Reform Prolog: The Language and its Implementation [*]

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

Reform Prolog is an (dependent) AND-parallel system based on recursion-parallelism and Reform compilation. The system supports selective, user-declared, parallelization of binding-deterministic Prolog programs (nondeterminism local to each parallel process is allowed). 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. The high performance is due to efficient process managment and scheduling, made possible by the execution model.

1 INTRODUCTION

Most systems for AND-parallel logic programming defines the procedural meaning of conjunction to be inherently parallel. These designs are based on an ambition to maximize the amount of parallelism in computations. We present and evaluate an approach to AND-parallelism aimed at maximizing not parallelism but machine utilization. The system supports selective, user-declared, parallelization of Prolog.

Reform Prolog supports parallelism only across the different recursive invocations of a procedure. Each such invocation constitutes a process, which gives the programmer an easy way of estimating the control-flow and process granularity of a program. We refer to this variant of (dependent) AND-parallelism as *recursion-parallelism.*

We implement recursion-parallelism by *Reform compilation* [9] (this can be viewed as an implementation technique for the Reform inference system [15]). 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 unification with normal control-flow, is performed.

2. All n instances of the calls to the *left* of the recursive call are computed in parallel.

[*]To appear in: *Proc. 10th Int. Conf. Logic programming*, MIT Press, 1993.

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.

This approach is somewhat akin to loop parallelization in imperative languages such as Fortran. However, an important difference is that the granularity of top-level (or near top-level) recursive Prolog procedures typically far exceeds that of parallelizable Fortran loops. A major feature of the approach is that it allows a static process structure: all parallel processes are initialized when the parallel computation starts. This has profound impact on performance.

This paper is organized as follows. The Reform Prolog execution model is defined in Section 2. In Sections 3 to 5, the design of the parallel abstract machine is discussed. The global dataflow analysis employed by the compiler is outlined in Section 6. The extension of the instruction set for parallel execution is described in Section 7. Section 8 presents our experimental results.

2 REFORM PROLOG

Reform Prolog parallelizes a deterministic subset of Prolog. Below we define the condition for when a recursive Prolog predicate can be parallelized. We then consider how this condition can be enforced by the implementation. We need to define two auxiliary concepts:

- A variable is *shared* if it is accessible 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.

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.**

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 neccessary to check safeness at runtime. If the call is not safe, then it is delayed until it becomes safe. This is done by suspending 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, parallelization fails.

The execution model described above has some similarities to the approach taken in Parallel NU-Prolog [10] in that both approaches parallelize a binding-deterministic subset of Prolog. 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.

Safeness allows local nondeterminism in each recursion level as long as no unsafe bindings are made. In Parallel NU-Prolog this is not done, since the entire proof tree is subject to parallelization. The consequence is that any variable may be accessible to other processes and so any binding in a nondeterminate state may be unsafe.

3 THE PARALLEL ABSTRACT MACHINE

The parallel machinery consists of a set of workers numbered 0,1,...,$n-1$, one per processor. Each worker is implemented as a separate process running a WAM-based Prolog engine with extensions to support parallel execution.

3.1 Execution phases

The execution of a program alternates between two modes: sequential execution and parallel execution. A phase of sequential execution is referred to as a *sequential phase* and a phase of parallel execution as a *parallel phase*. One worker is responsible for sequential execution (the *sequential* worker). During sequential execution all other workers (the *parallel* workers) are idle, during parallel execution the sequential worker is idle.

3.2 Memory Layout

A standard WAM implementation has three main data areas: heap, stack and trail. The heap contains variables and data structures, the stack contains choicepoints and environments, and the trail contains references to bound variables. Environments contain only variables which are either unbound, referring to other variables in the stack, or referring to variables or structures on the heap.

In the Reform engine each worker has its own distinct data areas (heap, stack and trail). The stack and the trail are local to each worker and cannot be accessed by other workers. All heaps are shared and all workers have restricted access to other workers' heaps. The motivation for the design is given below.

Heap. Each parallel worker might require access to data built on the heap of the sequential worker. Moreover, if there are data dependences in the Prolog program, then parts of each worker's data might have been created by other workers during the current parallel phase. The easiest way to support this is to let all workers have access to each others heaps. This access is restricted so that no worker may create new objects on another worker's heap, but only bind existing variables. Such an arrangement ensures simple memory management of each heap.

Stack. The stacks can remain unshared if it can be ensured that no references to stack objects will occur in shared objects. Since heap variables cannot be bound to stack objects, the only restriction that has to to be imposed is to disallow stack objects in the arguments to the parallel call.

Trail. If each worker is responsible for any unconditional bindings it has created, then the trail can remain local. For this to work, the sequential worker must notify all workers when it backtrack across a parallel phase and let each worker undo the conditional bindings it created during that parallel phase.

3.3 Heap-allocated vectors

Some variables, that would be allocated on the stack in a sequential WAM, are allocated in vectors stored on the heap. Consider, for example, the 'naive reverse' program:

```
nrev([],[]).
nrev([A|X],Y) :- nrev(X,Z), append(Z,[A],Y).
```

Here the variable Y on each recursion level is bound to the variable Z on the next recursion level. In a sequential WAM both Y and Z would be stored on the stack. This is not possible in our machine for two reasons. First, stacks are not shared between workers. Second, all instances of the variables are created *before* the parallel execution of the append calls is initiated, as a consequence of our execution model.

Hence all instances of the variables Y and Z are allocated in vectors on the heap rather than in binding environments on the stack. This is an example on the tradeoff between efficient memory usage (stack allocation) and increased potential for parallelism (heap allocation). For tail-recursive programs this effect is even more dramatic.

The Reform engine also exploits heap-allocated vectors for another purpose. Consider the list in the first argument position of nrev/2 above. The sequential engine traverses the list and builds a vector of its elements before the parallel phase is initiated. This is neccessary since we (*i*) need to know the final recursion depth (i.e. the length of the list) *before* the parallel phase is initiated, in order to determine how many parallel processes (recursion levels) to spawn, and (*ii*) need to index into the list during the parallel phase.

A problem is that after the parallel phase, the vector should be viewed as a linked list again. Our solution to this problem is to build a 'vector-list' by allocating the cons cells densely one after the the other. In this way we can index into the list during the parallel phase, and still view it as a linked structure in the following sequential phase.

4 DATA SHARING

There are two aspects of data sharing between workers. First, terms created by the sequential worker (variables, numbers, structures and lists) must be accessible to parallel workers. Second, terms created by parallel workers must likewise be accessible to other parallel workers, and to the sequential worker.

4.1 Binding variables

The memory layout described above gives each worker the ability to refer to objects shared with other workers. This ability might lead to a situation where two workers try to simultaneously bind the same variable, potentially with the result of one binding destroying the other.

We therefore use an atomic exchange operation when binding a variable. If another worker has managed to bind the variable ahead of this worker, then the exchange operation will return that binding. The other worker's binding must then be unified with this worker's, to ensure consistency. A similar method is used in the implementation of Parallel NU- Prolog [10].

We have found that in our system this method is significantly faster than spin locking [8].

4.2 Creating shared structures

When building a structure on the heap other workers should not have access to the structure until it has been fully initialized. In WAM a structure is built using either put instructions or get instructions. When put instructions are used there is no problem, since no variable is bound to the structure until it has been fully initialized. A get instruction, on the other hand, might bind a variable to an incomplete structure and then proceed to fill in the missing parts using unify instructions.

We avoid references to uninitialized structures by modifying the get instructions so that they do not bind the variable, but save it for later binding. A new instruction has to be introduced after the last unify instruction (when the structure is complete) to bind the variable. This method is also discussed in Naish's paper on Parallel NU-Prolog [10].

4.3 Trailing

Variables must be trailed in the parallel phase, even though only safe programs are parallelized. There are two reasons for this.

1. There might be local nondeterminism within each recursion level. If that is the case, the worker must be able to backtrack locally. This forces the worker to, at least, trail bindings of local heap and stack variables ('local' variables reside in a data area managed by the worker).

2. A parallel worker might bind variables created before the parallel execution started. Since sequential backtracking is allowed across parallel calls, the machine must be able to undo bindings created during parallel phases.

The problem with trailing is that each worker has its own heap to work on. It is no longer possible to simply compare a variable's position on the heap with a heap pointer to determine whether it should be trailed or not. The situation gets even worse if we consider what might happen when the sequential worker continues executing after a parallel execution. In this case there might be variables distributed over all workers' heaps. A simple comparison between pointers cannot be used to determine whether a variable should be trailed or not, no matter how the heaps are arranged.

Our solution to this problem is to extend heap variables with a *timestamp*. This implies that the WAM has to be extended with a counter that is incremented whenever a choice point is created. This timestamp has to be saved in the choice point and restored on backtracking. A timestamp comparison is then used for determining whether to trail a variable or not.

5 PROCESS MANAGMENT

Process management and scheduling are critical points in many of the parallel Prolog systems existing today.

In recursion-parallel systems much of the scheduling is done at compile time. It is then possible to determine which code is going to be executed in parallel. The number of processes executing the parallel code is determined immediately before parallel execution begins. These properties greatly simplify the process management problem; in fact, the scheduling overhead becomes negligible.

5.1 Suspension

The set of programs that can be parallelized in Reform Prolog has been restricted to those which are binding-deterministic with respect to shared variables. This condition is verified at compile time.

Recursion levels suspend only to preserve sequential semantics and safeness. To ensure sequential semantics, a recursion level suspends before binding variables subject to time-sensitive tests; the level resumes when the variable is instantiated or it becomes leftmost. In both cases, preceding recursion levels cannot be affected by the binding.

To ensure safeness, a recursion level must not conditionally bind shared variables. In this situation, the level suspends until the variable is instantiated. If the level becomes leftmost, instantiation would occur and a safeness violation is signalled.

Some operations, notably general unification, unpredictably instantiate terms. Recursion levels suspend until leftmost before general unifications with terms containing time-sensitive variables (in the sense of above); if a general unification might create conditional bindings to shared variables, parallelization currently fails.

We implement these tests by busy-waiting. For instance, a worker suspending to preserve sequential semantics repeatedly checks if the variable has become instantiated or if the goal has become leftmost.

This method has the drawback that a suspended worker can tie up a processor for a long time. Whether this is a problem or not depends on the application program. If the average waiting time is short, then the overhead of this method is negligible.

The advantage of this method is that it does not slow down processes that do not wait for data—only the waiting worker is slowed down. No extra overhead is imposed on the Reform engine as compared to a sequential implementation.

5.2 Scheduling

One of the advantages of the Reform execution model is that much of the scheduling can be done at compile time. It is possible to determine at compile time which code is going to be executed in parallel, and the number of recursion levels to run can be determined prior to parallel execution. In the Reform engine this is done by examining the recursion argument.

The scheduling process is thus reduced to dividing recursion levels among workers. There are two different approaches to scheduling: static scheduling and dynamic scheduling.

Static scheduling. Static scheduling minimizes the need for synchronization. Most parallel Prolog systems cannot use static scheduling since too little information is available about the structure of the parallel execution before it is initiated. In recursion-parallel systems, information about which code is going to be executed, as well as the number of parallel processes, is available. This makes it possible to distribute recursion levels to workers statically.

Of course, some programs are not well-suited for static scheduling, e.g., programs with poor load balancing due to significantly varying execution times of recursion levels.

Dynamic scheduling. The goal of dynamic scheduling methods is to optimize the tradeoff between large process granularity and good load balance.

The simplest dynamic algorithm for scheduling is *self-scheduling* [14]. In this algorithm each processor executes one recursion level at a time until all levels have been executed. This method achives almost perfect load balancing.

The problem with self-scheduling is, not surprisingly, that it tends to create too many processes with too fine granularity. However, this is less a problem in a recursion-parallel Prolog system than in loop-parallel Fortran systems, since the granularity of a single recursion level typically is greater than the granularity of a single iteration of a parallel loop.

More sophisticated dynamic algorithms has been proposed [6, 12, 7]. In these algorithms each processor is allocated a chunk of iterations at a time, instead of a single iteration. The chunk size may be fixed or variable. These algorithms have not yet been tested in Reform Prolog.

Task switching. Each worker is responsible for calculating which recursion levels it is going to execute. With static scheduling no synchronization is necessary to schedule work. With dynamic scheduling, on the other hand, it is necessary to synchronize accesses to a global variable holding the remaining number of unscheduled processes.

Task switching (i.e., starting a new process) amounts to a simple jump operation. With 'chunking' dynamic scheduling methods, the jump operation is preceded by an arithmetic calculation of how many processes to schedule in one chunk.

Mapping processes to processors. There are two simple ways to map recursion levels to processors regardless of whether we use static or dynamic scheduling (assuming chunks of more than one level in the latter case). Either consecutive recursion levels are mapped to consecutive processors (*horizontal* mapping), or consecutive recursion levels are mapped to the same processor (*vertical* mapping). If there are data dependencies in the program, then horizontal mapping is to be preferred since that enables process pipelining. If there are no data dependencies, on the other hand, then it might be better to use vertical mapping since it improves data locality.

6 COMPILING RECURSION PARALLEL PROGRAMS

Compilation has two main components: ensuring safeness and introduce suspension and locking unification instructions where required. This is managed by combining three analyses: type inference, safeness analysis and locality analysis.

The type inference domain is an extension of the Debray-Warren domain [4], with the addition of support for lists and difference lists. The compiler uses both parallel and sequential types in code generation; parallel types hold at all times in the program, while sequential types hold when leftmost.

Safeness analysis investigates when the computation is in a nondeterminate, parallel state. Determinacy changes on procedure entry and cuts. Note that there is no attempt to prove determinacy of a goal; instead the analyser merely records when determinacy may change. To get reasonable results, the analyser simulates indexing on procedure entry.

Locality analysis has two tasks: to find terms that are local to one recursion level, and to mark shared terms subject to time-sensitive operations (such terms are called *fragile* since they must be handled with care). Local operations do not require suspension or locking unification; shared terms require locking unification but no suspension, while fragile terms may not be instantiated out of the sequential order. If the compiler were not to respect fragility, the system might stray from simulating sequential behavior [3].

The goal of locality analysis is to generate precisely WAM code for parallel operations on unshared data. When this is possible, there is no parallelization overhead once the parallel execution has started. That is, the compiler attempts to localize the overheads of parallel execution to the points where the full machinery is actually needed.

7 INSTRUCTION SET

The sequential WAM instruction set is extended with new instructions for supporting recursion-parallelism.

7.1 New instructions

The new instructions can be divided into five groups as follows.

Creating shared structures. These instructions create structures that might be accessed by other workers while they are created.

Creating vectors. These instructions build vector-lists that are used in the parallel phase. They are executed by the sequential worker. There are instructions that convert lists into vector-lists, as well as instructions that create vector-lists corresponding to binding environments.

Accessing global arguments. These instructions are used by the parallel workers to fetch data needed at their recursion levels from the sequential working space.

Process control. These instructions are used by the sequential worker to spawn parallel workers, and by parallel workers to switch from one recursion level to the next.

Runtime safeness tests. These instructions perform runtime tests to enforce safeness.

7.2 Examples

The following programs illustrate the use of some of the new instructions.

In the description of each instruction we use the term *step*, by which we understand the number of elements the recursive argument is reduced by in each recursive call. Some registers are denoted Gn, this refer to the sequential worker's register Xn, which is globally accessible to all workers during a parallel phase.

Example 1:

```
        map([],[]).
        map([X|Xs],[Y|Ys]) :- foo(X,Y), map(Xs,Ys).
```

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:     % Sequential code for first clause of map/2

Lb:     % Sequential code for second clause of map/2

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              % while (I++ < N) do X2 := I;
        put_nth_head G3 X2 0 X0         %  X0 := G3[I]
        put_nth_head G4 X2 0 X1         %  X1 := G4[I]
        call foo/2 0                    %  foo(G3[I],G4[I])
        jump L3                         % od
```

Let us describe the effects of the code for the recursive clause (label L1).

- `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 list length of X0 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.

- `start_left_body L2` starts parallel execution of the foo/2 calls. The code at label L2 is run in parallel by all active workers. The sequential execution continues with the next instruction (execute map/2) when the parallel phase is finished.

- `initialize_left 1` initializes a worker for parallel execution. The step 1, given as argument, and the worker number is used for calculating the initial recursion level in static scheduling mode. In dynamic scheduling mode this instruction is ignored.

- `spawn_left 1 X2 G2` calculates the next recursion level for this worker and stores it in X2. The new level is calculated from the step 1 and the internal level count. If the new level is greater than the value stored in the global register G2 (i.e., the register X2 in the sequential worker), then then parallel computation is finished, otherwise the execution continues with the next instruction.

- `put_nth_head G3 X2 0 X0` performs the assignment X0 := G3[X2+0]. G3 points to a vector-list and X2+0 is the offset into the vector-list.
- `put_nth_head G4 X2 0 X1` similarly assigns X1 := G4[X2+0].

Example 2:

```
        nrev([],[]).
        nrev([X|Xs], Y) :- nrev(Xs,Z), append(Z,[X],Y).

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

The program is compiled into the following extended WAM code.

```
nrev/2: switch_on_term fail La L1 fail  % fail cases never occur

   La:  get_nil X0
        get_nil X1
        proceed

   L1:  allocate
        build_rec_poslist X0 X3 X6 X0   % first list
        build_variables X1 X3 X5 X1     % second list
        /* code for saving X3, X5 and X6 in environment */
        call nrev/2, 0                  % call base case
        /* code for restoring X3, X5 and X6 from environment */
        start_right_body X3 L1          % execute L1 in parallel
        deallocate
        proceed                         % done
   L1:  initialize_right 1 G3           % I := initial recursion level
   L2:  spawn_right 1 X7                % while(I-- > 0) do X7 := I;
        allocate
        put_nth_head G5 X7 1 X0         %  X0 := G5[I+1]
        put_list X1                     %  X1 := [
        unify_nth_head G6 X7 0          %         G6[I]
        unify_nil                       %              ]
        put_nth_head G5 X7 0 X2         %  X2 := G5[I]
        call append/3, 0                %  append(G5[I+1],[G6[I]],G6[I])
        deallocate
        jump L2                         % od

append/3:
        await_nonvar X0                 % wait until first arg nonvar
        switch_on_term fail La L1 fail  % fail cases never occur

   La:  get_nil X0
        get_value X1 X2
        proceed

   L1:  get_list X0                     % X0 = [
        unify_variable X3               %       X|
        unify_variable X0               %         Xs]
        lock_and_get_list X2 X4         % lock X2; X4 = [
        unify_x_value X3                %                X|
        unify_x_variable X5             %                  Zs]
```

```
    unlock X2 X4                    % unlock X2; X2 = [X|Zs]
    put_value X5 X2                 % X2 := Zs
    execute append/3                % append(X0,X1,X2)
```

We describe below the effects of the new instructions in the above code.

- `build_variables X1 X3 X5 X1` builds a vector-list containing X3+1 distinct unbound variables, storing a pointer to it in X5. A reference to the last variable in the vector-list is stored in X1.
- `start_right_body X3 L1` initiates parallel execution of the append/3 calls in the body of nrev/2. The code at label L1 is run in parallel by all workers. The sequential execution continues with the following instruction (deallocate) when the parallel phase is finished. The length of the recursion list is given in X3.
- `initialize_right 1 G3` initializes a worker for parallel execution. The step 1, given in the first argument, the length of the recursion list, given in G3, and the worker number is used for calculating the initial recursion level in static scheduling. In dynamic scheduling mode this instruction is ignored.
- `spawn_right 1 X7` calculates the next recursion level for this worker, using the step given in the first argument, and stores it in X7. If all recursion levels have been executed, the worker suspends and awaits the next parallel phase.
- `unify_nth_head G6 X7 0` writes the element G6[X7+0] onto the heap. G6 contains a pointer to a vector-list and X7+0 is the offset into the vector-list. This instruction never occurs in read mode.
- `await_nonvar X0` suspends until X0 contains a nonvariable or the recursion level has become leftmost in the resolvent.
- `lock_and_get_list X2 X4` checks the value in X2. If X2 contains a variable, it creates a list on the heap, stores a pointer to it in X4, and enters write mode. If X2 contains a list, the S register is set. Otherwise failure occurs.
- `unlock X2 X4` is ignored in read mode. In write mode, X2 and X4 are unified.

Other Instructions. The extended instruction set contains some instructions not used in the two examples above. These instruction are described below.

We use the following notation. If x is a cons cell of a vector-list, then $x.tl$ denotes the tail of the cons cell.

- `build_poslist_value Xa Xn Xv Xt` This instruction is to `build_poslist` as `unify_value` is to `unify_variable`.
- `build_neglist Xa Xn Xv Xt` A 'reverse list' vector-list of length Xn is created and stored in Xv. Xt is set to the head and Xa to the last tail of the vector-list, respectively. The last tail of the vector-list is in this case also the tail of the first element of the vector-list. See figure 1.
- `build_neglist_value Xa Xn Xv Xw Xt` is to `build_neglist` as `unify_value` is to `unify_variable`.
- `put_global_arg Gn Xi` stores the value of the sequential worker's register Xn in Xi.

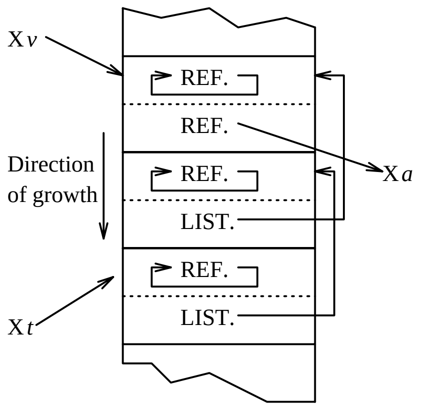

Figure 1: List (viewed as a vector) created by `build_neglist` .

- `put_nth_tail Gv Xl 0 Xi` Similar to `put_nth_ head` but performs the assignment Xi := Gv[Xl+0].tl.
- `unify_nth_tail Gv Xl 0` Similar to `unify_nth_head` but writes Gv[Xl+0].tl to the heap.
- `unify_global_arg Gg` writes the value of the sequential worker's register Xg on the heap. This instruction never occurs in read mode.
- `await_leftmost` forces the current recursion level to suspend until it is leftmost in the resolvent, i.e., until all preceeding recursion levels have terminated.
- `await_strictly_nonvar Xi` suspends the current recursion level until Xi contains a non-variable. If the recursion level becomes leftmost in the resolvent and Xi still contains an unbound variable, then a run-time error is signaled and execution fails.
- `await_variable Xi` suspends the current recursion level until it is leftmost in the resolvent. If the variable Xi becomes bound during suspension then a run-time error is signaled and the execution fails.
- `lock_and_get_structure F Xi Xn` Similar to `lock_and_get_list` but for structures with functor F.

8 EXPERIMENTAL RESULTS

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

8.1 Experimental methodology

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).

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.

8.2 Benchmarks.

Programs and input data. We have studied the performance of four benchmark programs. One of the programs exploits independent AND-parallelism and the others dependent AND-parallelism. Two programs are considerably larger than the others.

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

Nrev. This is the classic 'naive reverse' program run on a list of 900 elements.

Match. A dynamic programming algorithm for comparing, e.g., DNA-sequences. A sequence of length 32 was compared with 24 other sequences and the resulting similarity-values collected in a 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.

Tsp. This 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. Parallelization occurs on a single level in Reform Prolog, and there are necessarily sequential startup portions of the programs. The startup portions of our benchmark programs include setting up the arguments for the parallel call, large head unifications, and spawning parallel processes.

According to Amdahl's law, the time spent in the sequential part of the program will ultimately limit the 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%

The unparallelized parts of the programs are negligible. As a consequence, we do not currently see a need for parallelizing head unifications of parallel predicates.

8.3 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.)

8.4 Discussion

We briefly compare our system with Andorra-I, another compiler-based implementation supporting deterministic dependent AND-parallelism [13]. Note, however, that Andorra-I parallelizes a wider class of computations than does Reform Prolog. In particular, Andorra-I also supports OR-parallelism. The results reported for Andorra-I were obtained on a 10 processor Sequent Symmetry.

As before, take the parallel efficiency of a program to be the speedup on N processors divided by N. We have computed the efficiency relative to parallel Andorra-I, sequential Andorra-I and SICStus 2.1.

- Relative speedups on a set of 12 benchmarks range from 3.32 to 9.66, with a median of approximately 6.5 and a mean of approximately 6.4. The relative efficiency is taken to be 64%.
- Normalized speedups w.r.t. sequential Andorra-I on the same 12 benchmarks range from 2.04 to 7.26, with a median of 4.1 and a mean of 4.6. Normalized efficiency is then 46%.

- Compared with SICStus 2.1, Andorra-I was apparently 344 times faster in running the constraint solving fly_pan benchmark. Apart from this datapoint, on a range of 9 benchmarks, Andorra-I exhibited normalized speedups of 0.76 to 3.80 . The median was 1.47 and the mean 1.98. Normalized efficiency is thus 20%.

Only one benchmark can be directly comparad to Reform Prolog, the nrv400 benchmark of naive reverse on 400 elements. The reported relative speedup is 8.25, the normalized speedup w.r.t. sequential Andorra-I is 6.55 and the normalized speedup w.r.t. SICStus 2.1 is 3.80. The efficiency is then 82%, 65% and 38%, respectively, on 10 processors. Reform Prolog has a relative efficiency of 95% and a normalized efficiency of 83% on 24 processors when running naive reverse on 900 elements.

We conclude that while Andorra-I can parallelize quite a wider range of programs, exploiting recursion-parallelism where available seems to have considerable benefits.

9 CONCLUSIONS

Reform Prolog implements parallelism across recursion levels by Reform compilation. One restriction is introduced on the recursive predicates subject for parallelization: bindings of variables shared between recursive calls of the predicate must be unconditional. This is not a severe restriction in practice.

The execution model has two major advantages:

First, a static process structure can be employed. That is, all parallel processes are created when the parallel computation is initiated. In most other systems for parallel logic programming, processes can be dynamically created, rescheduled and destroyed during the parallel computation.

A consequence of the static process structure is that process managment and scheduling can be implemented very efficiently. This opens up for high parallel efficiency (91–99% on the programs tested). Another consequence is that it is easy for the programmer to see which parts of the program are going to execute in parallel. This facilitates the task of writing efficient parallel programs.

Second, it is possible by global dataflow analysis to optimize the code executed by each parallel worker very close to ordinary sequential WAM code. This results in very low overheads for parallelization (2–12 % on the programs tested).

The apparent drawback of this approach is that not all available parallelism in programs are exploited. This is, however, a deliberate design decision: exploiting as much parallelism as possible is likely to lead to poor machine utilization on conventional multiprocessors.

Acknowledgments

We thank the Swedish Institute of Computer Science (SICS) for making their 26 processor Sequent Symmetry available to us, and the Andorra-I team of Bristol University for providing us with their latest benchmarks results.

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, Exploiting recursion-parallelism in Prolog, *Proc. PARLE'93*, Lecture Notes in Computer Science, Springer-Verlag, 1993.

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

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

[6] C.P. Kruskal & A. Weiss, Allocating Independent Subtasks on Parallel Processors, *IEEE Trans. Software Engineering*, Vol. 11, No. 10, 1985.

[7] E.P. Markatos & T.J. LeBlanc, Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors, Technical Report 410, University of Rochester, March 1992.

[8] J.M. Mellor-Crummey & M.L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems*, Vol 9, Febr., 1991.

[9] H. Millroth, Reforming Compilation of Logic Programs, *Proc. Int. Symp. Logic Programming*, MIT Press, 1991.

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

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

[12] C.D. Polychronopoulos & D.J. Kuck, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, *IEEE Trans. Computers*, Dec. 1987.

[13] R. Yang, T. Beaumont, I. Dutra, V.S. Costa, D.H.D. Warren, Performance of the Compiler-Based Andorra-I System, *Proc. 10th Int. Conf. Logic Programming*, MIT Press, 1993.

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

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

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