Parallel Bounded Quantification—Preliminary Results

Henrik Arro, Jonas Barklund and Johan Bevemyr
UPMAIL, Computing Science Dept.
Uppsala University,
E-mail: arro@csd.uu.se, jonas@csd.uu.se, bevemyr@csd.uu.se
Box 311, S-751 05 Uppsala, Sweden
Phone: +46–18–18 25 00
Fax: +46–18–51 19 25

To be presented at the
Joint Workshop on Distributed and Parallel
Implementations of Logic Programming Systems
Washington, D.C., November 13-14, 1992

Abstract
We have extended D. H. D. Warren's abstract machine for sequential
Prolog with parallel instructions that implement bounded quantifi-
cations, an extension to Prolog proposed by Barklund and Bevemyr.
These instructions are intended for parallel computers supporting the
data parallel programming model. Luther, an emulator for the sequen-
tial abstract machine implemented in the C programming language,
has been extended with these instructions, implemented in the data
parallel C* programming language.

We have coded three example programs that use bounded quan-
tifications, and run them in the emulator on a Connection Machine
model 200, a SIMD parallel computer. We compare the run times on
this parallel computer with the run times obtained with a sequential
implementation of bounded quantifications on a SUN-4M sequential
computer.

The best result is that one parallel bounded quantification pro-
gram runs 30 times faster on the Connection Machine than an ordi-
inary recursive program implementing the same algorithm running
on the sequential computer. It is conceivable that one can obtain far
higher speed-ups when running on larger data parallel computers using
bounded quantifications.
1 INTRODUCTION

Recently, Barklund & Millroth (1992) have proposed bounded quantifications as an extension to sequential and parallel logic programming languages. Barklund & Bevemyr (1992) have defined a particular extension of Prolog with bounded quantifications and also defined extensions to D. H. D. Warren's sequential abstract Prolog machine (1983), a.k.a. WAM, that implement this extension. The resulting language is a strict superset of Prolog where some computations which otherwise would be written as recursive programs can instead be written as bounded quantifications. The program

\[
\text{factorial}(N, F) :- \\
F = (\text{#product } I : 1 \leq I \leq N : I).
\]

is a simple example of a program that uses a bounded quantification. The corresponding recursive programs, e.g.,

\[
\text{factorial}(1, 1).
\]

\[
\text{factorial}(N, F) :- N > 1, \\
N1 is N-1, \\
\text{factorial}(N1, F1), \\
F is N*F1.
\]

and

\[
\text{factorial}(N, F) :- \text{factorial1}(N, 1, F).
\]

\[
\text{factorial1}(1, F, F).
\]

\[
\text{factorial1}(N, A, F) :- N > 1, \\
N1 is N-1, \\
A1 is A*N, \\
\text{factorial1}(N1, A1, F).
\]

are in our opinion less elegant and also run less efficiently in the abstract machine (but there are still some computations which are more naturally expressed using recursion).

In this work we have taken the same extension of Prolog with bounded quantifications but defined a parallel extension to Warren's machine. We will briefly present these extensions to the machine and present our first results when comparing the speed of computations on the parallel machine with those on the sequential machine. Finally we will outline our intended continuation of this work.

2 OUR EXTENSION TO PROLOG

Let us first state that the extensions to Prolog proposed herein are certainly not definitive, but they suffice to show that an extension along these lines
brings several interesting properties to the language. For a more extensive
description of this language extension, please see our earlier paper (Barklund

We extend the well formed expressions of Prolog with those on the form

\[
\#\text{quantifier vars} : \text{range} : \text{body}
\]

where \text{quantifier} is an expression recognized as a quantifier, \text{vars} an expres-
sion on the form \text{var}_1, \text{var}_2, \ldots, \text{var}_k, where every \text{var}_i is a variable and
\(k \geq 1\), \text{range} is an expression which is recognized as a range formula and \text{body}
is an expression (which is interpreted differently for different quantifiers).

The quantifiers include, of course, the universal and existential quantifiers,
but also numeric quantifiers, such as sums and products. Bounded quan-
tifications having a universal or existential quantifier may appear wherever
an arbitrary Prolog formula might. Bounded quantification with numeric quan-
tifiers may appear wherever terms may appear. (We allow ourselves
to assume that the equality theory for numerical bounded quantifications
and arithmetical expressions is not Herbrand equality but rather arithmetic
equality, as in Gödel [Hill & Lloyd, 1991].) We have also mentioned some
other quantifiers, for example, one which has a term-valued expression as
body and returns an array containing one term for each possible value of
the quantified variable.

The variables listed in a bounded quantification are local to the quantifi-
cation. In the parallel context below, that will usually mean that they are
local in each processor.

Range formulas are, informally, formulas that are "obviously" true for
only a finite number of values of some variable. For now we require that
they consist of one of several predefined \textit{iterators}, or a conjunction of such
an iterator with an arbitrary formula. These iterators are such that it is
possible to enumerate their solutions efficiently, e.g., the integers in a range,
the elements of an array, or the suffixes of a list.

For example,

\[
\#\text{and I : 1\leq I \leq K} : r(I , I+3)
\]
is a bounded universal quantification and the \textit{body} expression is thus inter-
preted as a goal to prove. Furthermore,

\[
\]

(which computes the sum of every combination of elements from the one-
dimensional arrays \text{X} and \text{Y}) is a bounded sum quantification and its \textit{body}
expression is interpreted as an arithmetical expression which is evaluated.
3 THE SEQUENTIAL WAM EXTENSION

We have added some new instructions to Warren's machine together with compilation schemas, in order to run bounded quantifications. We have presented these extensions and some of the schemas elsewhere (Barklund & Bevemyr, 1992). We will present a subset of them here, in order that they can be compared with the corresponding parallel extensions below.

There are three compilation schemas for universal bounded quantifications, and three corresponding schemas for the numeric `reduction' quantifications; the choice of an applicable schema depends on the body of the quantification. The schemas can be characterized as "deterministic body with no out-of-line calls", "deterministic body with out-of-line calls" and "potentially non-deterministic body". (For numeric quantifications it may be that only the two first cases are relevant.) Existential bounded quantifications and the other kinds of quantifications proposed before require further schemas and instructions, which are not presented here.

The compilation schema for the first case of a universal quantification

\[ \text{\#and } i : m \leq I < n : \Phi \]

(we assume that the registers $X_i$ and $X_j$ are free) is

- code for putting the value of $m$ in $X_i$
- code for putting the value of $n$ in $X_j$
- inline `$<$' $L_2$ $X_i$ $X_j$
- $L_1$: code for $\Phi$
- iterate_int $X_i$ $X_j$ $L_1$
- $L_2$:

Register $X_i$ is used for the counter while register $X_j$ contains the upper limit; they are set up by the first two instructions (which may be put_value instructions if the limits are given as variables). The following in-line test verifies that there are any values to try. If the compiler can prove that $m < n$, the instruction can be omitted. The iterate_int instruction is the only cost in the loop. It tests whether the value in $X_i$ is less than that in $X_j$ and, if so, increments the contents of $X_i$ by 1 and goes back to label $L_1$; otherwise simply proceeds. (For the `list' iterators we use another instruction but the principle is the same. Other `enumeration' iterators can also be handled by the iterate_int instruction but we leave them aside for now.)

The corresponding compilation schema for a numeric quantification is identical except that it requires using a third temporary register $X_k$ in which the result is computed. The register should be initialized before the in-line test and then manipulated (e.g., added to) in each pass through the loop.
When label L2 is reached, the value of the bounded quantification is that contained in Xk. For example, a sum quantification

\[ \sum_{I : m \leq I \leq n} E \]

is compiled as

```
code for putting the value of m in Xi
code for putting the value of n in Xj
code for putting 0 in Xk
inline '<' L2 Xi Xj
L1: code for adding the value of E to Xk
iterate_int Xi Xj L1
L2:
```

We will not give the other two schemas here, because all examples below use the first schema.

In our extended sequential WAM many predefined relations are implemented through two instructions `builtin` and `inline`. They both have an argument which is a key, determining the predefined relation to be computed. The only difference between `builtin` and `inline` is that `inline` takes an extra argument telling where to jump if computing the relation fails, while `builtin` fails to the most recent choice point. The `inline` instructions are relevant here because they are used when compiling bounded quantifications.

## 4 PARALLEL MACHINE MODEL

We are considering any parallel computer that supports the data parallel programming model. This programming model has been realized both on SIMD computers and on MIMD computers, with uniform or non-uniform memory access.

A program contains sequential and parallel parts. The latter parts will be run by many processors, usually applying some procedure to all data in some collection. It is not in principle relevant whether the code implementing a parallel part of the program is run in 'lock step' by the parallel processors, or not. The important property is that synchronization occurs before communication is attempted.

In our presentation arrays are of special importance. Some arrays, which we will call **sequential arrays**, are stored totally in the memory of a single processor, the one that runs sequential code. That processor could be one of the parallel processors or a separate processor.

Other arrays have their storage distributed over the memories of many parallel processors (assuming non-uniform memory access) so that each of these processors has direct access to some elements of the array. We call such arrays **parallel arrays**.
5 THE PARALLEL WAM EXTENSION

Let us begin by pointing out that although this data-parallel extension of WAM is sufficient for running several interesting programs, it certainly does not cover all programs with bounded quantifications. Nevertheless, it is sufficient for illustrating the general idea.

The compilation schema for a bounded universal quantification

\[ \# \text{and } I : m \leq I \leq n : \Phi \]

where the body \( \Phi \) is deterministic and contains no out-of-line calls (we again assume that the registers \( X_i \) and \( X_j \) are free) is

- code for putting the value of \( m \) in \( X_i \)
- code for putting the value of \( n \) in \( X_j \)
- inline \( \langle \cdot \rangle \) \( \langle X_i \rangle X_j \)
- \text{par\_begin } X_i X_j
- \text{parallel code for } \Phi
- \text{par\_end}

\[ L : \]

Registers \( X_i \) and \( X_j \) remain unchanged by the \text{par\_begin} and \text{par\_end} instructions and serve only to contain the limits for the bounded quantification. The \text{par\_begin} instruction configures the parallel machine to appear as having \( X_j - X_i \) abstract processors and initiates parallel computation in them, while the \text{par\_end} instruction returns to the sequential emulator.

At the WAM level, the abstract machine appears to have an unlimited number of processors. Given a certain number of physical processors these abstract processors must be simulated. On a Connection Machine (Hillis, 1985; TMC, 1987, 1991), this is provided for through ‘virtual processors’, which are supported at a low level. On other actual parallel computers the emulator itself must provide this facility.

Within parallel code some new registers and instructions are available. There is a limited number of parallel temporary registers, called ‘P-registers’, each of which may contain a different value for each parallel processors. In the current, very limited implementation, they may only contain numbers or uninstantiated variables. Those variables may only be bound to numbers.

The instruction \text{put\_p\_constant } e P_i stores the (numeric) constant \( e \) in P-register \( i \) for every processor.

The instruction \text{put\_p\_void } P_i initializes the P-register \( i \) of each abstract processor with a unique unbound variable.

Assuming that the registers given to the preceding \text{par\_begin} contained the values \( \text{low} \) and \( \text{high} \), respectively, the instruction \text{put\_p\_index } P_i will store the value \( \text{low} \), \( \text{low} + 1 \), \ldots, \( \text{high} - 1 \) in P-register \( i \) of each abstract processor 0, 1, \ldots, \( \text{high} - \text{low} \).
The instruction `get_p_value P; X_j` makes every abstract processor retrieve the contents of X-register \( j \) and unify the value with the contents of its P-register \( i \).

The parallel WAM contains a number of new predefined parallel relations (called by the `inline` and `builtin` instructions), which are computed in every processor. The most important of these, for our purpose here, are various arithmetic and comparison relations. For example, the instruction `par_reduce+_ P; X_j` adds together the values contained in P-register \( i \) of each processor and unifies the sum with the contents of X-register \( j \).

## 6 Parallel and Sequential Arrays

We have also extended Warren’s machine with support for arrays. Ordinary, sequential, arrays are stored in the same memory as other structured terms. Each such array resides in the memory of a single processor.

We introduce a ternary relation `size` such that `size(i, a, d)` is true whenever \( i \) is a non-negative integer, \( a \) is an array and \( d \) is the number of elements in the \( i \)th dimension of \( a \).

We also predefine the ternary relation `elt`, letting `elt(j, a, x)` be true whenever \( j \) is a non-negative integer, \( a \) is an array and \( x \) is the \( j \)th element of \( a \). Array indices thus always go from 0 to \( k - 1 \), where \( k \) is the size of the first dimension of the array. We will assume that a term \( a[j] \) denotes “the value \( x \) such that `elt(j, a, x)` is true”. If there is no such element (i.e., \( j \) is not a valid index of \( a \)) then the computation should fail.

The `builtin` and `inline` instructions of our emulator implement the `size` and `elt` relations. Our examples below all use one-dimensional arrays and the first argument to `size` is therefore always 0. The code for the examples therefore uses a special case, `builtin size0 X; X_j`, where registers \( X_i \) and \( X_j \) contain the second and third arguments of the call to `size`, respectively.

For the parallel extension of WAM we have also added parallel arrays. The restriction above on the contents of P-registers also applies to the elements of parallel arrays. In the future this restriction will be partially lifted.

The language is extended with ternary relations `par_size` and `par_elt` which are the analogues of `size` and `elt` above. The first of these relations is implemented by an ordinary sequential case of the `builtin/inline` instruction (`par_size0` in the examples) while the second is implemented by a parallel case of the same instruction (`par_elt`) when occurring in parallel code.

## 7 Sample Programs

Let us present bounded quantification programs, and extended WAM code for running them on sequential or parallel computers, for three problems: computing the euclidean inner product of a pair of vectors, computing an approximation to an integral using Simpson’s method, and finding roots in an oriented forest.
**Inner product**

The following program uses bounded quantification for the inner product problem. The relation $e_{i,p}$ is true of values $x$, $y$ and $s$ whenever $s$ is the euclidean inner product of the arrays $x$ and $y$.

$$
e_{i,p}(X, Y, S) :-
\text{size}(0, X, N), \text{size}(0, Y, N),
S = (\sum I : 0 \leq I < N : X[I] \ast Y[I]).$$

Its sequential extended WAM code is

```prolog
e_i_p/3:
  put_void X3
  builtin size0 X0 X3
  builtin size0 X1 X3
  put_constant 0 X4
  zero_p X3 L2
  put_constant 0 X5
L1:  put_void X6
     builtin elt X5 X1 X6
     put_void X7
     builtin elt X5 X0 X7
     builtin '•' X6 X6 X7
     builtin '+' X4 X4 X6
  iterate_int X5 X3 L1
L2:  get_value X2 X4
     proceed
```

while the corresponding parallel code is

```prolog
e_i_p/3:
  put_void X3
  builtin par_size0 X0 X3
  builtin par_size0 X1 X3
  put_constant 0 X4
  inline '<' L1 X4 X3
  par_begin X4 X3
     put_p_index P0
     put_p_void P1
     put_p_void P2
     builtin par_elt P0 X0 P1
     builtin par_elt P0 X1 P2
     builtin '•' P1 P1 P2
     builtin par_reduce_+ P1 X2
  par_end
```

7
Integral approximation
The relation \textit{intsimp} holds for three numbers \(a\), \(b\) and \(i\), and a positive integer \(n\) if \(i\) is an approximation to the integral \(\int_a^b f(x)\,dx\) where \(n\) is the number of intervals. We assume that the expression \(\varphi[x, y]\) is such that it is true if and only if \(f(x) = y\), where \(f\) is the function being integrated. The following program uses bounded quantification to code Simpson’s method for solving the integral approximation problem.

\begin{verbatim}
intsimp(A, B, N, I) :-
    W = (B-A)/N, 
    size(0, G, 2*N+1),
    and J,Y : G[J] = Y : \varphi[A+J*W/2,Y],
    I = sum J : 0 \leq J < N :
        W * ( \varphi[A+J*W] + 4*\varphi[A+J+1] + \varphi[A+2*J+2]) / 6,

and when \(\varphi[X,Y]\) is chosen to be \(Y=4/(X*X+1)\) the sequential extended WAM code is

\begin{verbatim}
intsimp/4:
    builtin '-' X4 X1 X0
    builtin '/' X4 X4 X2
    put_constant 2 X5
    builtin '*' X6 X5 X2
    builtin '+' X6 X6
    put_void X7
    builtin size0 X7 X6
    zero X6 L2
    put_constant 0 X1
L1:   builtin elt X9 X7 X1
    builtin '*' X8 X1 X4
    put_constant 2 X5
    builtin '/' X8 X8 X5
    builtin '+' X8 X8 X0
    builtin '*' X8 X8 X8
    builtin '+' X8 X8
    put_constant 4 X5
    builtin '/' X8 X5 X8
get_value X8 X9
\end{verbatim}
\end{verbatim}

8
iterate_int X1 X6 L1
L2:  zerop X2 L4
put_constant 0 X1
put_constant 2 X10
put_constant 4 X11
put_constant 6 X12
put_constant 0 X13
L3:  builtin '* X0 X10 X1
     builtin elt X14 X7 X0
     builtin '+ X0 X0
     builtin elt X15 X7 X0
     builtin '*' X15 X11 X15
     builtin '+ X0 X0
     builtin elt X16 X7 X0
     builtin '+' X14 X14 X15
     builtin '+' X14 X14 X16
     builtin '*' X14 X4 X14
     builtin '/' X14 X14 X12
     builtin '+' X13 X13 X14
iterate_int X1 X2 L3
L4:  get_value X3 X13
     proceed

while the corresponding parallel code is
intsimp/4:
     builtin '-' X4 X1 X0
     builtin '/' X4 X4 X2
     put_constant 2 X5
     builtin '*' X5 X5 X2
     builtin '+' X5 X5
     put_void X6
     builtin par_size0 X6 X5
     put_constant 0 X7
par_begin X7 X5
     put_p_index P0
     get_p_value P1 X4
     builtin '*' P1 P0 P1
     put_p_constant 2 P2
     builtin '/' P1 P1 P2
     get_p_value P3 X0
     builtin '+' P1 P3 P1
     builtin '*' X1 X1 X1
     put_p_constant 4 P4
Roots of an oriented forest

Finally, the following bounded quantification program is a solution for the problem of computing the roots of an oriented forest. The relation \( \text{find} \) holds for two vectors \( p_0 \) and \( p \) if \( p_0 \) and \( p \) are oriented forests representing the same equivalence relation and the depth of \( p \) is one.

\[
\text{find}(P,P) :- \\
\quad \text{#and I : P[I] = P[I] : P[I] = P[P[I]]}.
\]

\[
\text{find}(P0,P) :- \\
\quad \text{#and I,J : P0[I] = J : P1[I] = P0[J],} \\
\quad \text{find}(P1,P).
\]

The odd range formula of the first quantification is simply one way of saying that the range of the iteration variable is the sequence of valid indices in \( P \). The program has the following corresponding sequential WAM code.

\[
\text{find}/2:
\]
try_me_else 3
  get_value X0 X1
  put_void X2
  builtin size0 X0 X2
  zerop X2 L2
  put_constant 0 X3
L1:  put_void X5
    builtin elt X3 X0 X5
    builtin elt X5 X0 X5
    iterate_int X3 X2 L1
L2:  proceed
L3: trust_me
  put_void X2
  builtin size0 X0 X2
  builtin size0 X1 X2
  put_void X6
  builtin size0 X6 X2
  zerop X2 L5
  put_constant 0 X3
L4:  put_void X4
    builtin elt X4 X0 X3
    builtin elt X5 X0 X4
    builtin elt X3 X6 X5
    iterate_int X3 X2 L4
L5:  put_value X6 X0
execute find/2

The data parallel WAM code is

find/2:
  try_me_else L2
  get_value X0 X1
  put_void X2
  builtin par_size0 X0 X2
  put_constant 0 X3
  inline '<' L1 X3 X2
  par_begin X3 X2
    put_p_index P0
    put_p_void P1
    builtin par_elt P0 X0 P1
    builtin par_elt P1 X0 P1
  par_end
L1:  proceed
L2: trust_me
<table>
<thead>
<tr>
<th>Program</th>
<th>Rec.</th>
<th>B.q., seq.</th>
<th>B.q., par.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>1020</td>
<td>825 (81%)</td>
<td>78/275 (7.6%/35%)</td>
</tr>
<tr>
<td>IA</td>
<td>1279</td>
<td>917 (72%)</td>
<td>42/214 (3.3%/20%)</td>
</tr>
<tr>
<td>OF1</td>
<td>919</td>
<td>894 (97%)</td>
<td>91/494 (9.9%/64%)</td>
</tr>
<tr>
<td>OF3</td>
<td>2760</td>
<td>2345 (81%)</td>
<td>—</td>
</tr>
<tr>
<td>OF9</td>
<td>8300</td>
<td>6611 (80%)</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 1: Comparing run times (ms).

put_void X2
builtin par_size0 X0 X2
put_void X3
builtin par_size0 X3 X2
put_constant 0 X4
inline '<' L3 X4 X2
par_begin X4 X2
    put_p_index P0
    put_p_void P1
    put_p_void P2
    builtin par_elt P0 X0 P1
    builtin par_elt P1 X0 P2
    builtin par_elt P0 X3 P2
par_end
L3:    put_value X3 X0
execute 'find'/2

8 EXPERIMENTAL RESULTS

We have also written recursive programs for the problems above. We do not show the programs or their WAM code here, begging the reader to trust that we do a fair comparison.

We have run the recursive programs and the bounded quantification programs using the extended sequential abstract machine on a computer with one SUN-4M (SPARC) processor. Then we have run the bounded quantification programs using the parallel abstract machine on a Connection Machine model 200 with 4096 processors.

The inner product programs (IP) were given two vectors of 4000 elements and the programs were run 10 times. The integral approximation programs (IA) were invoked to approximate the integral $\int_1^3 4/(x^2 + 1) dx$ using 2000 ‘stripes’ and were run 4 times. The oriented forests programs (OF) were given a degenerated tree of 4000 elements and were run 1, 3, and 9 times.\footnote{One notes that the sequential bounded quantification program wins more and more
Each of these experiments was repeated 50 times\(^2\) and Table 1 shows the average run times that were obtained, in milliseconds.

In the ratios, the times for the recursive programs are defined as 100%.

The times for the Connection Machine are given as a pair, where the first number is the amount of time spent by the parallel (back-end) computer and the second number is the time spent by the sequential (front-end) computer.

The first ratio for the Connection Machine relates the time spent in the parallel computer to the time for the sequential recursive program. The second ratio relates the sum of the front-end and the back-end times to the same.

We present all these figures because we do not know yet how much of the time spent in the front-end that actually should be included for a fair comparison. All significant computation takes place in the parallel processors so we believe that most of the front-end time is overhead that does not increase when the problem size or the number of processors increases.

What we would like to measure is the ratio of run time to problem size for these and other problems, on sequential and parallel computers. These data points are too few to draw any firm conclusions. That we save up to 96% of the run time may not be very impressive considering the 4096 processors spent to achieve it. However, it is well known that the sequential performance of the parallel processors of a Connection Machine model 200 is indeed very much lower than that of a SUN-4M processor. On a Connection Machine model 5, which uses such processors as parallel elements, we predict a uniform and linear speed-up when related to the uniprocessor.

Viewed another way: few, if any, other logic programming systems can exploit the massive parallelism of a Connection Machine at all, and solving a problem approximately 30 times faster than a SUN-4M, using the same algorithm, is certainly promising for the future.

9 RELATED WORK

Kacsuk (1990) has proposed DAP Prolog, a parallel Prolog with an array extension. His integration of arrays is quite different from ours; arrays appear in the programs as predicates and Prolog's `is`-relation is extended to operate on array elements. However, his idea for parallel execution is similar to ours. There is a particular array mode, corresponding to certain bounded quantifications, which allows array programs to use the data parallelism of the SIMD-type DAP computer (Reddaway, 1979). The array mode of

\(^2\)On the Connection Machine the first time was always dramatically higher than the others and we decided to ignore it, thus the averages on the CM are instead for 49 times.
Kacsuk's language could certainly be realized also on other data parallel computers.

The most significant related theoretical work is that of Voronkov who has studied a resolution-based inference system that incorporates something very close to our bounded quantifications. It appears that his work provides suitable theoretical foundations for ours.

10 FUTURE WORK

The benchmark figures above are the first that we have measured. We will immediately go on to do more elaborate benchmarks in order to evaluate our idea properly.

Our examples above were compiled by hand and we aim to extend a Prolog compiler with support for (sequential and parallel) bounded quantifications.

We ought to write bounded quantification programs for some larger problems, to find a suitable set of quantifiers and iterators for real programs. For example, we expect that it is important to handle efficiently nested bounded quantifications, drawing an analogy from conclusions by Blelloch (1990).

We also consider running bounded quantifications on a shared-memory multiprocessor (SUN 630MP) and expect nearly linear speed-ups.

It would be interesting to add bounded quantifications also to other sequential logic programming languages, e.g., Gödel (Hill & Lloyd, 1991).

ACKNOWLEDGEMENT

This work was supported by the Swedish National Board for Technical and Industrial Development (NUTEK) under contract No. 221-91-344 (Massively Parallel Logic Computation).

REFERENCES


