Executing Bounded Quantifications on Shared Memory Multiprocessors[§]

(Extended Abstract)

Jonas Barklund and Johan Bevemyr

Uppsala Univ., Comp. Sci. Dept.; Box 311; S-751 05 Uppsala; Sweden
Phone: +46-18-182500, fax: +46-18-521270
{jonas,bevemyr}@csd.uu.se

Introduction

Bounded quantifications have been proposed as a useful linguistic construct for logic programs [2, 8]. Firstly, bounded quantifications are more convenient than recursion for expressing many algorithms. Secondly, bounded quantifications with integer ranges express definite iteration and therefore have a high potential for parallelization, for example, on computers supporting the data parallel model of computation.

In previous reports we have discussed implementation of Prolog extended with bounded quantifion on sequential computers [3] and on an SIMD computer (the Connection Machine by Thinking Machines Corp.) [1]. In this paper it is shown how we can exploit the affinity between bounded quantification and recursive programs to run bounded quantifications on an existing implemented abstract machine that was developed to run a class of recursive programs efficiently on shared-memory MIMD multiprocessors [5].

In order to do so one can transform programs with bounded quantifications to recursive programs, which can be compiled to run efficiently on the abstract machine developed for recursion parallel execution [6]. Some design decisions for this machine may not be optimal for running bounded quantifications, but from the performance of our example programs we conclude that bounded quantifications nevertheless map quite well on the machine. More details are provided in the full version of this paper [4].

Bounded Quantifications

In its most general form a *quantification* is an expression that applies an abstraction (its body) to a set of values, combining the results in a way that depends on the quantifier. A *bounded quantification* is a quantification where the set of values is finite. Voronkov has shown how to incorporate a form of bounded quantifications in a Horn clause language with SLD-resolution [8].

In the context of logic programming, the most obvious quantifiers are the logical universal and existential quantifiers of first-order predicate calculus that combine the values of the body by conjunction and disjunction, respectively. However, we can also think of other quantifiers. In mathematics, sums and products are readily thought of a quantifiers, combining the values of the body by addition and multiplication, respectively, but there are several other similar operations from mathematics that combine the values of the body through reduction with some binary operator.

In this abstract we shall consider only the universal and sum quantifiers. We will also only express the finite sets as integer ranges. Other quantifiers and expressions for the finite sets have been discussed elsewhere [2, 3, 8]. We write a universal quantification as `all(`m`leq`i`<`n`,`

[§]This is an abridged version of a full paper submitted to PLILP '93.

$\beta[i]$), where m and n are integer expressions, and the body $\beta[i]$ is a goal. The expression i must be a variable, called the *iteration variable*, and is considered locally scoped in the quantification. The universal quantification succeeds if $\beta[m]$, $\beta[m{+}1]$, ..., $\beta[n{-}1]$ all succeed (with the same instantiation of any global variables). We write an arithmetic quantification as κ(*m*leq*i*<*n*, $\beta[i]$, τ), where κ is an arithmetic quantifier such as `sum`, `prod`, `min` or `max`, m, n and i are as above, the body $\beta[i]$ is an arithmetic expression, and τ is a variable or a number. The arithmetic quantification succeeds if the result of reducing $\beta[m]$, $\beta[m+1]$, ..., $\beta[n-1]$ with the appropriate binary operator equals the value of τ (possibly instantiating τ).

The following three example programs incorporate bounded quantifications and are used below as benchmarks for the parallel implementation. `e_i_p`: computing the Euclidean inner product S of arrays X and Y. `intsimp`: computing an approximation I of $\int_A^B f(x)dx$ with N slices, where $r(X,Y) \equiv f(X) = Y$, using Simpson's method. `find`: finding roots in an oriented forest represented by a vector.

```
e_i_p(X, Y, S) :-
    size(X, N), size(Y, N),
    sum(0<=I<N, X[I] * Y[I], S).

intsimp(A, B, N, I) :-
    W is (B-A)/N, Z is 2*N+1,
    size(G, Z),
    and(0<=J<Z, r(A+J*W/2, G[J])),
    sum(0<=J<N, W * (  G[2*J] +
                     4*G[2*J+1] +
                       G[2*J+2])/6, I).
```

```
find(P0, P) :-
    size(P0, N),
    and(0<=I<N, P0[I] = P0[P0[I]]),
    !,
    P0 = P.
find(P0, P) :-
    size(P0, N), size(P1, N),
    and(0<=I<N, P1[I] = P0[P0[I]]),
find(P1, P).
```

Parallel Execution

The general idea for running bounded quantification on parallel computers is analogous to loop parallelization in, e.g., Fortran. Often many iterations of a bounded quantification, sometimes all iterations, can be run simultaneously.

We view each iteration of a bounded quantification as a separate task, that can potentially be run in parallel. Synchronization primitives are used to handle data dependencies between iterations. We can understand the compilation process here as a transformation of bounded quantifications to recursive predicates. These recursive predicates are subsequently compiled using methods developed for running recursive programs [5, 7].

Unfortunately, the transformed arithmetic bounded quantifications use accumulators for computing sums and products, etc. Clearly, these accumulators introduce strict data dependencies. These can be dealt with using synchronization primitives: an efficient method is to let each worker have a local accumulator and add (or multiply) all local accumulators when the parallel execution terminates. For example, a bounded quantification `sum(0leq I<N, A[I])` is compiled corresponding to this schema:

```
for(I = 0, localacc = 0 ; I < N ; I++) {
    /* code for localacc = localacc+A[I] */
}
/* code for summing all local accumulators */
```

Abstract Machine

We use the abstract machine designed for recursion-parallel execution developed by Bevemyr, Lindgren and Millroth [5]. This machine 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 that support parallel execution. One worker is responsible for sequential execution (the *sequential worker*). During sequential execution all other workers (the *parallel workers*) are idle.

Data Sharing. A common code area is used and all workers have restricted access to each others heaps. All other data areas are private to each worker. The shared heaps are used to communicate terms created during sequential and parallel execution.

Scheduling. It is possible to use static scheduling since the set of parallel processes is known when the parallel execution is initiated.

Performance

We have measured the above programs on a 26 processors Sequent Symmetry using 24 processor (leaving 2 processors for operating system activities). We compare the parallel programs with recursive programs and sequential programs using bounded quantifications. Normalized speed-up expresses the ratio of uniprocessor execution time of the sequential program (compiled without parallelization) to the time using p processors on the program compiled with parallelization. Normalized parallel efficiency is defined as the normalized speed-up on N processors divided by N.

Workers	Execution time (s)	Relative speed-up	Normalized speed-up Rec.	Normalized speed-up BQ	Relative efficiency	Normalized efficiency Rec.	Normalized efficiency BQ
Inner Product (120000 elements)							
1	16.92	1	1.52	0.85	100 %	152 %	85 %
24	0.75	22.56	34.38	19.18	94 %	144 %	80 %
Numerical Integration (20000 intervals)							
1	26.19	1	1.16	0.89	100 %	116 %	89 %
24	1.26	20.78	24.20	18.55	87 %	101 %	77 %
Oriented Forest (10000 elements)							
1	19.13	1	1.07	0.76	100 %	107 %	76 %
24	1.49	12.84	13.73	9.82	54 %	57 %	41 %

Parallelization Overhead. The parallelization overhead is low for all programs when compared with a sequential bounded quantification program. The negative overhead, when compared with recursive programs, can be explained by the fact that recursion, as implemented in WAM, has a considerably higher overhead than the iteration instructions used to encode bounded quantifications on sequential and parallel machines.

Program	Overhead (rec.)	Overhead (BQ)
Inner product	-52 %	15 %
Numerical integration	-16 %	11 %
Root find (initialized)	11 %	16 %
Root find (uninitialized)	-7 %	24 %

References

[1] Arro, H., Barklund, J. and Bevemyr, J., Parallel Bounded Quantifications — Preliminary Results, UPMAIL Tech. Rep. No. 74, Comp. Sci. Dept., Uppsala Univ., 1993.

[2] Barklund, J. and Millroth, H., Providing Concurrency and Iteration in Logic Programming through Bounded Quantifications, in: H. Tanaka (ed.), *Proc. Intl. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha, Tokyo, 1992.

[3] Barklund, J. and Bevemyr, J., Prolog with Arrays and Bounded Quantifications, to appear in: A. Voronkov (ed.), *Proc. 3rd Intl. Conf. on Logic Programming and Automated Reasoning*, LNCS, Springer-Verlag, Berlin.

[4] Barklund, J. and Bevemyr, J., Executing Bounded Quantifications on Shared Memory Multiprocessors, submitted to PLILP'93, Comp. Sci. Dept., Uppsala Univ., 1993.

[5] Bevemyr, J., Lindgren, T. and Millroth, H., Reform Prolog: The Language and its Implementation, to be presented at the *International Conference on Logic Programming*, 1993.

[6] Bevemyr, J., *A Recursion Parallel Prolog Engine*, Ph.L. Thesis, Comp. Sci. Dept., Uppsala Univ., 1993.

[7] Millroth, H., Reforming Compilation of Logic Programs, in: V. Saraswat, K. Ueda (eds.) *Proc. 1991 Intl. Symp. on Logic Programming*, MIT Press, Cambridge, Mass., 1991.

[8] Voronkov, A., Logic Programming with Bounded Quantifiers, in: A. Voronkov (ed.), *Logic Programming—Proc. Second Russian Conf. on Logic Programming*, LNCS 592, Springer-Verlag, Berlin, 1992.