Optimization Principles for Parallel Implementation of Nondeterministic Languages and Systems*

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

Abstract

Naive parallel implementation of nondeterministic systems (such as a theorem proving system) and languages (such as a logic, constraint, or a concurrent constraint language) can result in poor performance. We present four optimization principles—the flattening principle, the procrastination principle, the sequentiality principle, and the reuse principle—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 principles is illustrated by applying them to the ACE parallel logic programming system. Performance data presented shows that considerable improvement in performance can result.

Keywords: Non-determinism, Logic and Constraint Programming, And-parallelism, Or-parallelism, Optimizations.

1 Introduction

Non-determinism arises in many areas of computer science. Artificial intelligence and constrained optimization are two such areas where non-determinism is commonly found. By non-determinism we mean 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) [8], concurrent constraint languages (e.g AKL) [21], and rule based languages (e.g. OPS5) [4] being some of the salient examples.

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

1. Or-parallelism: the different potential solutions can be explored in parallel (i.e. given a subgoal, there may be more rules that can be used to reduce it).

2. And-parallelism: while looking for a specific solution, the different operations involved can be executed in parallel (i.e., conjunctive subgoals may each be reduced in parallel).

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

Solution of problems with non-determinism has been abstractly represented using *and/or trees*. Or-parallelism can be exploited by exploring the branches emanating from an *or-node* of this tree in parallel. Likewise, and-parallelism is exploited by exploring the branches emanating from *and-nodes* in parallel. Thus, the parallel processing of non-deterministic problems can be abstracted as the problem of exploring (or constructing) the corresponding and-or tree 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

*Some figures in this paper are in color for convenience of the reviewers; these will be changed to black and white in the final paper, if accepted. The research is supported by NSF grants CCR 92-11732, HRD 93-53271, and INT 95-15256, by NATO Grant CRG 921318, and by a fellowship from Phillips Petroleum to Enrico Pontelli.

naive parallel implementation may lead to a slow down, or, may incur a severe overhead compared to a corresponding sequential system. Excessive parallel overhead may cause a naive parallel system to run many times slower on one processor compared to a similar sequential system.

This paper presents a number of general optimization principles 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 and *systematically* apply our principles, considerably reducing the parallel overhead in the process, and obtaining a more efficient system. Reduction of parallel overhead results in improved *sequential efficiency* (performance of the parallel system on one processor) of the system. It should be noted that the objective of parallel execution is to obtain better *absolute performance* and not just better speed-ups.[1] This implies that it is absolutely imperative that the sequential performance of a parallel system be very close to that of state-of-the-art purely sequential systems. Systematic application of optimization principles presented in this paper can take one close to this goal. While these optimization principles 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), some of them can also be applied to deterministic systems and languages (e.g. C or Fortran).

The optimization principles we present are arguably not new, and have been applied in the past in various situations to optimize execution. However, ours is the first attempt at presenting them in their general form and showing how they can be *systematically* applied to improve execution performance. We developed these principles and applied them to the *ACE system* [14], a high performance parallel Prolog system being developed by us, which resulted in vast improvement in its performance. The concrete performance improvement figures for the ACE system are presented as a testimony to the effectiveness of these optimization principles (note that all performance figures in this paper are on the Sequent Symmetry Multiprocessor). As a result of these optimizations, in many applications the parallel overhead in ACE was reduced to less than 2%, a remarkably small overhead given the enormous complexity of and-or parallel systems (for instance, in the ROPM parallel Prolog system [18], in absence of any special granularity control measures, the parallel overhead can be 200% to 300%). These optimization principles are intended to be applied at runtime (rather than at compile-time), therefore, the benefits accrued have to be balanced against the cost of applying them. We believe that, in general, the benefits accrued will be more than the cost of applying these optimizations; at least that what is borne out by our experiments with the ACE system, where the cost is limited to very simple runtime checks.

In the rest of the paper, we take logic programming systems as representatives of non-deterministic systems. Thus, we present our optimizations principles and their applications in the context of logic programming, though they apply equally well to parallel implementations of arbitrary non-deterministic systems. The rest of the paper is organized as follows: Section 2 describes the and-or tree, the commonly used structure for abstractly representing parallel execution of non-deterministic systems/languages, and presents insights into its nature and construction process. Sections 3, 4, and 5 present each of the principles and their applications. Finally, Section 6 presents conclusions.

2 And-Or Tree

Parallel execution of a logic programming system can be viewed as parallel traversal of an and-or tree. An and-or tree has or-nodes and and-nodes. Or-nodes are created when there are multiple ways to solve a goal[2]. An and-node is created when a goal invokes several conjunctive subgoals. The or- and and-nodes represent *sources of parallelism*, i.e. points of the execution where it is possible to fork parallel computations. Figure 1 illustrates an and-or tree for a logic program ('&' in the program stands for parallel conjunction). Note that the and-nodes and the or-nodes may be descendents of each other during parallel execution of the and-or tree. This results in nesting of and-parallelism and or-parallelism within each other, making the management and implementation of parallel execution enormously complex[3]. The complexity is exacerbated by the fact that non-deterministic systems admit backtracking, and incorporation of backtracking implies that a computation should be restorable to every point where a choice was made. In the

[1] Where speedup is defined as time taken on 1 processor divided by time taken on n processors.

[2] In logic programming terminology or-nodes are termed choice points.

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

context of and-or parallelism this means saving extra information and allocation of extra data-structures, so that in the event of backtracking the state of the computation can be restored.

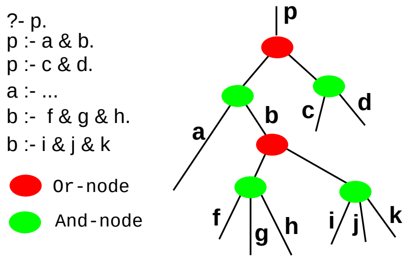

Figure 1: And-Or parallel Tree

2.1 Overheads of Parallel Exploration

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

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

• 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 computing agents. These traversal and synchronization activities are absent from a sequential computation (in which the management of nondeterminism is reduced to a linear and fast scan of the branches in a predetermined order).

2.2 Reducing Overheads in And-Or Tree Traversal

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 we can improve the activity of traversing the complex structure of a parallel computation:
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 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 Structures 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 a double purpose: *(i)* support the management of the parallel parts of the computation; *(ii)* support the management of nondeterminism.

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

These observations can be concretized into four optimization principles that are presented in the following sections. Note that the optimization principles we present are not generally applicable to deterministic languages and systems because deterministic systems do not have to perform the two operations that our optimization principles seek to eliminate: (i) traversal of an and-or tree; and, (ii) leaving state information behind so as to be able to backtrack. Before describing our optimization principles we give a brief introduction to the ACE system next, being this the testbed for our optimization techniques.

2.3 The ACE System

ACE is an and-or parallel system for full Prolog developed by the authors [14, 6, 13]. The ACE system has been implemented by extending the SICStus Prolog system. For the purpose of illustrating the optimization principles we need only understand the and-parallel component of ACE (called &ACE). &ACE executes *independent* goals in and-parallel, i.e., conjunctive goals that are determined to not influence each other's execution *at runtime* are run in and-parallel. Programs supplied to the systems are annotated in order to make parts of the program that can be safely run in and-parallel explicit (through the use of the "&" parallel conjunction operator; the ',' indicates the usual sequential conjunction). Note that independence is a runtime property. This annotation has to be either done manually by the programmer, or automatically by an abstract-interpretation based parallelizing compiler (the &ACE system uses the latter [10]). 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*) of the conjunction is created (containing also a descriptor, called *slot*, for each subgoal belonging to the parallel call) and the subgoals are made available to the other computing agents for parallel execution. Each agent that selects a subgoal for execution will initially allocate a *marker* on its control stack to indicate the beginning of a new execution; analogously, a marker will be allocated to mark the end of the subgoal's execution. These marker are required in order to guide backtracking activity on the parallel call in the case of failure.

The &ACE system is fully operational, has very high sequential efficiency, and has shown excellent speed-ups on a variety of benchmarks including some a couple of thousand lines long [14]. The unoptimized &ACE engine incurs an average parallel overhead of 10% to 25% compared to sequential SICStus Prolog.

3 Simplification of the Computation 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 nondeterminism (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., choice points in a control stack). In parallel systems, by contrast, the computation structure gets distributed across the stacks of multiple processing agents.

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. This observation inspires the *flattening principle.*

Flattening Principle The flattening principle can be simply 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 principle helps in reducing the levels of nesting of and- and or-parallelism thereby reducing overhead. Of course, the flattening principle should be applied 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 principle 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 [23]) is an instance of the flattening principle where nested recursive calls get flattened into one single call.

• the use of the *unfold* [12] transformation, frequently used in both sequential and parallel systems, is another instance of the flattening principle. 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 [7] and implementations of contraint languages [8].

• In parallel implementation of flat committed choice languages [22] 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.

• The flattening principle has also been used for efficient implementation of nested data-parallel operations in languages such as Paralation Lisp [19], C*, etc. Guy Blelloch was the first one to advocate the use of flattening in implementing data-parallel languages [3]. Recently it has also been incorporated in other data-parallel systems [11].

• The flattening principle 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 [17].

We now illustrate the application of the flattening principle to the ACE system. We term the optimization that results from the application of the *flattening principle* the *last parallel call optimization* since it's similar in spirit to the last call optimization found in sequential implementations of Prolog [23].

3.1 Flattening: Last Parallel Call Optimization

The intent of the Last Parallel Call Optimization (LPCO) is to merge, whenever possible, distinct parallel conjunctions, which are nested one inside the other. The final aim of this optimization is to reduce the depth of nesting of parallel calls.

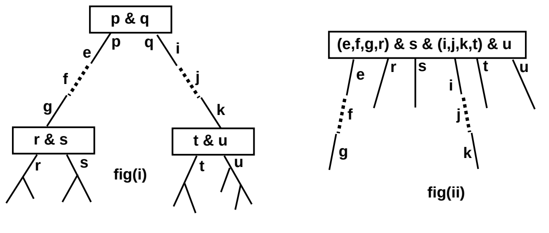

Figure 2: Reusing Parcall Frames

LPCO cannot be applied every time a nested and-parallel call arises inside another and-parallel call, since doing so in an unrestricted way may change the operational semantics when backtracking over parcalls. However, under certain restrictions, the operational semantics is preserved. To illustrate LPCO in the most general case, consider a parallel conjunction of the form *(p & q)*, where (See Figure 2(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[4] 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 expressing a data-parallel behaviour), 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)*[5], if we determine at the time of the parallel call *(r & s)* that *e, f,* and *g* are determinate. This is illustrated in Figure 2(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.

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.

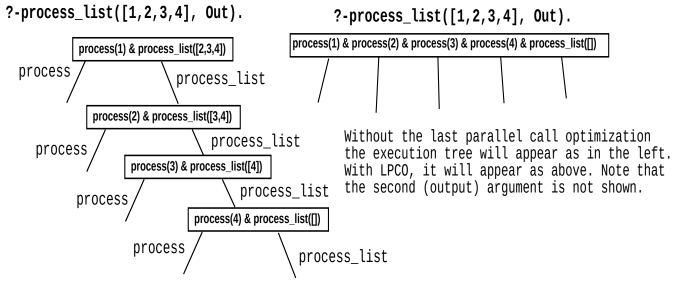

Figure 3: Reuse of Parcall Frames for Recursive Programs

[4] Actually, these conditions can be relaxed, but this is not discussed further due to lack of space [15].

[5] Although, in our case, the eventual bindings 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:

```
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 runtime technique, given that the goal *process_list* is determinate, nesting of parcall frames can be completely eliminated (Figure 3). 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.

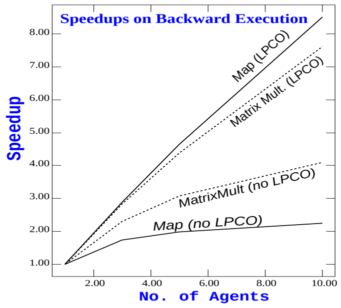

Figure 4: Speedup Curves for exec. with backtracking

On applying the LPCO in a situation such as the one shown in Figure 3, control parallelism that is deeply nested transforms itself into data-parallelism resulting in improved efficiency. In fact, our data-parallel version of Prolog, based on LPCO is comparable in performance with dedicated data-and parallel Prolog systems such as Reform Prolog [2].

Benchmark executed	&ACE agents 1	3	5	10
map^2	7.14/ 6.39	2.51 / 2.32	1.99 / 1.48	1.91 / 1.48
occur(5)	3.65/3.15	1.25/1.02	.75/.64	.43/.35

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

The results of applying LPCO to the ACE system are shown in Tables 1 and 2 (times in sec.). 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.[6] The performance improvement in terms of speed-up curves are shown in Figure 4. The most noticeable is the map benchmark: without the optimization almost no speedup can be observed (due to the heavy overhead of backtracking), while using the optimization an almost linear speedup can be obtained. Use of LPCO produces also a considerable improvement in memory consumption: experiments have shown that the usage of control stack can be decreased of almost 50% [13].

Benchmark executed	&ACE agents 1	3	5	10
matrix	6.30/5.36	2.73/ 1.90	2.05/ 1.22	1.54/ .70
pderiv	9.49/ 5.61	5.88/ 2.75	5.19/ 2.34	6.67/ 2.342
map^1	24.21/ 14.98	14.01/ 5.20	12.24/ 3.23	10.73/ 1.76
annotator	3.94/ 3.86	1.35/1.34	.88/ .87	.49 / .47

Table 2: Exec. Time in sec. (LPCO with Backward Exec.)

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)* has been also tested in the ACE system with excellent results [7]. The LAO can also be used for parallelizing and optimizing constraint languages (e.g. CHIP).

4 Avoidance of Unnecessary Operations

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

There are two ways to avoid these worst-case induced overheads, and they can be concretized in the following two

[6] For each optimization, results have been reported for benchmarks which trigger that optimization. In no case, it should be stressed, does the application of the optimization produces a slowdown.

principles: the *Procrastination Principle* and the *Sequentiality Principle.*

Procrastination Principle The procrastination principle has been widely used, not only in the context of parallel systems but also in sequential systems. The principle can be summarized as follows: *"The execution of an operation should be delayed until its effects are needed by the rest of the computation."*

The intuition is that certain operations may be delayed indefinitely, i.e. their effect may never be needed by the computation. The principle of procrastination has been repeatedly utilized in the design of Warren Abstract Machine (WAM) [24] for efficient execution of Prolog programs (e.g., space for caller goal is allocated by callee clause). Another example of the principle of procrastination is the *shallow backtracking* optimization of M. Carlsson [5] and implemented in both sequential (SICStus) as well as or-parallel (Aurora [9] and Muse [1]) 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 unification 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 procrastination principle also manifests itself in lazy evaluation and demand driven execution of functional and data-flow languages.

Sequentiality Principle The sequentiality principle can be stated as follows: *"Two consecutive branches of the same node of the computation tree executed by the same computing agent should produce minimal overhead."*

The principle has been implicitly used in various parallel systems. For example, many or-parallel implementations of Prolog (like Muse [1] and Aurora [9]) allow the sequential exploitation of consecutive alternatives from a parallel choice point with minimal overhead (using almost standard backtracking).

Granularity control in parallel systems can also be seen as an application of the sequentiality principle. The idea behind granularity control is to break up 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. The sequentiality principle has also been applied to committed choice logic languages [22] and improvement in performance obtained.

Next we illustrate the application of procrastination and sequentiality principles to the ACE system.

4.1 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 of a data structure on the stack, a *(input) marker.* This marker is used to indicate the point at which the execution of a subgoal was initiated, partition the stack in *sections*, one for each parallel subgoal, and maintain the logical connections between a parallel subgoal and its parent computation. The same considerations need to be applied when a subgoal is completed—a marker (*end marker*) needs to be allocated to indicate the completion of a subgoal and the end of the corresponding stack section. The presence of these section markers is fundamental in order to enforce the proper behaviour during backtracking—encountering an input marker indicates that a subgoal has been completely backtracked over and backtracking needs to proceed in the logically preceding subgoal, while encountering an end marker indicates that we are entering a parallel call. 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 determinate then backtracking should proceed directly from g_3 to g_1. We only have to make sure that every binding trailed during the execution of g_2 is untrailed (undone) before backtracking moves into g_1. This implies that actually there is no need to allocate the input marker node and the end marker node for goal g_2 during forward execution—and this applies to every determinate and-parallel subgoal. Allocation of marker, as described above, can be avoided only if the deterministic nature of the subgoal is known. Determinacy information is not available *a priori* (unless some form of compile-time analysis is applied, which will discover some of the cases), therefore the question is how do we avoid the allocation of the input marker. That's where the procrastination principle 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.

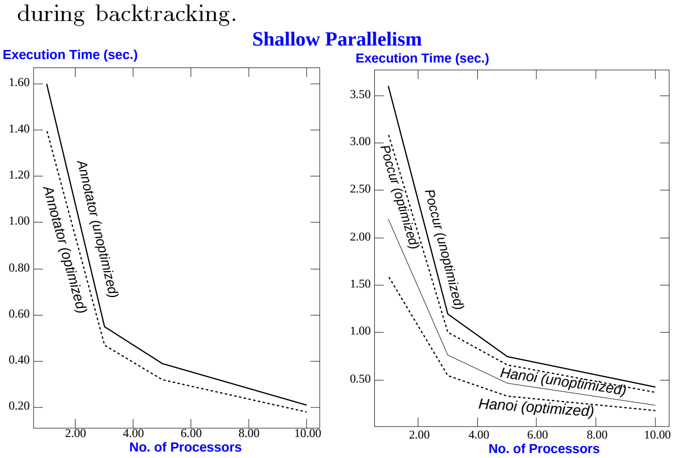

Figure 5: Execution time (ms.) with Shallow Parall.

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	&ACE agents 1	3	5	10	Max %
matrix_mult	5.59/5.2	1.9/1.7	1.1/1.0	.57/.53	10%
takeuchi	2.4/1.8	.83/.58	.52/.36	.25/.20	30%
hanoi	2.2/1.6	.76/.55	.47/.33	.23/.18	29%
occur	3.6/3.1	1.2/1.0	.75/.66	.43/.37	14%
bt_cluster	1.4/1.3	.52/.48	.34/.31	.20/.18	10%
annotator	1.6/1.4	.55/.47	.39/.32	.21/.18	18%

Table 3: Unoptimized/Optimized Exec. times in seconds for Shallow Parallelism (% improvement 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 implementation is obtained due to this optimization alone. In table 3 the execution times and percentage improvement obtained on some common benchmarks, all creating quite large and-or computation trees, are listed. The optimized and unoptmized speed curves are shown in figure 5.

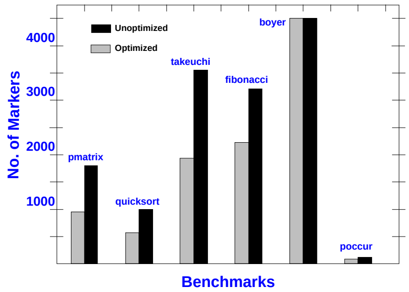

Figure 6: Shallow Parallelism: No. of markers allocated

The shallow parallelism optimization also leads to considerable reduction in the number of markers allocated and, thus, in the amount of memory consumed. Figure 6 shows the savings on the number of markers allocated obtained for some of the benchmarks. We 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).

4.2 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 sequentiality principle can be applied in such a situation to reduce this overhead. We call the optimization that results from applying the sequentiality principle the *Processor De-*

terminacy Optimization (PDO) because it comes into affect after the processor that is going to execute a given goal is determined.

The application of the PDO is also *a posteriori*. Once a situation in which the optimization can be applied is detected, i.e. the scheduler returns (or, it manages to select) a subgoal which, considering sequential semantics, immediately follows the one just completed, we can avoid most of the overhead associated with the second subgoal. This saving is obtained by simply avoiding allocating any marker between the two subgoals and—in general—treating them as a single, contiguous piece of computation. Thus, given *(a & b & c)* if processor executing *a* picks up *b* after finishing then the effect of the PDO is 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.

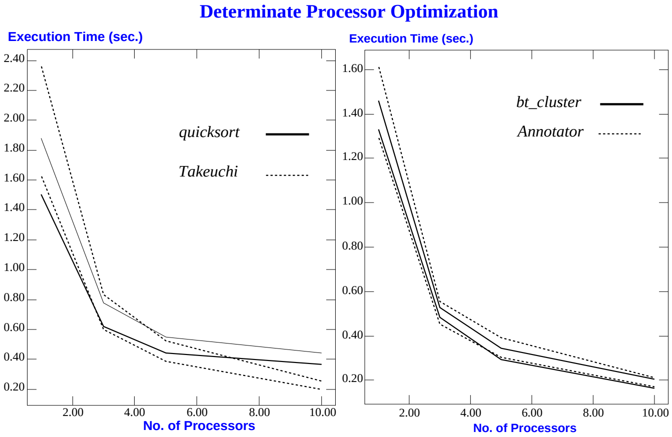

Figure 7: Improvement in Execution Time due to PDO

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 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 4. 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 7, 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: as we can see from table 4 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. The improved speed curves are represented in figure 7.

Benchmark executed	&ACE Execution Unopt.	&ACE Execution Optim.	Markers Used Unopt.	Markers Used Opt.
bt_cluster(0)	1461	1391 (5%)	120	60
occur(5)	3561	3418 (4%)	100	50
matrix_mult(30)	5598	5336 (5%)	1798	899
listsum	2333	2054 (12%)	3000	1500
hanoi	2183	1790 (18%)	4094	2047
takeuchi	2366	1963 (17%)	3558	2372

Table 4: Improvement with PDO, time (ms.) and Markers

An optimization, which combines a restricted version of Shallow Parallelism and Processor Determinacy and relies on static analysis, has been proposed by others [16]. 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).

5 Reuse of Resources

A common way of improving performance, in general, is by *reusing* elements of the computation.

Reuse Principle *"Reuse a data-structure or computation whenever possible."*

In the context of parallel nondeterministic systems, reuse can improve performance in two ways: *(i)* allow different parts of the computation to reuse certain data structures; *(ii)* avoid recomputation of certain parts of the computation (i.e. reuse parts of the computation).

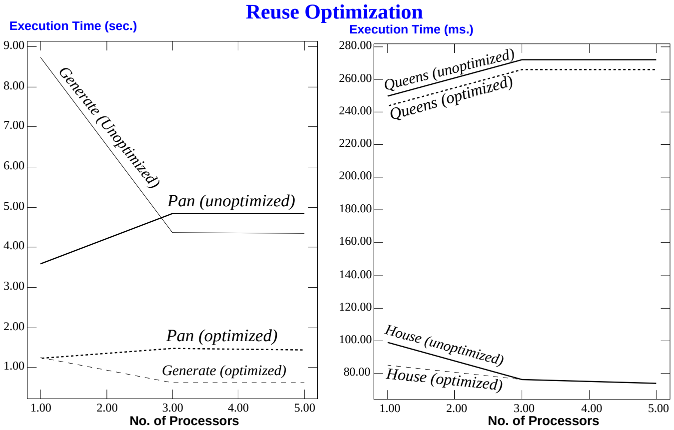

Figure 8: Unoptimized vs Optimized Execution Times for Goal Reuse

The reuse principle has found considerable number of applications. For example:

• the well-known tail recursion optimization and last call optimization lead to reuse of data structures [23];

• reuse of subgoals forms the basis of many models supporting parallel execution of logic programming languages. For example the *Basic Andorra Model* [20] allows considerable amount of reuse of deterministic subgoals—since they are virtually moved to the left of the nondeterministic ones, avoiding their recomputation during backtracking. Goal reuse is also employed in parallel logic programming systems exploiting or- and and-parallelism [18].

In fact, the LPCO also reuses space, because it reuses (modifies) older parcall frames. However, the reuse principle can also be applied to the ACE system to save computation. We next describe the application of the reuse principle to the ACE system.

5.1 Reuse: Goal List Rearrangement

Consider the following facts for the &ACE system:

• the subgoals belonging to the parallel call are independent;

• a deterministic subgoal will produce exactly one solution, independently of anything else;

This suggests that deterministic subgoals in a parallel call may be *reused* during backtracking. In the standard Prolog-like backtracking, given a goal $\ldots, p\ ,\ q\ ,\ r\ , \ldots$ *re-computation* is used, which means that whenever a new solution is found for subgoal p, then all the subgoals to its right (e.g., q and r) are recomputed from scratch. If, for example, q is deterministic and independent from other subgoals, then every time it is re-executed it will always return the same unique solution. If we do not remove the solution produced by q on backtracking, we can simply avoid its re-execution—by reusing the result previously computed.[7] In our implementation determinacy of goals is determined *a posteriori*, i.e., if after executing q, q is discovered to be deterministic (i.e. all but one alternative branches inside q fail), then it is moved to the leftmost position in the parallel conjunction. This idea is essentially the dual of the idea of shallow parallelism optimization: shallow parallelism avoids deterministic subgoals during backward search for a new successful alternative, while reuse skips deterministic subgoals during the forward activity which follows a backtracking process.

Reuse leads to: (i) savings in execution time because whenever backtracking discovers a new successful alternative and forward execution is reactivated, deterministic subgoals are not recomputed (this can speed up execution on its own, without employing any extra processors); (ii) improvement in backward execution since deterministic subgoals will not be repeatedly backtracked over (they will be backtracked over only once, when the whole parallel call has been completely explored); and, (iii) saving in memory consumption: after the first execution, deterministic subgoals will not be recomputed and stack space saved. Furthermore, the cost incurred in applying the reuse optimization is very small and does not lead to any additional overhead.

The optimizations have been tested on several benchmarks, and results are presented in Table 5 (times in sec.). Speedup curves are shown in Figure 8. Note that reuse will benefit those benchmarks that involve heavy backtracking over the parallel call. This heavy backtracking over parallel

[7] One could argue, that we could carry this further and compute solutions to p, q, and r once and for all, and compute their cross-product to find all possible combination, however, this destroys Prolog semantics and therefore we have to limit ourselves to deterministic goals.

calls is the cause for drop in speedups for some benchmarks (LPCO doesn't apply to these benchmarks, unfortunately).

Query	No. of Agents						Max
	1		3		5		%
	Unop.	*Opt.*	*Unop.*	*Opt.*	*Unop.*	*Opt.*	
Queens	.25	.24	.27	.26	.27	.26	2%
House	.99	.85	.76	.76	.74	.74	14%
Pan	3.59	1.23	4.84	1.47	4.84	1.44	70%
Gen&test	8.73	1.25	4.35	.63	4.34	.62	86%

Table 5: Improvements in Execution Time due to Goal Reuse

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 four optimization principles that can help in improving execution efficiency of parallel non-deterministic systems. These principles, termed the *procrastination principle*, the *sequentiality principle*, the *reuse principle*, and the *flattening principle* were illustrated by applying them to ACE, a high-performance parallel logic programming system. Application of these principles to ACE results 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. While these principles have been illustrated on the ACE system, they are fairly general, and can be readily applied to other nondeterministic systems such as parallel theorem proving systems, parallel rule based and AI systems, and parallel implementations of constraint and concurrent constraint languages.

Our experiments on applying these principles show that the field of software implementation can be a science (as opposed to being an art): one can systematically improve execution performance of a software system by applying well known optimization principles, such as those presented in this paper.

References

[1] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model. In *N. American Conf. on Logic Prog.* MIT Press, 1990.

[2] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the Language and its Implementation. In *Proc. Int'l Conf. on Logic Prog.*, MIT Press, 1993.

[3] G.E. Blelloch. *Vector Models for Data-parallel Computing.* MIT Press, 1990.

[4] L. Brownston. *Programming Expert Systems in OPS5.* Addison-Wesley, 1985.

[5] M. Carlsson. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In *Proc. Int'l Conf. on Logic Prog.*, MIT Press, 1989.

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

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

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

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

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

[11] D.W. Palmer et al. Work-Efficient Nested Data-Parallelism. In *Frontiers of Massively Parallel Computing*, 1994.

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

[13] E. Pontelli and G. Gupta and D. Tang. Determinacy-driven Optimization of And-parallel Systems. In *Proc. Int'l Conf. on Logic Programming*, MIT Press, 1995.

[14] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-performance Parallel Prolog System. In *Proc. Int'l Parall. Proc. Symp.*, IEEE, 1995.

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

[16] E. Pontelli et al. Efficient Implementation of Independent And Parallelism. Tech. report, New Mexico State University, 1994.

[17] V. Ramakrishnan et al. Efficient Techniques for Fast Nested Barrier Synchronization. In *ACM Symp. on Parallel Algo. & Arch.*, 1995.

[18] B. Ramkumar and L. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proc. North American Conf. on Logic Prog.*, MIT Press, 1989.

[19] G.W. Sabot. *The Paralation Model.* MIT Press, 1988.

[20] V. Santos Costa et al. Andorra-I: A Parallel Prolog System. *1990 Symp. on Princ. and Practice of Parallel Prog.*

[21] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, Pittsburgh, 1989.

[22] E. Tick. Concurrent Logic Programs *a la mode. Implementation of Logic Programming Systems.* Kluwer, 1994.

[23] D. H. D. Warren. An Improved Prolog Implementation Which Optimises Tail Recursion. Tech. Rep., University of Edinburgh, 1980.

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