Nested Parallel Call Optimization

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

Abstract

We present a novel optimization called Last Parallel Call Optimization (LPCO) for parallel systems. The last parallel call optimization can be regarded as a parallel extension of last call optimization found in sequential systems. While the LPCO is fairly general, we use and-parallel logic programming systems to illustrate it and to report its performance on multiprocessor systems. The last parallel call optimization leads to improved time and space performance for a majority of and-parallel programs. We also present a generalization of the Last Parallel Call Optimization called Nested Parallel Call Optimization (NPCO). A major advantage of LPCO and NPCO is that parallel systems designed for exploiting control parallelism can automatically exploit data parallelism efficiently.  

Keywords: Implementation optimizations, Parallel logic programming, And-parallelism.

1 Introduction

Non-determinism is an inherent component of problem solving in many areas of computer science. Search problems, generate-and-test problems, constraint relaxation applications are all examples of situations where non-determinism arises naturally. By non-determinism we mean the existence of multiple execution paths, each leading to (potentially) multiple solutions to the original problem.

Non-determinism is found in various programming languages: logic programming languages (e.g. Prolog), constraint programming languages (e.g. Chip [11]), rule-based languages (e.g. OPS5 [4]), etc.

Non-determinism represents not only a powerful way of expressing solutions to complex problems (by allowing programs to be written at a very high level of abstraction), it also offers a very rich source of parallelism. The possibility of extracting parallelism from the execution without affecting the structure of the solution and without heavy user intervention is making these languages more and more attractive.

A non-deterministic problem in such a language is generally expressed as a goal to be achieved/proved using a set of rules (or clauses) that specify how a goal can be reduced to “smaller” subgoals.

The process of solving a goal can be abstractly visualized as the construction of an and/or tree. Each node of the tree represents a goal to be solved. A goal composed of multiple subgoals is reduced by solving each of the individual subgoals (and-node). A single subgoal that can be solved using different rules represents, instead, an or-node. Figure 1 shows an example of and/or tree for a simple logic program (“?- gf(john, X)” is the goal to be solved).

Languages that include non-determinism allow considerable freedom in the way programs are executed, with particular reference to the order in which (i) different execution paths are explored (i.e. different rules for the same goal are attempted), and (ii) the different subgoals composing a goal are solved. This latitude permits one to exploit parallelism implicitly (without the need for programmer intervention) during program execution.

Two principal forms of parallelism are typically considered: (i) Or-parallelism: the different potential solutions to a goal can be explored in parallel (i.e., given a subgoal which can be reduced with different rules, the different reductions are concurrently attempted); (ii) And-parallelism: while looking for a specific solution, the different operations involved can be executed in parallel (e.g., the different subgoals composing a goal can be solved in parallel). And-parallelism is the “traditional” form of parallelism found, for example, in imperative programming languages, while or-parallelism is a direct result of the presence of non-determinism. With reference to the and/or tree construction, each or- and and-node represents a source of parallelism, i.e. a point of execution where parallel activities can be forked. The parallel construction of the tree should be realized respecting the dataflow dependencies of the program—
similarly to what happens in loop parallelization of Fortran programs [13].

A parallel system that builds an and/or tree to solve a non-deterministic problem may look trivial to implement at first glance, but experience shows that this is indeed a challenging task. A naive parallel implementation may incur an excessive overhead compared to a corresponding sequential system, which, in turn, may effectively translate to a slow-down of the execution.

A simple analysis shows that the major sources of overhead are related to the need of managing the tree structure of the computation at run-time (i.e., the tree need to be explicitly maintained, using additional data structures, and repeatedly traversed in search for multiple solutions).

One can formulate a number of principles that an implementor of a parallel system should follow to minimize parallel time and space overhead [7]. Two such principles are:

- **Reduced Nesting Principle**: The level of nesting of control structures in a computation should be reduced whenever possible.
- **Memory Reuse Principle**: Memory should be reused whenever possible.

These two principles step from the most intuitive way to simplify execution: reduce the complexity of the tree by flattening it. A simpler tree structure will lead to saving of memory (less data structures are required) and to faster execution (simpler structure to traverse). In fact, if an execution spawns \( n \) parallel computations and, furthermore, one of these subcomputations spawns other \( m \) parallel branches, then two distinct descriptors (for the two parallel calls) are required, one “nested” inside the other. The principle of reduced nesting will instead attempt to use a single descriptor (i.e., a tree of depth one) associated to a parallel call of \( m + n \) computations. Of course, the reduce nesting principle should be applied in a way such that program semantics are unaltered (e.g., in the case of logic programming this means that the order in which backtrack literals are chosen is preserved).

The reduced nesting principle manifests itself in many situations, both in non-deterministic as well as deterministic systems. The tail recursion optimization [12], the flattening used in Paralation Lisp [3], the distributed last call optimization [8] are just some examples of applications of the reduced nesting principle.

In this paper we present the **Last Parallel Call Optimization (LPCO)**, and its generalization, the **Nested Parallel Call Optimization (NPCO)**. These are meant to be the instantiation of the two principles above to the case of generic non-deterministic systems. In particular we will describe them in the context of logic programming (they can be applied to independent and-parallel systems, like &ACE [6], dependent and-parallel systems, like DDA[9], and more general and- and or-parallel implementations, like MUSE [1] and ACE [5]), although the concepts can be applied in a straightforward way to other non-deterministic systems (AI systems, theorem provers, etc.).

The LPCO is triggered when the last call in a Prolog clause is itself a parallel conjunction (from now on a parallel conjunction will also be referred to as a parcall for brevity). The NPCO is triggered when a Prolog clause has a nested parallel conjunction, and all goals following the parallel conjunction (these goals are termed continuation of the parcall) satisfy certain conditions.

The same principles can be used in the case of or-parallelism, giving rise to the **Last Alternative Optimization (LAO)**.

Even though the LPCO and NPCO have been developed in the context of non-deterministic programming, they can also be applied to parallel implementations of traditional languages, like Fortran. Figure 2 shows an ideal situation for the application of an optimization like LPCO: if at runtime the condition of the test in the if statement is satisfied, then the innermost loop can be parallelized and merged with the outermost one, potentially saving one barrier during execution.

2 Last parallel call optimization

The intent of the Last Parallel Call Optimization (LPCO) is to merge, whenever possible, distinct parallel conjunctions. Last Parallel Call Optimization can lead to a number of advantages (discussed later).

To illustrate LPCO in its generality, let us consider (fig. 3(i)) the parallel conjunction \((p \& q)\) where

\[
p := e, f, g, (e \& s).
\]

\[
q := i, j, k, (t \& u).
\]

LPCO will apply to \(p\) (resp. \(q\)) if (i) There is only one (remaining) matching clause for \(p\) (resp. \(q\)); (ii) All goals preceding the parallel conjunction in the clause for \(p\) (resp. \(q\)) are determinate. If these conditions are satisfied then a new parcall frame is not needed for the parallel conjunction in the clause (see figure 3). Rather, we can pretend as if the clause for \(p\) was defined as \(p := (e, f, g, t) \& s\)

![Figure 2. Fortran Code](image-url)
(although the bindings generated by \( e, f, g \) would be produced before starting the execution of \( s \)). Following the previous example, we extend the parcall descriptor for \((p \& q)\) with an appropriate number of slots and insert the nested parallel call in place of \( p \). Likewise for the clause for \( q \), if it contains a parallel call as its last call. This is akin to last call optimization in sequential systems [12], where the optimization is triggered when the last clause for a 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. \( q \)).

The advantages of LPCO are amplified in presence of backtracking. The search for alternative solutions is considerably simplified if the computation has the structure in figure 3(ii) (a simple linear scan of the parallel call is sufficient). It is obvious that LPCO indeed leads to saving in space as well as time during parallel execution. In fact: (i) space is saved by avoiding allocation of the nested parcall descriptors; (ii) time is saved during forward execution; and, (iii) considerable time is always saved during parallel backtracking\(^1\), since the number of control structure to traverse is considerably reduced.

### 3 Nested parallel call optimization

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. These requirements can be relaxed and two different generalizations are discussed below.

#### Nondeterministic Computations: LPCO cannot be applied whenever a non-deterministic computation is performed between the two nested parallel calls, like in \((p \& q)\), where a clause \( p : e \), \((f \& g)\) is used and \( e \) has multiple solutions. Extending LPCO to these cases is possible but it requires more involved changes to maintain the correct backtracking semantics. In particular, some information regarding the depth of nesting of each subgoal need to be kept, to limit the extent of propagation of backtracking.

#### Continuations: LPCO cannot be applied whenever a computation is present in the continuation of the nested parallel calls. In a goal \( :- (p \& q)\), \( c \), if the clause \( p :- (p_1 \& p_2)\), \( c_1 \) is used, LPCO will not be able to guarantee that the execution of \( c_1 \) is started after \( p_1 \) and \( p_2 \) but before \( c \). A safe possibility is to delay all the continuations of the nested parallel calls until the main parcall has completed. In the example, this equates to executing the goal \((p_1 \& p_2 \& q)\), \( c_1, c \). The soundness of this solution is guaranteed by the independence of the subgoals. Nevertheless, in order to have soundness we must also guarantee that the continuations are executed in the proper order (i.e., if a subgoal \( b \) is expanded with a clause containing \((c \& d)\), \( h \) and \( d \) is expanded with \((e \& f)\), \( i \), then \( h \)

\(^1\)In conventional languages there is no backtracking, however, the descriptors stored in the stack have to be removed at the end of the parallel computation. Reducing the level of nesting of parcalls is going to make space reclamation from the stack faster.

cannot be executed before \( i \). Furthermore, if it is known that the continuation is deterministic and non-failing, then the continuation goals can be executed without regard to continuation goals of other parcalls. In practice, it turns out that for most parcalls, either the continuation is empty, or it contains deterministic, non-failing goals.

Note that relaxing the conditions imposed on LPCO makes it a more general optimization scheme, increasing its applicability to a wide family of computations involving nested parallel calls. For this reason we term the generalization of LPCO the Nested Parallel Call Optimization (NPCO).

### 4 Experimental results

The LPCO optimization has been implemented as part of the current version of the \&ACE and-parallel system running on Sequent Symmetry [6] and Sun Sparc Multiprocessors\(^2\). The experimental tests that we have performed consist of running various benchmarks, measuring the time elapsed and memory consumed during execution. We selected the benchmarks in order to separately study the effects of the LPCO on programs whose executions present different degrees of backtracking.

Furthermore, we have separated our experimental analysis into two phases, by first running the benchmarks on the system with only LPCO and next executing them on the system with both LPCO/NPCO and other optimizations. Often, other optimizations [7] become applicable because of the flattening of the computation tree.

**LPCO:** The execution of the system with the use of LPCO produces considerable speed-ups while maintaining a good efficiency in execution; table 1 shows execution times obtained on some commonly used benchmarks (like MatrixMult—which computes multiplication between matrices, and BT-Cluster—an extract of a clustering program used by British Telecom) and on a “real-life” application, like the abstract analyzer (pam) used by the \&ACE compiler.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Execution Time</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixMult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BT-Cluster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract Analyzer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 compares the execution times obtained during sequential execution. The \( fw \) columns analyze the forward execution (i.e., without any backtracking), while the \( bw \) columns describe the execution times in presence of heavy backtracking. The improvements obtained using LPCO are particularly evident in presence of backtracking (e.g., execution is 74% faster in the list_search benchmark).

Interesting results are also seen by examining the effect of inside failures (i.e., failure of one goal within a parallel call) during execution. The presence of a single parcall descriptor considerably reduces the delay of propagating Kill signals to sibling parallel goals. In programs with sufficient nesting of parcalls, the improvement in total execution time due to faster killing improves by as much as 42%.

Figure 4 summarizes memory savings obtained by LPCO: the picture compares the usage of control stack measured

\(^2\)The results obtained on a Sun Sparc 10 are consistent with those presented in this paper—which are obtained on the Sequent Symmetry.


<table>
<thead>
<tr>
<th>Goals executed</th>
<th>&amp;ACE Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fw/no lpco</td>
</tr>
<tr>
<td>BtCluster(0)</td>
<td>890</td>
</tr>
<tr>
<td></td>
<td>(5% )</td>
</tr>
<tr>
<td>Deriv(0)</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>(64% )</td>
</tr>
<tr>
<td>Occur(5)</td>
<td>3216</td>
</tr>
<tr>
<td></td>
<td>(5% )</td>
</tr>
<tr>
<td>pann(5)</td>
<td>1327</td>
</tr>
<tr>
<td>MatrixMult(20)</td>
<td>1724</td>
</tr>
<tr>
<td></td>
<td>(4% )</td>
</tr>
<tr>
<td>Ilist_search(1500)</td>
<td>2354</td>
</tr>
<tr>
<td></td>
<td>(17% )</td>
</tr>
</tbody>
</table>

Table 1. Unoptimized/Optimized Execution times in msec (single processor)

during the execution of some benchmarks, comparing the unoptimized vs. the optimized case. The percentages indicated show the reduction in memory consumption obtained.

NPCO: In the present version of &ACE only the second extension described in section 3 has been implemented. This version of NPCO has been tested on several benchmarks and the general results obtained are consistent with those presented in the previous section for LPCO: a moderate improvement in execution time for purely deterministic executions, a more considerable speed-up for computations involving backtracking across parallel executions, and, in general, a dramatic improvement in memory usage. Table 2 shows the results obtained for two benchmarks, hanoi and quicksort (both cannot take advantage of LPCO since the parallel call is not the last call in the clause).

In the case of both the optimizations, as mentioned before, backtracking across parallel computations is considerably improved. Figure 5 compares the speedup curves obtained with and without these optimizations in the case of benchmarks with backtracking. As we can observe in certain cases (e.g., Map—a program which applies a function to the elements of nested lists) this optimization results in an exceptionally good speedup, which was lost in the unoptimized case due to the intense overhead during backtracking.

5 Last alternative optimization

It is possible to develop an optimization analogous to LPCO for or-parallelism, named Last Alternative Optimization (LAO). LAO applies whenever a new choice-point is created in the execution of the last alternative of a previous choice-point. In a sequential execution, techniques like shallow backtracking allows the new choice-point to reuse the space previously occupied by the old choice-point. In an or-parallel execution this is not possible in general, since when the last alternative is started some of the previous ones may still be active. The idea is to avoid creation of a new choice point and promote the new alternatives to the old choice-point.

We have tested a prototypical implementation of the LAO on the MUSE System [1], which is based on stack copying (i.e., an agent steals work from another one by creating a local copy of the computation). The use the LAO produced some improvements, as illustrated in table 3. These are mainly due to the fact that, by promoting alternatives to the previous choice-point, the new work locally produced is automatically made available to all the other agents that have copied the older choice-point. In this way the number of sharing operations (i.e., operations in which one agent steals work) is reduced, minimizing the parallel overhead. The optimization has been shown to scale with larger number of processors.

LAO is conceptually independent from the way in which or-parallelism is managed (stack copying or some other technique) and we expect it to produce improved execution speed also in other or-parallel systems.
6 Data-parallel programming

LPCO and NPCO can be seen as instruments for taking advantage of occurrences of Data Parallelism in Prolog programs. Typical instances of data-parallelism are represented by recursive clauses whose iterations can be performed in parallel. For example, the process_list program:

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

is a data-parallel program, because once the recursion is completely unfolded a number of identical looking calls for the process goal are produced. LPCO can be seen as a way to efficiently executing data parallel programs. Given the process_list program, although a system like &ACE will produce one iteration at a time, the LPCO will actually collect all the iterations under a single parallel call—obtaining an effect analogous to a complete unfolding of the recursion. The efficiency of execution of data-parallel programs using LPCO and NPCO compares favorably to other proposals made in the literature for exploitation of data parallelism (like Reform Prolog [2]).

The same considerations apply to LAO, which can be used to efficiently exploit data or-parallelism. This form of parallelism occurs quite frequently in various application areas; a good example is represented by constraint optimization problems based on finite domain constraints (e.g., as in the chip system [11]). Use of forward checking techniques to reduce the domains associated to the variables in the problem can be simply rephrased as an or-parallel activity originated by applying a member predicate to each domain and solving the corresponding problem instantiation—and this can be easily optimized by LAO. Applying LAO should allow a generic or-parallel system to achieve speedups close to those obtained by dedicated data or-parallel systems, like MultiLog [10].

7 Conclusions

In this paper we presented a novel optimization, called Nested Parallel Call Optimization. This optimization applies well known optimization principles to practical parallel programming languages implementations. It not only allows considerable savings in memory consumption, it speeds up the execution of a majority of parallel programs. This optimization has been implemented in the &ACE parallel systems, and the experimental results confirm its effectiveness. This optimization was illustrated in the context of and-parallel logic programming system, but it can be easily applied to any arbitrary parallel system that allows nested parallel computations.

References