A Scheme for Executing Nested Recursion Parallelism

Johan Bevemyr

Computing Science Dept., Uppsala University
P.O. Box 311, S-751 05 Uppsala, Sweden
e-mail: `Johan.Bevemyr@csd.uu.se`
Phone: +46 18 18 10 52, Fax: +46 18 51 19 25

We propose an extension for nested parallelism in the Reform Prolog execution model. The proposed modification preserves the simplicity and efficiency of the original design and its performance model.

This is achieved by exploiting properties of deterministic execution and careful scheduling of work.

1. Introduction

We propose a scheme for executing nested recursion parallelism. The scheme requires only minimal extensions to the current execution model of Reform Prolog [3].

It is possible to transform some nested recursions into a single recursive loop, as shown by Millroth [18]. Flattening nested data-parallel computations has been investigated by Blelloch [4], Palmer, Prins and Westfold [20] among others.

However, it is not always feasible to flatten a nested recursion in Reform Prolog, e.g. when the size of the nested recursion cannot be statically determined. Suppose we have the following program:

```
p([],[]).
p([H|T],[NewH|NewT]) :-
        state_size(H,State,N),
        q(N,State,NewH),
        p(T,NewT).

q(0,_,Result) :- !, Result = [].
q(N,State,[H|T]) :-
        foo(N,State,H),
        N2 is N - 1,
        q(N2,State,T).
```

Then we would have to choose between parallelising the outer loop (p/2) or the inner loop (q/2). p/2 is the obvious choice provided that not both p/2 and q/3 are shallow recursions. It would then be desirable to parallelise both and distribute the work among the workers.

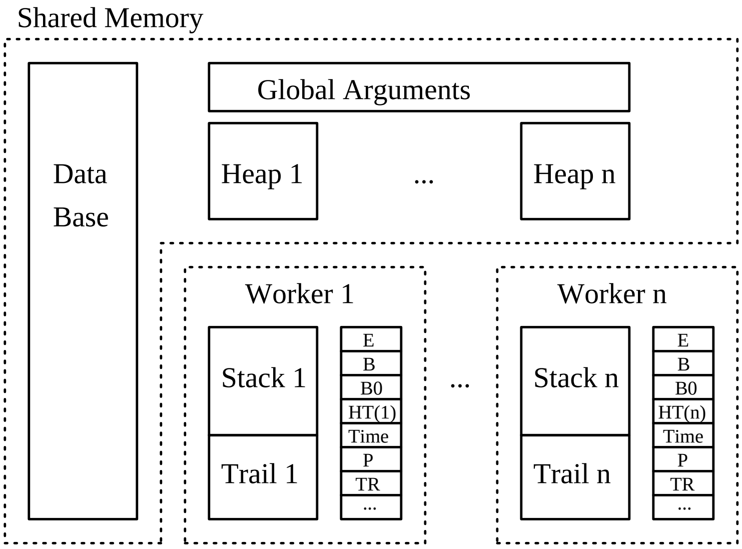

Figure 1: Overview of the recursion parallel machine.

We investigate how this nesting of parallel loops can be supported while maintaining the efficient implementation and the simple execution model of Reform Prolog.

Note that the intention of Reform Prolog is not to parallelise arbitrary Prolog programs. The programmer is responsible for rewriting the programs into a form suitable for parallel execution. Reform Prolog provides a clean execution and performance model which makes it possible for the programmer to coherently express parallelism with predictable performance.

2. Reform Prolog: The Original Design

Reform Prolog concentrates on parallelising structure recursion, e.g. recursion over lists or integers, in a manner similar to loop parallelisation in imperative languages. This gives the programmer a simple and predictable execution model.

The execution of a recursive loop can be seen as a four step computation (or as program transformation through unfolding).

1. Perform all initial head unifications, building the necessary vectors for parts 2 and 4.
2. Execute the goals to the left of the recursive call in a parallel loop.
3. Execute the base case.
4. Execute the goals to the right of the recursive call in a parallel loop.

This model is efficiently implemented using a set of WAM [28] engines (workers) which compute one recursion level at a time (see Figure 1). Instructions are added to WAM for performing the vectorising unifications in step 1 and for synchronisation in step 2 and 4 to keep the sequential semantics of the program.

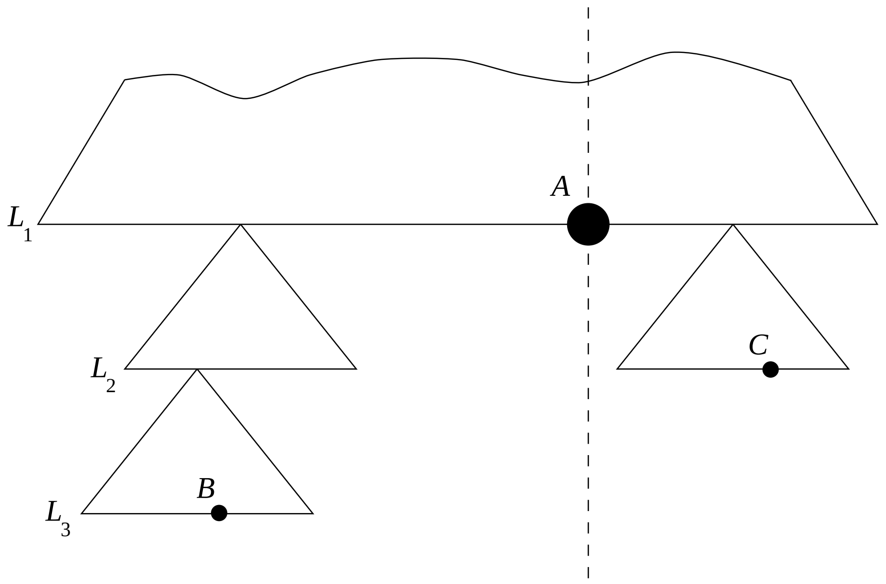

Figure 2: L_1, L_2 and L_3 represent parallel recursive loops where L_2 is initiated by an iteration level in L_1 and L_3 is run inside an iteration level in L_2. A is a suspended recursion level in L_1. B and C represent available work.

No nesting of parallel recursive calls is allowed. Data sharing and communication is allowed as long as bindings of shared variables are deterministic. Scheduling, memory and process management are simple and efficient. Almost sequential code is executed by each worker, with the exception of some synchronisation instructions (which are only present in programs with data dependencies between recursion levels). Bevemyr [1] describes the full machinery in detail.

All this result in small overheads for parallel execution.

3. Nested Recursion Parallelism

How can this machinery be extended to support nested parallel execution?

We make a few observations about the current implementation:

1. The stack is restored to its initial state when a recursion level terminates, i.e. no choice points or environments are left by a recursion level.
2. The trail and the heap grow when executing forward and shrink on backtracking.
3. Work to the left is less speculative than work to the right in the tree (resolvent).

Let us consider the following situation in the nested case: a worker w suspends at point A waiting for some data to be produced or to become leftmost (see Figure 2). Work is available both to the left (B) and the right (C) in the tree (resolvent).

If B is chosen then we can execute B until it terminates. This means that the stack will be restored to the state it was in when A suspended. However, the heap and the trail will not be

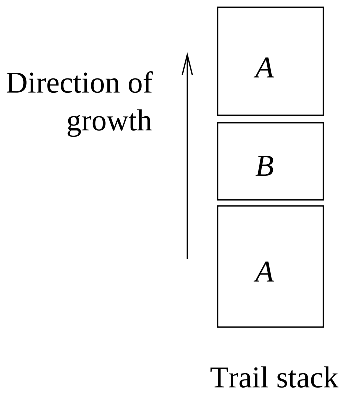

Figure 3: Trail entries are first created while executing A which suspends and B is executed, leaving new trail entries. A is then resumed creating another section of trail entries.

restored. Portions of data and trail entries will be left on the heap and the trail after executing B, as shown in Figure 3. First there is a portion produced when executing A, then a portion belonging to B followed by a section produced when continuing to execute A.

Consider what happens if the entire parallel recursion L_3 fails. Then the trail section belonging to B must be unwinded. This is tricky since it resides between sections produced by A. A solution is to only allow L_3 to be executed in parallel if the initiating worker is in a deterministic state, i.e. no choice point has been created in the current thread since the parallel execution started. If L_3 fails, then the entire parallel execution fails (L_1, L_2 and L_3) and all trail entries created during the current parallel phase are undone.

A similar situation occurs if A fails back to a point before w executed B. The trail entries created while running B are unwinded since they are in the middle of the trail stack. This situation can be treated in the same manner as above. A worker is only allowed to search for work when it is suspended in a deterministic state. If the worker fails to a point before the suspension, then the entire parallel phase fails and all conditional bindings are undone.

If C is chosen then we might end up in a situation where C must be suspended and A resumed, since C may wait for data produced by A. That is, we must be able to switch between suspended recursion levels which is not a trivial task.

To summarise: no new data areas or data structures are needed to support nested parallel execution if the following three restrictions are used.

1. Nested recursion parallelism is only allowed if no choice points have been created in the current thread of recursion levels.
2. Suspended workers are only allowed to search for work if they are in a deterministic state, i.e. no choice points have been created by the worker in the current iteration level.
3. Suspended workers are only allowed to search for work to their left in the tree.

The first restriction can be verified at compile time using information obtained by the existing compiler [17].

The drawback of this approach is that some nested parallelism might not be exploited. On the other hand, the machinery for nested parallelism is extremely simple and no extra overheads are associated with its implementation.

3.1. Optimisations

The compiler generates suspension instructions to enforce the sequential semantics for producer-consumer dependencies. A worker is suspended until a variable becomes sufficiently instantiated, or until the current recursion level becomes leftmost. The leftmost condition can sometimes be reduced to waiting until the current level is locally leftmost. That is, until it is the leftmost unfinished recursion level in the current recursive loop. This is possible when a variable is locally shared and not shared between recursion levels at a higher level.

Lindgren [17] shows how specialised versions of the suspension instructions can be generated by the compiler.

4. Discussion

4.1. Restrictions

Are the restrictions posed on nested recursion parallelism reasonable?

Let us consider the first restriction. Nested parallel recursive loops are only allowed to be initiated in deterministic states. We do not think this is a severe restriction. The condition is intuitive and programmers would have no problem determining when it is satisfied.

This would allow the two loops in the program on page 1 to be nested provided that state_size/3 and foo/3 are deterministic. Whereas programs such as testing, in parallel, if each element in a list satisfies some condition are not allowed inside an outer parallel loop.

The second restriction states that a worker is only allowed to search for work if it is idle, or suspended, in a deterministic state. This restriction is harder to reason about for the programmer. It is not as obvious when and how a worker suspends. However, removing this restriction requires some form of “cactus” heaps and trails which is less efficient than standard stacks.

The third condition is a restriction on the scheduler. It means that some workers will be unable to obtain parallel work since they are almost leftmost (the leftmost worker is never suspended).

It would be possible to allow workers to obtain work from the right, but such work might have to be suspended in favour of work to the left. The machinery for this is cumbersome and it is questionable whether the gain is worth the added complexity.

4.2. Related Work

Hermenegildo [9, 11, 14] and others have noted the significance of left-biased scheduling mainly to ensure the no-slowdown criteria, meaning that less speculative work is performed first. Combined with determinism, we argue that parallel implementation overheads can be made minimal.

Our scheme can be viewed as merging deterministic parallel subgoals with the topmost parallel computation. This is similar to AKL [16, 15, 19] where deterministic AND-boxes are promoted. Gupta, Pontelli [22, 24] and Tang [23] use a similar technique in &ACE for reducing the overheads for independent AND-parallelism.

Carro, Gupta, Hermenegildo, Pontelli, Santos Costa, Tang [9, 10, 12, 14, 21, 23, 8] and others have proposed different schemes for reducing the overheads for nested independent AND-parallelism. These schemes reduce the performance difference between various independent AND-parallel systems and Reform Prolog, at the cost of added implementation complexity.

Our restriction on nested parallelism greatly simplifies the design compared with other nested AND-parallel systems like &-Prolog [9, 10, 12], DASWAM [26, 27] and ACE [7].

- No distributed stacks with markers [6, 9, 25] or multiple stack sets [10] are needed.
- No intelligent backtracking, or any other non-standard sequential backtracking scheme is needed.
- Very small overheads for suspension are possible.
- Busy wait can be used without deadlock detection machinery.
- Large number of processes can be created simultaneously.

Admittedly other designs may be able to exploit more implicit parallelism. Our design relies on programmers to rewrite their programs to fit our execution model.

Hermenegildo and Carro [13] observe that &-Prolog can be extended with support for data-parallel execution, similar to Reform Prolog.

There are some significant differences.

1. Reform Prolog is a (somewhat restricted) dependent AND-parallel system (not as general as DASWAM [27] which, however, imposes some significant performance penalties). It is possible to have producer-consumer dependencies in parallel loops.

   Data dependencies are carefully tracked by the compiler and necessary synchronisation primitives are generated.

   It is not clear how difficult it would be to incorporate these compilation techniques into &-Prolog.

2. We argue that the performance and execution models for Reform Prolog are clearer than for implicitly AND-parallel systems.

Possible future work include investigating to which extent the results for scheduling data-parallelism in imperative languages, e.g. Blelloch, Gibbons and Matias [5], can be applied in Reform Prolog.

5. Conclusion

This paper aims to show that there is no need for a complicated implementation technique to efficiently take advantage of nested data parallelism, and consequently a substantial part of nested *dependant* AND-parallelism using Lindgren's transformation [17].

We have proposed simple extensions to the execution model of Reform Prolog which make it possible to efficiently exploit nested recursion parallelism.

Initiating parallel execution, and obtaining work when suspended, is only allowed in deterministic states. Nested work is only allowed to be obtained from the left of the suspension point in the resolvent.

These restrictions make it possible to efficiently exploit nested parallelism with exceptionally small overheads in term of execution time and implementational complexity.

It is reasonable to assume that our approach will be easy to adopt in high performance implementations due to its streamlined design and the minimal changes to the sequential execution model.

6. Acknowledgements

Thanks to Thomas Lindgren and Håkan Millroth for valuable discussions, and to Manuel Hermenegildo and Enrico Pontelli for pointers to related research.

References

[1] J. Bevemyr, *A Recursion-Parallel Prolog Engine*, PhL Thesis, Computing Science Department, Uppsala University, 1993.

[2] J. Bevemyr, T. Lindgren & H. Millroth, Exploiting recursion-parallelism in Prolog, *Int. Conf. PARLE-93* (eds. A. Bode, M. Reeve & G. Wolf), Springer LNCS 694:279–290, Springer-Verlag, 1993.

[3] J. Bevemyr, T. Lindgren & H. Millroth, Reform Prolog: The language and its implementation, *Proc. 10th Int. Conf. Logic Programming*, MIT Press, 1993.

[4] G. Blelloch, *Vector Models and Data-Parallel Computing*, MIT Press, 1990.

[5] G. Blelloch, P. Gibbons, Y. Matias, Provably Efficient Scheduling for Languages with Fine-Grained Parallelism, *7th ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[6] P. Borgwardt, Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors, *IEEE Symp. Logic Programming*, 1984.

[7] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa, ACE:And/Or-parallel Copying-based Execution of Logic Programs, *Proc. Int. Conf. Logic Programming*, MIT press, 1994.

[8] G. Gupta, E. Pontelli, Last Alternative Optimization, *8th IEEE Symposium on Parallel and Distributed Processing*, IEEE Computer Society, 1996.

[9] M. V. Hermenegildo, *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, PhD thesis, The University of Texas At Austin, 1986.

[10] M. V. Hermenegildo, An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs, *Proc. 3rd Int. Conf. Logic Programming*, LNCS 225, Springer-Verlag, 1986.

[11] M. V. Hermenegildo, Relating Goal Scheduling, Precedence and Memory Management in AND-Parallel Execution of Logic Programs, *Fourth Int. Conf. Logic Programming*, MIT Press, 1987.

[12] M. V. Hermenegildo, K. J. Green, &-Prolog and its Performance: Exploiting Independent And-Parallelism, *Proc. 7th Int. Conf. Logic Programming*, MIT Press, 1990.

[13] M. V. Hermenegildo, M. Carro, Relating Data-Parallelism and (And–) Parallelism in Logic Programs, to be published in *Computer Languages Journal.*

[14] M.V. Hermenegildo, F. Rossi, Non-strict independent and-parallelism, in *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press 1990.

[15] S. Janson, *AKL A Multiparadigm Programming Language*, Ph. D. thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1994.

[16] S. Janson, J. Montelius, The Design of the AKL/PS 0.0 Prototype Implementation of the Andorra Kernel Language, ESPRIT Deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992.

[17] T. Lindgren, Compiling for Nested Recursion-Parallelism, submitted to the Parallelism and Implementation Technology for (Constraint) Logic Programming Languages workshop at JICSLP'96.

[18] H. Millroth, *Reforming Compilation of Logic Programs*, PhD thesis, Computing Science Department, Uppsala University, 1990.

[19] J. Montelius, K. A. M. Ali, An And/Or-Parallel implementation of AKL, *New Generation Computing*, 13(4), 1995.

[20] D. Palmer, J. Prins, S. Westfold, Work-efficient Nested Data Parallelism, *Frontiers of Massively Parallel Computing*, 1994.

[21] E. Pontelli, G. Gupta, D. Tang, M. Carro, M. Hermenegildo, Improving the Efficiency of Nondeterministic Independent And-parallel Systems, to appear in *Journal of Computer Languages.*

[22] E. Pontelli, G. Gupta, Data Parallel Logic Programming in &ACE, *Proc. of the IEEE International Symposium on Parallel and Distributed Processing*, IEEE, 1995.

[23] E. Pontelli, G. Gupta, Determinacy Driven Optimization of And-Parallel Prolog Implementations, *Int. Conf. Logic Programming*, MIT Press, 1995.

[24] E. Pontelli, G. Gupta, Nested Parallel Call Optimisation, Technical Report, New Mexico State University,

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

[26] K. Shen, Implementing Dynamic Dependent And-parallelism, *Proc. 10th Int. Conf. Logic Programming*, MIT Press, 1993.

[27] K. Shen, Exploiting And-parallelism in Prolog: the Dynamic Dependant And-parallel Scheme (DDAS), *Proc. Joint Int. Conf. and Symp. Logic Programming*, MIT Press, 1992.

[28] Warren, D. H. D., An Abstract Prolog Instruction Set, SRI Tech. Note 309, SRI International, Menlo Park, Calif., USA, 1983.