Data Parallel Logic Programming in &ACE*

Enrico Pontelli & Gopal Gupta
Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003-0001

Abstract

&ACE is a high performance Parallel Prolog System developed at the Laboratory for Logic, Databases, and Advanced Programming that exploits *and-parallelism* from Prolog programs. &ACE was developed to exploit *MIMD parallelism*. However, SPMD parallelism also arises naturally in many Prolog programs. In this paper we develop runtime techniques that allow systems that have primarily been designed to exploit MIMD parallelism (such as &ACE) to also efficiently exploit SPMD parallelism. These runtime techniques have been incorporated in the &ACE system. Performance of &ACE augmented with these techniques on programs containing SPMD parallelism is presented.

1 Introduction

Parallelism and Logic Programming Technology

Parallel processing of numerical problems has been an active area of research for almost two decades now [12, 22, 23]. There are parallelizing compilers and other tools available for parallel numerical processing that programmers can use without being experts on parallel processing. However, the same is not true of parallel processing of Symbolic and Artificial Intelligence (AI) applications. There has been some work [11, 19], but there are still no systems that are general enough and that programmers can use without being experts on parallel processing. &ACE [5] is a a system that is a step towards the goal of building a general automatic tool for parallelizing symbolic and AI applications. &ACE is based on the logic programming paradigm and the Prolog technology. A programmer wishing to parallelize his/her symbolic or AI program will code it in Prolog on a uniprocessor. The &ACE compiler and runtime system will then run this Prolog program automatically in parallel. Our parallelizing system supports the full Prolog language, complete with all its extra-logical (the cut), meta-logical (var, assert, retract, etc.), and side-effect (read, write, etc.) predicates that are sensitive to order of execution of the Prolog program.

Logic programming is a paradigm of programming where programs are expressed as logical implications. Logic programming languages are suited for a wide range of applications, from compilers to databases to symbolic applications, as well as for general purpose programming. The most popular logic programming language is Prolog. An important property of logic programming languages is that they are single assignment languages. Unlike conventional programming languages they disallow destructive assignment and explicit control information. Not only does this allow cleaner (declarative) semantics for programs, and hence a better understanding of them by their users, it also makes it easier for an evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can be executed in any order without affecting the (declarative) meaning of the program. In particular, these operations can be performed by the evaluator *in parallel*.

An important characteristic of logic programming languages, thus, is that parallelism can be exploited in an *implicit* way. This can be done directly by the program evaluator (the runtime system) as suggested above, or, alternatively, it can also be done by a parallelizing compiler. The task of the parallelizing compiler is essentially to unburden the evaluator from making run-time decisions regarding which parts of the program to run in parallel. Note that the program can also be parallelized by the user (through suitable annotations). In all cases, the advantage of-

*Some of the figures in this paper are in color for the convenience of the reviewers. They will be changed to black-and-white for the final paper, if accepted.

fered by logic programming is that the process is easier because of the more declarative and high level nature of the language. Furthermore, the exploitation of parallelism at run-time or parallelization at compile-time can be done quite successfully in an automatic way and without requiring any input from the user. Clearly, implicit exploitation of parallelism can in many cases have significant advantages over explicit parallelization.[1] In that sense, one can look in some ways towards Prolog for solving a new form of "(parallel) software crisis" that is posed to arise with the new wider availability of multiprocessors[2]—given systems, such as the one described in this paper, one can run Prolog programs written for sequential machines in parallel with no or minimal effort. For the rest of the paper we assume that the reader is familiar with Prolog and its execution model.

It must be pointed out that while the target application area of the &ACE system are Symbolic and AI applications, in fact, it can also be used for parallel execution of general purpose (non-numerical) programs. This stems from the fact that Prolog is an excellent language for writing problem solving programs.

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

1. *Or-parallelism* arises when more than one clause defines some predicate and a literal unifies with more than one clause head—the corresponding bodies can then be executed in parallel with each other, giving rise to or-parallelism.

2. *And-parallelism* arises when more than one goal is present in the query or in the body of a clause, and at runtime these goals are executed in parallel. And-parallelism can be further classified into *independent and-parallelism* and *dependent and-parallelism*. *independent and-parallelism* arises when bindings for the variables in the goals being parallelized are such that two or more goals are *independent* of one another, i.e., their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables. Parallel execution of such goals gives rise to (independent) and-parallelism. *Dependent and-parallelism* arises when two or more goals in the body of a clause have a common variable and are executed in parallel.

In this paper, for simplicity we will mainly concern ourselves with independent and-parallelism, although the results reported in this paper are just as applicable to systems that exploit dependent and-parallelism.

The main objective or this paper is to to show that there is no need of an specialized parallel implementations to take the advantage of SPMD parallelism[3] present in Prolog applications. Rather simple runtime techniques can allow parallel logic programming systems designed for exploiting MIMD[4] parallelism to also exploit data-parallelism with near-maximum efficiency. These techniques, devised by us, are very simple and are called Last Parallel Call Optimization (LPCO) and Nested Parallel Call Optimization (NPCO) respectively. They are analogs of the last call optimization found in modern sequential Prolog systems. LPCO and NPCO allow &ACE to execute SPMD programs with the efficiency of dedicated SPMD parallel logic programming systems such as Reform Prolog [10]. The efficiency of &ACE (augmented with LPCO and NPCO) for programs containing SPMD parallelism is demonstrated by presented performance figures. Thus, our research shows that there is no need of an specialized parallel implementations to take the advantage of data-parallelism present in Prolog applications. Rather simple runtime techniques can allow one to exploit data-parallelism efficiently in general purpose parallel Prolog systems.

The rest of this paper is organized as follows: Section 2 discusses MIMD and SIMD (SPMD) parallelism in general as well as in the context of logic programming. It also presents a critical survey of earlier efforts for exploiting SPMD parallelism in Prolog. Section 3 describes the parallel Prolog system called &ACE [5] that is designed to exploit MIMD parallelism from Prolog programs. Section 4 presents runtime implementation techniques, called Last Parallel Call Optimization (LPCO) and Nested Parallel Call Optimization (NPCO), that allow &ACE to execute SPMD programs with the efficiency of dedicated SPMD parallel logic programming systems such as Reform Prolog [10]. Section 4.3 presents several benchmarks that show the excellent performance of &ACE augmented with LPCO and NPCO for SPMD programs.

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

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

[3] The terms SPMD parallelism, SIMD parallelism, and data-parallelism are interchangeably used in the rest of this paper.

[4] The terms MIMD parallelism and control-parallelism are interchangeably used in the rest of the paper.

2 MIMD vs SIMD Parallelism

Broadly speaking, parallelism can be exploited in one of two forms: MIMD parallelism and SIMD (or SPMD) parallelism. MIMD parallelism arises when different operations are applied to different data-items in parallel. SIMD parallelism arises when the same operation is applied in parallel to more than one different data-items. Of course, SIMD-parallelism can viewed as a special case of MIMD parallelism, but the chief advantage of data-parallelism is that it incurs less implementation overhead. In a more general form SIMD is referred to as SPMD (Single Program Multiple Data) parallelism in which an identical program/function/procedure operates over different data items in parallel. SPMD parallelism is usually exploited in conjunction with aggregate data-structures such as arrays and lists. For example, a data-parallel operation applied to an array, will operate on each element of the array.

The forms of parallelism found in logic programs, namely, or-parallelism and and-parallelism, are usually regarded as MIMD parallelism. However, their SPMD analogs do exist. The main aggregrate data-structure found in Prolog is the list. Thus any program that operates over list can be cast as an SPMD program (though it should be remembered that logic programs operating over lists are not the only programs that can exploit SPMD parallelism). Consider the following program: (& denotes parallel conjunction)

```
map(_, [ ], [ ]).
map(P, [H | T], [Ho | To]) :-
         call(P(H, Ho)) & map(P, T, To).
```

In this program above the predicate `P` is being applied to every element of a list. If this program was executed in an MIMD parallel logic programming system then a nesting of parallelism equal to the length of the list will be produced, resulting in considerable space and time inefficiency, especially during backtracking. However, if this nesting could somehow be flattened, and the SPMD nature of the parallelism in the `map` predicate exploited, then considerable efficiency can result. Thus, given an inherently SPMD logic program it can always be treated as an MIMD logic program and parallelism can be exploited from it using a system that has been designed to exploit MIMD parallelism. However, the efficiency that comes about from SPMD parallelism is lost. The objective of this paper is to present runtime implementation techniques that allow a parallel logic programming system designed for exploiting MIMD parallelism to also exploit SPMD parallelism with nearly the same efficiency as that of dedicated SPMD systems.

2.1 SPMD in Prolog

In traditional programming languages SPMD is exploited by parallelization of repetitive operations on elements of arrays (*loop-parallelization*), or by supplying special features in the programming language (e.g., C^* and $^*LISP$) which provide built-in data-parallel operations that operate on complex data-structures whose components can be processed in parallel (e.g., arrays, lists).

In the area of logic programming, and more specifically of Prolog, most of the attempts for the parallelization of the language have focused on the exploitation of control parallelism. Nevertheless, it has been observed that programs written in Prolog often follow a quite well-defined pattern, in which structural recursion is applied to process the components of the input to produce the corresponding output (e.g. the `map` predicate above). Note that recursion is the main control structure available to programmers in Prolog. Thus, even iterative computations have to be expressed using recursion. Such recursive control structures are excellent candidates for data-parallel execution. For example, situations like the one depicted in the following program are common to a large majority of Prolog programs:

```
process_list([],[]).
process_list([X|Y],[Z|W]) :-
          process_element(X,Z),
          process_list(Y,W).
```

The exploitation of this form of parallelism, where all the iterations of the recursion are executed by different parallel threads, has also been named *Recursion Parallelism* in the literature [10].

Two main cases can be identified regarding direct exploitation of data-parallelism from recursive programs:

1. Deterministic Programs: if the iterations of the recursion are *deterministic*, i.e. they do offer at most one solution, then the problem is relatively simpler. It can be seen as analogous to the problem of automatically parallelizing loops in imperative programming languages, like FORTRAN [23].

2. Nondeterministic Programs: if some of the iterations admit multiple solutions, then the exploita-

tion of data-parallelism becomes harder, since other problems need to be tackled, like:

- bindings for a common variable coming from different iterations need to be compatible;
- a form of backtracking on computations spread across different processing units needs to be developed, guaranteeing that all the solutions will be found and respecting the desired semantics of the execution (e.g. Prolog semantics).

These problems have been proven to be hard to solve—and indeed many approaches presented disallow parallelization of nondeterministic computations.

2.2 Approaches to Data-Parallel in L.P.

2.2.1 Standard Control Parallelism

As we have mentioned above, standard MIMD parallelism can be employed to exploit parallelism from data-parallel programs (like the `process_list` above). In particular, many occurrences of data-parallelism in Prolog programs can be seen as special instances of and-parallelism, where and-parallelism is obtained by executing an iteration of the recursion in parallel with the recursive call.

In order to support this pattern of execution we require a minimal support for control and-parallelism. In particular, independent and-parallelism (like the one supported by systems like &ACE [5] and &-Prolog [7] and) is sufficient if the recursive call is independent from the outcome of the processing of one element of the recursive structure, otherwise a form of dependent and-parallelism is required (like DDAS [17]).

Nevertheless, the use of pure control parallelism to exploit data-parallelism leads to some undesired side-effects, in terms of space and time overheads.

Often, simple control parallelism (e.g., and-parallelism) will perform poorly on data-parallel programs, because they lack the ability of identifying the iterative nature of the execution and they maintain at the run-time the "nested" structure of the recursion (see fig. 1(i)). This causes an excessive overhead both in space (all the intermediate descriptors for the nested parallel calls exist simultaneously) and in time (due to the additional costs of traversing the deep and complex computation tree). In particular, these overheads become excessive when dealing with nondeterministic executions, in which the complex nesting of control structures needs to be traversed over and over while searching for alternative successful paths of computation.

Systems aiming to exploit data-parallelism from Prolog programs should remove any additional overhead from situations shown in Fig 1(i), and, possibly, also spawn different parallel threads for each level of the recursion right at the beginning itself. The ideal structure that should be realized is illustrated in figure 1(ii).

Note that in most cases data-parallelism present in Prolog is an instance of and-parallelism; a recursion over a recursive structure essentially specifies the application of a quantification to the operation applied to the elements of the data structure. In situations like `process_list` above the quantification is universal which translates to a conjunction of the iterations. In other situations (e.g., the classical definition of the `member` predicate for a list) the quantification is existential and translates to a disjunction of the iterations. This form of data-parallelism (with existential quantification) represents an instance of or-parallelism. However, in the rest of this work we will focus only on the and-parallel version of data-parallelism[5].

2.2.2 Compile time transformation

One of the first approaches to be considered for efficient exploitation of data-parallelism from Prolog programs is based on the use of traditional Control Parallel system (like &-Prolog) in conjunction with compile time transformations. The compile-time transformation was used used to achieve a computation whose structure approximates the ideal data-parallel one. The basic idea is quite simple: as mentioned earlier, the main drawbacks of systems that exploit control parallelism is that they unroll the recursion (and backtrack over) one step at the time, leading to inefficient distribution of parallel work. The compile time transformation aims at unrolling the recursion in some suitable way so that parallel work is available sooner and the granularity of parallelism is coarser (see later). In figure 2, to the left a normal control-parallel execution of a recursive program is shown, while to the right the same program is shown with a 'smarter' splitting of the iterations performed during its execution.

Two proposals in this area have been made that are extremely similar in nature.

Hermenegildo's Approach

[5] It should be noted that does not lead to any loss of generality since in most cases or-parallelism and and-parallelism are interconvertible [4].

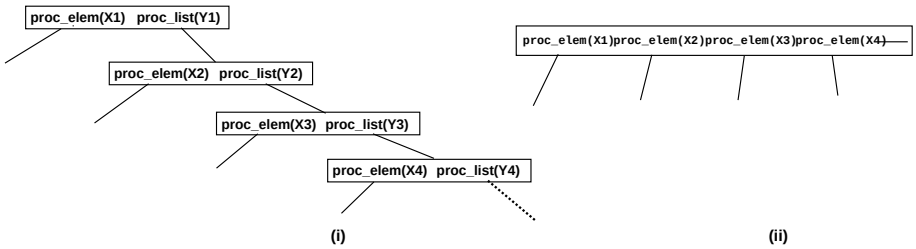

Figure 1: Nested Structure of Recursion in Control Parallelism

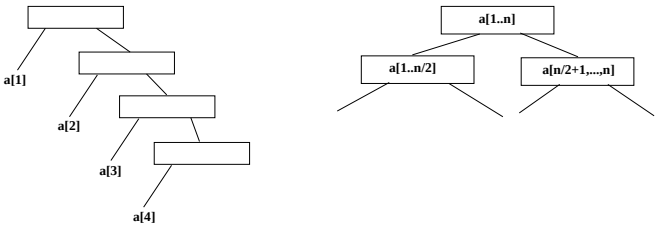

Figure 2: Ideal Compile time transformation

The first approach, taken by Hermenegildo's group [1], focuses on the transformation of list processing programs. Hermenegildo's group has proposed two techniques. The first is targeted to the generation of code which unrolls recursion over a list by N elements at the time, where N is a fixed constant. For example, the `process_list` code will be transformed as follows ($N = 4$):

```
vproc([H1,H2,H3,H4|R],[R1,R2,R3,R4|S]) :-
        !,
        vproc(R,S) &
        process_element(H1,R1) &
        process_element(H2,R2) &
        process_element(H3,R3) &
        process_element(H4,R4).

vproc([H|R],[HR|TR]) :-
        vproc(R,TR) &
        process_element(H,HR).
```

The second type of transformation is based on extending the host system (that exploits MIMD parallelism) with some constructs capable of dynamically splitting the list by a specified N.

Although these transformation can improve the behaviour of the parallel execution, we are still far from the ideal. In particular, to obtain the ideal with this technique we would need knowledge about the size of the recursive structure as well as knowledge about the availability of computational resources, information that is typically unknown and/or expensive to compute.

Debray's Approach

A similar approach has been recently proposed by Debray [2] and it applies to recursions where:

1. the depth is known;
2. the outcome of the recursion is obtained by applying an associative operator to combine the outcomes produced by each iteration.

Schematically, given a recursion like

```
p(X,Y) :-
        a(X), p(e(X),Y1),
        Y = d(X) ⊕ Y1.
p(X,Y) :-
        ¬a(X), b(X,Y).
```

where `e(.)` is the function applied to the recursion parameter and $\oplus$ is an associative operator, it is possible to obtain the following transformed program:

```
p(X,Y) :-
        q(X,1,size(X),Y1),
        Y = Y1 ⊕ e^size(X)(X).
q(X,N,N,Y) :-
        Y = d(e^M-1(X)).
q(X,M,N,Y) :-
        M < N, M1 = (M+N)/2, q(X,M,M1,Y1),
        q(X,M1+1,N,Y2), Y = Y1 ⊕ Y2.
```

Nevertheless, this sort of transformation can be effective only when the original recursion satisfies the desired properties (e.g., associative nature of the composition $\oplus$) and, most important, when the depth of the recursion is known in advance, which though always computable may be quite expensive to compute at runtime.

2.2.3 Data-parallel Logic Programming Systems

Given the limited applicability of the techniques based on converting data-parallelism to control-parallelism various approaches have emerged recently to directly exploit data-parallelism from recursive programs. This systems extract parallelism exclusively from structural recursions.

Their main advantage is that since they target a very specific form of parallelism which occurs only in well-behaved situations, they manage to extract parallelism with very low overhead. Furthermore, at least in theory, this approach is completely automatic and transparent to the user.

The most significant proposals in this class are represented by the *Bounded Quantification* approach [9] and the *Reform Prolog* system [10].

The Bounded Quantifications approach is based on the idea of supplying to the user a way of directly expressing iterations through the use of bounded quantifications (i.e., quantifications over a specified finite range). The quantification is then automatically compiled into the proper sequence of actions required to execute the different iterations in parallel. The idea is that a quantification like

$$\forall x\{\Theta[x] \rightarrow \Phi[x]\}$$

where $\Theta[x]$ identifies the (finite) range of x and $\Phi[x]$ is the predicate to be applied to each element of the iterations, can be reduced to the parallel execution of $\Phi[c_0], \ldots, \Phi[c_n]$, assuming that $\Theta[x] \leftrightarrow x = c_0 \vee \ldots \vee x = c_n$. Thus, a query `process_list(Lin, Lout)` will be transformed to:

$\forall$ `A` $\forall$ `I` { `Lin[I] = A` $\rightarrow$ `process_element(A,B)` $\wedge$ Lout[I] = B }

(note that the rules defining `process_list` will not be needed any more). The user is responsible for inserting the quantifiers in the program. All quantified computations will be executed in data-parallel.

The idea behind Reform Prolog is very similar to that of bounded quantification except that no special syntactic constructs are added to the language. Rather, static analysis is proposed as a mechanism to identify well-behaved (or, using Reform Prolog terminology, safe) recursions and produce code capable of:

- computing the depth n of the recursion;
- performing a single head unification, corresponding to the n head unifications performed in the n different iterations;
- executing these n iterations in parallel, each with the proper parameter;

This system has been proven to be quite effective, capable of efficiently extracting data-parallelism from recursive programs. Nevertheless, Reform Prolog has some major drawbacks:

1. it is limited only to recursive parallelism;
2. the use of compile time technology to identify all the source of parallelism can be a limiting factor—due to lack of exact information on the nature of the arguments (which are completely known only at run-time) the compiler may fail to identify good sources of parallel work.
3. the mechanism requires run-time knowledge of the depth of the recursion in order to unroll the whole recursion in a single step—this may constitute a considerable (and unavoidable) overhead.
4. Reform Prolog requires determinacy of the bindings made to variables which are shared between different iterations—this rules out the possibility of parallel execution for nondeterministic programs.

3 &ACE: A High Performance And-parallel System

In this section we briefly describe independent and-parallelism and how it is implemented. Foundations of much of the work described in this section were laid down in [3, 8], however, the specific implementation described is that of the &ACE system.

Conventionally, an and-parallel Prolog system works by executing a program that has been annotated with parallel conjunctions. These parallel conjunction annotations are either inserted by a parallelizing compiler [7] or hand-coded by the programmer. Execution of all goals in a parallel conjunction is started in parallel when control reaches that parallel conjunction. Whenever a parallel conjunction is encountered during execution, a data structure—the *parcall frame*—describing the parallel conjunction is allocated on the (control) stack. It contains various bookkeeping information (like number of subgoals in the conjunction), together with a descriptor—called *slot*—for each subgoal in the conjunction. At the same time appropriate data structures (e.g. work queue, termed goal stack in the logic programming literature) are initialized to allow remote execution of the newly generated subgoals (see Figure 3).

A processor taking a subgoal from a parallel conjunction will initially allocate an *input marker* on its

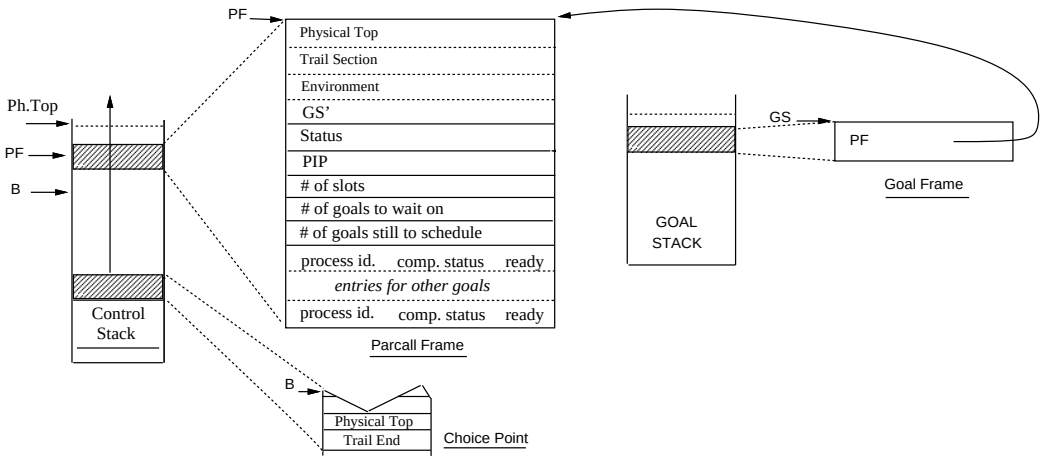

Figure 3: Data-structures needed for and-parallelism

stack, to identify the beginning of the section of computation dedicated to this subgoal, and start the corresponding computation. At the completion of the subgoal another marker, the *end marker*, will be allocated to close the section[6].

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 is such that all solutions are reported. One such backtracking semantics has been proposed by Hermenegildo and Nasr [7]: consider the subgoals shown below, where ',' is used between sequential subgoals (because of data-dependencies) and '&' for parallel subgoals (no data-dependencies): `a, b, (c & d & e), g, h`. Assuming that all subgoals can unify with more than one rule, there are several possible cases depending upon which subgoal fails: If subgoal `a` or `b` fails, sequential backtracking ("Prolog-like") occurs, as usual. Since `c`, `d`, and `e` are mutually independent, if either one of them fails before finding any solution, then a limited form of intelligent backtracking can be applied and backtracking can be forced to continue directly from `b`—but see further below. If `g` fails, backtracking must proceed to the right-most choice point within the parallel subgoals `c & d & e`, and recompute all goals to the right of this choice point (following Prolog semantics). If `e` were the rightmost choice point and `e` should subsequently fail, backtracking would proceed to `d`, and, if necessary, to `c`. Thus, backtracking within a set of and-parallel subgoals occurs only if initiated by a failure from outside these goals, i.e., "from the right" (also known as *outside backtracking*). If initiated from within, backtracking proceeds outside all these goals, i.e., "to the left" (also known as *inside backtracking*). When backtracking is initiated from outside, once a choicepoint is found in a subgoal p, an untried alternative is picked from it and then all the subgoals to the right of p in the parallel conjunction are restarted (in parallel).

Independent and-parallelism with the backtracking semantics described above has been implemented quite efficiently by the authors in the &ACE system. 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 [7][7]. &ACE is an extension to the sequential WAM (Warren Abstract Machine [20]) for and-parallel execution of Prolog programs with and-parallel annotation (such as CGEs [3]). 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 4 shows the speedups obtained on some of these benchmarks[8]. For

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

[7] Other implementations based on the principles of RAPWAM have also been proposed in the past, like &-Prolog [7] and DDAS [17].

[8] Data produce by running the system on a Sequent Symmetry Multiprocessors.

further details regarding the structure of &ACE and its performance the reader is referred to [6, 5].

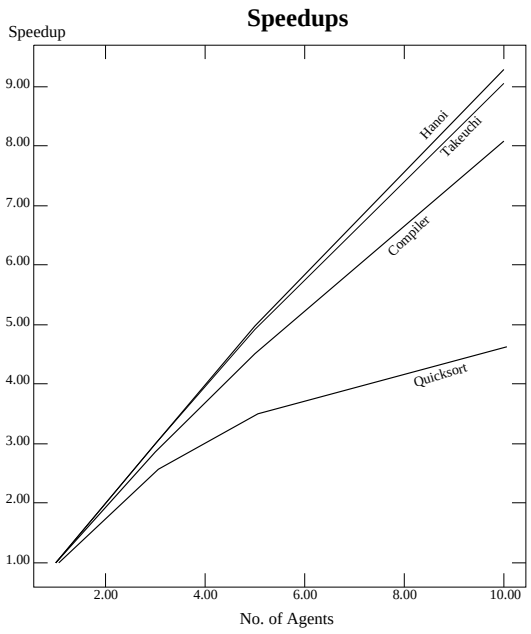

Figure 4: Speedups of &ACE

4 Last Parallel Call Optimization

In this section we present the last parallel call optimization that allows us to execute data-parallel programs efficiently in the &ACE system. The intent of the Last Parallel Call Optimization (LPCO) is to merge, whenever possible, distinct parallel conjunctions. Last Parallel Call Optimization produces the following advantages in an and-parallel system:

- *(i)* It speeds up forward execution by avoiding allocation of certain parcall frames (and, eventually, triggering other optimizations — like the shallow parallelism one);
- *(ii)* It speeds up the process of killing computations during an inside backtracking;
- *(iii)* It speeds up the process of backtracking, in general;
- *(iv)* It saves space on the stacks and allows earlier recovering of space on backtracking.

The advantages of LPCO are very similar to those for last call optimization [21] in the WAM. The conditions under which the LPCO applies are also very similar to those under which last call optimization is applicable in sequential systems.

Consider first an example that covers a special case of LPCO: `?- (p & q).` where

```
p :- (r & s).        q :- (t & u).
```

The and-tree constructed is shown in Figure 5(i). One can reduce the number of parcall nodes, at least for this example, by rewriting the query as `?- (r & s & t & u).` Figure 5(ii) shows the and-tree that will be created if we apply this optimization. Note that executing the and-tree shown in Figure 5.(ii) on RAPWAM will require less space because the parcall frames for `(r & s)` and `(t & u)` will not be allocated. The single parcall frame allocated will have two extra goal slots compared to the parcall frame allocated for `(p & q)` in Figure 5(i). It is possible to detect cases such as above at compile time. However, our aim is to accomplish this saving in time and space at runtime. Thus, for the example above, our scheme will work as follows. When the parallel calls `(r & s)` and `(t & u)` are made, the runtime system will recognize that the parallel call `(p & q)` is immediately above and instead of allocating a new parcall frame some extra information will be added to the parcall frame of `(p & q)` and allocation of a new parcall frame avoided. The extra information added will consist of adding slots for the goals `r`, `s`, etc. Note that no new control information need be recorded in the parcall frame of `(p & q)`. However, some control information, such as the number of slots, etc., need to be modified in the parcall frame of `(p & q)`); it is also necessary to slightly modify the structure of a slot in order to adapt it to the new pattern of execution[9].

Note also that if the goal `r` is to fail in inside mode, then in case (ii) (Figure 5(ii)) killing of computation in sibling and-branches will be considerably simplified. In case (i) the failure will have to be propagated from parcall frame f2 to parcall frame f1. From f1 a kill message will have to be sent out to parcall frame f3.

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 are not sufficient static information to detect the determinacy of `p` and `q`, then the compile-time analysis will not be able to detect the eventual applicability 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 program:

[9] For example it is necessary to keep in each slot a pointer to the environment in which the execution of the corresponding subgoal should start.

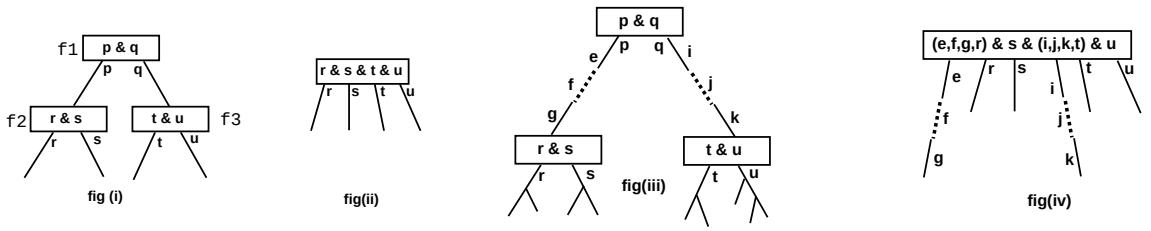

Figure 5: Reusing Parcall Frames

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

?-process_list([1,2,3,4], Out).
```

In such a case, compile time transformations cannot unfold the program to eliminate nesting of parcall frames because it will depend on the length of the input list. However, using our runtime technique, given that the goal `proc_list` is determinate, nesting of parcall frames can be completely eliminated (Figure 6). 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.

Next we present the most general case of LPCO. This arises when there are goals preceding the parallel conjunction in a clause that matches a subgoal that is itself in a parallel conjunction (figure 5.(iii)). Thus, given a CGE of the form: `(p & q)` where

```
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.

These conditions can be more easily understood by looking at 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.

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)`[10]. Thus, if we determine at the time of the parallel call `(r & s)` that `e, f,` and `g` are determinate then we pretend as if the clause for `p` is defined as `p :- ((e,f,g,r) & s).` This is illustrated in Figure 5(iv).

Note that the two determinacy conditions above require that when the parallel conjunction is encountered at the end of clause for `p` then there are no intervening choicepoints between the parcall frame for `(p & q)` and the current point on the stack. Thus even though goal `p` (resp. `q`) was not determinate in the beginning, the determinacy conditions will be satisfied when the last matching clause for `p` (resp. `q`) is tried. LPCO can be applied at that point. This is akin to last call optimization in sequential systems when even though a goal is not determinate, last call optimization is triggered when the last clause for that goal is tried. 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. When outside backtracking takes place in the tree in Figure 5(iv), because of the organization of the parcall frame, backtracking will proceed through `u`, `t`, `i,j,k` (without finding here any further solution since, by hypothesis, `i,j,k` represents a deterministic computation) and so on. Backtracking over `i,j,k` will be immediate (since no choice points are present); the presence of a slot in the parallel computation descriptor dedicated to this deterministic part of the execution may seem superfluous, but it is actually necessary in order to guarantee a proper unwinding of the bindings created[11]. A possible optimization in this framework is briefly sketched in section 4.4.

Suppose now an untried alternative is found within `s`, then the subgoals on the right of `s` have to be restarted. In this case the whole computation of `p` will be reactivated. This examples shows one of the focal point in the implementation of the LPCO, the need to maintain a *backtrackable* description of the

[10] Although, in our case, the eventual bindings generated by `e, f, g` should be produced *before* starting the execution of `s`

[11] And, additionally, simplifies the management of the local stack

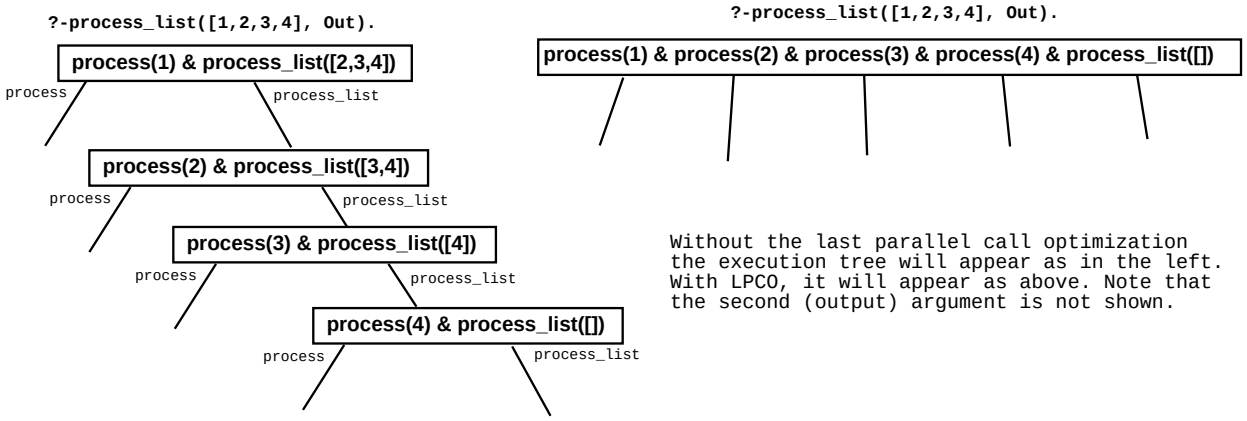

Figure 6: Reuse of Parcall Frames for Recursive Programs

subgoals associated to a given parallel call. In the example above, once we have backtracked over `i,j,k` we need to *undo* the application of the LPCO, removing the new subgoals introduced (`i,j,k`, `r`, and `s`) and restoring the previously existing one (`q`). This can be avoided only if we have further evidence that actually only one clause (the one indicated above) matches subgoal `q`.

Generalizations of the LPCO to more complex cases (like those when the clause used has a continuation, e.g. `p :- e, f, g, (r & s), h`) are analyzed in section 4.4.

4.1 LPCO and Data Parallelism

A precondition for applying data-parallelism is that goals to be tried in data-parallel should be all available at once. The LPCO has the effect of flattening the nested control structure created in the control stack by MIMD parallel execution. This results in all goals that can be executed in parallel being collected at the same level of nesting, thus lending themselves to data-parallel execution. It should be noted that the nested control structure has to be traversed once to transform it to a flat control structure, but once this has been done, all operations (including backtracking) can be done in a data-parallel way resulting in considerable saving. Because of this requirement of having to traverse the nested control structure once (during recursion unrolling at the time of execution) the performance of our system is slightly worse than dedicated data-parallel systems such as Reform Prolog (of course, our system can parallelize and speed-up a much larger range of Prolog programs than the Reform system can). The overhead of having to traverse the nested control structure once is inevitable if we are to rely on purely runtime techniques because the data-parallelism is hidden in the recursive program which has to be unraveled. This overhead, linear in the size of the nesting structure, can be reduced to a factor of *logn* if Debray's approach (Section 2) is adopted at compile time to transform linearly recursvie structure to a dichotomous one (see Figure 2).

The LPCO illustrates an important principle of optimization of parallel systems, termed the *reduced nesting principle* [14] that states:

> *The level of nesting of control structures in a computation should be reduced whenever possible.*

4.2 Implementation of LPCO

To implement LPCO, the compiler will generate a different instruction when it sees a parallel conjunct at the end of a clause. This instruction, named `opt_alloc_parcall`, behaves the same as the `alloc_parcall` instruction of the &ACE[12], except that if the conditions for LPCO are fulfilled last parallel call optimization will be applied.

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., behaving like a normal `alloc_parcall` instruction). The above mentioned check is immediate and does not introduce any relevant overhead—it is sufficient to check

[12] The instruction used to create parallel conjunctions

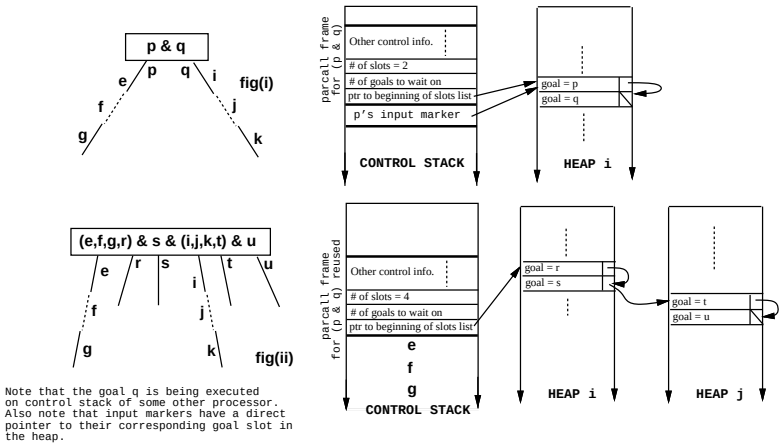

Figure 7: Allocating Goal Slots on the Heap

that the data structure currently lying on the top of the stack is a parcall frame or an input marker.

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 incremented by n in the enclosing parcall frame (this operation should be done atomically).

Introducing the LPCO in the &ACE system requires only one relevant change in the architecture. In the original &ACE (as in RAPWAM, DDAS, etc.) 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 7). 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 7. Note that the modification of the slot list will have to be an atomic (backtrackable) operation. 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 operation on the slots requires a linear scanning 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 often 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.

4.3 Experimental Results

The LPCO optimization has been incorporated in the current version of the &ACE and-parallel system. A first comment that is to be made is the limited amount of time and work that was needed to add the LPCO to the existing implementation, thanks to its inherent simplicity. Introducing the LPCO took only a week of work and we strongly believe that porting it to different and-parallel systems will not require any larger effort.

The experimental tests that we have performed consist of running various benchmarks, measuring the execution time and the corresponding memory consumption (a factor that has been proven to be of very im-

portant for parallel logic programming systems [18]). In particular we selected the benchmarks in order to separately study the effects of the LPCO on programs whose execution is

- ◇ pure forward execution (i.e. no backtracking);
- ◇ mainly backward execution.

The benchmarks adopted are mainly classical benchmarks commonly used in parallel logic programming. `Matrix` is matrix multiplication program, `Pderiv` is a benchmark performing symbolic differentiation, `Poccur` is a search of an element in nested lists, `BTCluster` is part of the clustering program using by British Telecom, `Annotator` is the core of the parallelizing compiler, and finally `Map` is program which applies a function to the elements of a list[13].

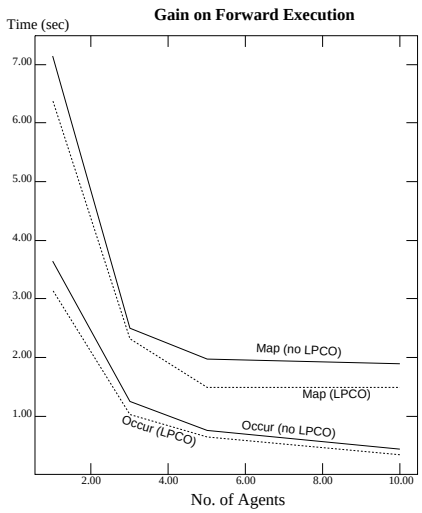

Figure 8: Gain in execution time (in ms.) during Forward Execution

The performance on these benchmarks is extremely good. On most of the benchmarks we managed to obtain very good speedups, often close to linear. Table 1 illustrates the execution times (in ms.) obtained by running our system on a Sequent Symmetry Multiprocessor; relative speedups are indicated in parenthesis.

Note that the results obtained for data-parallel benchmarks shown in this section are considerably better than those for general purpose benchmarks (shown in Figure 4). This underscores the well known fact that data-parallelism, whenever available, can be exploited much more efficiently than control-parallelism. The use of LPCO allows to avoid the allocation of a considerable number of data structures

[13]Two versions have been used, the one indicated with [1] makes use of a large granularity function, while the one indicated with [2] uses a fine granularity function.

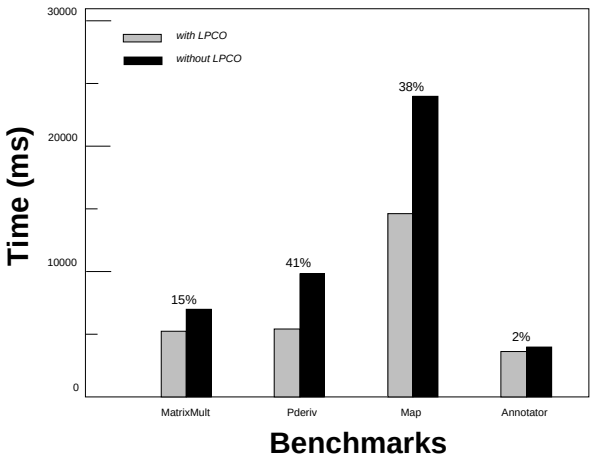

Figure 10: Gain in Execution time using LPCO on programs with Backtracking

(i.e., markers and parcall frames) reducing the complexity of the structure of the computation. In particular, the consistent application of LPCO in many situations leads to exposure of the deterministic nature of the parallel subgoals (otherwise hidden by the complex nesting of parallel calls), and this considerably simplifies the execution pattern—since there is no need to keep track of backtracking-related information for these subgoals. A graphical representation of the speedup curves is presented in figure 9.

A couple of benchmarks (e.g., `BTCluster` and `Map`[2]) show a slight reduction of speedup for a higher number of processors: this is not due to any shortcoming in the implementation but it derives from:

- lack of sufficient parallel work;
- fine granularity of the parallel tasks;

The presence of the LPCO during the execution of these benchmarks is important. First of all it allows some savings in execution time, since the allocation of many control structures is avoided. These savings can vary, with respect to a standard MIMD and-parallel execution, from program to program. In certain benchmarks (e.g., `Pderiv`) these savings are almost negligible, since the nesting of the parallel computation is very shallow, and the actual cost of applying the optimization (i.e., modifying an existing parallel call, generating the new slots) ends up being very close to the cost of actually opening a new parallel call. In other benchmarks the savings are more relevant. Table 2 and figure 8 illustrate the savings obtained on a some benchmarks.

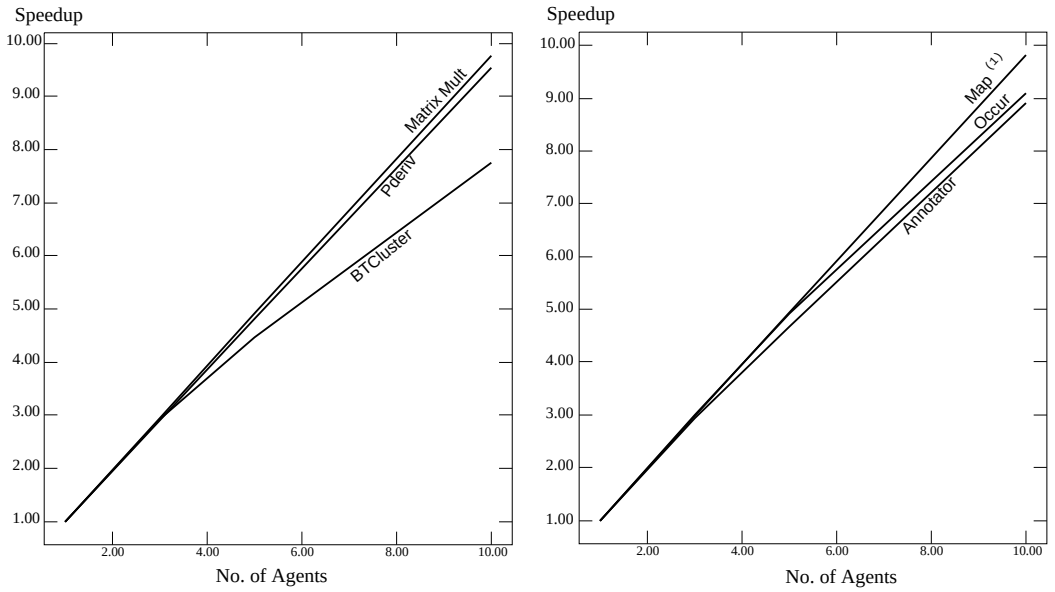

Figure 9: Speedup curves for Forward Execution

Goals executed	&ACE agents 1	3	5	10
matrix_mult(30)	5214	1768	1059	534
pderiv	5145	1766	1067	539
bt_cluster	1320	450	296	170
annotator(15)	3902	1330	834	438
poccur(5)	3146	1020	640	346
$map^1(2000)$	14517	4885	2936	1480
$map^2(5000)$	6396	2318	1482	1480

Table 1: Execution times in msec of Data Parallel Benchmarks

More interesting are the results for memory consumption. On most of the examples considered we managed to cut the number of parcall frames down to one, i.e. only a main parcall frame is created and all the nested ones are removed. Table 3 indicates the percentage of memory required to execute some benchmarks using LPCO with respect to the corresponding unoptimized execution. A considerable amount of memory is also saved because of (as mentioned earlier) deterministic subgoals get exposed by the flattening of the computation tree, and their determinacy can be used to avoid allocation of markers and storing of backtracking-related information. Table 3 summarizes these results: for each entry the memory consumption in terms of $\langle$ *No. of Markers, No. of Parcall Frames* $\rangle$ is reported, both for unoptimized and optimized execution. The second line of the table instead measures the percentage of improvement in stack consumption obtained introducing the LPCO. The gainings in memory consumption are graphically described in figure 11.

The results are even better when dealing with programs involving backward execution[14]. The presence of the LPCO allows a saving in execution time, generally proportional to the depth of the nesting of parallel calls. Some results are illustrated in table 4. On some of the benchmarks the improvement is not very considerable (for example for the British Telecom clustering benchmark we have an improvement in execution time of 6%, and for the `occur` benchmark this goes down to 3%), due to the shallow nesting of parallel calls (50 or 60) and to the predominance of the actual

[14]Backtracking in the benchmarks has been forced by adding a `fail` at the end of the query.

Goals executed	&ACE agents 1	3	5	10
map²(5000)	7149/ 6396	2515 / 2318	1991 / 1482	1912 / 1482
poccur(5)	3651/3146	1255/1020	759/640	430/346

Table 2: Savings in Execution time (w.r.t. MIMD And-parallel exec.)

Goals executed	&ACE Execution BTcluster	Deriv	Occur(5)	Map²(2000)	Matrix(40)
Markers/Parcall	119/60 — 1/1	134/87 — 0/1	95/50 — 40/1	4000/2000 — 1/1	1638/1599 — 0/1
% Improvement	49%	45%	34%	48%	39%

Table 3: Memory Usage (Markers/Parcall-Frames for Unoptimized — Optimized)

computation time over parallel overhead. For other benchmarks (like `deriv`) the results are considerably better.

The speedup curves obtained on the benchmarks containing backtracking are plotted in figure 12. It is important to observe that for various benchmarks the purely MIMD execution is not able to show speedups (because of the intense interactions required to propagate backtracking between the different computing agents), while the use of LPCO leads to considerable speedups. Gains in execution time are compared in figure 10.

Interesting results can be also seen by examining the effect of *inside failures* during execution: the use of LPCO should allow a further improvement; the presence of a unique parcall frame should in fact allow to considerably reduce the delay of propagating killing signals along the computation tree. Table 5 shows the result obtained causing an inside failure during the execution of the matrix multiplication benchmark.

4.4 Extensions to the basic LPCO

As we mentioned in the previous sections, LPCO can be applied whenever certain conditions on the determinacy of given parts of the computations are met. The obvious question that comes to mind is whether these conditions can be relaxed and what would eventually be the cost of this relaxation. Three different cases can be analyzed:

- LPCO cannot be applied whenever a nondeterministic computation has been performed between the two nested parallel calls, e.g. in `(p & q)` we are using a clause `p :- e , (f & g)` where `e` has multiple solutions. Extending LPCO to these cases is possible but it requires some more involved changes in the slot list management and in the way in which backtracking is performed. The main issue here is related to inside backtracking: if we apply the optimization to the case above we will obtain an *expanded* parallel call `( (e, (f & g)) & q )` where different slots of the parcall frame will be assigned to `e`, `f`, `g`, and `q`. If the execution of `f` fails, this time we are not allowed to apply inside backtracking and abort the whole parallel call (`q` included), since there could be other solutions for `e` which may cure the failure. Anyway a more sophisticated representation of the list of goals (in which the information about the nesting of parallel conjunctions are not completely lost) and a slight improvement in the backtracking mechanism are sufficient to allow extension of the LPCO to these cases.
- LPCO cannot be applied whenever a computation is present in the continuation of the nested parallel calls. The main problem in this case is related to respecting the desired order of execution: if a clause like `p :- (r & s), h` is used in solving `(p & q)` and the LPCO is applied, we need to make sure that the execution of `h` is not started before both `r` and `s` have completed. A safe possibility would be to delay all the continuations of the nested parallel calls until the main parcall has completed. This anyway pose some problems during outside backtracking (the necessity of identifying which continuations should be re-executed and which not). A scheme for dealing with this extension (in case the continuations in the innermost parallel calls are deterministic) has been developed and is currently in its implementation stage.

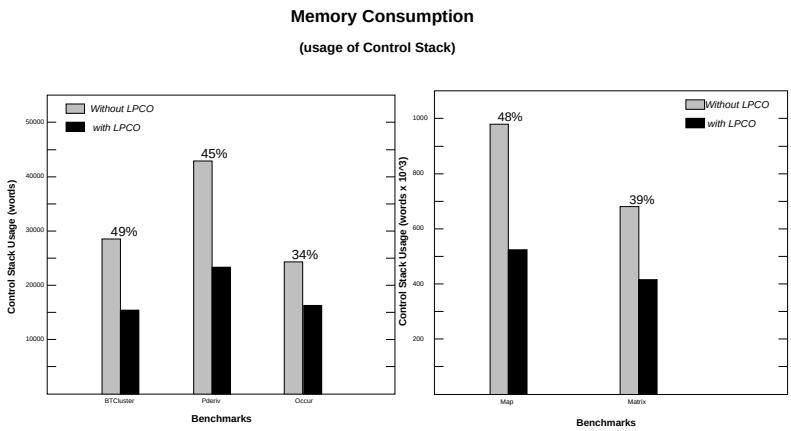

Figure 11: Improvement in Control Stack usage

Goals executed	&ACE agents 1	3	5	10
matrix_mult(30)	6302/5364	2739/ 1896	2056/ 1224	1540/ 705
pderiv	9493/ 5618	5881/ 2756	5195/ 2346	6672/ 2342
map[1]*(2000)*	24210/ 14988	14014/ 5206	12246/ 3233	10739/ 1762
annotator(15)	3949/ 3862	1353/1346	884/ 870	497 / 476

Table 4: Backward Execution

- given a query `:- p & q`, let us assume that `q` has been reduced using the clause `q :- a, (b & c)`$^{(*)}$ and LPCO has been applied. During outside backtracking, once the slot for `q` is reached, an action to restore the original slot list needs to be performed (as described in the previous sections). On the other hand, if we have knowledge that the clause $^{(*)}$ is the only one that could be used to solve `q`, then we could consider the possibility of avoiding the recomputation of `a`, by simply
  - ◇ avoiding untrailing the bindings associated to `a` (information stored in the slot of `q` at the moment of application of LPCO);
  - ◇ avoiding restoring the original slot list (i.e., keeping `b` and `c` in the slot list).

  This simple further optimization may save considerable amount of work — whenever the computation `a` is very large.

The introduction of these extensions allows to extend the applicability of this model to a considerable larger set of programs. This extended form of Data-parallelism management has been named *Nested Parallel Call Optimization* (NPCO) and it has been already partially implemented in &ACE [13].

5 Related Work and Conclusions

Ramkumar's distributed last call optimization and adaptations of last call optimization to committed choice languages are two efforts in a direction similar to LPCO. Neither one is motivated by a desire to exploit data-parallelism efficiently.

- Ramkumar's distributed last call optimization: The only work that one could think of as coming close is that of Ramkumar [15]. In it he describes what he calls *distributed last call optimization*, designed for his and Kale's ROPM system [16]. This optimization is specific to process based systems (like ROPM) and its main objective is to reduce the message flow between goals during parallel executions. The main ideas is that, if a goal g calls a goal g' which in turns calls a goal g", then the solutions produced by g" can be directly communicated to g, if g" is the last call in g' and g' is the last call in g. In this way the communication through g' is avoided. The sole aim of the last distributed call optimization is to reduce message passing traffic in the multiprocessor system — so that its aim, scope, as well as its results are quite different from the traditional last call optimization or from our last parallel call optimization.

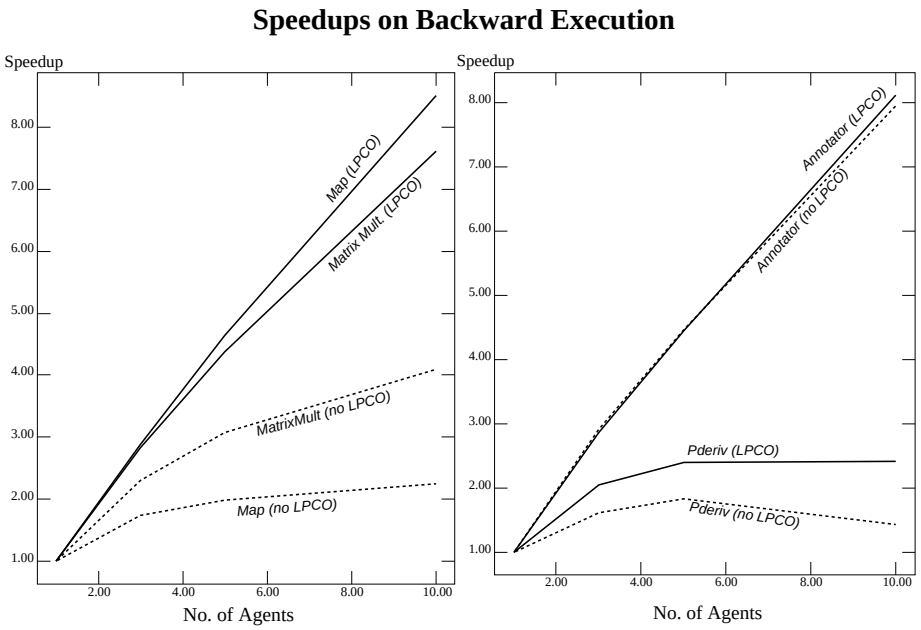

Figure 12: Speedup Curves for Backtracking executions (with and without LPCO)

Goals executed	&ACE Execution	
	Unoptimized	Optimized
pm-inside-fail-first	5346	3100 (42%)

Table 5: Unoptimized/Optimized Execution times in msec (single processor)

- It should be noted that another notion of 'call optimization' is present in committed choice languages (like Parlog). This optimization has quite a different nature from our LPCO: whenever a subgoal p commits to a certain clause, instead of spawning n new processes (one for each element of the body of the clause), it spawns only $n-1$ while one of the clause's subgoals (typically the last one) is automatically executed by the same process running p. Clearly the scope and aim of this optimization are different from those of LPCO.

In this paper we presented runtime techniques for allowing efficient execution of data-parallel Prolog programs in parallel Prolog systems primarily designed for MIMD parallel execution. These runtime techniques developed as optimizations result in considerable efficiency both in time and space. The techniques have been incorporated in an existing parallel Prolog system developed by us for the Sequent Symmetry, called &ACE, and have shown excellent results.

Acknowledgements

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

References

[1] M. Carro and M. Hermenegildo. A Note on Data-Parallelism and (And-Parallel) Prolog. In *ICLP94 Workshop on Parallel and Data Parallel Execution of Logic Programs*, 1994.

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

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

[4] G.Gupta E. Pontelli. On the Duality Between Or- and And-Parallelism in Logic Programming. Technical report, New Mexico State University, 1995.

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

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

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

[8] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Proc. 3rd ICLP, LNCS 225*, pages 25–40. Springer-Verlag, 1986.

[9] H. Millroth J. Barklund. Providing Iteration and Concurrency in Logic Program through Bounded Quantifications. In *Proc. of the Conf. on Fifth Generation Computer Systems*, 1992.

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

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

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

[13] E. Pontelli and G. Gupta. Nested Parallel Call Optimization. Technical report, New Mexico State University, 1994.

[14] E. Pontelli, G. Gupta, and D. Tang. Determinacy Driven Optimizations of Parallel Prolog Implementations. In *Proc. of the Int'l Conference on Logic Programming 95*. MIT Press, 1995.

[15] B. Ramkumar. Distributed Last Call Optimization for Portable Parallel Prolog Systems. *ACM Letters on Prog. Languages and Systems*, 3(1), 1992.

[16] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proc. NACLP'89*, pages 313–331. MIT Press, 1989.

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

[18] K. Shen and M. Hermenegildo. A Flexible Stack Memory Management Scheme for Non-Deterministic, And-parallel Execution of Logic Programs. Technical report, Facultad de Informatica, U. P. Madrid, 28660 Boadilla del Monte, Madrid, Spain, January 1993.

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

[20] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[21] D.H.D. Warren. An Improved Prolog Implementation Which Optimises Tail Recursion. In *Proc. of the 2nd ICLP*. Academic Press, 1984.

[22] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.

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