Exploiting Recursion-Parallelism in Prolog [*]

Johan Bevemyr   Thomas Lindgren   Håkan Millroth

Computing Science Dept., Uppsala University
Box 311, S-75105 Uppsala, Sweden
Email: {bevemyr,thomasl,hakanm}@csd.uu.se

Abstract

We exploit parallelism across recursion levels in a deterministic subset of Prolog. The implementation extends a convential Prolog machine with support for data sharing and process managment. Extensive global dataflow analysis is employed to facilitate parallelization. Promising performance figures, showing high parallel efficiency and low overhead for parallelization, have been obtained on a 24 processor shared-memory multiprocessor.

1 INTRODUCTION

The Single Program Multiple Data (SPMD) model of parallel computation has recently received a lot of attention (see e.g. the article by Bell [1]). The model is characterized by each parallel process running the same program but with different data.[1] The attraction of this model is that it does not require a dynamic network of parallel processes: this facilitates efficient implementation and makes the parallel control-flow comprehensible for the programmer.

We are concerned here with the SPMD model in the context of logic programming. For recursive programs, the different recursive invocations of a predicate are all executed in parallel, while all other calls are executed sequentially. We refer to this variant of (dependent) AND-parallelism as *recursion-parallelism.* A recursive invocation minus the head unification and the (next) recursive call is referred to as a *recursion level.* Each recursion level constitutes a process, which gives the programmer an easy way of estimating the process granularity of a given program or call.

We implement recursion-parallelism by *Reform compilation* [6] (this can be viewed as an implementation technique for the Reform inference system [11]).

[*]To appear in: *Proc. PARLE'93*, Springer LNCS, 1993.

[1]This should not be confused with the Single Instruction Multiple Data or SIMD model, where processes are synchronized instruction by instruction.

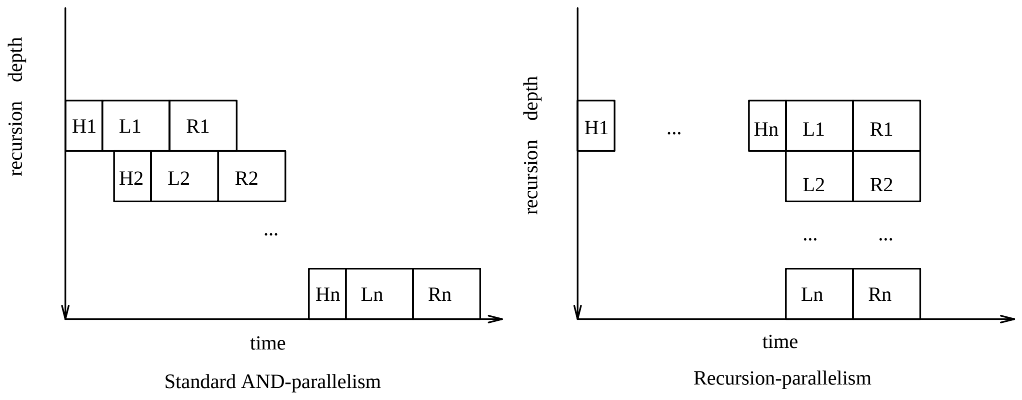

Figure 1: Executing a clause $H \leftarrow L, H', R$ with standard AND-parallelism and recursion-parallelism.

This is a control-structure transformation that changes the control-flow of a recursive program quite dramatically. When invoking a recursive program with a call of size n (corresponding to a recursion depth n) a four-phase computation is initiated:

1. A big head unification, corresponding to the n small head unifications with normal control-flow, is performed.
2. All n instances of the calls to the *left* of the recursive call are computed in parallel.
3. The program is called recursively. This call is known to match the base clause. Hence, in practice, this call is often trivially cheap.
4. All n instances of the calls to the *right* of the recursive call are computed in parallel.

The difference between standard AND-parallelism and recursion-parallelism is illustrated in Figure 1. The figure shows execution of a recursive clause $H \leftarrow L, H', R$, where H is the head, H' is the recursive call and L, R are (possibly empty) conjunctions. Note that the figure shows a situation where there are no data dependencies between recursion levels, and no unification parallelism.

This paper is organized as follows. In Section 2 we define Reform Prolog. We discuss some programming techniques and concepts in Section 3. An overview of the parallel abstract machine is presented in Section 4. Section 5 provides an overview of the compile-time analyses employed for parallelization. Extensions to the sequential instruction set are presented by means of an example in Section 6. Experimental results obtained when running benchmark programs on a parallel machine are presented and discussed in Section 7.

2 REFORM PROLOG

Reform Prolog parallelizes a deterministic subset of Prolog. This is similar to the approach taken in Parallel NU-Prolog [7]. However, Reform Prolog exploits recursion-parallelism when parallelizing this subset, whereas Parallel NU-Prolog exploits AND-parallelism.

With Reform Prolog, as with Parallel NU-Prolog, it is straight-forward to call parallel subprograms from a nondeterministic program. Thus, there is a natural embedding of parallel code in ordinary sequential Prolog programs. Since the entire call tree below a parallel call is not parallelized, some non-deterministic computations can be supported in a parallel context as shown below. This is not done in Parallel NU-Prolog.

Below we define the condition for when a recursive Prolog predicate can be parallelized, and how this condition can be enforced by the implementation. We need to define two auxiliary concepts:

- A variable is *shared* if it is visible from more than one recursion level. Note that a variable can be shared at one point of time and unshared (local) at another.
- A variable binding is *unconditional* if it cannot be undone by backtracking.

We say that a call in a parallel computation is *safe* if all bindings made to its shared variables are unconditional when the call is executed. The condition for when a predicate can be parallelized is then:

> **A recursive predicate can be parallelized only if all calls made in the parallel computation are safe.**

Hence, limited non-determinism is allowed: when conditional bindings are made only to variables local to a recursion level, the computation is safe.

Safeness of a call is defined w.r.t. to the *instantiation* of the call (i.e., what parts of the arguments are instantiated). We can distinguish between the parallel instantiation and the sequential instantiation of a call. These might differ as a parallel call can 'run ahead' of the sequential instantiation: recursion levels that would execute after the current one sequentially, may already have bound shared variables.

We say that a call is *par-safe* when it is safe w.r.t. the parallel instantiation, and that it is *seq-safe* when it is safe w.r.t. the sequential instantiation.

The compiler is responsible for checking that programs declared parallel by the programmer are safe. For calls that can be proven par-safe at compile-time, there is no need for extra safeness checking at runtime. For calls that can be proven seq-safe at compile-time, but not par-safe, it is necessary to check safeness at runtime. If the call is not safe, then it is delayed until it becomes safe. This is done by suspending the call until:

1. The unsafe argument has become sufficiently instantiated by another recursion level; or
2. The current call becomes leftmost.

If neither par-safeness nor seq-safeness can be proven at compile-time, then parallelization fails.

3 RECURSION PARALLEL PROGRAMMING

Before describing the execution machinery, we briefly consider some programming techniques and concepts.

3.1 Machine utilization

The parallel programmer is concerned with utilizing the available parallel machine as efficiently as possible. In Reform Prolog, this means creating sufficient work and avoiding synchronization due to data dependences. A parallel machine where most of the workers are inactive due to lack of work is underutilized; a parallel machine where most workers are waiting for data is inefficiently used.

The number of processes available to the workers is precisely the number of recursion levels of the parallel call. To keep a large machine busy, there should consequently be many recursion levels – far more than the number of workers, ideally.

3.2 Safeness

To illustrate the concept of safeness, consider the following two programs:

```
split([],_,[],[]).
split([X|Xs],N,[X|Ys],Zs) :- X =< N, split(Xs,N,Ys,Zs).
split([X|Xs],N,Ys,[X|Zs]) :- X > N, split(Xs,N,Ys,Zs).

split*([],_,[],[]).
split*([X|Xs],N,Ys,Zs) :- X =< N, !, Ys = [X|Ys0], split*(Xs,N,Ys0,Zs).
split*([X|Xs],N,Ys,[X|Zs]) :- split*(Xs,N,Ys,Zs).
```

Assume that the third arguments of both predicates are shared, as might be the case if they were called from a parallel predicate. The predicate split/4 is then unsafe, since the third argument might be conditionally bound in the second clause, and then unbound again if the comparison $X \leq N$ fails. In contrast, the third argument of split∗/4 is only bound in a determinate state and so split∗/4 is safe for parallel execution (the binding of the fourth argument in the last clause does not affect safeness, since clauses are tried in textual order).

3.3 Suspension

The programmer would like to avoid suspension as far as possible. However, in the implementation described in this paper, cheap suspension and simple, efficient scheduling combine to lessen synchronization penalties considerably. Consider the following program:

```
nrev([],[]).
nrev([X|Xs],Zs) :- nrev(Xs,Ys), append(Ys,[X],Zs).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

The compiler detects that the first argument of append/3 is shared. Hence, indexing must suspend until the first argument Ys is instantiated by the previous recursion level. The third argument Zs is also found to be shared, but the situation is reversed: the next recursion level will wait for Zs to be bound.

The inner loop of append/3 is then: wait for the input to be instantiated; when this occurs, write an element on the output list and go back to the beginning again. Thus, there is considerable scope for overlapping computations. As can be seen in the benchmark section, speedups are almost linear on 24 processors.

We also ran a second version of nrev/2, where the data dependence of append/3 was removed by a simple transformation: the length of the first argument is known at call-time by an extra parameter. Every call to append/3 can then construct the list Ys asynchronously (the elements of the list will be filled in later) and there is no suspension under execution.

```
nrev*(0,[],[]).
nrev*(N+1,[X|Xs],Zs) :- nrev*(N,Xs,Ys), append*(N,Ys,[X],Zs).

append*(0,[],Ys,Ys).
append*(N+1,[X|Xs],Ys,[X|Zs]) :- append*(N,Xs,Ys,Zs).
```

Surprisingly, the nrev∗/3 program is slower than the suspending nrev/2 program. Measurements show that this is because recursion levels of nrev/2 usually suspend very briefly, due to simple, fast suspension and straightforward scheduling of processes. For 16 processors, no processor was suspended more than 0.6% of the total execution time on the nrev/2 program; when run on 8 processors, the program suspended each worker less than 0.1% of the execution time.

On the other hand, the asynchronous nature of constructing the answer lists in nrev∗/3 led to an increase in the number of general unifications, due to later recursion levels overtaking earlier ones. The cost is time and memory. (Note that nrev∗/3 still exhibited a speedup of approximately 13 on 16 processors; nrev/2, however, was clocked at a speedup of over 15 on 16 processors.)

4 THE PARALLEL ABSTRACT MACHINE: OVERVIEW

The Reform engine consists of a set of *workers*, at least one per processor. Each worker is a separate process running a WAM-based [12] Prolog engine with extensions to support parallel execution. The Prolog engine is comparable in speed with SICStus Prolog (slightly faster on some programs, slightly slower on others).

The Reform engine alternates between two modes: sequential execution and parallel execution. One dedicated worker (the sequential worker) is responsible for the sequential execution phase. During this phase all other workers (the parallel workers) are idle. The sequential worker initiates parallel execution and resumes sequential execution when all parallel workers have terminated.

A common code area is used and all workers have restricted access to each others heaps. All other data areas are private to each worker. The shared heaps are used to communicate data created during sequential and parallel execution (an atomic exchange operation is employed when binding possibly shared heap variables). When there are several shared heaps it is no longer possible to use a simple pointer comparison for determining whether a binding should be trailed or not. We solve this problem by extending the representation of each variable with a timestamp.

The sequential worker's temporary registers can be read by the parallel workers. These registers are employed for passing arguments to the parallel computation (one such register contains the precomputed recursion depth, i.e., the total number of parallel processes).

Synchronization is implemented by busy-waiting, i.e., suspended processes actively check if they can be resumed. The drawback of this method is that a suspended process tie up a processor. The advantage is that non-suspended processes are not slowed down. In particular, very simplisitic and efficient approaches to process scheduling are possible. The Reform Prolog implementation currently supports *static scheduling* and *dynamic self-scheduling* [10]. With both approaches, the actual task switching amounts to a simple jump operation.

5 COMPILING RECURSION PARALLEL PROGRAMS

The compiler ensures the safeness of the computation, guarantees that time-dependent operations are performed correctly and employs suspending and locking unification when necessary.

These tasks depend on compile-time program analyses that uncover type, locality and determinacy information. The compiler then emits instructions based on this information, possibly falling back to more conservative code generation schemes when high-precision analysis results cannot be obtained.

5.1 Type analysis

The type inference phase employs an abstract domain based on the standard mode-analysis domain [5], augmented with support for lists and difference lists as well as handling of certain aliases.

The compiler distinguishes the parallel and sequential types of a predicate. The sequential type is the type that must hold when the current recursion level is leftmost, while the parallel type holds for any recursion level. The parallel type is the most frequently used for compilation.

5.2 Locality analysis

Locality analysis tries to find what terms are local to a process (recursion level), what terms are shared between processes and what terms contain variables subject to time-dependent tests. Consider the following program:

```
rp([],[]).
rp([X|Xs],[Y|Ys]) :- p(X,Y), rp(Xs).

p(a,X) :- q(X).                   p(b,c).

q(X) :- var(X),!,X = b.           q(c).
```

Given the call rp([a,b],[Y,Y]), we get two processes, p(a,Y) and p(b,Y). Both are safe and can thus proceed in parallel. Now assume p(b,Y) precedes p(a,Y), and binds Y to c. Then p(a,c) will reduce to q(c) which succeeds.

Sequential Prolog would have a quite different behaviour: first, p(a,Y) reduces to q(Y) and in turn to Y = b. Then p(b,b) fails. Hence, for this example, the parallel execution model is unsound w.r.t. sequential Prolog execution. This is due to the time-dependent behaviour of the primitive var/1.

Our solution to this problem is to mark, at compile-time, certain variables as being time-sensitive, or *fragile*. In the example, the argument of q/1 is fragile and the compiler must take this into account.

Furthermore, knowledge that a variable is *not* fragile, or *not* shared is extremely useful to the code generator. Using such information, operations with a very complex general case, such as unification, can in some cases be reduced to the same code as would be executed by a sequential WAM.

5.3 Safeness analysis

Safeness analysis aims to ensure that no conditional bindings are made to shared variables. In this respect, it is quite different from determinacy analysis: while determinacy analysis attempts to prove that a given call yields at most a single solution, safeness analysis instead proves no unifications with shared terms are made in a non-determinate, parallel state. Safeness analysis thus employs the results of type inference (to see whether the call is determinate or not) and locality analysis (only shared terms can be unsafe).

6 INSTRUCTION SET

The sequential WAM instruction set is extended with new instructions for supporting recursion-parallelism. Due to space limitations, we can only describe these by means of a simple example and refer to other sources for a full discussion [3, 2, 4]. Consider the program:

```
map([],[]).
map([A|As],[B|Bs]) :- foo(A,B), map(As,Bs).
```

The program is compiled into the following extended WAM code.

```
map/2:  switch_on_term Lv La L1 fail

Lv:     try La
        trust L1

La:     get_nil X0
        get_nil X1
        proceed

L1:     build_rec_poslist X0 X2 X3 X0     % first list
        build_poslist X1 X2 X4 X1         % second list
        start_left_body L2                % execute L2 in parallel
        execute map/2                   % call base case

L2:     initialize_left 1               % I := initial recursion level
L3:     spawn_left 1 X2 G2              % X2 := I++; while(I < N) do
        put_nth_head G3 X2 0 X0         %  X0 := G3[I]
        put_nth_head G4 X2 0 X1         %  X1 := G4[I]
        call foo/2 0                    %  call foo(G3[I],G4[I])
        jump L3                         % next iteration
```

The instructions `build_rec_poslist` and `build_poslist` employs a data structure called a *vector-list*. This is a list where the cells are allocated in consequtive memory locations, to allow constant-time indexing into the list. The instructions work as follows:

- `build_rec_poslist X0 X2 X3 X0` traverses the list in X0 and builds a vector-list of its elements, storing a pointer to it in X3, and storing the vector length in X2. The last tail of the list is stored in X0.

- `build_poslist X1 X2 X4 X1` traverses the list X1 to its X2'th element, and builds a vector-list of its elements in X4. If the list has fewer than X2 elements and ends with an unbound variable, then it is filled out to length X2. The X2'th tail of the vector-list is unified with the X2'th tail of the list in X1, and finally X1 is set to the X2'th tail of the vector-list.

The sequential worker's registers X2, X3 and X4 are referred to as G2, G3 and G4 in the parallel code.

The instruction `initialize_left` calculates the initial recursion level in static scheduling mode. In dynamic scheduling mode, this instruction is ignored.

7 EXPERIMENTAL RESULTS

In this section we present the results obtained when running some benchmark programs in Reform Prolog on a parallel machine.

7.1 Experimental methodology

Parallel machine. Reform Prolog has been implemented on the Sequent Symmetry multiprocessor. This is a bus-based, cache-coherent shared-memory machine using Intel 80386 processors. The experiments described here were conducted on a machine with 26 processors, where we used 24 processors (leaving two processors for operating systems activitites).

Evaluation metrics. The metric we use for evaluating parallelization is the speedup it yields. We present *relative* and *normalized* speedups.

Relative speedup expresses the ratio of execution time of the program (compiled with parallelization) on a single processor to the time using p processors.

Normalized speedup expresses the ratio of execution time of the original sequential program (compiled without parallelization) on a single processor to the time using p processors on the program compiled with parallelization.

7.2 Benchmarks.

Programs and input data. We have parallelized four benchmark programs. Two programs (Match and Tsp) are considerably larger than the other two. One program (Map) exploits independent AND-parallelism, whereas the other three exploits dependent AND-parallelism.

Map. This program applies a function to each element of a list producing a new list. The function merely decrements a counter 100 times. A list of 10,000 elements was used.

Nrev. This program reverses a list using list concatenation ('naive reverse'). A list of 900 elements was used.

Match. This program employs a dynamic programming algorithm for comparing, e.g., DNA-sequences. One sequence of length 32 was compared with 24 other sequences. The resulting similarity-values are collected in an sorted binary tree.

Tsp. This program implements an approximation algorithm for the Travelling Salesman Problem. A tour of 45 cities was computed.

Load balance. One way of estimating the load balance of a computation is to measure the finishing time of the workers. We measured the execution time for each worker when executing our benchmarks. Static scheduling was used in all experiments.

Map. This program displayed a very uniform load balance (less than 0.3% difference between workers). This is hardly surprising since the number of recursion levels executed by each worker is large, and there is no difference in execution time between recursion levels.

Nrev. The execution time of each worker only varied about 3% when executing this program. There is a slight difference in the execution time of each recursion level but the large number of recursion levels executed by each worker evens out the differences.

Match. When 16 workers were used, 8 workers executed 2 recursion levels each, while 8 workers executed a single recursion level. This explains the relatively poor speedup on 16 workers. When 24 workers were used the execution time varied less than 0.3% between workers. This is explained by the fact that each worker executed one recursion level, and that all recursion levels executed in the same time.

Match and Tsp. These program displayed an uniform load balance on all but three workers. This is explained by the fact that 45 recursion levels were executed in all; 21 workers executed 2 recursion levels each while 3 workers executed 1 recursion level each. Despite this the program displays good speedup (21.85). Using dynamic scheduling would not have improved the results in this case.

Sequential fraction of runtime. Each parallelized benchmark program has an initial part which is not parallelized. This includes setting up the arguments for the parallel call. It also includes head unification, and spawning of parallel processes.

According to Amdahl's law, the ratio of time spent in this sequential part of the program to that spent in the part which is parallelized (measured on a sequential machine) determines the theoretically best possible speedup from parallelization.

The following table shows for each benchmark program how large fraction of the execution time on a sequential machine is not subject to parallelization.

Map	Nrev	String	Tsp
0.3%	0.04%	0.003%	0.005%

We conclude from this data that the unparallelized parts represent negligible fractions of the total execution times. Another conclusion is that there is no point in parallelizing the head unification of parallelized predicates, since it represents such a tiny fraction of the computation.

7.3 Results

The results of the experiments are summarized in the tables below. In the tables P stands for number of workers, T for runtime (in seconds), S_R for relative speedup, and S_N for normalized speedup. The sequential runtime for each program is given below each table.

P	T	S_R	S_N	T	S_R	S_N
1	40.40	1.00	0.98	30.80	1.00	0.88
4	10.12	3.99	3.89	8.08	3.81	3.43
8	5.07	7.96	7.76	3.96	7.77	6.99
16	2.54	15.91	15.50	2.01	15.32	13.78
24	1.70	23.76	23.15	1.36	22.65	20.36

Map. (39.59 sec.) **Nrev**. (27.70 sec.)

P	T	S_R	S_N	T	S_R	S_N
1	68.88	1.00	0.95	258.22	1.00	0.90
4	17.22	3.99	3.80	68.85	3.75	3.37
8	8.61	7.99	7.60	34.55	7.47	6.73
16	5.76	11.95	11.35	17.25	14.96	13.47
24	2.91	23.70	22.52	11.82	21.85	19.67

Match. (65.44 sec.) **Tsp**. (232.40 sec.)

7.4 Discussion

From the above results we calculate parallel overhead and efficiency of parallelization and make a comparison with other systems.

We compare Reform Prolog with the only other Prolog systems, which supports deterministic dependent AND-parallelism, that we are aware of, Andorra-I [9] and NUA-Prolog [8]. The Andorra-I system is an interpreter written in C. NUA-Prolog is a compiled system using a WAM-based abstract machine.

It should be noted that these systems to some extent exploit different forms of parallelism. Reform Prolog and NUA-Prolog exploit deterministic dependent AND-parallelism (recursion parallelism in the case of Reform Prolog). Andorra-I exploits deterministic dependent AND-parallelism and OR-parallelism (here we are only interested in the AND-parallel component of the system).

Unfortunately, we can only make a very limited comparison with NUA-Prolog, since the published benchmark programs stress the constraint-solving capabilites of the system, rather than it potential for raw AND-parallel speedup. However, we have compared their result on the nrev benchmarks with ours.

Let us define

$$\text{parallel efficiency} = (\text{speedup on } N \text{ processors})/N$$

The table below displays the parallel efficiency of Reform Prolog on 24 processors. It also indicates the parallelization overhead on a single processor as compared to the sequential Prolog implementation.

Program	Relative efficiency	Normalized efficiency	Parallelization overhead
Map	99 %	96 %	2 %
Nrev	95 %	83 %	12 %
Match	99 %	94 %	5 %
Tsp	91 %	82 %	10 %

The Andorra-I system shows relative efficiency in the range of 47–83 % and normalized efficiency in the range of 35–61 % (assuming parallelization overhead of 35 %). We have excluded benchmarks that mainly exhibits OR-parallelism from this comparison. The figures are obtained on a Sequent Symmetry with 10 processors. NUA-Prolog shows a relative efficiency of 71 % and a normalized efficiency of 36 % on the nrev benchmark on an 11 processor Sequent Symmetry. (Note that the Reform Prolog figures were obtained using more than twice the number of processors the other systems used—using less processors improves the result.)

The Andorra-I system shows parallelization overheads in the range of 35–40 % on a set of benchmarks [9]. NUA-Prolog shows a parallelization overhead of 50 % on the nrev benchmark.

8 CONCLUSIONS

The developments and results discussed in this paper suggests that recursion-parallelism is an efficient method for executing Prolog programs on shared-memory multiprocessors. Our implementation exhibits very low overhead for parallelization (2–12 % on the programs tested).

We believe that the high parallel efficiency of Reform Prolog is due to efficient process management and scheduling. An important factor is that all parallel processes can be initiated simultaneously. These properties of the system are due to the static recursion-parallel execution model made possible by Reform compilation.

Acknowledgments

We thank the Swedish Institute of Computer Science (SICS) for making their 26 processor Sequent Symmetry available to us.

REFERENCES

[1] G. Bell, Ultracomputers: a Teraflop before its time, *Comm. ACM*, Vol. 35, No. 8, 1992.

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

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

[4] T. Lindgren, *The Compilation and Execution of Recursion-Parallel Prolog on Shared Memory Multiprocessors*, PhL Thesis, Computing Science Department, Uppsala University, May, 1993 (expected).

[5] S.K. Debray & D.S. Warren, Automatic Mode Inference for Logic programs, *J. Logic Programming*, Vol. 5, No. 3, 1988.

[6] H. Millroth, Reforming Compilation of Logic Programs, *Proc. Int. Symp. Logic Programming*, San Diego, Calif., MIT Press, 1991.

[7] L. Naish, Parallelizing NU-Prolog, *Proc. 5th Int. Conf. Symp. Logic Programming*, MIT Press, 1988.

[8] D. Palmer and L. Naish, NUA-Prolog: An Extension to the WAM for Parallel Andorra, *Proc. 8th Int. Conf. Logic Programming*, Paris, MIT Press, 1991.

[9] V. Santos Costa, D.H.D. Warren and R. Yang, The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model, *Proc. 8th Int. Conf. Logic Programming*, Paris, MIT Press, 1991.

[10] P. Tang and P.-C. Yew, Processor Self-Scheduling for Multiple Nested Parallel Loops, *Proc. 1986 Int. Conf. Parallel Processing*, 1986.

[11] S.-Å. Tärnlund, Reform, report, Computing Science Dept., Uppsala University, 1991.

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