

Explicit parallelization of logic programs using low-level threading primitives

Diptikalyan Saha

IBM Research Labs, Delhi, India
diptsaha@in.ibm.com

Paul Fodor

State University of New York at Stony Brook, USA
pfodor@cs.sunysb.edu

Abstract

In this paper, we present a way to parallelize logic programs, using a transformation to low level threading primitives. Although, much work has been done in parallelization of logic programming more than a decade ago (e.g., Aurora, Muse), the current state of parallelizing programs under the well-founded semantics is still very poor. To our knowledge, our new framework surpasses most of the current proposals for parallelization. We present the transformation using merging of answer-tables from multiple children threads and study its behavior for classic standard logic programs. This transformation has better results and more intuitive behavior than the current well-founded model systems that allow parallelization. The transformation and its lower-level answer merging predicates were implemented as an extension to the XSB system.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Logic Programming.

Keywords parallelization, threads, tabling logic programming

1. Introduction

In the last few years, more and more applications are turning towards logic programming as a better way to represent knowledge and a logical semantics for huge amounts of data. Unfortunately, although logic programming works well for medium size programs, it is still lacking the capability to handle huge programs from areas as the program analysis, semantic web, data mining, security policy frameworks, and most researchers agree that we need to understand better parallelization of logic programs.

We believe that the area of logic programming was better understood more than a decade ago [3], when special WAM engines were developed to run on multiple processors (see Aurora [4], Muse [1], and Yap-Or [5]). All these approaches extend the WAM in some way for shared memory architectures: Aurora keeps a trail tree "derivation tree" and uses "binding arrays" in each process to point to nodes in the trail tree, while Muse copies process bindings for each spawn thread. Unfortunately, during those times multiple core processors were extremely expensive and those approaches never made it in large communities or into full-

fledged systems. Lately, there is a higher interest in parallelization using common multi-core processors for answer-set programming. However, the work for well-founded models is still weak and we are aware of only one deductive database which allows parallelization of computation, namely OntoBroker. The improvements are not very high (~10% improvement for the parallel computation compared to its single process computation) due to its bottom-up computation and lower possibility of parallelization.

Multi-Core is one of the current buzz-words around along with related buzz like Cloud Computing. It seems that the era of Moore's law is coming to an end and processor speeds are not going to increase as they way it was increasing in the pre-2002 era. Instead, the future performance gain is going to be exploiting the multiple CPUs available in the single processor by thread level parallelism. The main question here is the effect of this change in computing paradigm to the logic programming model. The WAM (Warren Abstract Machine) emulator and its tabled extensions such as SLG-WAM were designed primarily for a single threaded execution. Question is whether a prolog program can be compiled to a set of set of instructions where each set is going to run in a single CPU, and still preserve the soundness and completeness of tabled execution.

An important question to the programming language community is how to cope and exploit the power of parallel executions. In theory, we don't really need to write our programs along with parallel-programming constructs that the compiler can generate efficient code which exploits parallelism as the user knows what is the intension of the program, or the compiler take the burden of taking any program and generate efficient parallel code from it, after all the compiler knows more about the system and architecture. I believe that it's a new opportunity for declarative community where programmer, by definition, writes down its intent through a declarative language and then the compiler can take the burden of generating efficient code which exploits parallelism in the executing machine.

More and more LP systems support the use of POSIX threads to perform separable computations with a simple and clear API (supported in Yap, SWI, XSB, Mercury). Our work describes parallelization of computing tabled predicates using a transformation into a parallel version which distributes the computation on multiple threads. This transformation for multi-core processors can be extended to a multi-computer MapReduce scheme [2] (i.e., "map" for distributing the data and "reduce" to combine the results). Due to its simple join for the "reduce" operation, such a transformation has a very good performance (tested on multiple computers in our network).