reduce this overhead by using a more careful implementation of these tests and by taking advantage of the information supplied by the compiler (e.g., distinction between sequential and parallel nodes).

Program	**No. of Agents**				
	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)

In rare cases (as for the *Maps* program, which solves the map coloring problem for a map of West Europe), anomalous behaviour was produced, due to the experimental nature of our scheduler. The problem should disappear if more sophisticated scheduling strategies are adopted [Beaumont and Warren 1993].

The overall effect observed in these experiments is that the data-parallel nature of the benchmarks, hidden by the use of recursion to express the application of a computation to a sequence of inputs, is again exposed as a collection of alternatives stored in a single choice point.

4.2.4 *LAO for Parallel CLP.* A parallel *Constraint Logic Programming (CLP)* system, based on finite domains, such as CHIP [Van Hentenryck 1988] or CLP(FD) [Codognet and Diaz 1993] can take substantial advantage of LAO. A typical CLP problem can be characterized as follows: given a collection of variables, each of which ranges over a finite domain (set of values), the task is to find out an assignment of values for these variables from their respective domains such that certain user stated constraints are satisfied. In traditional logic programming this will be stated as a *generate and test* problem, where all possible assignments of values for the variables from the domains will be considered and then tested for consistency under the given constraints. CLP over finite domains makes this process efficient, by setting up the constraints first, which significantly prunes the domains of variables (any constraints that cannot be solved in this phase are suspended), and then performing a generate and test, where the generation phase considers values from the pruned domain (programmed by using built-ins like `labeling` or `indomain`) and the test phase consists of testing the assignments generated in the `generate` phase for the remaining suspended constraints.

The predicate `indomain` is nothing but a call to a member predicate, where the variable in question is assigned successive values from the pruned domain. Most CLP systems provide efficient ways of doing this by providing `deleteff` or `labelingff` [Van Hentenryck 1988] predicates which will first assign to variables that have smaller domains or are more likely to generate a failure (*first-fail* heuristic [Van Hentenryck 1988]).

In a parallel CLP system that incorporates LAO, the `indomain` predicate unfolds into a data-parallel call due to LAO, thus, automatically improving the efficiency of parallel execution. Thus, with the incorporation of LAO, one can incorporate data-parallelism automatically into a parallel CLP system.

5. 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 non-determinism 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, i.e., recognizing situations where potentially parallel computations are actually executed sequentially.

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) [Warren 1983] 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 [Carlsson 1989] and implemented in both sequential (SICStus) as well as or-parallel (Aurora [Lusk et. al. 1990] and MUSE [Ali and Karlsson 1990]) versions of Prolog. Shallow backtracking delays creation of the choice point until the head unification (and eventually execution of the guard) has succeeded. If none of the head unifications is successful, then backtracking is activated without creation of any choice point (i.e., the creation of the choice point is postponed indefinitely). Carlsson reported an average speedup of 7% to 15% on a set of complex benchmarks.

The idea of procrastination has also been applied to or-parallel systems such as MUSE and Aurora in the context of choice points. Typically, in an or-parallel system the choice points have to save a larger state than in a purely sequential system. However, if all alternatives of a choice point are explored by only one processor, then saving this extra state is a pure overhead. In MUSE and Aurora, this overhead is delayed until the scheduler explicitly declares a choice point to be parallel (in MUSE and Aurora terminology, it is *made public* [Ali and Karlsson 1990; Lusk et. al. 1990]; processors can come and pick untried alternatives only from public choice-points of other processors). Thus, in MUSE and Aurora, when a choice point is allocated, space is allocated for the extra part of the state that is related to parallel execution (scheduling information, value in extra registers introduced to support parallelism, etc.), however, this space is not filled in at the time of allocation. The information is filled in only later, when the choice point is made public by the scheduler. If a choice point is never made public, then the overhead associated with saving the extra state is never incurred.

Procrastination has also been employed in the context of *lazy evaluation* of functional languages. However, our schema urges system designer to procrastinate *overheads* as long as the operational semantics are not altered. This is not the case with lazy evaluation where the overheads are not procrastinated, rather the evaluation of operations themselves in the program is procrastinated. Also, lazy evaluation alters the operational semantics of the language.

The procrastination schema can be applied to parallel logic programming systems by delaying the allocation of control data-structures, such as the choice pointes and the markers, as much as possible. The hope is that the allocation of the structures can be indefinitely delayed, i.e., the structures are not needed by the execution. On the other hand, the allocation of the data structures cannot be arbitrarily delayed, as this may lead to modification of the semantics of the execution. For example, in the case of the optimization in MUSE and Aurora discussed in the previous paragraph, where saving of state data related to parallelism is procrastinated, it can be procrastinated only up to the point when that choice-point is declared public by the scheduler. Delaying it any further will cause the the information to be saved to become unrecoverable. Ideally, the saving operation should have been delayed until we are certain that alternative at this choice-point will be executed by at least two processors, because it is possible that only one processor may execute all the alternatives after a choice point becomes public, resulting in the time spent in saving the parallel state going waste.

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 incur minimal overhead."*
This schema has been implicitly used in various parallel systems. For example, many or-parallel implementations of Prolog (like MUSE [Ali and Karlsson 1990] and Aurora [Lusk et. al. 1990]) 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. Granularity control attempts to break up the parallel execution into large pieces of sequential computation such that the total parallel overhead is insignificant compared to the execution time. Large grain sizes are obtained by taking consecutive tasks (consecutive in the sense of sequential execution) and putting them together for execution by one processor so that parallel overhead associated with the smaller tasks that constitute the large grain is eliminated. A similar idea has also been applied to committed choice logic languages [Tick 1994] and improvements in performance obtained.
Like the flattening and procrastination schemas, the sequentialization schema is more relevant to non-deterministic systems. This is because in non-deterministic system, two pieces of computation that are contiguous in sequential execution have to be delimited (with markers) during parallel execution in order to facilitate their parallel execution. Delimiters are needed because processors may backtrack over these computations. If two contiguous pieces of computation that

were anticipated to be executed in parallel, but are eventually executed by the same processor, then the overhead incurred in delimiting these two contiguous computations can be avoided. Since backtracking is absent in deterministic systems, and computations need not be marked as in non-deterministic systems, this schema has little applicability to deterministic systems. For example, during parallel execution of iterations of a Fortran loop, no overhead is saved if a processor knows, while executing the iteration with loop index k, that the next iteration it will execute is the next consecutive iteration with loop index $k + 1$. This is because no substantial parallel overhead is incurred in picking work (iterations) while executing Fortran `DO` loops in parallel. In contrast, in non-deterministic systems substantial overhead may be incurred after picking work and before starting its execution because of the need to allocate markers for backtracking purposes.

Next we illustrate the application of procrastination and sequentialization schemas to the case of parallel logic programming.

5.1 Procrastination: Shallow Parallelism Optimization

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 an 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—an 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. Indeed the majority of independent and-parallel systems proposed rely on this memory model. The expense incurred in allocating these markers is considerable, since each marker stores various 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 (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—see figure 18. Thus, in general, a goal of the form $(\ldots \& G_{i-1} \& G_i \& G_{i+1} \& \ldots)$ is trans-

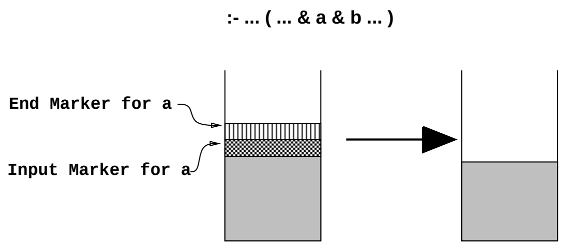


Fig. 18. Shallow Parallelism Optimization

formed into $(\ldots \& G_{i-1} \& G_{i+1} \& \ldots)$ for backtracking purposes.

Allocation of markers, 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.

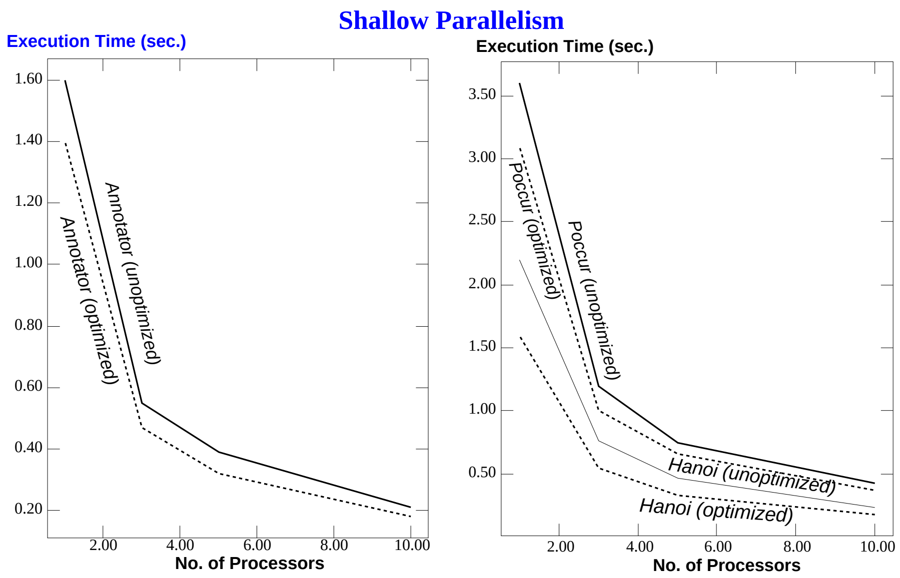


Fig. 19. Exec. time w/ Shallow Parallelism

We term this optimization, where we avoid allocation of input and end marker nodes, *shallow parallelism optimization*. The name is inspired by the similarity at the implementation level between this optimization and the previously described *shallow backtracking* optimization of Carlsson.

Benchmark executed	Number of Processors 1	3	5	10
MatrixMult	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%)
BTcluster	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%)
Boyer	2.4/2.4 (0%)	.89/.89 (0%)	.61/.61 (0%)	.4/.4 (0%)

Table 4. Unoptimized/Optimized Exec. times (sec.) for Shallow Parallelism (% improv. shown in parenthesis)

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 19.

The benchmarks used to evaluate this optimization are:

—*Annotator:* the complete abstract interpreter used by ACE for automatic program parallelization;

—*Hanoi:* solution of the Hanoi's Tower problem;

—*Occur:* symbolic search of occurrences of elements in a deeply nested list structure;

—*MatrixMult:* program for matrix multiplication;

—*Takeuchi:* computation of Takeuchi's function;

—*Boyer:* theorem proving application;

—*BTcluster:* telecommunication clustering program from British Telecom.

Shallow parallelism, by reducing the number of markers created, allows to considerably reduce the memory usage during the parallel execution. Figure 20 compares the memory usage for a selected set of benchmarks (the figure reports the number of markers created during an execution). We

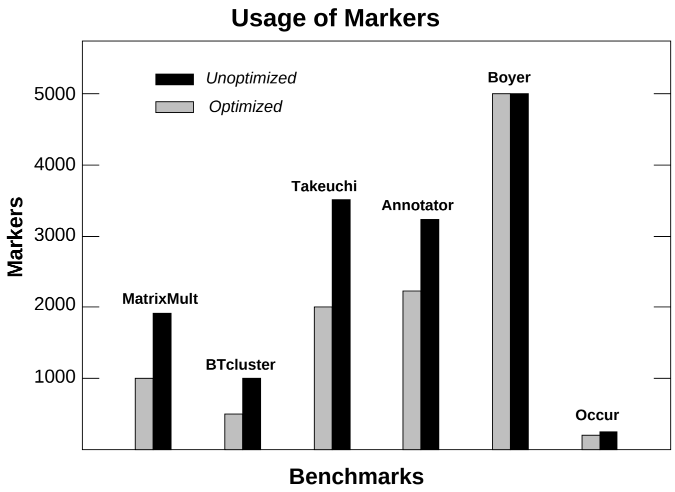


Fig. 20. Shallow Parallelism: No. of markers allocated

go from an extreme case like *Boyer* (no savings at all), to some excellent results, like for *Takeuchi* (50% savings on the number of markers allocated). In general, the improvement is proportional to the number of deterministic goals present in the computation—e.g., in *Boyer* a negligible amount of deterministic computations appears.

5.2 Sequentiality: Processor Determinacy Optimization

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 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—see figure 21.

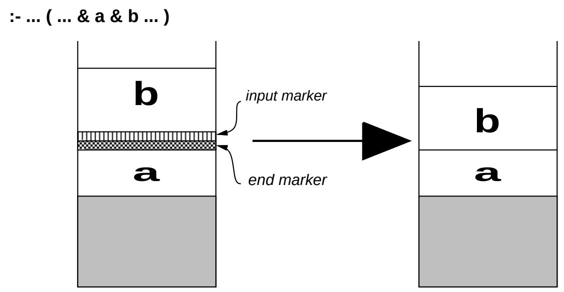
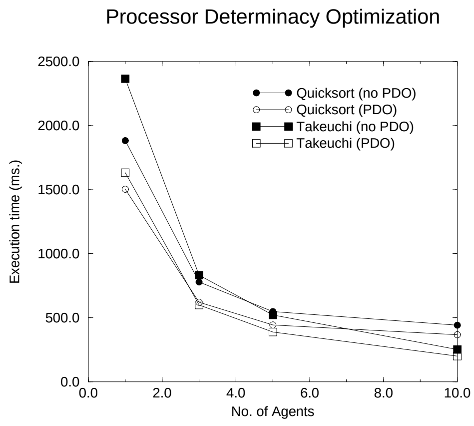


Fig. 21. Processor Determinacy Optimization

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.



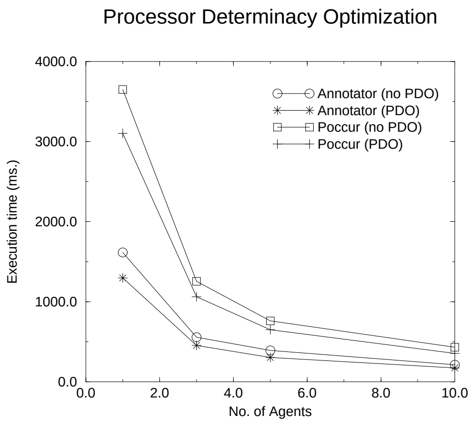


Fig. 22. PDO: Improvement in Exec. Time

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. The check is immediate, at least in an architecture in which knowledge of the order of subgoals is maintained, which is readily available in almost all parallel

Goals executed	Number of Processors 1	3	5	10
MatrixMult	5598/5207 (8%)	1954/1765 (11%)	1145/1067 (7%)	573/536 (7%)
Quicksort	1882/1503 (25%)	778/621 (25%)	548/443 (23%)	442/367 (20%)
Takeuchi	2366/1632 (45%)	832/600 (39%)	521/388 (34%)	252/200 (26%)
Occur	3651/3104 (15%)	1255/1061 (18%)	759/649 (17%)	430/353 (22%)
BTcluster	1461/1330 (10%)	528/482 (10%)	345/294 (17%)	202/165 (22%)
Annotator	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)

logic programming systems that have been built. Improved results can be obtained by modifying the scheduler itself, giving preference to selection of consecutive subgoals over other goals.

Regarding forward execution, the results obtained are extremely encouraging, as can be observed from table 5. For many examples the optimization manages to improve the execution time by 3% to 20%. The variations in improvement depend on the number of parcall frames generated, on the effect of the marker allocation overhead on the overall execution time, and on the number of processors available. For this reason we obtain considerable improvements in benchmarks like *takeuchi*, where we have deep nestings of parallel calls and the marker allocation represents the main component of the parallel overhead. The optimization maintains its effects when we consider execution spread across different processors, as we can see in figure 22, which shows the execution times of both the optimized and the unoptimized version for some benchmarks. The advantages are even more evident in terms of memory consumption: the number of markers allocated is cut in almost all the cases to half of the original value—this is because most of the examples analyzed have parallel calls of size 2 and the optimization allows to avoid the allocation of the marker between the two subgoals. Figure 23 shows the reduction in number of markers created thanks to the use of PDO. The improved speed curves are represented in figure 22.

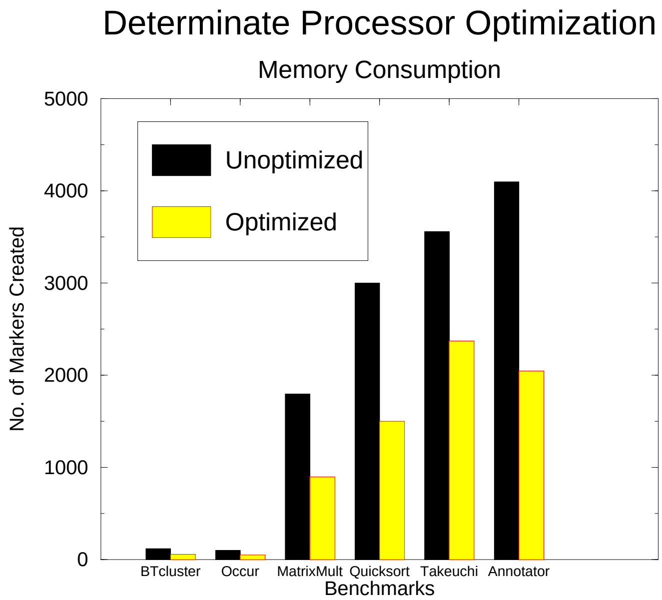


Fig. 23. No. of Markers allocated in presence of PDO

An optimization (*Backtracking Families*), which combines a restricted version of Shallow Parallelism and Processor Determinacy and relies on static analysis, has also been proposed by Carro and Hermenegildo [Carro and Hermenegildo 1993; Pontelli et al. 1996]. The idea is to use compile time analysis to detect determinacy of subgoals and use this knowledge to extend applicability of the Processor Determinacy Optimization. In fact if we have a parallel call a & b & c and it is known that b is deterministic, then if a and c are executed by the same computing agent the PDO can still be applied—this is because from an operational semantics point of view *(a & b & c)* is equivalent to *(b & a & c)* if b is deterministic (modulo side-effects, like I/O calls).

6. CONCLUSIONS

Non-determinism arises in many important areas of computer science such as Artificial Intelligence and constrained optimization. Non-determinism has also been incorporated in several language families (such as logic, constraint, and concurrent constraint languages). Parallel non-deterministic systems are more overhead prone due to the need of saving state information at every step, so that in the event of backtracking correct state can be restored. Naive parallel implementations of non-determinism have resulted in systems that perform very poorly.

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 used to develop novel optimizations for the ACE system, 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—similar improvements were observed also on Sun Sparc Multiprocessors. While these schemas have been illustrated on the ACE system, they are fairly general, and can be readily applied to other non-deterministic systems such as parallel theorem proving systems, parallel rule based and AI systems, and parallel implementations of constraint and concurrent constraint languages.

Our experiments show that one can systematically improve execution performance of parallel non-deterministic systems and languages by designing optimizations based on schemas presented in this paper. We hope this paper will induce other researchers to look for more such general optimization schemas, as well as to develop other useful optimizations based on schemas presented in this paper.

Acknowledgements

Thanks are due to Manuel Carro, Eduardo Correia, Vitor Santos Costa, Ines Dutra, Manuel Hermenegildo, Kish Shen, Fernando Silva, and David H.D. Warren for many stimulating discussions and suggestions.

REFERENCES

Ali, K. and Karlsson, R. 1990. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.* (1990), pp. 757–776. MIT Press.

Beaumont, T. and Warren, D. 1993. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *10th Int'l Conference on Logic Programming* (1993), pp. 135–149. MIT Press.

Bevemyr, J., Lindgren, T., and Millroth, H. 1993. Reform Prolog: the Language and its Implementation. In *Proc. of the 10th Int'l Conference on Logic Programming* (1993), pp. 283–298. MIT Press.

Blelloch, G. 1990. *Vector Models for Data-parallel Computing.* MIT Press.

Brownston, L. 1985. *Programming Expert Systems in OPS5: an introduction to rule-based programming.* Addison-Wesley.

Bueno, F., de la Banda, M. G., and Hermenegildo, M. 1994b. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *Parallel Symbolic Computation* (September 1994), pp. 63–73. World Scientific Publishing Company.

Bueno, F., de la Banda, M. G., and Hermenegildo, M. 1994a. Effectiveness of Global Analysis in Strict Independence-based Automatic Program Parallelization. In M. Bruynooghe Ed., *International Logic Programming Symposium* (1994). MIT Press.

Carlsson, M. 1988. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden.

Carlsson, M. 1989. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In *Sixth International Conference on Logic Programming* (June 1989), pp. 3–16. MIT Press.

Carro, M. and Hermenegildo, M. 1993. Backtracking families for independent and-parallelism.

Codognet, P. and Diaz, D. 1993. A Minimal Extension of the WAM for `clp(fd)`. In *International Conference on Logic Programming* (1993). MIT Press.

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

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

Debray, S. K. and Warren, D. S. 1986. Detection and Optimization of Functional Computations in Prolog. In *Third International Conference on Logic Programming*, Number 225 in Lecture Notes in Computer Science (July 1986), pp. 490–505. Imperial College: Springer-Verlag.

Debray, S. K. and Warren, D. S. 1988. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 207–229.

DeGroot, D. 1984. Restricted AND-Parallelism. In *Int'l Conf. on 5th Generation Computer Systems* (Nov. 1984), pp. 471–478. Tokyo.

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

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

Hermenegildo, M. 1986. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Proc. 3rd ICLP, LNCS 225* (1986), pp. 25–40. Springer-Verlag.

Hermenegildo, M. 1994. Towards Efficient Parallel Implementation of Concurrent Constraint Logic Programming. In T. Chikayama and E. Tick Eds., *Proc. of ICOT/NFS Workshop on Parallel Logic Programming and its Programming Environments*. University of Oregon.

Hermenegildo, M. and Carro, M. 1995. Relating Data-Parallelism and (And-)Parallelism in Logic Programs. In *EuroPar'95* (1995), pp. 43–54. Springer Verlag.

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

Hermenegildo, M. and Nasr, R. I. 1986. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, Number 225 in Lecture Notes in Computer Science (July 1986), pp. 40–55. Imperial College: Springer-Verlag.

Hermenegildo, M., Warren, R., and Debray, S. 1991. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*.

Lampson, B. 1983. Hints for Computer System Design. In *ACM Symposium on Operating Systems Principle* (1983). ACM.

Lusk, E., Mudambi, S., Overbeek, R., and Szeredi, P. 1993. Applications of the Aurora Parallel Prolog System to Computational Molecular Biology. In *International Logic Programming Symposium* (1993). MIT Press.

Lusk et. al., E. 1990. The Aurora Or-parallel Prolog System. *New Generation Computing 7*, 2,3.

Meier, M. 1991. Recursion vs. Iteration in Prolog. In *International Conference on Logic Programming* (1991). MIT Press.

Muthukumar, K., Bueno, F., de la Banda, M. G., and Hermenegildo, M. 1997. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*.

Muthukumar, K. and Hermenegildo, M. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming* (June 1991), pp. 49–63. MIT Press.

Palmer, D., Prins, J., and Westfold, S. 1994. Work-Efficient Nested Data-Parallelism. In *Frontiers of Massively Parallel Computing* (1994). McLean, VA.

Pettorossi, A. and Proietti, M. 1994. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming 19/20.*

Pfenning, F. 1992. *Types in Logic Programming.* MIT Press.

PONTELLI, E. AND GUPTA, G. 1995. Data And-parallel Logic Programming in &ACE. In *Proc. of 7th IEEE Symposium on Parallel and Distributed Processing* (1995).

PONTELLI, E. AND GUPTA, G. 1996. Nested Parallel Call Optimization. In *International Parallel Processing Symposium* (1996). IEEE Computer Society.

PONTELLI, E. AND GUPTA, G. 1997a. Implementation mechanisms for dependent and-parallelism. In *International Conference on Logic Programming* (1997). MIT Press.

PONTELLI, E. AND GUPTA, G. 1997b. Parallel Symbolic Computation with ACE. *Annals of AI and Mathematics*.

PONTELLI, E. AND GUPTA, G. 1998. Efficient backtracking in and-parallel implementations of non-deterministic languages. In *International Conference on Parallel Processing* (1998). IEEE Computer Society.

PONTELLI, E., GUPTA, G., AND HERMENEGILDO, M. 1995. &ACE: A High-performance Parallel Prolog System. In *International Parallel Processing Symposium* (April 1995), pp. 564–571. IEEE Computer Society: Santa Barbara, CA.

PONTELLI, E., GUPTA, G., PULVIRENTI, F., AND FERRO, A. 1997. Automatic compile-time parallelization of prolog programs for dependent and-parallelism. In *International Conference on Logic Programming* (1997). MIT Press.

PONTELLI, E., GUPTA, G., AND TANG, D. 1995. Determinacy Driven Optimizations of Parallel Prolog Implementations. In *Proc. of the Int'l Conference on Logic Programming 95* (1995), pp. 615–629. MIT Press.

PONTELLI, E., GUPTA, G., TANG, D., CARRO, M., AND HERMENEGILDO, M. 1996. Efficient Implementation of Independent And Parallelism. *Computer Languages 22*, 2/3.

RAMAKRISHNAN, V., SCHERSON, I., AND SUBRAMANIAN, R. 1995. Efficient Techniques for Fast Nested Barrier Synchronization. In *Seventh ACM Symposium on Parallel Algorithms and Architectures* (1995).

RAMKUMAR, B. AND KALE, L. V. 1989. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proc. NACLP'89* (1989), pp. 313–331. MIT Press.

ROY, P. V. 1993. *How to Get the Most out of Aquarius Prolog.* Digital Equipment Corporation.

SABOT, G. 1988. *The Paralation Model: Architecture Independent Parallel Programming.* MIT Press.

SANTOS COSTA, V., WARREN, D., AND YANG, R. 1991. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proc. 3rd ACM SIGPLAN PPoPP* (1991), pp. 83–93. ACM Press.

SARASWAT, V. 1989. *Concurrent Constraint Programming Languages.* Ph. D. thesis, Carnegie Mellon, Pittsburgh. School of Computer Science.

SHEN, K. 1992. *Studies in And/Or Parallelism in Prolog.* Ph. D. thesis, U. of Cambridge.

SMITH, D. 1993. Multilog: Data or-parallel logic programming. In *International Conference on Logic Programming* (1993). MIT Press.

SMITH, D. 1994. Why Multi-SLD beats SLD (even on a uniprocessor). In *LPAR* (1994). Springer Verlag.

TICK, E. 1994. Concurrent Logic Programs *a la mode.* In E. TICK AND G. SUCCI Eds., *High Performance Logic Programming Systems*. Kluwer.

VAN HENTENRYCK, P. 1988. *Constraint Handling in Prolog.* MIT Press.

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

WARREN, D. H. D. 1983. An Abstract Prolog Instruction Set. Technical Report 309, SRI International.

WARREN, R., HERMENEGILDO, M., AND DEBRAY, S. 1988. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming* (August 1988). MIT Press.

ZIMA, H. AND CHAPMAN, B. 1991. *Supercompilers for Parallel and Vector Computers.* ACM Press.