To be published in the *Computer Languages Journal*

Improving the Efficiency of Nondeterministic Independent And–parallel Systems

Enrico Pontelli, Gopal Gupta
DongXing Tang

Laboratory for Logic and Databases
Dept. of Computer Science
New Mexico State University
Las Cruces, NM, USA
{epontell,gupta,dtang}@cs.nmsu.edu

Manuel Carro
Manuel Hermenegildo

Facultad de Informática
Universidad Politécnica de Madrid
28660-Boadilla del Monte
Madrid, Spain
{mcarro,herme}@fi.upm.es

Abstract

We present the design and implementation of the and-parallel component of ACE. ACE is a computational model for the full Prolog language that simultaneously exploits both *or-parallelism* and *independent and-parallelism*. A high performance implementation of the ACE model has been realized and its performance reported in this paper. We discuss how some of the standard problems which appear when implementing and-parallel systems are solved in ACE. We then propose a number of optimizations aimed at reducing the overheads and the increased memory consumption which occur in such systems when using previously proposed solutions. Finally, we present results from an implementation of ACE which includes the optimizations proposed. The results show that ACE exploits and-parallelism with high efficiency and high speedups. Furthermore, they also show that the proposed optimizations, which are applicable to many other and-parallel systems, significantly decrease memory consumption and increase speedups and absolute performance both in forwards execution and during backtracking.

Keywords: Independent And-parallelism, Or-parallelism, Implementation Issues, Memory Management, Performance Evaluation, Logic Programming.

1 Introduction

1.1 Logic Programming and Prolog

Logic programming is a programming paradigm where programs are expressed as logical rules [35, 8]. Logic programming languages have been shown to be suited to a wide range of applications, from compilers to databases and to symbolic applications, as well as for general purpose programming (see, e.g., [49]). Arguably, the most popular logic programming language nowadays is Prolog. Unlike conventional programming languages, Logic Programming languages disallow destructive assignment and include little explicit control information. Not only this allows cleaner (declarative) semantics for programs, and hence a better understanding of them by their users,

it also makes it easier for an evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can often be executed in any order without affecting the (declarative) meaning of the program.[1] In particular, these operations can be performed by the evaluator in parallel. Furthermore, the cleaner semantics also make logic languages more amenable to automatic compile-time analysis and transformation.

An important characteristic of logic programming languages is that they greatly facilitate exploiting parallelism in an *implicit* way. This can be done directly by the program evaluator, as suggested above, or, alternatively, it can be done by a parallelizing compiler, whose task then is essentially unburdening the evaluator from making run-time decisions regarding when to run in parallel. Finally, of course, the program can be parallelized by the user. In all cases, the advantage offered by logic programming is that the process is easier because of the more declarative nature of the language and its high level, which contribute in preventing the parallelism in the application from being hidden in the coding process. Furthermore, the parallelization process can be done quite successfully in an automatic way, requiring little or no input from the user. Clearly, implicit exploitation of parallelism can in many cases have significant advantages over explicit parallelization.[2] In that sense, Prolog offers a possible path for solving the new form of "(parallel) software crisis" that is posed to arise with the new wider availability of multiprocessors[3]—given systems, such as the one described in this paper, one can run Prolog programs written for sequential machines in parallel with little or no effort. For the rest of the paper we assume that the reader is familiar with Prolog and its execution model.

It must be pointed out that while the preferred target areas of Prolog are Symbolic and AI applications, our system, as any other Prolog system (parallel or not), can also be used for the execution of general purpose programs [49], retaining the advantages in performance of parallel execution. This is borne out from some of the benchmarks we have used in Section 5 of this paper.

1.2 Parallelism in Logic Programming

Three principal kinds of (implicitly exploitable) control parallelism can be identified in logic programs (and, thus, Prolog) [9].

1. *Or-parallelism* arises when more than one clause defines some predicate and a literal unifies with more than one clause head—the corresponding bodies can then be executed in parallel with each other [38, 1]. Or-parallelism is thus a way of efficiently searching for solutions to the query, by exploring alternative solutions in parallel.

2. *Independent and-parallelism* arises when more than one goal is present in the query or in the body of a clause, and it can be determined that these goals do not "affect" each other in the sequential execution–they can then be safely executed (independently) in parallel [16, 28, 36, 24, 27].

[1] Data dependencies or side effects however do pose constraints in the evaluation order.

[2] This does not mean, of course, that a knowledgeable user should be prevented from parallelizing programs manually or even programming sequentially but in a particular way that makes it possible for the system to uncover more parallelism.

[3] For example, affordable (shared memory) multiprocessor workstations are already being marketed by vendors such as Sun (Sun Sparc 10–2000), SGI (Challenge), etc.

3. *Dependent and-parallelism* arises when two or more non-independent goals (in the sense above) are executed in parallel. In this case the shared variables are used as a means of communication. Several proposals and systems adhere to this execution paradigm. Some of them try to retain the Prolog semantics and behavior, either relying on low–level machinery [46] or on a mixture of compile–time techniques and specialized machinery [10]. Other proposals depart from standard Prolog semantics, mainly restricting or disallowing backtracking and using matching instead of general unification [50, 32, 34, 12, 2]. In general, these decisions simplify the architecture of the system.

1.3 ACE: An And-Or Parallel System and Execution Model

The ACE (And-or/parallel Copying-based Execution) model [21, 41] uses stack-copying [1] and recomputation [19] to efficiently support combined or- and independent and-parallel execution. ACE represents an efficient combination of or- and independent and-parallelism in the sense that it strives to pay for the penalties for supporting either form of parallelism only when that form of parallelism is actually exploited. It achieves this by ensuring that, in the presence of only or-parallelism, execution in ACE be essentially the same as in the MUSE [1] system—a stack-copying based purely or-parallel system, while in the presence of only independent and-parallelism, execution be essentially the same as in the &-Prolog [24] system—a recomputation based purely and-parallel system. This efficiency in execution is accomplished by extending the stack-copying techniques of MUSE to deal with an organization of processors into *teams* [10].

It is important to observe that reaching this goal goes far beyond solving a simple engineering problem in combining two existing systems. The experience of ACE showed that the combination of two forms of parallelism leads to question most of the design choices and requires new solutions to previously solved problems (e.g. memory management schemes). This allowed us to get a better insight in the issues to be tackled in implementing general parallel logic programming systems. Some of these fundamental issues are briefly sketched in Section 2.5.

The ACE system is an efficient implementation of the ACE model supporting the full Prolog language, that has been developed at the Laboratory for Logic, Databases, and Advanced Programming of the New Mexico State University, in collaboration with the CLIP group at the Technical University of Madrid, Spain. In this paper we will present briefly how some of the standard problems which appear when implementing and-parallel systems are solved in ACE. We then propose a number of optimizations aimed at reducing the overheads and the increased memory consumption. Finally, we present results from an implementation of the system which includes the optimizations proposed. The results show that ACE exploits and-parallelism with very high efficiency and excellent speedups. These results are comparable and often superior than those presented for other pure and-parallel systems. Furthermore, they also show that the proposed optimizations, which are applicable to many other and-parallel systems, significantly decrease memory consumption and increase speedups and absolute performance both in forwards execution and during backtracking. The ACE implementation belongs to the second generation of and-parallel systems, since it combines the techniques used in older, first generation systems (e.g. the first versions of &-Prolog [24]) with new innovative optimizations to obtain a highly efficient system.

As mentioned before, in this paper we are exclusively concerned with the analysis of the and-parallel component of the ACE system; for further details on the whole ACE system the interested reader is referred to [21].

2 Independent And-parallelism

As pointed out above, the main purpose of this paper is to illustrate the structure, features, and optimizations of the and-parallel engine developed for the ACE system, and evaluate its performance. In this section we explain the computational behavior of the and-parallel engine in more detail.

Much work has been done to date in the context of independent and-parallel execution of Prolog programs. Practical models and systems which exploit this type of parallelism [28, 36, 24, 46] are generally designed for shared memory platforms and based on the "marker model", and on derivations of the RAP-WAM/PWAM abstract machines, originally proposed in [28, 30] and refined in [24, 47, 48]. This model has been shown to be practical through its implementation in the &-Prolog system, which proved capable of obtaining quite good speedups with respect to state of the art sequential systems. Our design of the and-parallel component of ACE is heavily influenced by this model and its implementation in &-Prolog. However, in addition to supporting or-parallelism, ACE also incorporates a significant number of optimizations which considerably reduce the parallel overhead and result in better overall efficiency. These optimizations are fairly general, and are applicable to any and-parallel system whose implementation is based on the markers model. The election of having a shared memory space, in contrast to many other proposals which use distributed memory models, is now supported by the availability of commercial multiprocessors in the market and their relative ease of programming. In addition, shared memory machines offer a model for the implementor which is simpler and more portable than that offered by distributed memory architectures.

2.1 Introduction

As in the RAP-WAM, ACE exploits independent and-parallelism using a recomputation based scheme [19]—no sharing of solutions is performed (at the and-parallel level). This means that for a query like `?- a,b`, where `a` and `b` are nondeterministic, `b` is completely recomputed for every solution of `a` (as in Prolog).

For simplicity and efficiency, we adopt the solution proposed by DeGroot [16] of *restricting* parallelism to a nested *parbegin-parend* structure. This is illustrated in Figure 1 which sketches the structure of the computation tree created in the presence of and-parallel computation with the previously mentioned *parbegin-parend* structure, where the different branches are assigned to different *and-agents* (and-agents are processing agents working in and-parallel with each other). Since and-agents are computing just different parts of the same computation (i.e. they are cooperating in building one solution of the initial query) they need to make available to each other their (partial) solutions. Doing this in a distributed memory machine would need a traffic of data which would impact negatively on performance. This is avoided in a shared memory machine by having *different* but *mutually accessible* logical address spaces. This can be seen in through an example: let us consider the following clause (taken from a program for performing symbolic integration):

$$\texttt{integrate(X + Y,Z)} \leftarrow \texttt{integrate(X,X}_1\texttt{), integrate(Y,Y}_1\texttt{), Z = X}_1 \texttt{ + Y}_1$$

The execution of the two subgoals in the body can be carried out in and-parallel. But at the end of the parallel part, the execution is sequential and it requires access to terms created in the stacks of different and-agents (see figure 2).