Or-Parallel Prolog with Heuristic Task Distribution

Wai-Keong Foong

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052, Australia.
Tel: +61 3 344-4247  Fax: +61 3 348-1184
Email: foong@cs.mu.oz.au

Abstract

An important determiner of the performance of or-parallel Prolog systems is the order in which the or-nodes in the computation tree are tried. Existing models of or-parallel Prolog select clauses for execution in much the same way as sequential systems do; clauses are tried chronologically (that is, in the order stated in the program) with the added advantage that some can be executed simultaneously on other processors. There is no attempt to examine clauses to assess their suitability in terms of finding answers or select those clauses that have low overhead in task migration for scheduling to other processors. Such systems do not exploit or-parallelism to the fullest and may sometimes have worse performance than sequential systems depending on the structure of the computation tree. Often in an or-parallel environment, it is more desirable to distribute nodes with larger work load to other processors leaving those nodes with lighter work load for local processing. This is so that scheduling costs, including task creation and task migration, are reduced and processors spend more time doing useful work. [8, 5] show that or-parallel Prolog systems are more suited to coarse-grain parallelism. We describe a method of gathering empirical data about clauses in the program. A heuristic function uses the data to deduce the relative “weights” of each clause and the system uses these “weights” to select the clauses for local processing or distributing to other processors.

1 Introduction

Prolog offers two types of parallelism, and-parallelism and or-parallelism. And-parallelism is the simultaneous execution of the goals in the body of a clause, while or-parallelism is the simultaneous execution of alternative clauses that match a goal. In this paper we are concerned only with or-parallelism.

Most modern sequential Prolog systems are implemented according to the WAM architecture [16], where the creation and access of variable bindings are fast constant-time operations (that is, the time for these operations is independent of the amount of work in the processor). To create and bind a variable requires nothing more than a write instruction to a memory word, while accessing its value involves a single read instruction. In the case of binding a variable belonging to an ancestral node (conditional binding), another write operation is needed in the trail. This area holds all conditional bindings and these need to be undone when a processor backtracks to try an alternative clause. Backtracking in Prolog is somewhat akin to task switching and corresponds to the try, retry and trust operations in the WAM.

In an or-parallel Prolog system, backtracking is replaced by spawning multiple tasks to examine alternative clauses. For an or-parallel Prolog system to be more time-efficient than a sequential Prolog system, the cost of creating multiple tasks must not be so much greater than the cost of the backtrack operation that it outweighs the gains from parallel execution. It is also desirable that any or-parallel implementation of Prolog maintains the efficiency of the operations to bind and access variables.

There are two important aspects of an or-parallel system. One is the selection of tasks for distribution to other processors; the other is the task switching operation itself. Often task switch operations are somewhat affected by the memory architecture of the machine. In shared memory systems like the Aurora model [1], task switching on a processor involves unwinding the private binding array which holds conditional bindings of variables in or-nodes when the processor moves up the tree and restoring bindings as it moves down the tree to the appropriate branch where work is available. In these cases the time taken for task switch operations is dominated by the bind and unbind operations in the binding array. With the Muse model [3], task switching involves copying of stack space (term stack) of the WAM data areas. This approach is more suitable to architectures with non-shared memory models. Evidently the main cost of task switching comes from the copying operations. The larger the heap of the task being distributed, the more time taken to replicate this heap on other processors. Our model is based on the stack copying approach similar to the Muse.

In [5] we measured the performance of two methods of scheduling or-nodes in an or-parallel environment. One involves work distribution giving priority to or-nodes higher up in the computation tree (top-most)
while the other gives priority to or-nodes closer to the leaves (bottom-most). The result shows that scheduling top-most nodes first performs better in load balancing, usage of processor resources and overall execution time of the system than bottom-most node scheduling. The model was successful because it was based on the assumption that or-nodes higher up the tree would have acquired a smaller heap and hence have a lower cost of copying than those at lower levels. Another important assumption was that higher nodes have larger workload and hence are more favourable for distribution. Such assumptions may be reasonable for a well-balanced computation trees. However, many Prolog computation trees have highly irregular (unbalanced) shape, and for these trees we can no longer assume that top-most nodes are necessarily heavier. Therefore to ensure optimal scheduling of large work to idle processors we need to determine the "weight" of each clause within the or-node. In our system the "weight" of a clause consists of two components; the number of steps it takes to reach it's leaf node, the amount of memory cells consumed by the clause. The formal gives an estimate of the actual amount of workload of the clause while the latter estimates the amount of copying overhead necessary to move this clause to another processor. These two components are antithetical to the scheduling decision and the overall weight estimation function must reflect this.

In this paper we describe a method of static and dynamic analysis of clauses in a or-node for a Prolog program. The static analysis involves collection of clause attributes that pertain to the two components and the dynamic analysis consists of applying a heuristic function to these attributes to give an estimate of the "weight" of the clause. Clauses in an or-node are then ranked according to their weights and the function of the scheduler is to choose the clause whose weight is the highest for distribution. For a full description of the scheduling process and the model, see [4, 5].

In Section 2 we describe the clause attributes relevant for our analysis, the method of gathering these attributes and the heuristic function that applies to the data. Section 3 gives a brief description of the system architecture from [4] incorporating this technique of clause selection. Section 4 gives an analysis of the new model and Section 5 presents the conclusion.

2 Heuristics

In this enhanced scheduling scheme each clause in an or-node is assigned a "weight". This "weight" reflects the favourability of the clause for distribution to other processors. The larger the weight the higher the priority for the clause to move to another processor.

The weight assigned to clauses in or-nodes is computed using an evaluation function which takes several clause attributes as inputs. Clause attributes can be collected during the compilation phase and the execution phase. In the compilation phase, static features of clauses are examined. For example, the number of literals in the clause (subgoals), the type of arguments in the head clause, the groundness of arguments and the number of distinct variables. In the execution phase, dynamic features are collected. This includes examining the arguments of clauses, especially those of type list and structure. Our analyser, in both phases, consists of two main components. One for analysing the explosiveness of clauses and the other for examining the arguments of the clause and the query (goal) presented. The result generated by them will be fed into the evaluation function to produce a weighted tuple of clause attributes and inserted into a table for easy reference during the execution phase. We discuss the two main components of the analyser below.

2.1 Explosiveness of Clauses

The number of literals is the primary measure of the explosiveness of the clause, and if we can determine the number of steps it takes to resolve all subgoals then an approximation can be made for the execution time of the clause. This gives us a fair indication of what the work load is for this clause. We can deduce the number of steps each subgoal takes with the help of a simple meta-interpreter. An example of a meta-interpreter for pure Prolog is,

```prolog
% solve(Goal, Steps)
solve(true, 0) :- !.
solve((GoalA, GoalB), Steps) :- !,
solve(GoalA, StepA),
solve(GoalB, StepB),
```