Smoothing the rough edges in Aurora[§]

(Extended Abstract)

Mats Carlsson
SICS
Box 1263, S-164 28
Sweden
matsc@sics.se

Ewing L. Lusk
Argonne National Laboratory
Argonne, IL 60439
U.S.A.
lusk@mcs.anl.gov

Péter Szeredi
IQSOFT
Teleki Blanka u.15–17
H-1142 Budapest, Hungary
szeredi@iqsoft.hu

Aurora is an OR-parallel implementation of full Prolog. It has been reported on in [8]. The system is nearing maturity, and we are beginning to use it for application work. Aurora is thus evolving from an experimental, research implementation to a complete, production-oriented system. The purpose of this paper is to describe our most recent enhancements to Aurora, motivated partly by the need to smooth the "rough edges" of the system, and partly by the needs of applications themselves. We will now briefly outline the issues involved in these most recent developments in Aurora.

Synchronisation

One of the principal goals of Aurora has been the support of the *full* Prolog language. Preserving the semantics of built-in predicates with side effects is achieved by *synchronisation*: whenever a non-leftmost branch of execution reaches a side-effect predicate, the given branch is suspended until it becomes leftmost [3]. This technique ensures that the side-effect predicates are executed in the same left-to-right order as in a sequential implementation, thus preserving compatibility with these implementations.

It is often the case that this strict form of synchronisation is unnecessary, and slows down parallel execution. Aurora therefore provides non-synchronised variants for most side-effect predicates which come in two flavours: the *asynchronous* form respecting the cut pruning operator, and the completely relaxed *cavalier* form.

There are groups of built-in predicates for which non-synchronised execution is not allowed, e.g. predicates relating to the programming environment such as `compile` and `save`. Such predicates are called *necessarily* synchronised. Most input-output and dynamic database predicates are *freely* synchronised, i.e. have non-synchronised counterparts.

All cavalier and asynchronous predicates are executed atomically, i.e. predicates using the same resource (e.g. the same input-output stream or the same dynamic predicate) are executed in mutual exclusion. More complex synchronisation primitives have also been built within Aurora, exploiting the atomicity of built-in predicates [9].

We will discuss the new non-synchronised built-in predicates both from the language and implementation point of view.

Dynamic Database Handling

Aurora provides two types of semantics for dynamic database updates. The virtual copy semantics [5] is used by default. This approach requires that a virtual copy is made of the

[§]This work was partially supported by the U.S.-Hungarian Science and Technology Joint Fund under project No. 031/90.

state of the dynamic predicate at each invocation, and so any further modification of the dynamic predicate will have no effect on the call in progress. The simpler, immediate update semantics causes the modification of a dynamic predicate to be carried out immediately, affecting all invocations in progress. The update semantics can be selected separately for each dynamic predicate.

Dynamic database handling is closely related to synchronisation. The built-in predicates for dynamic database handling are freely synchronised only if the immediate update semantics is prescribed for the dynamic predicate in question.

The Top-Level Interpreter

Every Prolog system has an interactive component, called the top-level interpreter, by which the user can enter goals to be executed and get their results. In an or-parallel environment the behaviour of the top-level interpreter raises several questions, such as the order of solutions returned by a query, and the speculativeness of the top-level goal (i.e. whether one should consider it being in the scope of a cut). We discuss the possible options for solving these issues and present the one adopted in Aurora.

Parallel Input-Output

Aurora provides asynchronous and cavalier versions of most input-output predicates, including character and term input-output, as well as open and close operations. This way parts of the Prolog program executing on different branches of the search tree can do their own asynchronous input-output to different files independently.

The present version of Aurora contains a dedicated process for handling input-output, which communicates with the workers through pipes [6]. The I/O process collects all input-output requests from the workers and passes them to UNIX. This makes it possible for several workers to cooperate in processing an input or output stream. The atomicity of input-output operations is implemented via locks associated with the streams handled by the I/O process.

A special predicate, `conversation(`*Goal*`)` is provided for carrying out an uninterruptible sequence of input-output operations on the terminal (the `user` stream). The `conversation` predicate is freely synchronised.

Parallel Tracing

Several visualisation tools have been developed or adapted for Aurora [1, 2, 4], which are extremely useful in analysing the parallel behaviour of programs. However, all these tools provide only a post-mortem analysis of log files created during the run. It is often the case that one would like to get an immediate feedback on the way a parallel program runs. Therefore we adapted the usual four-port Prolog trace for the parallel environment. We decided to make the trace non-interactive (output-only). We also modified the WAM-level "predtrace", which was introduced into the system as a debugging tool for the implementors themselves, to become a feature manageable by a normal user of the system.

Applications

We conclude by describing our experience of using the new features of Aurora in two application areas. The first is an experimental implementation of a general algorithm for solving optimisation problems [10]. The second area is computational molecular biology, where two search type applications have been ported to Aurora [7].

References

[1] M. Carro, L. Gómez, and M. Hermenegildo. Some event-driven paradigms for the visualization of logic programs. In *Proceedings of the Tenth International Conference on Logic Programming*. The MIT Press, June 1993. To appear.

[2] Terrence Disz and Ewing Lusk. A graphical tool for observing the behavior of parallel logic programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 46–53, 1987.

[3] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[4] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with upshot. Technical Report ANL-91/15, Argonne National Laboratory, 1991.

[5] Timothy G. Lindholm and Richard A O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In *Proceedings of the 4th International Conference on Logic Programming*, pages 21–39, 1987.

[6] Ewing Lusk. Remote I/O handling package specification. Internal Report, Gigalips Project, October 1989.

[7] Ewing Lusk, Shyam Mudambi, Ross Overbeek, and Péter Szeredi. Applications of the Aurora parallel Prolog system to computational molecular biology. In Doug DeGroot et al., editor, *Proceedings of the JICSLP'92 Joint Workshop on Distributed and Parallel Implementations of Logic Programming Systems*, November 1992.

[8] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[9] Péter Szeredi. Using dynamic predicates in an or-parallel Prolog system. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Logic Programming Symposium*, pages 355–371. The MIT Press, October 1991.

[10] Péter Szeredi. Exploiting or-parallelism in optimisation problems. In Krzysztof R. Apt, editor, *Logic Programming: Proceedings of the 1992 Joint International Conference and Symposium*, pages 703–716. The MIT Press, November 1992.