Compilation for Nested Recursion-Parallelism

Thomas Lindgren
Computing Science Dept., Uppsala University
Phone: +46 18 18 10 54
e-mail: thomasl@csd.uu.se

September 9, 1996

In this paper, we show that compilation for nested recursion-parallel execution can be accomodated by extending the existing analyses of Reform Prolog, a recursion-parallel Prolog implementation.

Second, we show how to translate control-parallel code into equivalent recursion-parallel programs, thus extending the class of programs handled by the proposed system.

1. Introduction

The first implementation of Reform Prolog [14, 4, 3, 13, 11] featured only parallel execution of a single parallel loop at a time: parallel computations could not themselves start parallel loops. While this lead to a simple and efficient execution model, it also restricted the programs that could be parallelized. In this paper, we consider how to extend recursion-parallel execution to more general programs.

2. Recursion parallelism

For list recursion, recursion-parallel predicates are on the following form.

```
p([],...) :- <base case>
p([X|Xs],...) :- <left body>, p(Xs), <right body>
```

Consider a call to a list recursive predicate.

```
p([T1,...,Tn],...)
```

We can unfold the call into the following, equivalent query.

```
<left body 1>,...,<left body n>,
p([],...),
<right body n>,...,<right body 1>
```

In his thesis [15, 14], Millroth developed a method for compiling this unfolding process and subsequent execution into parallel loop code, yielding a schematic form shown below.

```
p/n:

 compute the length of the input list as n;
 build all instances of left and right body
  as a collection of vectors;

 /* each instance accesses its variables
  * in vectors at offset i
  */
 forall i := 1 to n do
    left_body[i];
 end

 call p(...)

 forall i := n to 1 do
    right_body[i];
 end
 return;
```

We call each loop iteration a *recursion level*. At runtime, the parallel recursion levels are executed by a collection of worker processes.

In order to ensure that the parallel loops can be implemented efficiently, Reform Prolog restricts their form by imposing an *execution model* on programs. If the program does not conform to the execution model, it is rejected by the parallelizing compiler and is run sequentially (alternatively, non-conforming portions of the program, e.g., individual predicates, may be rejected and run sequentially).

2.1. The flat execution model

Mainly, the execution model of Reform Prolog restricts programs in how shared data may be used. For a given parallel loop, we define the following terms.

A goal is *leftmost* if it is leftmost in the resolvent. In a sequential Prolog system lacking dynamic scheduling, the leftmost goal is always chosen next for resolution.

A variable X is *robust* if it appears in several recursion levels (i.e., is shared by the recursion levels).

A variable X is *tested* when it appears in a type test, such as var(X) or atom(X), or when it is used in indexing. In essence, X is tested when the instantiation of X determines the control of the process. Unifications (other than indexing) do not count as tests in this sense, since unification is commutative. Two adjacent unifications can be freely reordered without affecting the end result.

A variable X is *fragile* if it is robust and tested by one or more recursion levels.

We can now state the restrictions.

- Binding a fragile variable X can only be done when the goal is leftmost. Tests on a fragile variable X must delay until X is bound or the recursion level is leftmost.
- Bindings to shared variables may never be conditional.
- When a recursion level is finished, it is always deterministic.
- A (parallel) recursion level can not start a parallel loop.

Further details, e.g., on handling predicates with side-effects, can be found in Refs. [2, 12].

By making the first restriction, we ensure that parallel and sequential execution yields the same answer.

By making the second and third restrictions, we can guarantee that no costly 'rollback-on-failure' operations need to be supported. (See for instance Shen's DDAS for a design where such rollbacks are supported [19, 18].)

The final restriction means the runtime system can be kept simple and efficient. All workers share a single parallel loop, which makes scheduling and leftmostness checking straightforward.

Obviously, nested parallel execution is not allowed, so we call this the *flat* execution model.

2.2. Supporting the flat execution model

The Reform Prolog compiler performs mainly two tasks to enable and optimize parallelization: it keeps track of the locality of variables, and it keeps track of whether recursion levels are deterministic or not.

The Reform Prolog compiler puts variables in three categories. A variable is normally local. When a parallel loop is entered, the variables shared between recursion levels are robust; unshared variables remain local. When a local term is unified with a robust term, it becomes robust as well. A robust term that is tested becomes fragile.

Robust and fragile variables can only be bound when the binding recursion level is deterministic; locking unification must be used when such variables are bound. Fragile variables can furthermore only be bound when the binding recursion level is leftmost.

Local data can be handled as in a sequential WAM — indeed, one of the points of our design was to ensure that working with local data was as efficient as sequential computation. The advantage of this approach is that standard high-performance Prolog compiler technology can be used on the local parts of a recursion level; overheads for parallelization are confined to places where data are actually shared.

The compiler inserts suspension and locking operations to ensure that these conditions hold. Experiments indicate that the compiler rarely classifies local data as robust or fragile, and that we can expect high speedups (using emulated SICStus Prolog as the reference system) for realistic programs fitting the flat execution model [11].

3. Nested recursion-parallelism

The main drawback of the previous execution model is that it does not allow nested parallel loops. In this section, we show how to extend the flat execution model to encompass nested parallel execution.

We still want to compile a recursion-parallel predicate into the schematic loop code of the flat model, but we now must face the potential problems of having several (nested or not nested) parallel loops active at the same time.

The following problems appear when extending the execution model.

1. How do nested parallel loops affect locality analysis?
2. How does the system check leftmostness efficiently?
3. How is scheduling done efficiently?

3.1. Locality analysis for nested parallel execution

Extending locality analysis to handle nested parallel loops boils down to the following issue: in the flat execution model, all variables are local prior to starting a parallel loop; in the nested execution model, this no longer holds. When a parallel loop starts, variables may already be robust or fragile by the actions of enclosing parallel loops.

The simplest solution is just to 'promote' the appropriate local variables into robust ones at the start of the loop (as is done in the flat execution model) and keep the externally robust and fragile variables as robust and fragile, respectively.

The drawback with this approach is that reading/writing fragile variables now must suspend until the recursion level is leftmost of all recursion levels in all loops. If the variable actually was fragile with respect to the current loop only, a recursion level could suspend for a needlessly long time, waiting for the loop to become leftmost of all loops.

In order to remedy this drawback, we refine the concepts of locality somewhat. First, we say that a goal (recursion level) is *locally leftmost* if it is leftmost of the goals (recursion levels) of its parallel loop. A goal (recursion level) is *globally leftmost* if it is leftmost of all goals (recursion levels).

A variable is *globally robust* if it appears in more than one parallel loop (and is not tested). A variable is *locally robust* if it appears in more than one recursion level but only in one parallel loop.

If a globally robust variable is tested, it becomes *globally fragile*. If a locally robust variable is tested, it becomes *locally fragile*.

A variable is *global* if it is globally robust or globally fragile.

Analysis sketch. We provide a preliminary sketch of locality analysis using the global/local distinction.

During analysis, we handle this as follows. When a parallel loop starts, (some) local variables become (locally) robust, just as in the flat execution model. At the same time, all locally robust variables are turned into globally robust variables, and likewise locally fragile variables become globally fragile. Global variables remain global.

Testing a robust variable makes it fragile as before, with the proper global/local prefix. Unifying two variables means taking the 'worst' locality of the two as new locality.

When the parallel loop is done, the variables that were local, and have not become global, are turned back into local variables. Variables that were locally robust/fragile before the parallel loop, and are not aliased to global variables after the loop, are turned back into locally robust/fragile again.

Locality use in nested parallel loops. We extend the execution model to consider global/local variables as follows.

- Binding or testing a locally fragile variable X must suspend until locally leftmost.
- Binding or testing a globally fragile variable X must suspend until globally leftmost.
- Bindings to locally robust, locally fragile or global variables must be unconditional.

Discussion. We note that this scheme may suspend recursion levels longer than necessary. Consider the following case: we have a parallel loop L1 where X is locally robust, with a recursion level R1 (among others). Now, R1 starts a parallel loop containing X, which makes X globally robust. Assume that some recursion level R' in R1 tests X. In this case, R' must suspend until *globally* leftmost, whereas it would be sufficient to suspend until leftmost in L1.

It is unclear how often such a situation would occur. If it occurs frequently, a more involved scheme should be devised, where a variable instead is fragile with respect to some given loop nest, and may be bound/tested when leftmost in that loop nest.

Future work includes measuring whether the global/local distinction is sufficient in practice, and if it is not, the design of a 'hierarchical' scheme on the lines of above.

3.2. Leftmostness checking

The nested execution model requires two forms of leftmostness checking: locally within a single loop, and globally among all loops. The local check is done as in the flat system. One way of implementing a global leftmostness check is the following: (a) maintain a total order between loops, e.g., by linking them together and (b) first check whether the current loop is leftmost, then check whether the current goal is leftmost.

3.3. Scheduling

A traditional problem in and-parallel systems is which goal to run next. A simple and cheap solution for Reform Prolog has been proposed by Bevemyr [1].

4. From control-parallelism to recursion-parallelism

In this section, we propose Nested Reform Prolog, a language unifying control- and recursion-parallel execution, and show how to convert control-parallelism into recursion-parallelism.

We extend Reform Prolog into Nested Reform Prolog as follows. Nested Reform Prolog uses the nested execution model defined in the previous section, and extends Reform Prolog with parallel conjunction (taken from &-Prolog), written as

$$A \& B$$

The operational semantics of $\&$ is to start executing A and B in parallel if possible (e.g., if the execution model is satisfied). In particular, we allow A and B to share variables and to communicate using these variables (e.g., producer/consumer parallelism).

Nested Reform Prolog retains recursion-parallel execution and sequential conjunction, and tolerates local nondeterminism. We could thus view it as an extension of Parallel NU-Prolog [16].

4.1. Transformation algorithm

In order to reuse the machinery of nested recursion-parallel execution, we transform a parallel conjunction

$$(B_1 \& B_2 \& \ldots \& B_n)$$

into a recursion parallel predicate p$*/k$ as follows.

1. Collect the free variables FV of each $B_i, 1 \leq i \leq n$ and intersect FV with the union of the free variables of all $B_j, j \neq i$. This is the *interface* of the parallel conjunct.

2. Compute the union $\{X_1, \ldots, X_k\}$ of the interfaces as the arguments to be added to the enclosing parallel predicate. We call this the total interface.

3. Emit the code for p$*/k+2$ according to the template shown in Figure 1. Replace the parallel conjunction with a call $p*(1, n, X_1, \ldots, X_k)$ to p$*/k+2$, where n is the number of conjuncts and $X_1, \ldots, X_n$ are the variables of the total interface.

If the parallel conjunction obeys the nested execution model, Reform Prolog parallelizes the recurrence, and generates the loop in Figure 2.

Note that the two extra arguments almost disappear from the generated code. M is local to each process and used briefly. N is used to set the loop bounds, then discarded.

Furthermore note that the compiler could simplify this code somewhat. For instance, the computation of the loop bound is unnecessary: both X[0] and X[1] are known at compile-time.

The proposed compilation technique extends to mixed parallel and sequential conjunctions in the obvious way.

```
p*(M,N,X1,...,Xn) :-
        M > N.
p*(M,N,X1,...,Xn) :-
        M =< N,
        M1 is M+1,
        ( M =:= 1 ->
          B1
        ; M =:= 2 ->
          B2
        ; ...
        ; M =:= N ->
          Bn
        ),
        p*(M1,N,X1,...,Xn).
```

Figure 1: Parallel conjunction as recursion-parallel predicate

```
p*/k+2
  n := X[1]+1-X[0];
  for i := 1 to n do
     M := i;
     if M =:= 1 then
       <load arguments>; B1
     else ...
     else if M =:= N then  /* N is a known constant */
       <load arguments>; Bn
     endif
  end
  call p(...)
  return;
```

Figure 2: Parallel loop code for $p*/k$. X[0] and X[1] denote X-registers.

5. Related work

Hermenegildo and Carro propose transforming Reform Prolog into &-Prolog [10]. Rather than generating parallel loop code, they use source-to-source transformation to express various strategies for task creation, including unfolding by k (to provide k independent loop bodies) and using a split/4 predicate to separating the list traversal from task creation, task scheduling and task execution.

They also discuss how to emulate Reform Prolog more closely. The main means are to add low-level support for traversing lists, creating vectors and tasks, while retaining the standard &-Prolog engine. The authors do not consider support for dependent and-parallelism.

Unfolding to increase parallelism was also proposed by Tärnlund in his thesis [20], and has been a mainstay of fine-grained parallel compilers for Fortran or C for many years.

Implementations such as &-Prolog [9] and &ACE [6] access the shared variables through an environment. The closest equivalent in Nested Reform Prolog is the set of shared arguments to the generated parallel predicate. We are not familiar with the precise implementation of the environments of &-Prolog or &ACE, and so cannot give a direct comparison with our proposal.

6. Future work

This paper has laid the groundwork for how to extend Reform Prolog to run nested parallel computations. At the time of writing, a source-to-source compiler translates control-parallel code into recursion-parallel code and generates code for recurrences. (The latter an extension of the scheme described in Ref. [12].)

The extensions to the compiler to allow nested recursion-parallelism (in particular the locality analysis) have not been implemented, nor has the runtime system necessary to support it. Future work is to implement and evaluate these systems.

7. Conclusion

We have shown how to extend the Reform Prolog compiler to compile nested recursion-parallel execution with minor changes to the existing compiler and system. The core concept is to extend the locality analysis of Reform Prolog, and to distinguish locally and globally shared data for the purpose of suspending processes. This admits a less restrictive view on when 'tests' can be performed than waiting until leftmost.

Our second contribution is to show how control-parallelism can be transformed into recursion-parallelism. The transformation is straightforward, and extends the data-parallel paradigm to include a class of control-parallel programs exploiting dependent and-parallelism.

7.1. Acknowledgements

My thanks to Johan Bevemyr and Håkan Millroth for discussions on nested recursion-parallel execution. Thanks also to the anonymous referee for valuable comments.

References

[1] J. Bevemyr, A scheme for executing nested recursion parallelism, in JICSLP'96 Post-Conference Workshop on Implementation Techniques.

[2] J. Bevemyr, *A Recursion Parallel Prolog Engine*, Licentiate of Philosophy Thesis, Uppsala Theses in Computer Science 16/93, 1993.

[3] J. Bevemyr, T. Lindgren, H. Millroth, Reform Prolog: the language and its implementation, in Proc. ICLP'93.

[4] J. Bevemyr, T. Lindgren, H. Millroth, Exploiting recursion-parallelism in Prolog, in Proc. PARLE'93.

[5] T. Chikayama, T. Fujise, D. Sekita, A portable and efficient implementation of KL1, in Proc. PLILP'94.

[6] G. Gupta, E. Pontelli, M. Hermenegildo, V. Santos Costa, ACE: And/Or-parallel copying-based execution of logic programs, in Proc. ICLP'94.

[7] G. Gupta, V. Santos Costa, R. Yang, M. Hermenegildo, IDIOM: Integrating dependent and-, independent and-, and or-parallelism, in Proc. ILPS'91.

[8] M. Hermenegildo, D. Cabeza, M. Carro, Using attributed variables in the implementation of concurrent and parallel logic programming languages, in Proc. ICLP'95.

[9] M. Hermenegildo, An abstract machine for restricted AND-parallel execution of logic programs, in Proc. ICLP'86.

[10] M. Hermenegildo, M. Carro, Relating data-parallelism and (And-)parallelism in logic programs, Computer Languages Journal, to appear.

[11] T. Lindgren, J. Bevemyr, H. Millroth, Compiler optimizations in Reform Prolog: experiments on the KSR-1 multiprocessor, in Proc. EURO-PAR'95.

[12] T. Lindgren, *The compilation and execution of recursion-parallel Prolog on shared-memory multiprocessors*, Licentiate of Philosophy Thesis, Uppsala Theses in Computer Science 18/93, November 1993.

[13] H. Millroth, SLDR-Resolution: parallelizing structural recursion in logic programs, Journal of Logic Programming.

[14] H. Millroth, Reforming compilation of logic programs, in Proc. ILPS'91.

[15] H. Millroth, *Reforming the compilation of logic programs*, Ph.D. Thesis, Uppsala Theses in Computer Science 10, 1991.

[16] L. Naish, Parallelizing NU-Prolog, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, 1988.

[17] V. Santos Costa, D.H.D. Warren, R. Yang, The Andorra-I engine: a parallel implementation of the basic Andorra model, in Proc. ICLP'91.

[18] K. Shen, *Studies of And-Or Parallelism*, Ph.D. Thesis, Cambridge University, rev. June 1992.

[19] K. Shen, Exploiting dependent and-parallelism in Prolog: the dynamic dependent and-parallel scheme (DDAS), in Proc. JICSLP'92.

[20] S.-Å. Tärnlund, Logic information processing, TRITA-IBADB 1034, Department of Information Processing and Computer Science, Royal Institute of Technology and University of Stockholm, 1975.