Reform Prolog [§]

(Extended Abstract)

Johan Bevemyr Thomas Lindgren Håkan Millroth

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

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

1. A big head unification, corresponding to the n small head unifications with normal control-flow, is performed.

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

3. The program is called recursively. This call is known to match the base clause. Hence, in practice, this call is often trivially cheap.

4. All n instances of the calls to the *right* of the recursive call are computed in parallel.

The Reform Prolog programmer tries to parallelize top-level (or near top-level) recursive procedures. When this is possible it ensures large-grain parallelism and minimizes the effect of Amdahl's law.

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. 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 process very close to ordinary sequential WAM code. The compiler emits standard WAM instructions, guarded by suspension instructions and atomic unification instructions. When the collected information permits, the extra instructions are removed. In particular,

[§]This presentation summarizes results presented at PARLE'93 and ICLP'93 in June 1993.

provably local computations execute precisely the sequential WAM code even when occuring in a parallel context. This results in low overheads for parallelization.

The apparent drawback of the execution model 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.

Parallelization is restricted to programs that do not make conditional bindings to variables shared between parallel processes. We say that such programs are *safe*. Note that safeness allows local nondeterminism within processes as long as it does not affect shared variables. The model also supports calling parallel subprograms from a nondeterministic program.

The implementation extends a convential Prolog machine with support for data sharing and process managment. The parallel machine consists of one 'worker' per processor, implemented as a separate UNIX process. The execution of a program alternates between two modes: sequential execution and parallel execution. One dedicated worker is responsible for sequential execution. 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.

Compilation has two main components: ensuring safeness and introduce suspension and locking unification instructions where required. This is done by extensive global dataflow analysis comprising type inference (with special support for lists and difference lists), safeness analysis (checking if the program satisfy the safeness requirement, possible with the aid of runtime tests) and locality analysis (checking when operations do not affect shared variables, in which case significant optimizations are possible).

Promising performance results, showing high parallel efficiency (91–99% on a set of benchmark programs) and low overhead for parallelization (2–12% on the same set of programs), have been obtained on a 24 processor shared-memory multiprocessor. These results compare favourably with other (more general) systems that supports dependent AND-aparallelism.