The Journal of Systems and Software 000 (2016) 1–10

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On scaling dynamic programming problems with a multithreaded tabling Prolog system

Miguel Areias*, Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto, Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal

ARTICLE INFO

Article history:
Received 15 May 2015
Revised 4 May 2016
Accepted 23 June 2016
Available online xxx

Keywords:
Dynamic programming
Multithreading
Tabling
Prolog
Scalability

ABSTRACT

Tabling is a powerful implementation technique that improves the declarativeness and expressiveness of traditional Prolog systems in dealing with recursion and redundant computations. It can be viewed as a natural tool to implement dynamic programming problems, where a general recursive strategy divides a problem in simple sub-problems that are often the same. When tabling is combined with multithreading, we have the best of both worlds, since we can exploit the combination of higher declarative semantics with higher procedural control. However, at the engine level, such combination for dynamic programming problems is very difficult to exploit in order to achieve execution scalability as we increase the number of running threads. In this work, we focus on two well-known dynamic programming problems, the Knapsack and the Longest Common Subsequence problems, and we discuss how we were able to scale their execution by using the multithreaded tabling engine of the Yap Prolog system. To the best of our knowledge, this is the first work showing a Prolog system to be able to scale the execution of multithreaded dynamic programming problems. Our experiments also show that our system can achieve comparable or even better speedup results than other parallel implementations of the same problems.

1. Introduction

Dynamic programming (Bellman, 1957) is a general recursive strategy that consists in dividing a problem in simple sub-problems that, often, are the same. The idea behind dynamic programming is to reduce the number of computations: once an answer to a given sub-problem has been computed, it is memorized and the next time the same answer is needed, it is simply looked up. Dynamic programming is especially useful in solving dynamic optimization problems and optimal control problems when the number of overlapping sub-problems grows exponentially as a function of the size of the input. Dynamic programming can be implemented using both *bottom-up* or *top-down* approaches. In bottom-up, it starts from the base sub-problems and recursively computes the next level sub-problems until reaching the answer to the given problem. On the other hand, the top-down approach starts from the given problem and uses recursion to subdivide a problem into sub-problems until reaching the base sub-problems. Answers to previously computed sub-problems are reused rather than being recomputed. An advantage of the top-down approach is that it might not need to compute all possible sub-problems.

However, dynamic programming has some limitations, such as, the *curse of dimensionality* (Bellman, 1957) which might occur in problems with high-dimensional spaces. One possible solution to overcome these limitations is the usage of adaptive dynamic programming (ADP) algorithms that approximate the optimal solution of the cost function in the dynamic programming problem. More recently, Zhang et al. studied the quality of the approximation of ADP algorithms by analyzing multiple factors, such as, their convergence and the execution time horizon (Zhang et al., 2013). In this work, we focus on problems with low-dimensional spaces.

Most of the proposals that can be found in the literature to parallelize dynamic programming problems with low-dimensional spaces follow the parallelization of a sequential bottom-up algorithm. All these proposals are usually based on a careful analysis of the sequential algorithm in order to find the best way to minimize data dependencies in the supporting data structures of memorization, which are often a matrix or an array. The resulting parallelization requires then a synchronization mechanism before recursively computing the next level sub-problems. Alternatively, a generic proposal to parallelize top-down dynamic programming algorithms is Stivala et al.'s work (Stivala et al., 2010), where a set of threads solve the entire dynamic program independently but with a randomized choice of sub-problems. In other words, each thread runs exactly the same function, but a randomized choice of sub-problems results in threads diverging to compute different

* Corresponding author.
E-mail addresses: miguel-areias@dcc.fc.up.pt (M. Areias), ricroc@dcc.fc.up.pt (R. Rocha).

http://dx.doi.org/10.1016/j.jss.2016.06.060
0164-1212/© 2016 Elsevier Inc. All rights reserved.

sub-problems, while reusing the sub-problem's results computed, in the meantime, by the other threads.

Tabling (Chen and Warren, 1996) is a recognized and powerful implementation technique that proved its viability and efficiency to overcome Prolog's susceptibility to infinite loops and redundant computations. Tabling consists of saving and reusing the results of sub-computations during the execution of a program and, for that, the calls and the answers to tabled subgoals are memorized in a proper data structure called the *table space*. Tabling can thus be viewed as a natural tool to implement dynamic programming problems. When tabling is combined with multithreading, we have the best of both worlds, since we can exploit the combination of higher declarative semantics with higher procedural control. However, such combination for dynamic programming problems is very difficult to exploit in order to achieve execution scalability as we increase the number of running threads. To the best of our knowledge, XSB (Marques and Swift, 2008) and Yap (Areias and Rocha, 2012b) are the only Prolog systems that support the combination of multithreading with tabling, but none of them showed until now to be able to scale the execution of multithreaded dynamic programming problems. This is a difficult task since we need to combine the explicit thread control required to launch, assign and schedule tasks to threads, with the built-in tabling evaluation mechanism, which is implicit and cannot be controlled by the user.

In this work, we focus on two well-known dynamic programming problems, the Knapsack and the Longest Common Subsequence (LCS) problems, and we discuss how we were able to scale their execution by taking advantage of the multithreaded tabling engine of the Yap Prolog system. For each problem, we present a multithreaded tabled top-down and bottom-up approach. For the top-down approach, we use Yap's mode-directed tabling support (Santos and Rocha, 2013) that allows to aggregate answers by specifying pre-defined modes such as *min* or *max*. For the bottom-up approach, we use Yap's standard tabling support (Santos Costa et al., 2012). To the best of our knowledge, no previous Prolog system showed to be able to scale the execution of multithreaded dynamic programming problems.

A key contribution of this work is our new asynchronous version of the table space data structures, where threads view their tables as private but are able to use the answers of a sub-problem, if another thread has already computed them. By sharing only completed tables, we avoid the problem of dealing with concurrent updates to the table space and, more importantly, the problem of dealing with concurrent deletes, as in the case of using mode-directed tabling.

Our experiments on a 32-core AMD machine show that using Yap's simple and efficient multithreaded table space design, we were able to scale the execution of both knapsack and LCS problems for both top-down and bottom-up approaches. To put our experiments in perspective, we compare our results with other systems and, in particular, we experimented with the state-of-the-art XSB Prolog system (Marques and Swift, 2008). In general, Yap's speedup results are comparable and sometimes better than other parallel implementations of the same problems. Regarding the particular comparison with XSB, Yap's results clearly outperform those of XSB for the execution time and for the speedups.

The remainder of the paper is organized as follows. First, we describe some background about Yap's standard, mode-directed and multithreaded tabling support and discuss XSB's approach to multithreaded tabling. Next, for both Knapsack and LCS problems, we introduce the problem and present in detail our parallel implementations using either a top-down and bottom-up dynamic programming approach. Then, we present a set of experiments and discuss the results. At the end, we discuss related work and outline some conclusions and further work.

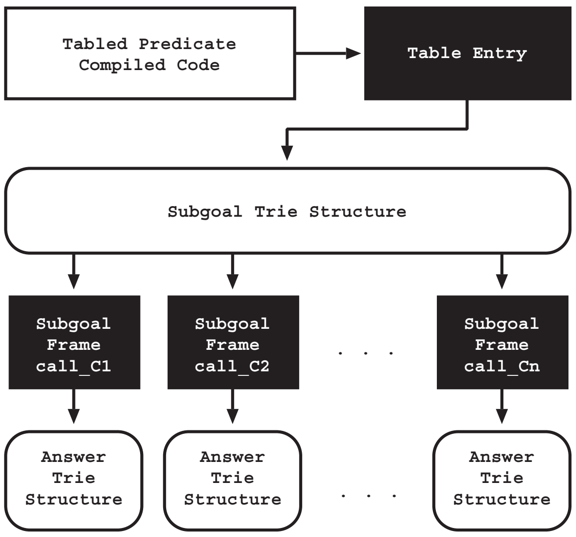

Fig. 1. Yap's table space organization.

2. Background

This section introduces some background needed for the following sections.

2.1. Standard tabling

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Variant calls[1] to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all variant calls.

With these requirements, the design of the table space is critical to achieve an efficient implementation. Yap uses *tries* which is regarded as a very efficient way to implement the table space (Ramakrishnan et al., 1999). Tries are trees in which common prefixes are represented only once. The trie data structure provides complete discrimination for terms and permits look up and possibly insertion to be performed in a single pass through a term, hence resulting in a very efficient and compact data structure for term representation. Fig. 1 shows the general table space organization for a tabled predicate in Yap.

At the entry point we have the *table entry* data structure. This structure is allocated when a tabled predicate is being compiled, so that a pointer to the table entry can be included in its compiled code. This guarantees that further calls to the predicate will access the table space starting from the same point. Below the table entry, we have the *subgoal trie structure*. Each different tabled subgoal call to the predicate at hand corresponds to a unique path through the subgoal trie structure, always starting from the table entry, passing by several subgoal trie data units, the *subgoal trie nodes*, and reaching a leaf data structure, the *subgoal frame*. The subgoal frame stores additional information about the subgoal and acts like an entry point to the *answer trie structure*. Each unique path through the answer trie data units, the *answer trie nodes*, corresponds to a different tabled answer to the entry subgoal.

[1] Two terms are considered to be variant [of each other, i.e., are equivalent] if they are the same up to variable renaming.