Abstract

Several abstract models have already been proposed for parallel logic programming, but the actual developments are essentially sequential simulations, often based on monoprocessor systems: this is widely due to the fact that the problem encounters fundamental limits. This paper describes a cooperative scheme between Prolog interpreters, based on the And-parallelism opportunities of logical programming. The scheme has recently been implemented on a Transputer-based architecture, with the Parallel language. The grain of parallelism is medium-sized, to fit the message-passing characteristic of the architecture and the choice of the interpretative way.

Keywords: Parallel programming - High level languages - Prolog -

1. Introduction

Sequential realizations of logic programming languages have benefited from progresses in the past such as the Warren machine compilation scheme[AIK91], but their performances are still a source of bottleneck. For improvements, parallel solutions able to involve several processors are very much desirable.

1.1. Logic programs

A logic program [SchaSter87] is a set of axioms, or rules, defining relationships between objects. A computation of a logic program is a deduction of consequences of these axioms. The concepts of logic programming are based on the Robinson's resolution[Rob65] principle and unification algorithm as engine of a logic-based computation model. Kowalski[Kow74] found that such a control mechanism can be applied to restricted class of logical theories, namely Horn Clause theories. His major insight was that universally quantified axioms of the form $A \iff B_1 B_2 B_3 \ldots B_n; n\geq 0$, can be read both declaratively, saying that $A$ is true if $B_1$ and $B_2$ and ... $B_n$ are true, and procedurally, saying that to prove the goal $A$ (execute procedure $A$, solve problem $A$), one can prove subgoals (execute subprocedures, solve subproblems) $B_1$ and $B_2$ and ... $B_n$. Such axioms called definite-clauses. A logic program is a finite set of definite-clauses.

1.2. The parallelism in logic programming

Logic programming languages, at first glance, seem to offer intrinsic parallel opportunities. The And- and Or-parallelism possibilities have been presented in the literature [CCRS89],[CDO88]. In an And-parallelism scheme, one tries to evaluate in parallel the different terms of the right hand size of a clause. The Or-parallelism consists of trying to solve in parallel the different Prolog rules which unify to a given goal. Then we can say that the difference between the sequential execution and the parallel execution in logic programming, lies in the way they deduce consequences from such axioms.
But the complexity of the interplaying of the different resolutions in parallel execution is considerable. The And-parallelism brings in read/write conflicts for shared variables, and a kind of synchronism must be provided. For the OR-parallelism model, this type of conflict may be avoided by the creation of a new process for solving each applicable rule. But this leads in turn to the problems of replicating and managing the exploding number of processes. Anyway, one is lead either to simplifications, by restricting the possibilities of the language, or to algorithms whose practical feasibility is questionable, at least in the short run.

In fact, several research results [Gup87], [Kacs87] conclude to the possibility of a ten-times speed-up or so. Moreover, this implies to give up several semantical possibilities, among the most interesting ones. In a same way, Concurrent_Prolog [Shap83], Parlog [Greg87] [Kla86], Guarded_Horn_Clause(GHC) [Ued85] replace non-determinism by choices guided by the programmer, and [Wes87] excludes complex terms of the scope of his language.

1.3. Our contribution

We present in this paper a parallel execution scheme, which is derived in some way from the And-parallelism. This scheme does not try And-parallelism systematically, but guides its application on a knowledge-partitioning scheme, and through this scheme, we look after a medium-size grained parallelism (as several authors we do not think that a fine-grained parallelism is practical in the short run). this is the reason of our choise for a cooperative scheme between interpreters.

Another reason is the kind of architecture: with a large number of processors, avoiding the loading-times seems to be an interesting feature; and the debugging of the interpreters may be made once for all times, we think that cooperation schemes may be in the future enlarged to encompass interpreters of different languages.

The chosen architecture is based on Transputers: communications between processors are in form of messages because there is no shared memory. Developments are made with the Parallel C programming language.[PaC88].

In this paper, we describe the principles and the realization of the And-cooperation between interpreters. A complementary scheme is already designed for combining in the future, this And-parallelism scheme with a kind of Or-parallelism scheme. A future goal is to obtained by this combination a ten-time order speed-up, with no loss for the semantic possibilities of Prolog.

Section 2 describes the principle and the main design of the And-cooperation scheme. Section 3 is dedicated to the implementation of the scheme on the Transputer architecture using a parallel language. And section 4 deals with the conclusion and further works.

2. And-Cooperation Scheme Between Interpreters

In this section, we describe the main concept of the And-cooperation with the definition of the different processes of the scheme.

2.1 - Principle : a priori knowledge-partitionning

The principle is to force different Prolog Interpreters (PI for short), placed on different processors, to cooperate for the resolution of a goal. Let us just recall from Prolog practice that:

. a goal is a predicate term (in which some of the variables may have previously been bound to values) which the interpreter tries to solve by recursive application of the program rules,

. a procedure is a group of rules applicable to a same predicate. (fig 1)