An Or-Parallel Prolog Execution Model for a Distributed Shared Memory Machine

Fernando M. A. Silva
LIACC, Universidade do Porto,
R. do Campo Alegre 823, 4100 Porto, Portugal
(Email: fds@ncc.up.pt)

Abstract. Recently, new parallel architectures, namely distributed shared memory architectures, have been proposed and built. These combine the ease-of-use of shared memory architectures with the scalability of the message-passing architectures. These architectures provide software and hardware support for shared virtual address space on physically distributed memory.

This paper describes Dorpp, an execution model that supports or-parallelism for these machine architectures, namely for the EDS parallel machine. Dorpp uses a shared memory model for or-parallelism. It attempts, however, at exploiting locality and at reducing communication overheads through scheduling and by caching accesses to remote shared data. The problem of memory coherency of cached data is discussed and solutions are proposed. Preliminary evaluation results of the execution model through simulation are presented.

Keywords: Languages for AI, Or-parallel Prolog, Distributed Shared Memory, Memory Coherency, Scheduling.

1 Introduction

Prolog is a popular programming language that is particularly important in Artificial Intelligence applications. Traditionally, Prolog has been implemented in the common, sequential general-purpose computers. More recently, Prolog implementations have also been proposed for parallel architectures where several processors work together to speedup the execution of a program. By giving better speedups, parallel implementations of Prolog should enable better performance for current problems, and expand the range of applications we can solve with Prolog.

One way to take advantage of parallel architectures for Prolog programs is by exploiting the implicit parallelism in the logic programs. Two main sources of implicit parallelism can be identified in Prolog programs: or-parallelism and and-parallelism. Or-parallelism arises from the parallel execution of multiple clauses capable of solving a goal, that is from exploring the non-determinism present in logic programs. And-parallelism arises from the parallel execution of multiple subgoals in a clause body.

Or-parallelism is particularly attractive because (i) Prolog's generality and simplicity can be preserved[LWH88]; (ii) it "offers good potential for large-scale,
large-granularity parallelism across a wide range of applications" [War87a]. In particular, applications in the area of artificial intelligence involving detection of all solutions or large searches such as expert systems, natural language processing, theorem proving or answering a database query, contain large amounts of or-parallelism [She86]; (iii) the efficient implementation techniques developed for Prolog can be easily extended to cope with or-parallelism [LWH88, Car90, AK90].

Indeed, the Aurora and Muse systems are examples of two successful or-parallel Prolog systems. Such systems support full Prolog, and have obtained good speedups for a wide range of applications. These systems were designed for bus-based shared-memory machines, where the several processors can access a common store via a bus. Unfortunately, current (bus-based) shared memory machines fail to scale over a few tens of processors due to the limited communication bandwidth of the shared bus. This restricts the maximum parallelism a parallel Prolog system can extract.

To attain more parallelism, traditionally one would use distributed memory machines which are scalable to very large numbers (thousands) of processors [AG89]. In these parallel machines, each processor has its own local memory. Access to remote memory, as well as communication and synchronisation between processors, is accomplished through a message passing mechanism. This approach is rather inflexible and expensive for the implementation of Prolog, as any shared datum must be explicitly copied between the processors.

Recently, new parallel architectures, namely distributed shared memory architectures, have been proposed and built. These architectures contain no physically shared memory, although they provide software and hardware support for shared virtual memory. Fundamentally, this is implemented by making each local memory to work as a cache for the shared virtual memory. One example of such architecture is the EDS machine [WTW90]. In this machine, whenever a processor accesses a remote memory location, a page containing that location is copied and stored in the processor's cache. This localises subsequent accesses to cached pages, hence reducing overall communication overheads. Note that a mechanism is required to ensure memory coherency (a similar requirement exists for shared memory architectures using caches). The shared virtual memory simplifies the programming process as it allows complex structures to be passed by reference and hides the remote communication mechanism from the processes.

By combining the advantages of large number of processors with the advantages of a shared memory programming model, distributed shared memory architectures are an ideal target for the execution of Prolog programs with much parallelism. In this paper, we describe Dorpp, an execution model that supports or-parallelism for these architectures, and particularly for the EDS machine.

The main problem for Dorpp to solve, in order to use a distributed shared machine efficiently, is to reduce the number of remote memory accesses, or in other words, to cache data as much as possible. In practice, read-only data is easier to cache, and to obtain the best results, we were interested in making the shared memory as much read-only as possible. This is supported by Warren's SRI