Using the Reform Inference System for Parallel Prolog

Håkan Millroth

Computing Science Dept., Uppsala University
Box 520, S-751 20 Uppsala, Sweden
Electronic mail: hakanm@csd.uu.se

Abstract. We show how a new method for parallel logic programming, based on compilation of Tärnlund's inference system Reform, can be applied to the logic programming language Prolog. We retain the sequential left-to-right depth-first backtracking scheme with one exception: the recursion levels of a recursive program, including the head unifications at each level, are computed in parallel. We discuss criteria for when a program is amenable to this kind of parallel processing and describe parallel Reform Prolog solutions of some programming problems.

1. INTRODUCTION

Previous attempts at developing parallel Prolog systems have focused on exploiting AND-parallelism, or OR-parallelism, or both. In this work we parallelize Prolog by exploiting parallelism in its fundamental control structure: recursion.

Our basic idea is that parallelization takes place across recursion levels: the recursion levels of a program, including the head unifications at each level, are computed in parallel. The sequential left-to-right depth-first backtracking scheme of Prolog is retained within each recursion level.

We compile structural recursion to bounded iteration (Millroth, 1990). The iterative programs are parallelized by standard methods developed for imperative programs. The technique is based on an analysis of the unification patterns of recursive programs.

This approach has a number of appealing consequences:

1. It gives the parallel program a natural and easy-to-understand parallel reading. The programmer can write efficient parallel programs by obeying some simple rules of programming.
2. It gives a natural partitioning of the computation and its data, since nondeterminism and producer-consumer relationships are often local to each recursion level. Nondeterminism and data dependencies *within* each recursion level does not impede parallelization, since we run the each level sequentially.
3. There is a simple mapping of the program onto a parallel machine whose processors are organized in, e.g., a ring: adjacent recursion levels are mapped to adjacent processors. The inter-processor communication on such a machine will mostly be between neighboring processors, since it is unusual that data is passed between nonadjacent recursion levels in a logic program. We thus achieve predominantly local communication, which is crucial on a distributed memory machine.
4. The workload will automatically be spread evenly among the parallel processors, assuming that each recursion level of the program requires approximately

the same amount of work. This assumption seems reasonable for most Prolog programs.

2. COMPILED REFORM COMPUTATIONS

In this section we briefly describe our method for compiling a program, using structural recursion, to a program that uses bounded iteration. The compilation technique makes use of Tärnlund's (1991) inference system Reform.

2.1. Reform

We shall define Reform for the special case of linear recursion. Let $R = (H \leftarrow \Psi \wedge T \wedge \Phi)$ be a clause. We enumerate its variants as:

$$R_i = (H_i \leftarrow \Psi_i \wedge T_i \wedge \Phi_i), \qquad i \geq 1.$$

The nth *reforment* ($n > 1$) is defined as:

$$R^n = (H_1 \leftarrow \Psi_1 \wedge \cdots \wedge \Psi_n \wedge T_n \wedge \Phi_n \wedge \cdots \wedge \Phi_1)(\sigma_1 * \cdots * \sigma_{n-1}),$$

where

$$\sigma_i = \mathrm{mgu}(H_{i+1}, T_i) \qquad \text{and} \qquad \sigma_i * \sigma_j = \mathrm{mgu}(\sigma_i, \sigma_j).$$

Let G_0 be an (initial) goal. Resolving R^n with G_0 gives

$$G_n = (\leftarrow \Psi_1 \wedge \cdots \wedge \Psi_n \wedge T_n \wedge \Phi_n \wedge \cdots \wedge \Phi_1)\theta,$$

where

$$\theta = \mathrm{mgu}(G_0, H_1(\sigma_1 * \cdots * \sigma_{n-1})).$$

It can be shown that G_n can be derived from the clause R and the initial goal G_0 in $\lceil \log n \rceil + 1$ steps. Note that G_n is the resolvent derived in n SLD-resolution steps from R and G_0. The derivation is thus significantly shorter with Reform.

Example. Consider the program for scaling each element of a list by a factor Y:

```
scale([], Y, []).
scale([X|Xs], Y, [Z|Zs]) :-
   Z is X*Y,
   scale(Xs, Y, Zs).
```

Assume that we have the following goal, where $n = 4$.

```
   scale([1,2,3,4], 10, W)
```

We may derive the $4th$ reforment of the program in two steps as follows. First, we unfold the recursive call, obtaining:

```
scale([X1,X2|Xs], Y, [Z1,Z2|Zs]) :-
   Z1 is X1*Y,
   Z2 is X2*Y,
   scale(Xs, Y, Zs).
```

Unfolding the recursive call of this clause (using the clause itself) now yeilds the $4th$ reforment:

```
scale([X1,X2,X3,X4|Xs], Y, [Z1,Z2,Z3,Z4|Zs]) :-
   Z1 is X1*Y,
   Z2 is X2*Y,
   Z3 is X3*Y,
   Z4 is X4*Y,
   scale(Xs, Y, Zs).
```

We can now resolve the goal with this clause in a single, direct step. This step opens up for parallelism: the first two list can be matched in parallel, the third list can be constructed in parallel, and the multiplications can be performed in parallel.

2.2. Compiling Reform

Carrying out the Reform transformation at run-time might impose considerable overhead. The question is thus: How do we obtain the parallel algorithm, represented by the nth reforment, at *compile-time*?

Let $\mu_n = \sigma_1 * \cdots * \sigma_{n-1}$, where σ_i $(1 \leq i \leq n)$ are defined in Section 2.1. Consider a variable x and its variant variables $x_1, \ldots, x_n$ (these correspond to the new variables created in each recursive invocation of the program in SLD-resolution). The *Reform series* of x is then the sequence

$$x_1\mu_n, \ldots, x_n\mu_n .$$

We have shown that the Reform series of a variable can be inferred from information available in the first unifier σ_1. Closed-form expressions for the Reform series can be inferred for different classes of variables. Hence explicit Reform transformation is not needed for determining the variable bindings obtained by Reform (Millroth, 1990; 1991a).

Let us see how this can be useful in compilation of linear recursion. Consider a recursive clause $H \leftarrow \Psi \wedge T \wedge \Phi$ and a goal G. Let $H(i)$, $\Psi(i)$, $T(i)$, and $\Phi(i)$ denote H, Ψ, T, and Φ, respectively, with all variables replaced by the ith elements of the corresponding Reform series. The compiled program can then schematically be described, in a parametric form, as:

```
unify H(1) with G
for i=1 to n do call Ψ(i)
call T(n)
for i=n to 1 do call Φ(i)
```

(In some cases, the recursive call $T(n)$ can be unfolded against the base clause. In the general case, however, this call can match both the base clause and the recursive clause.)

Example (continued). The code for the recursive `scale` clause is:

```
n := length(arg1);
xs[1] := arg1;
```

```
y[1] := arg2;
for i = 1 to n+1
   zs[i] := new_variable();
endfor
bind arg3 to zs[1];
for i = 1 to n
   xs[i+1] := tail(xs[i]);
   y[i+1] := arg2;
endfor
for i = 1 to n
   x[i] := head(xs[i]);
   z[i] := x[i]*y[i];
   zs[i] := cons(z[i],zs[i+1]);
endfor
call scale(xs[n+1],y[n+1],zs[n+1]);
```

The first and third loop are parallel whereas the second loop is sequential. (In the actual implementation the second loop distributes data to the third.)

2.3. Scope of the method

Our compilation method is applicable to *structural* recursion: The 'step' function (how the recursion argument is decomposed or composed) of the recursion must be derivable from the program text and not be dependent on input data. Moreover, this step function must be the same in all recursive clauses of the predicate.

For other data structures than lists and integers (of which we assume the compiler to be aware) the compiler must have type information available.

Compilation of nonlinear structural recursion follows the same principles as in the linear case. This is possible since a temporary linear representation of the recursion tree is obtained in the single large head unification that replaces the smaller unifications of a traditional system (Millroth, 1990).

We shall limit the discussion in this paper to linear recursion.

3. REFORM PROGRAMMING IN PROLOG

We now apply Reform to the logic programming language Prolog. The idea is to retain the usual left-to-right depth-first backtracking scheme with one exception: the recursion levels of a recursive program, including the unifications at each level, are computed in parallel.

An attractive consequence of this idea is that the nondeterminism of the language does not impede efficient parallelization in most cases, assuming that nondeterminism most often occurs locally *within* recursion levels. This assumption seems reasonable for most Prolog programs.

3.1. Criteria for running in parallel

Let us refer to variables shared between recursion levels as *global* variables. (Which variables are shared between recursion levels is easily seen by doing one or a few Reform transformation steps: they are the ones that get bound in the transformation.)

We shall now give two conditions for running the different recursion levels efficiently in parallel.

1. *Each recursion level is deterministic with respect to the bindings made to the global variables.*

We say that the recursion levels are *binding deterministic* (Naish, 1988) with respect to the global variables, if this condition is fulfilled. Note that the concept of binding determinism is different from the concept of data dependency in programming languages lacking logical variables. As an example, in the context of traditional AND-parallelism, consider appending three lists x, y and z to form a new list w. Given the logic program

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

we can do this with the query

```
?- append(Y, Z, YZ), append(X, YZ, W).
```

The two calls in this query can be computed in parallel, since unification allows us to use the result of a computation (the variable YZ in our example) before it is actually computed.

In order to achieve binding determinism with respect to global variables, it must be assured that such variables are bound only after it has been determined which clause of the recursive program to use. Hence one must be careful not to bind any global variables before tests or before *cut* in a clause. Consequently, one must sometimes defer output unification to the clause body. We observe that it would be convenient to do this program transformation automatically. The declarations used in Parallel NU-Prolog (Naish, 1988) seems to be adequate for this purpose.

What happens if we try to run the recursion levels in parallel when the condition of binding determinism is violated? The computation will toggle back and forth between recursion levels by means of backtracking until appropriate bindings of the global variables are found. This is clearly a very inefficient method of control.

2. *There is no cross-level dereferencing.*

(Dereferencing is the process of following variable-to-variable bindings until a non-variable term, or an unbound variable, is found. In effect, it amounts to retreiving the value of the variable.) Cross-level dereferencing amounts to dereferencing a global variable bound to a variable or term on another recursion level; we will see in Section 4 how the programmer easily can detect this situation.

Consider, as an example, the following program which computes the sum of the elements of a list using an accumulating parameter (initialized to zero).

```
sum_list([], Y, Y).
sum_list([X|Xs], Y, Z) :-
   W is X+Y,
   sum_list(Xs, W, Z).
```

Here Y is a global variable, and binding W requires dereferencing Y. Hence the recursion levels must be computed in sequence, although they are deterministic. Let us now contrast this program with a slightly contrived variant of the usual list reversal program using an accumulating parameter:

```
rev([], Y, Y).
rev([X|Xs], Y, Z) :-
   W = [X|Y],
   rev(Xs, W, Z).
```

This program is very similar in structure to the `sum_list` program. The difference is that instead of binding W to the evaluated value of X+Y, we bind W to the list [X | Y]. We may put it like this: `sum_list` *uses* Y whereas `rev` *mentions* it (the distinction between use and mention is a nice feature of logical variables). Mentioning Y does not enforce dereferencing and all recursion levels of `rev` can run in parallel.

Assume now that a recursion level that performs a cross-level dereferencing suspends execution until the dereferenced value is obtained. Then, if we try to run recursion levels in parallel and the condition of no cross-level dereferencing is violated, the effect is that the recursions level are *synchronized*. Of course, this does not mean that such programs always lack potential for parallel speed-up. Although the recursion levels are synchronized, they may overlap in time to smaller or larger extent.

Let us summarize. If condition 1 (binding determinism) is violated, then the parallel computation may be very ineffcient. If condition 2 (no cross-level dereferencing) is violated, then the computation is more or less sequentialized due to synchronization.

4. EXECUTION MODEL

In this section we describe an execution model for parallel Prolog, suitable for a large-scale distributed memory multiprocessor. The idea is to increase performance of a restricted, yet interesting, class of predicates with parallelism, running other predicates sequentially.

The sequential and parallel parts of a Prolog program interact as follows. The sequential predicates of the program run on a *root processor*. When a parallel predicate is called, control is transferred to a network of parallel *node processors*. Upon termination, the parallel predicate returns control to the root which continues sequential execution or invokes another parallel predicate.

A parallel computation is thus initiated by calling a parallel predicate on the root processor. The parallel computation can then conceptually be divided into three phases (in reality, they may overlap in time):

1. Supplying the input data to the node processors.
2. Computing one instance of the body of the recursive clause on each node processor.
3. Returning control to the root processor.

The parallel abstract machine used in the Reform project at Uppsala University (Tärnlund *et al.*, 1991) has an inter-processor topology incorporating both a ring and

a binary tree. The ring is used for communication between adjacent recursion levels of a parallel predicate whereas the tree is used for transmitting data between the sequential and the parallel parts of the computation.

Example. Consider the following program for partitioning a list X of numbers with respect to a particular pivot number P. Those elements of X that are less than or equal to P are collected in a list Y, and those elements that are greater than P in a list Z.

```
split([], _, [], []).
split([U|X], P, Y, Z) :-
   (   U =< P ->
       Y = [U|S], Z = T
   ;   Y = S,   Z = [U|T]
   ),
   split(X, P, S, T).
```

A few steps of Reform transformation yield the following clause:

```
split([U1,U2,U3|X], P, Y1, Z1) :-
   (   U1 =< P ->
       Y1 = [U1|Y2], Z1 = Z2
   ;   Y1 = Y2,   Z1 = [U1|Z2]
   ),
   (   U2 =< P ->
       Y2 = [U2|Y3], Z2 = Z3
   ;   Y2 = Y3,   Z2 = [U2|Z3]
   ),
   (   U3 =< P ->
       Y3 = [U3|Y3], Z3 = T
   ;   Y3 = S,   Z3 = [U3|T]
   ),
   split(X, P, S, T).
```

By studying the transformed clause we can get an accurate view of what will happen when we run the program in parallel. Although explicit Reform transformation is not used in the system, this can be a convenient method for the programmer to understand the parallel program. The clause obtained after one or a few Reform transformation steps can tell us if bindings to global variables are deterministic, if there is any cross-level dereferencing, if backtracking across recursion levels may occur, etc.

So what happens more exactly at the node processors when we we run the `split` program? Once the ith node processor has received its input data (one element Ui of the input list and the pivot number P), it can start computing the ith instance of `split`'s body. Assuming we have an input list of length n and n node processors, the processors run the following code (each processor employs the sequential computation rule of standard Prolog):

```
  Node 1:              Node 2:                 Node n:
(  U1 =< P ->       (  U2 =< P ->          (  Un =< P ->
   Y1 = [U1|Y2],       Y2 = [U2|Y3],          Yn = [Un|S],
   Z1 = Z2             Z2 = Z3        ...     Zn = T
;  Y1 = Y2          ;  Y2 = Y3             ;  Yn = S
   Z1 = [U1|Z2]        Z2 = [U2|Z3]           Zn = [Un|T]
)                   )                      )
```

If we bind Y1 ← [U1 | Y2] on processor 1, we can simultaneously bind Y2 ← [U2 | Y3] on processor 2, since the action taken on processor 1 does not require Y2 to be dereferenced. The situation is analogous on the other processors. Hence the second condition of Section 3.1 is fulfilled: no cross-level dereferencing takes place.

The first condition is also fulfilled, since the bindings are deterministic. Hence we can run the recursion levels efficiently in parallel.

5. PROGRAM EXAMPLES

In this section we describe parallel Reform Prolog solutions of some programming problems.

In the first example all work is done by head unification. Such programs can exploit parallel unification (Barklund, 1990) with our method, but are hard to parallelize efficiently in other parallel models.

Since mutual recursion cannot be handled directly with our compilation technique it must be, automatically or manually, transformed to straight recursion. Our second example illustrates this.

In the third example we discuss a program which allows parallelism in head unification and execution of the left-body, but has a sequential right-body.

5.1. Parentheses matching

We shall first consider an example proposed by Shapiro (1983). We are given a list of left and right parentheses of two kinds. The task is to check that the parentheses are balanced.

```
balanced(L) :-
   balanced(L, Ss),
   stack(Ss, []).

balanced([], []).
balanced(['('|Ins], [push('(')|Outs]) :-
   balanced(Ins, Outs).
balanced(['{'|Ins], [push('{')|Outs]) :-
   balanced(Ins, Outs).
balanced([')'|Ins], [pop('(')|Outs]) :-
   balanced(Ins,Outs).
```

```
balanced(['}'|Ins], [pop('{')|Outs]) :-
   balanced(Ins, Outs).

stack([], []).
stack([push(X)|Ss], S) :-
   stack(Ss, [X|S]).
stack([pop(X)|Ss], [X|S]) :-
   stack(Ss, S).
```

Let us consider running this program with Reform parallelism. In `balanced/1`, we run the calls to `balanced/2` and `stack/2` in sequence. However, both `balanced/2` and `stack/2` can exploit Reform parallelism. That is, all recursion levels of each of these predicates are computed simultaneously. Since neither predicate has any goals in the residual body, this amounts to two large parallel head unifications.

It is interesting to compare Reform with other paradigms of parallel logic programming with respect to this program:

1. The program cannot benefit from traditional AND-parallelism: running `balanced/2` in parallel with `stack/2` is not possible unless some notion of communicating processes is adapted. Neither `balanced/2` nor `stack/2` are, by themselves, amenable for traditional AND-parallelism since each clause contains at most one literal.
2. In a concurrent logic programming language, like Parlog (Clark & Gregory, 1983) or GHC (Ueda, 1986), the `balanced/2` and `stack/2` calls can run as concurrent processes. The variable Ss acts as a communication channel on which `balanced/2` writes messages to be consumed by `stack/2`. However, it is unlikely that this program will be accelerated by concurrent execution in this manner when run on parallel machine. The reason is simple, yet quite typical: the concurrent processes are too small to make the overhead associated with spawning parallel processes worthwhile.

When considering this example, the reader may wonder how programs which have more than one recursive clause are handled by the compiler. The answer is that in a preprocessing step such programs are transformed into equivalent programs having only one recursive clause (this is called *clause fusing*). For example, the `stack` program is transformed to:

```
stack([], []).
stack([Y|Ss], U) :-
   s(Y, U, W),
   stack(Ss, W).
s(push(X), S, [X|S]).
s(pop(X), [X|S], S).
```

5.2. Dutch national flag

We shall now consider the following program, adapted from O'Keefe (1990), for solving Dijkstra's Dutch national flag problem. The problem reads: "Given a list of elements coloured red, white, and blue, reorder the list so that all the red elements appear first,

then all the white elements, followed by the blue elements. This reordering should preserve the original relative order of elements of the same colour." For clarity, we modify O'Keefe's program to represent each d-list explicitly rather than as a pair of lists.

```
dutch_national_flag(Input, Output) :-
   dnf(Input, Output-X, X-Y, Y-[]).

dnf([], R-R, W-W, B-B).
dnf([Item|Items], R0-R, W0-W, B0-B) :-
   colour(Item, Colour),
   dnf(Colour, R0-R, W0-W, B0-B, Item, Items).

dnf(red, [Item|R1]-R, W0-W, B0-B, Item, Items) :-
   dnf(Items, R1-R, W0-W, B0-B).
dnf(white, R0-R, [Item|W1]-W, B0-B,Item, Items) :-
   dnf(Items, R0-R, W1-W, B0-B).
dnf(blue, R0-R, W0-W, [Item|B1]-B, Item, Items) :-
   dnf(Items, R0-R, W0-W, B1-B).
```

We transform the program to use straight recursion. First we merge the three clauses of `dnf/6` to one clause by introducing an auxiliary predicate `dnf_branch/5`:

```
dnf(Colour, R0-R, W0-W, B0-B, Item, Items) :-
   dnf_branch(Colour, R0-R1, W0-W1, B0-B1, Item),
   dnf(Items, R1-R, W1-W, B1-B).

dnf_branch(red, [Item|R1]-R1, W0-W0, B0-B0, Item).
dnf_branch(white, R0-R0, [Item|W1]-W1, B0-B0, Item).
dnf_branch(blue, R0-R0, W0-W0, [Item|B1]-B1, Item).
```

Now we can unfold the call to `dnf/6` in `dnf/4`, thereby obtaining a program using straight recursion. The resulting procedure for `dnf/4` is:

```
dnf([], R-R, W-W, B-B).
dnf([Item|Items], R0-R, W0-W, B0-B) :-
   colour(Item, Colour),
   dnf_branch(Colour, R0-R1, W0-W1, B0-B1, Item),
   dnf(Items, R1-R, W1-W, B1-B).
```

This program is amenable to Reform parallel execution: Traversal of the input list, construction of the output lists, and all calls to `colour/2` and `dnf_branch/5` can be computed in parallel. That gives a very high degree of parallelism to this seemingly sequential program.

5.3. Linear regression

The presentation of our next example is adopted from Press *et al.* (1989). The problem is to fit a set of n data points (x_i, y_i) to a straight line $y = a + bx$. We assume that the uncertainty σ_i associated with each data y_i is known, and that the $x_i's$ (values of the dependent variable) are known exactly.

Let us first define the following sums.

$$S = \sum_{i=1}^{n} 1/\sigma_i^2 \qquad S_x = \sum_{i=1}^{n} x_i/\sigma_i^2 \qquad S_y = \sum_{i=1}^{n} y_i/\sigma_i^2$$
$$S_{xx} = \sum_{i=1}^{n} x_i^2/\sigma_i^2 \qquad S_{xy} = \sum_{i=1}^{n} x_i y_i/\sigma_i^2$$

The coefficients a and b of the straight-line equation can now be computed as:

$$\Delta = SS_{xx} - (S_x)^2$$
$$a = \frac{S_{xx}S_y - S_x S_{xy}}{\Delta}$$
$$b = \frac{SS_{xy} - S_x S_y}{\Delta}$$

The following is a Prolog program for computing the five sums needed in the calculation of a and b.

```
lin_regr_sums([], [], [], 0, 0, 0, 0, 0).
lin_regr_sums([X|Xs], [Y|Ys], [E|Es], S, Sx, Sy, Sxx, Sxy) :-
   E1 is 1/(E*E), X1 is X*E1, Y1 is Y*E1,
   XX is X*X1, XY is X*Y1,
   lin_regr_sums(Xs, Ys, Es, S1, Sx1, Sy1, Sxx1, Sxy1),
   S is S1+E1, Sx is Sx1+X1, Sy is Sy1+Y1,
   Sxx is Sxx1+XX, Sxy is Sxy1+XY.
```

We have broken up the arithmetic calculations in the body in two parts: The parallel operations are done in the left-body, and all calculations which are dependent on earlier recursion levels are done in the right-body.

Notice that we could, of course, have used accumulating parameters for the five sum-arguments, and thus obtained a tail-recursive version of the program. This would have been a good idea if we intended the program to run sequentially. However, we would then have missed most opportunities for parallelism.

6. RELATED WORK ON PARALLELIZATION OF RECURSION

The idea of running the goals of a conjunction concurrently was described by Kowalski (1974) in his seminal paper on predicate logic as a programming language. This kind of parallel processing was later coined AND-parallelism.

Let us consider running a recursive logic program in AND-parallel mode. Assume that, at each recursion level, the recursive call and the other body calls are made in parallel. Then the different recursion levels are initiated one after the other but their work may overlap in time. Clearly, this kind of parallel processing can only speed up the computation of recursive programs with a constant factor: it takes $O(n)$ time to spawn all n recursion levels.

In our model, the time complexity for getting all recursion levels into work is bounded only by the time it takes to distribute the input data of the program. This

may take logarithmic time on, for example, a machine with tree topology. On a shared-memory machine (or on a distributed-memory machine where the input data is already distributed) work on all recursion levels can be initiated simultaneously.

Now, let us look outside the logic programming world. A parallelization technique for recursion in Lisp, which gives essentially the same degree of parallelism as with AND-parallelism, is described by Larus (1991).

PARCEL (Harrison, 1989) is a compiler, for the Lisp-dialect Scheme, that parallelizes recursion for execution on shared-memory multiprocessors. This work addresses the same problem as we do: compiling recursion to parallel iteration. Let us point out some notable differences in the solutions.

In the first place, PARCEL does not parallelize programs that use other data structures than lists (a nonstandard, vector-like, representation of lists is employed). Our compilation method is neither restricted to any particular data structure, nor to linear recursion. The only restriction is that the 'step' function of the recursion must be independent of input data.

Secondly, PARCEL does not parallelize programs that destructively modify list structures. The Prolog counterparts of such programs use difference lists which allow constant-time concatenation. The use of difference lists does not impede parallelization with our method. On the contrary, difference lists offer a very efficient way of constructing lists in parallel, as is shown in Section 5.2.

Thirdly, PARCEL depends on solving (at run-time) recurrences which gives expressions corresponding to our Reform series. Our classification of variables and derived expressions for Reform series allow us to determine the bindings of the variables involved in the recursion at compile-time (modulo the particular input data of the call).

7. CONCLUSIONS

The important issues in parallel computation on large-scale distributed memory machines are programming simplicity, locality of reference, and workload balance. We have described a method which successfully addresses these questions for a class of recursive Prolog programs.

Locality of reference is assured by the natural mapping of parallel programs to a ring of parallel processors. Adjacent recursion levels are mapped to adjacent processors on such a machine. Thus, inter-processor communication is local for the majority of Prolog programs which do not pass data between nonadjacent recursion levels.

Even workload among processors is assured (without dynamic load balancing) if the recursion levels of the parallel program contain approximately the same amount of work. This is the case for many Prolog programs.

Reform offers a conceptually simple model of parallelism for the Prolog programmer: the operational behaviour of a parallel program is no more complex than the corresponding sequential program. Here it is interesting to draw an analogy between our parallelization of recursion and parallelization (or vectorization) of iteration in Fortran. In both cases one starts from a sequential program and parallelizes it by exploiting parallelism in the language construct for repetition. In the Fortran case,

this is often a very complicated procedure which requires expertise (Fox, 1990). In our case, a few simple rules of programming is sufficient for writing efficient parallel programs.

ACKNOWLEDGEMENT

Jonas Barklund, Johan Bevemyr, Thomas Lindgren and Margus Veanes gave valuable comments on earlier drafts of this paper.

REFERENCES

BARKLUND, J. (1990) *Parallel Unification*, Ph.D. Thesis, Computing Science Dept., Uppsala University.

CLARK, K. L. & S. GREGORY (1983) PARLOG: a parallel logic programming language. Research report DOC 83/5, Dept. of Computing, Imperial College, London.

FOX, G. (1990) Talk given at a workshop on Massively Parallel Reasoning Systems, Syracuse, New York, December 1990.

HARRISON III, W. L. (1989) The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation* 2, No. 3/4, 179–396.

KOWALSKI, R. A. (1974) Predicate logic as a computer language. In *Information Processing 74*, pp. 569–574. North-Holland, Amsterdam.

LARUS, J. R. (1991) Compiling Lisp programs for parallel execution. *Lisp and Symbolic Computation* 4, No. 1, 29–99.

MILLROTH, H. (1990) *Reforming Compilation of Logic Programs*, Ph.D. Thesis, Computing Science Dept., Uppsala University. (Summary to appear at Int. Logic Programming Symp., San Diego, CA., October 1991)

MILLROTH, H. (1991) Compiling Reform, (to appear in) *Massively Parallel Reasoning Systems* (eds. J. A. Robinson & E. E. Siebert), MIT Press.

MYCROFT, A. & R. A. O'KEEFE (1984) A Polymorphic Type System for Prolog. *Artificial Intelligence* 23, No. 3, 295–307.

NAISH, L. (1988) Parallelizing NU-Prolog. *Proc. 5th Int. Conf./Symp. Logic Programming* (eds. K. A. Bowen & R. A. Kowalski), Seattle, Washington.

O'KEEFE, R. A. (1990) *The Craft of Prolog.* MIT Press, Cambridge, Mass.

PRESS, W. H. *et al.* (1989) *Numerical Recepies. The Art of Scientific Computing.* Cambridge U. P., Cambridge.

SHAPIRO, E. Y. (1983) *A Subset of Concurrent Prolog and its Interpreter.* Technical report TR-003, ICOT, Tokyo.

TÄRNLUND, S.-Å. (1991) Reform, (to appear in) *Massively Parallel Reasoning Systems* (eds. J. A. Robinson & E. E. Siebert), MIT Press.

TÄRNLUND, S.-Å, H. MILLROTH, J. BEVEMYR, T. LINDGREN & M. VEANES (1991) Perform: a Parallel Reform Machine, submitted for publication.

UEDA, K. (1986) *Guarded Horn Clauses*, Eng.D. Thesis, University of Tokyo.