On Scaling Dynamic Programming Problems with a Multithreaded Tabling Prolog System

Miguel Areias and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
{miguel-areias, ricroc}@dcc.fc.up.pt

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.

Keywords: Dynamic Programming, Multithreading, Tabling, Prolog, Scalability

1. Introduction

Dynamic programming [1] 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* [1] 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 [2]. 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 [3], 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 sub-problems, while reusing the sub-problem's results computed, in the meantime, by the other threads.

Tabling [4] 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 [5] and Yap [6] 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 [7] 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 [8]. 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 [5]. 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.

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 [9]. 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. Figure 1 shows the general table space organization for a tabled predicate in Yap.

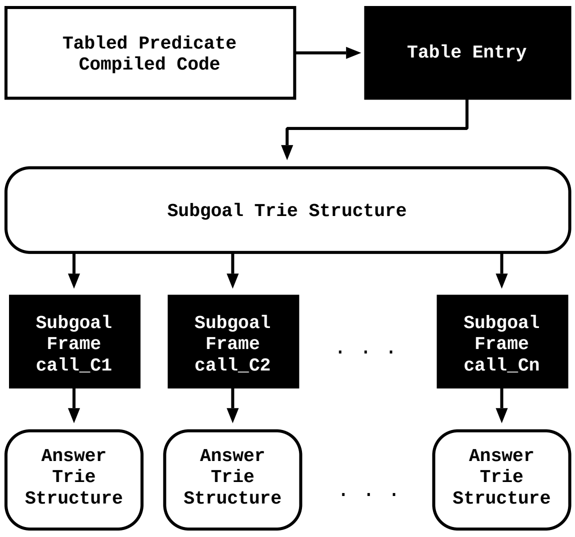

Figure 1: Yap's table space organization

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

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.

2.2. Mode-Directed Tabling

In a traditional tabling system, all the arguments of a tabled subgoal call are considered when storing answers into the table space. When a new answer is not a variant of any answer that is already in the table space, then it is always considered for insertion. Therefore, traditional tabling is very good for problems that require storing all answers. However, with dynamic programming, usually, the goal is to dynamically calculate optimal or selective answers as new results arrive. Writing dynamic programming algorithms can thus be a difficult task without further support.

Mode-directed tabling is an extension to the tabling technique that supports the definition of *modes* for specifying how answers are inserted into the table space. Within mode-directed tabling, tabled predicates are declared using statements of the form '*table* $p(m_1, ..., m_n)$', where the m_i's are *mode operators* for the arguments. The idea is to define the arguments to be considered for variant checking (the index arguments) and how variant answers should be tabled regarding the remaining arguments (the output arguments). In Yap, index arguments are represented with mode *index*, while arguments with modes *first*, *last*, *min*, *max*, *sum* and *all* represent output arguments [7]. After an answer is generated, the system tables the answer only if it is *preferable*, accordingly to the meaning of the output arguments, than some existing variant answer.

2.3. *XSB's Approach to Multithreaded Tabling*

The XSB system supports two types of models for the combination of multithreading with tabling: *private tables* and *shared tables* [5, 10]. On the private tables model, each thread keeps its own copy of the table space. On one hand, this avoids concurrency over the tables but, on the other hand, the same table entry can be repeatedly computed by several threads, thus increasing the memory usage necessary to represent the table space. Moreover, since no information is shared between threads, a thread cannot reuse the results being tabled by another thread to reduce execution time in a parallel environment.

For shared tables, the running threads store only once the same table entry, even if multiple threads use it. This model can be viewed as a variation of the *table-parallelism* proposal [11], where a tabled computation can be decomposed into a set of smaller sub-computations, each being performed by a different thread. Each sub-computation is computed independently by the first thread calling it, the *generator thread*, and each generator is the sole responsible for fully exploiting and obtaining the complete set of answers for a sub-computation. Variant sub-computations by other threads are resolved by consuming the answers stored by the generator thread. When a set of sub-computations being computed by different threads is mutually dependent, then a *usurpation operation* [12] synchronizes threads and a single thread assumes the computation of all sub-computations, turning the remaining threads into consumer threads. This maintains the correctness of the table space but has a major disadvantage: it restricts the potential of concurrency to non-mutually dependent sub-computations. As our experiments will show, this severely constrains the goal of scaling multithreaded dynamic programming problems.

Concerning the single threaded version, XSB does not support mode-directed tabling, but instead supports two *answer subsumption* mechanisms [13] that can reproduce the same behavior. Since the main focus of this work is the multithreaded tabling environment, we will use XSB without support for answer subsumption in the performance analysis section.

3. Our Approach to Multithreaded Tabling

Yap implements a SWI-Prolog compatible multithreading library [14]. Like in SWI-Prolog, Yap's threads have their own execution stacks and only share the code area where predicates, records, flags and other global non-backtrackable data are stored. For tabled evaluation, a thread views its tables as private but, at the engine level, we use a common table space. From the thread point of view, the tables are private but, from the implementation point of view, the tables are shared among all threads.

In previous work [6], we have proposed three designs for the common table space. This work uses the *Subgoal Sharing (SS)* design. In the SS design, the subgoal trie structures are shared among all threads and the leaf data structures representing each tabled subgoal call C_i, instead of pointing to a single subgoal frame, point to a list of private subgoal frames, one per thread that is evaluating the call C_i. The answers for call C_i for each thread are then also stored in an answer trie structure, but private to each thread. As a consequence, no sharing of answers between threads is done.

In this work, we propose a new asynchronous version of the SS design, 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. The idea is as follows. Whenever a thread calls a new tabled subgoal, first it searches the table space to lookup if any other thread has already computed the answers for that call. If so, then the thread reuses the available answers. Otherwise, it computes the subgoal call itself in a private fashion. Several threads can work on the same subgoal simultaneously, i.e., we do not protect a subgoal from further evaluation while other threads have picked it up already. The first thread completing a computation, shares the results by making them available (public) to the other threads.

Figure 2 illustrates the table data structures for the subgoal sharing design with shared answers. Threads can concurrently access the subgoal trie structures, for both read and write operations, and the answer trie structures, but only for reading (black data structures in Fig. 2). Concurrent updates to the subgoal trie structures are implemented through a simpler and efficient lock-free design based on hash tries that minimizes the problems of false sharing and cache memory effects by dispersing the concurrent areas as much as possible [15]. Lock-freedom is implemented by using the CAS atomic instruction that nowadays is widely found on many common architectures.

All subgoal frames and answer tries are initially

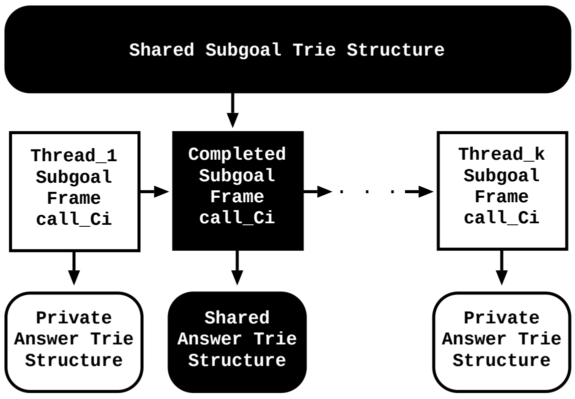
Shared Subgoal Trie Structure
Thread_1 Subgoal Frame call_Ci
Completed Subgoal Frame call_Ci
Thread_k Subgoal Frame call_Ci
Private Answer Trie Structure
Shared Answer Trie Structure
Private Answer Trie Structure

Figure 2: Subgoal sharing design with shared answers

private to a thread. Later, when the first subgoal frame is completed, i.e., when we have found the full set of answers for it, it is marked as completed and put in the beginning of the list of private subgoal frames (configuration shown in Fig. 2). Following calls made by other threads to this subgoal call simply consume the answers from the completed subgoal frame, thus avoiding recomputing the subgoal call at hand. By sharing only completed answer tries, we avoid the following problems: (i) dealing with concurrent updates to the answer tries; (ii) managing the different set of answers that each thread has found; (iii) dealing with concurrent deletes, as in the case of using mode-directed tabling.

4. 0-1 Knapsack Problem

The Knapsack problem [16] is a well-known problem in combinatorial optimization that can be found in many domains such as logistics, manufacturing, finance or telecommunications. Given a set of items, each with a weight and a profit, the goal is to determine the number of items of each kind to include in a collection so that the total weight is equal or less than a given capacity and the total profit is as much as possible. In this paper, for the sake of simplicity, we will use a special case of the *0-1 Knapsack problem* with integer weights and profits. The *0-1 Knapsack problem* restricts the number of copies of each kind of item to be zero or one. We formulate the problem as follows. Given a set of items $i \in \{1, ..., n\}$, each with a weight $w_i \in \mathbb{N}_0^+$ and a profit $p_i \in \mathbb{N}_0^+$, and a Knapsack with capacity $C \in \mathbb{N}_0^+$, the following formulas define the Knapsack problem (KS) and the restriction (KS_R) that avoids any trivial solution, by insuring that each item fits into the Knapsack and that the total weight of all items exceeds the Knapsack capacity.

$$KS = \begin{cases} max \sum_{i=1}^{n} p_i.x_i, \\ \text{s.t.} \sum_{i=1}^{n} w_i.x_i \leq C, \\ \quad x_i \in \{0,1\}, i \in \{1, ..., n\}. \end{cases}$$

$$KS_R = \begin{cases} \forall_i \in \{1, ..., n\}, w_i \leq C, \\ \sum_{i=1}^{n} w_i > C. \end{cases}$$

4.1. Top-Down Approach

We first introduce a standard top-down approach that solves the Knapsack problem using mode-directed tabling. Figure 3 shows our Yap's implementation adapted from [17] to include the dimension of profitability.

```
% table declaration
:- table ks(index, index, max).
% base case
ks(0, C, 0).
% exclude case
ks(I, C, P) :-
  I > 0, ks_exc(I, C, P, 1).
% include case
ks(I, C, P) :-
  I > 0, ks_inc(I, C, P, 1).
% exclude N items starting from I
ks_exc(I, C, P, N) :-
  J is I - N, ks(J, C, P).
% include I and
% exclude the next N-1 items
ks_inc(I, C, P, N) :-
  item(I, Ci, Pi), Cj is C - Ci,
  Cj >= 0, J is I - N,
  ks(J, Cj, Pj), P is Pi + Pj.
```

Figure 3: A top-down approach for the Knapsack problem with mode-directed tabling

The table directive declares that predicate *ks* with arity 3 (or *ks*/3 for short) is to be tabled using modes (*index*, *index*, *max*), meaning that the third argument (the profit) should store only the maximal answers for the first two arguments (the index of the number of items being considered and Knapsack's capacity). The remaining part of the program implements a recursive top-down definition of the Knapsack problem. The first clause is the base case and defines that the empty set is a solution with profit 0. The second clause excludes the current item from the solution set and the third includes the current item in the solution if its inclusion does not overcome the current capacity of the Knapsack. For simplicity of integration with the parallel approach presented next, we are already using two auxiliary predicates, *ks_exc*/4 and *ks_inc*/4, as a way to implement the exclude and include cases. These auxiliary predicates take an extra argument N (fourth argument) that represents the number of items to jump (or exclude) in the recursion procedure. Here, for the sequential version of the problem, N is always 1, i.e, we always move to the next item.

To parallelize top-down dynamic programming algorithms, we followed Stivala et al.'s work [3] where a set of threads solve the entire program independently but with a randomized choice of the sub-problems. For the Knapsack problem, we have two sub-problems, the exclude and include cases. We can thus consider two alternative execution choices at each step: (i) exclude first and include next (as in the sequential version presented in Fig. 3), or (ii) include first and exclude next. The randomized choice of sub-problems results in the threads diverging to compute different sub-problems simultaneously while reusing the sub-problems' results computed in the meantime by the other threads. Since the number of overlapping sub-problems is usually high in these kind of problems, it is expected that the available set of sub-problems to be computed will be evenly divided by the number of available threads resulting in less computation time required to reach the final result.

For the parallel version of the Knapsack problem, we have implemented two alternative versions. The first version simply follows Stivala et al.'s original random approach. The second version extends the first one with an extra step where the computation is first moved forward using a random displacement of the number of items to be excluded and only then the computation is performed for the next item, as usual. By doing this, it is expected that the sub-problems closer to the base cases are computed earlier, meaning that their subgoal frames are also marked as completed earlier, which avoids recomputation when other threads call the same sub-problems. Figure 4 shows the implementation. The difference between the two versions is that the first version does not consider the first extra clause in the *aux_exc*/4 and *aux_inc*/4 auxiliary predicates.

```
% table declaration
:- table ks(index, index, max).
% base case
ks(0, C, 0).
% random choice
ks(I, C, P) :-
  I > 0, random(2, maxRandom, N),
  R is N mod 2,
  ( R = = 0 ->
    aux_exc(I, C, P, N)
  ;
    aux_inc(I, C, P, N)).
% try exclude first and include next
aux_exc(I, C, P, N) :-
  ks_exc(I, C, P, N).
aux_exc(I, C, P, _) :-
  ks_exc(I, C, P, 1).
aux_exc(I, C, P, _) :-
  ks_inc(I, C, P, 1).
% try include first and exclude next
aux_inc(I, C, P, N) :-
  ks_inc(I, C, P, N).
aux_inc(I, C, P, _) :-
  ks_inc(I, C, P, 1).
aux_inc(I, C, P, _) :-
  ks_exc(I, C, P, 1).
```

Figure 4: A top-down parallel version of the Knapsack problem with mode-directed tabling

4.2. Bottom-Up Approach

A straightforward method to solve the Knapsack problem bottom-up, for a fixed capacity c, is to consider all 2^n possible subsets of the n items and choose the one that maximizes the profit. The recursive application of this algorithm to increasing capacities $c \in \{1, ..., C\}$, yields a Knapsack of maximum profit for the given capacity C [18]. The bottom-up characteristic comes from the fact that, given a Knapsack with capacity c and using i items, $i < n$, the decision to include the next item j, $j = i+1$, leads to two situations: (i) if j is not included, the Knapsack profit is unchanged; (ii) if j is included, the profit is the result of the maximum profit of the Knapsack with the same i items but with capacity $c - w_j$ (the capacity needed to include the weight w_j of item j) increased by p_j (the profit of the item j being included). The algorithm then decides whether or not to include an item based on which choice leads to maximum profit. Figure 5 shows the $KS[n, C]$ matrix. The rows define the items and the columns define the Knapsack capacities. The first column and row are filled with zeros, which are the initial profit for the Knapsacks with no items or no capacity.

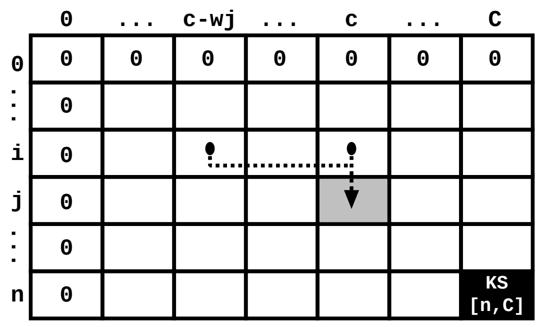

Figure 5: Knapsack bottom-up matrix

The sequential version of the algorithm can be constructed row by row or column by column. The computation of each sub-problem $KS[j, c]$ considers the maximum profitability obtained between $KS[j-1, c]$ and $KS[j-1, c-w_{j-1}]+p_j$. When all sub-problems are computed, $KS[n, C]$ holds the best profitability for the full problem. Figure 6 shows Yap's implementation. For simplicity of presentation, we are omitting the predicate that implements the main loop used to recursively traverse the matrix and launch the computation for each sub-problem.

The table directive declares that predicate *ks*/3 is to be tabled using standard tabling. Since here a sub-problem can be computed from the results of its sub-problems, standard tabling is enough and there is no need for mode-directed tabling. The first two clauses of *ks*/3 are the base cases and define that the Knapsacks with no items or no capacity have profit 0. The third clause deals with the cases where an item's weight exceeds the Knapsack capacity and the fourth clause is the one that implements the main case discussed above.

Filling cells in subsequent rows requires accessing two cells from the previous row: one from the same column and one from the column offset by the weight of the current item. Thus, computing a row i depends only on the sub-problems at row $i-1$. A possible parallelization is, for each row,

```
% table declaration
:- table ks/3.
% base cases
ks(0, C, 0).
ks(I, 0, 0).
% item I exceeds capacity C
ks(I, C, P) :-
  I > 0, item(I, Ci, Pi), Ci > C,
  J is I - 1, ks(J, C, P).
% item I fits in capacity C
ks(I, C, P) :-
  I > 0, item(I, Ci, Pi), Ci =< C,
  Cj is C - Ci, Cj >= 0, J is I - 1,
  ks(J, Cj, Pj), P1 is Pj + Pi,
  ks(J, C, P2), max(P1, P2, P).
```

Figure 6: A bottom-up approach for the Knapsack problem with standard tabling

to divide the computation of the C columns between the available threads and then wait for all threads to complete in order to synchronize before computing the next row.

Here, since we want to take advantage of the built-in tabling mechanism, which is implicit and cannot be controlled by the user, we want to avoid this kind of synchronization between iterations. Hence, when a sub-problem in the previous row was not computed yet (i.e., marked as completed in one of the subgoal frames for the given call), instead of waiting for the corresponding result to be computed by another thread, the current thread starts also its computation and for that it can recursively call many other sub-problems not computed yet. Despite this can lead to redundant sub-computations, it avoids synchronization. In fact, as we will see, this strategy showed to be very effective.

We next introduce our generic multithreaded scheduler used to load balancing the access to a set of concurrent tasks. We assume that the number of tasks is known before execution starts and that tasks are numbered incrementally starting at 1. For the Knapsack problem, we will consider that the number of tasks is the number of capacities $c \in \{1, ..., C\}$ (alternatively, we could have considered the number of items $i \in \{1, ..., n\}$). In a nutshell, the scheduler uses a user-level *mutex* to protect a concurrent *queue* that stores the indices of the available tasks. In fact, since tasks are numbered incrementally, the queue simply needs to store the index of the next available task. When a thread gets access to the queue of tasks, it picks a chunk of consecutive tasks and updates the queue's stored index accordingly. Figure 7 shows the Prolog code that implements the main execution loop of each thread.

```
% initialize mutex
:- mutex_create(queueLock).
% initialize queue
:- set_value(queueIndex, 0).
% main execution loop
do_work(NumberOfTasks, ChunkSize) :-
  mutex_lock(queueLock),
  get_value(queueIndex, Current),
  ( Current = NumberOfTasks ->
    % terminate execution
    mutex_unlock(queueLock)
  ;
    First is Current + 1,
    Last is Current + ChunkSize,
    set_value(queueIndex, Last),
    mutex_unlock(queueLock),
    compute_tasks(First, Last),
    % get more work
    do_work(NumberOfTasks, ChunkSize)
  ).
```

Figure 7: The generic execution loop of each thread for the bottom-up approach

The top declarations initialize the *queueLock* mutex and the *queueIndex* queue. The predicate *do_work*/2 implements the main execution loop of each thread and is recursively executed until no more tasks exist in the queue. It receives two arguments: the total number of tasks in the problem (*NumberOfTasks*); and the chunk size to be considered when retrieving tasks from the queue (*ChunkSize*). In each loop, a thread starts by gaining access to the mutex and then it checks the queue. If the queue is empty, case in which the test *Current* = *NumberOfTasks* succeeds[2], the mutex is released and the thread terminates execution. Otherwise, the thread picks a new chunk of consecutive tasks and updates the queue's stored index accordingly. Variables *First* and *Last* define the lower and upper bounds of the chunk of tasks obtained. The tasks are then evaluated using the *compute_tasks*/2 predicate, which calls the *ks*/3 predicate for the set of Knapsack sub-problems associated with the task. After the *compute_tasks*/2 finishes, the *do_work*/2 predicate is called again to get more tasks from the queue. The process repeats until no more tasks exist.

[2]In order to avoid low-level details which are not relevant to this work, the reader can assume that *NumberOfTasks* is a multiple of *ChunkSize*.

5. Longest Common Subsequence Problem

The problem of computing the length of the Longest Common Subsequence (LCS) is representative of a class of dynamic programming algorithms for string comparison that are based on getting a similarity degree. A good example is the sequence alignment, which is a fundamental technique for biologists to investigate the similarity between species. The LCS problem can be defined as follows. Given a finite set of symbols S and two sequences $U = \langle u_1, u_2, ..., u_n \rangle$ and $V = \langle v_1, v_2, ..., v_m \rangle$ such that $\forall_{i \in 1,...,n}, u_i \in S$ and $\forall_{i \in 1,...,m}, v_i \in S$, we say that U has a common subsequence with V of length k if there are indices $i_1, i_2, ..., i_k, j_1, j_2, ..., j_k : 1 \leq i_1 < i_2 < ... < i_k \leq n$ and $1 \leq j_1 < j_2 < ... < j_k \leq m$ such that $\forall_{l \in 1,...,k}, u_{i_l} = v_{j_l}$. The length k is considered to be the longest common subsequence if it is maximal.

5.1. Top-Down Approach

We next introduce a standard top-down approach that solves the LCS problem using mode-directed tabling. Figure 8 shows Yap's implementation adapted from [17].

The first two clauses of *lcs*/3 are the base cases defining that for empty sequences the LCS (third argument) is 0. The third clause deals with the cases where the current symbols in both sequences match (arguments I_u and I_v represent, respectively, the current indices in sequences U and V to be considered). The fourth and fifth clauses represent the opposite case, where the symbols do not match, and each clause moves one of the sequences to the next symbol (note that recursion is done in descending order until reaching index 0). Again, for simplicity of integration with the parallel approach presented next, we are already using two auxiliary predicates, *lcs_u*/4 and *lcs_v*/4, as a way to implement the unmatched cases. As for the Knapsack problem, these two auxiliary predicates take an extra argument N (fourth argument) that represents the number of symbols to jump in the recursion

```
% table declaration
:- table lcs(index, index, max).
% base cases
lcs(I, 0, 0).
lcs(0, I, 0).
% matched case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  symbol_u(Iu, S), symbol_v(Iv, S),
  Ju is Iu - 1, Jv is Iv - 1,
  lcs(Ju, Jv, Lj), L is Lj + 1.
% sequence U case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  lcs_u(Iu, Iv, L, 1).
% sequence V case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  lcs_v(Iu, Iv, L, 1).
% jump N symbols in sequence U
lcs_u(Iu, Iv, L, N) :-
  symbol_u(Iu, Su), symbol_v(Iv, Sv),
  Su =\= Sv,  Ju is Iu - N,
  lcs(Ju, Iv, L).
% jump N symbols in sequence V
lcs_v(Iu, Iv, L, N) :-
  symbol_u(Iu, Su), symbol_v(Iv, Sv),
  Su =\= Sv, Jv is Iv - N,
  lcs(Iu, Jv, L).
```

Figure 8: A top-down approach for the LCS problem with mode-directed tabling

procedure. For the sequential version of the problem, N is always 1, meaning that we always move to the next symbol.

Similarly to Knapsack's problem, to parallelize the LCS sequential top-down approach, we have implemented two alternative versions. The first version follows Stivala et al.'s original random approach. The second version extends the first one with an extra step where the computation is first moved forward using a random displacement of the number of symbols to jump and only then the computation is performed for the next symbol, as usual. Figure 9 shows the implementation. The difference between the two versions is that the first version does not consider the first extra clause in the *aux_u*/4 and *aux_v*/4 auxiliary predicates.

```
% table declaration
:- table lcs(index, index, max).
% base cases
lcs(I, 0, 0).
lcs(0, I, 0).
% matched case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  symbol_u(Iu, S), symbol_v(Iv, S),
  Ju is Iu - 1, Jv is Iv - 1,
  lcs(Ju, Jv, Lj), L is Lj + 1.
% random choice
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  random(2, maxRandom, N),
  R is N mod 2,
  ( R = = 0 ->
    aux_u(Iu, Iv, L, N)
  ;
    aux_v(Iu, Iv, L, N)).
% try sequence U first and V next
aux_u(Iu, Iv, L, N) :-
  lcs_u(Iu, Iv, L, N).
aux_u(Iu, Iv, L, _) :-
  lcs_u(Iu, Iv, L, 1).
aux_u(Iu, Iv, L, _) :-
  lcs_v(Iu, Iv, L, 1).
% try sequence V first and U next
aux_v(Iu, Iv, L, N) :-
  lcs_v(Iu, Iv, L, N).
aux_v(Iu, Iv, L, _) :-
  lcs_v(Iu, Iv, L, 1).
aux_v(Iu, Iv, L, _) :-
  lcs_u(Iu, Iv, L, 1).
```

Figure 9: A top-down parallel version of the LCS problem with mode-directed tabling

5.2. Bottom-Up Approach

We now introduce our bottom-up approach to the LCS problem, which is based on [18]. In a nutshell, the bottom-up characteristic comes from the fact that, the maximum length of a common subsequence between two sequences U and V is: (i) if the initial symbols of both sequences match, then they are part of the longest common subsequence and the length of the longest common subsequence can be incremented by one; (ii) if the initial symbols do not match then two situations arise: the longest common subsequence may be obtained from U and V without the initial symbol or from V and

U without the initial symbol. Since we want the longest subsequence, the maximum of these two must be selected. The following formula formalizes the LCS problem as described above:

$$LCS[j,l] = \begin{cases} LCS[j-1,l-1]+1, \\ \quad \text{if } u_j = v_l. \\ max\,\{LCS[j,l-1], LCS[j-1,l]\}, \\ \quad \text{otherwise.} \end{cases}$$

Figure 10 shows the LCS matrix that represents the bottom-up approach. The rows define the indices to be considered in sequence U and the columns define the indices in sequence V. The first column and the first row are filled with zeros, meaning that for empty sequences the LCS is 0. The sequential version of the algorithm can be constructed row by row or column by column, since the computation of each sub-problem $LCS[j,l]$ only depends on the sub-computations done for the preceding row and column. At the end, $LCS[n,m]$ holds the LCS for the problem.

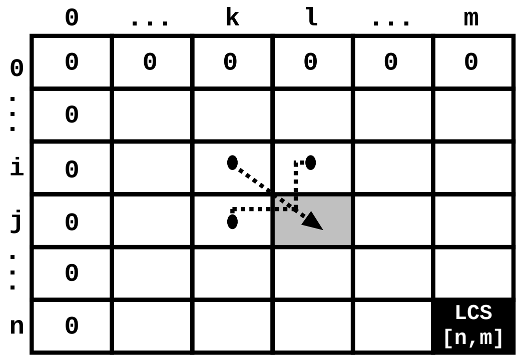

Figure 10: LCS bottom-up matrix

Figure 11 shows Yap's implementation. Again, for simplicity of presentation, we are omitting the predicate that implements the main loop used to recursively traverse the matrix and launch the computation for each sub-problem.

The table directive declares that predicate *lcs*/3 is to be tabled using standard tabling. The first two clauses of *lcs*/3 are the base cases and the third and fourth clauses deal with the cases where the initial symbols of both sequences match and do not match, respectively.

Concerning the parallelization of the matrix, a possible approach is, for each row, divide the computation of the m columns between the available threads or, for each column, divide the computation of the n rows between the available threads. Here, we will follow the same approach as for the Knapsack problem and we will use the generic multithreaded scheduler that implements the thread execution loop presented in Fig. 7. The number of concurrent tasks to be considered is the size of sequence U (alternatively, we could have considered the size of sequence V) and the evaluation of the *compute_tasks*/2 predicate calls the *lcs*/3 predicate for the set of LCS sub-problems associated with a given task.

```
% table declaration
:- table lcs/3.
% base cases
lcs(I, 0, 0).
lcs(0, I, 0).
% matched case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  symbol_u(Iu, S), symbol_v(Iv, S),
  Ju is Iu - 1, Jv is Iv - 1,
  lcs(Ju, Jv, Lj), L is Lj + 1.
% unmatched case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  symbol_u(Iu, Su), symbol_v(Iv, Sv),
  Su =\= Sv,
  Ju is Iu - 1, lcs(Ju, Iv, L1),
  Jv is Iv - 1, lcs(Iu, Jv, L2),
  max(L1, L2, L).
```

Figure 11: A bottom-up approach for the LCS problem with standard tabling

6. Performance Analysis

The environment for our experiments was a machine with 32-core AMD Opteron (tm) Processor 6274 @ 2.2 GHz with 32 GBytes of main memory and running the Linux kernel 3.16.7-200.fc20.x86_64 with TcMalloc 4.2 [19]. We used Yap Prolog, version 6.3.2, with the SS design and the memory allocator from [20]. To put our results in perspective, we also experimented with XSB Prolog version 3.4.0, using the shared tables model [5]. To verify the correctness of the experiments, we have confirmed that the final results of all benchmarks and approaches were correct on both Prolog systems.

Table 1: Execution time, in milliseconds, for one thread (sequential and multithreaded version) and corresponding speedup (against one thread running the multithreaded version) for the execution with 8, 16, 24 and 32 threads, for the top-down and bottom-up approaches of the *Knapsack* problem using the YAP and XSB Prolog systems

System/Dataset		**Seq. Time (T_{seq})**	**# Threads (p)** Time (T_1) 1	Speedup (T_1/T_p) 8	16	24	32	**Best Time (T_{best})**
Top-Down Approaches								
	D_{10}	**9,508**	12,415	n.c.	n.c.	n.c.	n.c.	9,508
YAP$_{TD_0}$	D_{30}	**9,246**	12,177	n.c.	n.c.	n.c.	n.c.	9,246
	D_{50}	**9,480**	12,589	n.c.	n.c.	n.c.	n.c.	9,480
	D_{10}	14,330	19,316	1.96	**2.12**	2.04	1.95	9,115
YAP$_{TD_1}$	D_{30}	14,725	19,332	3.57	**4.17**	4.06	3.93	4,639
	D_{50}	14,729	18,857	4.74	6.28	**6.44**	6.41	2,930
	D_{10}	19,667	24,444	6.78	12.35	15.44	**18.19**	1,344
YAP$_{TD_2}$	D_{30}	19,847	25,609	7.15	13.83	17.37	**20.47**	1,251
	D_{50}	19,985	25,429	7.27	13.70	17.35	**20.62**	1,233
Bottom-Up Approaches								
	D_{10}	12,614	17,940	7.17	13.97	18.31	**22.15**	810
YAP$_{BU}$	D_{30}	12,364	17,856	7.23	13.78	18.26	**21.94**	814
	D_{50}	12,653	17,499	7.25	14.01	18.34	**21.76**	804
	D_{10}	**32,297**	38,965	0.87	0.66	0.62	0.55	32,297
XSB$_{BU}$	D_{30}	**32,063**	38,007	0.86	0.61	0.56	0.53	32,063
	D_{50}	**31,893**	38,534	0.84	0.58	0.57	0.57	31,893

For the Knapsack problem, we fixed the number of items and capacity, respectively, $1,600$ and $3,200$. For the LCS problem, we used both sequences with a fixed size of $3,200$ symbols each. Then, for each problem, we created three different datasets, D_{10}, D_{30} and D_{50}, meaning that the values for the weights/profits for the Knapsack problem and the symbols for LCS problem were randomly generated in an interval between 1 and 10%, 30% and 50% of the total number of items/symbols, respectively. For the top-down approaches, we only experimented with Yap since XSB does not support mode-directed tabling. For Yap, we tested both problems without randomization (YAP_{TD_0}), with randomization using Stivala et al.'s original version (YAP_{TD_1}) and with the extended version using the extra random displacement clause (YAP_{TD_2}). For both Knapsack and LCS problems, we used a *maxRandom* value corresponding to 10% of the total number of items/symbols in the problem. For the bottom-up approaches, we experimented with Yap (YAP_{BU}) and XSB (XSB_{BU}) and we used a *ChunkSize* value of 5.

Table 1 and Table 2 show the average of 10 runs results obtained, respectively, for the Knapsack and LCS problems for both top-down and bottom-up approaches using the YAP and XSB Prolog systems. The columns in both tables show the following information. The first column describes the configurations of approaches and datasets used. The second column (T_{seq}) shows the sequential execution time in milliseconds. For T_{seq}, the Prolog systems where compiled without multithreaded support and ran without multithreaded code. The next five columns show the execution time for one thread (T_1) and the corresponding speedup for the execution with 8, 16, 24 and 32 threads (columns T_1/T_p). For each system/dataset configuration, the results in bold highlight the column where the best execution time was obtained and the last column (T_{best}) presents such result in milliseconds.

Analyzing the general picture of both tables, one can observe that the YAP_{TD_2} top-down and YAP_{BU} bottom-up approaches have the best results with excellent speedups for 8, 16, 24 and 32 threads. In particular, with 32 threads, they obtain speedups around 21 and 20, respectively, for the Knapsack and LCS problems. Column T_{best} shows that the YAP_{BU} approach running with 32 threads obtains also the best execution times of all systems in all datasets. The results for the YAP_{TD_1} top-down approach are not so interesting, regardless of the fact

Table 2: Execution time, in milliseconds, for one thread (sequential and multithreaded version) and corresponding speedup (against one thread running the multithreaded version) for the execution with 8, 16, 24 and 32 threads, for the top-down and bottom-up approaches of the *LCS* problem using the YAP and XSB Prolog systems

System/Dataset		**Seq. Time (T_{seq})**	**# Threads (p)**					**Best Time (T_{best})**
			Time (T_1)	**Speedup (T_1/T_p)**				
			1	**8**	**16**	**24**	**32**	
Top-Down Approaches								
	D_{10}	**21,191**	26,225	n.c.	n.c.	n.c.	n.c.	21,191
YAP_{TD_0}	**D_{30}**	**20,809**	26,146	n.c.	n.c.	n.c.	n.c.	20,809
	D_{50}	**20,775**	26,028	n.c.	n.c.	n.c.	n.c.	20,775
	D_{10}	26,030	33,969	**1.58**	1.53	1.50	1.42	21,509
YAP_{TD_1}	**D_{30}**	26,523	34,213	**1.60**	1.54	1.50	1.42	21,424
	D_{50}	26,545	34,234	**1.60**	1.54	1.51	1.40	21,408
	D_{10}	34,565	44,371	7.23	13.23	16.45	**19.74**	2,248
YAP_{TD_2}	**D_{30}**	34,284	44,191	7.12	13.09	16.52	**19.77**	2,235
	D_{50}	33,989	44,158	7.06	13.30	16.49	**19.58**	2,255
Bottom-Up Approaches								
	D_{10}	20,799	28,909	6.47	12.21	16.48	**20.32**	1,423
YAP_{BU}	**D_{30}**	21,174	28,904	6.94	12.61	16.63	**20.40**	1,417
	D_{50}	21,166	28,857	6.44	12.31	16.44	**20.52**	1,406
	D_{10}	**60,983**	74,108	n.a.	n.a.	n.a.	n.a.	60,983
XSB_{BU}	**D_{30}**	**59,496**	74,410	n.a.	n.a.	n.a.	n.a.	59,496
	D_{50}	**59,700**	74,628	n.a.	n.a.	n.a.	n.a.	59,700

that it can slightly scale the Knapsack problem up to 16 threads. The speedup results for the YAP_{TD_0} approach were *not considered* (*n.c.*) since without randomization this approach is unable to take advantage of our framework. All threads would replicate the same evaluation sequence and, thus, they would not be able to use answers from subproblems computed by the other threads.

An important aspect of these results is that they show the potential of our framework to scale the execution of multithreaded dynamic programming problems. This is the scenario that our YAP_{TD_2} and YAP_{BU} approaches show, which kept reducing the execution time as we increased the number of threads. Nevertheless, to take advantage of our framework, the user still needs to explicitly implement the thread management and scheduler policy for task distribution. Without such a good policy, the framework can get stuck in not so good results. This is the scenario that our YAP_{TD_0} and YAP_{TD_1} approaches show.

Despite the similar average speedups for YAP_{TD_2} and YAP_{BU}, their execution times are quite different. In this regard, when comparing the T_{seq} and T_1 results of YAP_{TD_0} with YAP_{TD_1} and YAP_{TD_2} approaches, we can observe that the randomized evaluation of YAP_{TD_1} and YAP_{TD_2} introduces an extra cost. This can be explained by the usage of a random function and by the fact that the Prolog code is sightly more complex. Consider, for example, the D_{50} dataset of the Knapsack problem with 32 threads. While the speedup 20.62 of YAP_{TD_2} corresponds to an execution time of 1.233 milliseconds, the speedup 21.76 of YAP_{BU} only corresponds to 804 milliseconds. Similarly for the LCS problem, if considering the D_{50} dataset with 32 threads, while the speedup 19.58 of YAP_{TD_2} corresponds to $2,255$ milliseconds, the speedup 20.52 of YAP_{BU} only corresponds to $1,406$ milliseconds.

Our results also seem to show that the execution times are not affected by the values for the weights/profits generated. In general, the speedups obtained for the different datasets (D_{10}, D_{30} and D_{50}) are always very close for the same number of threads. Note that for the bottom-up approaches this was expected since the complete matrix of results has to be computed. For the top-down approaches, it can be affected by the values for the weights/profits due to the depth in the evaluation tree where solutions can be found. However, since we are using randomized values in the datasets, we are aiming for the average case.

Regarding the comparison with XSB's shared tables model, Yap's results clearly outperform those of XSB. For the execution time with one thread, XSB shows higher times than all Yap's approaches (more than two times the execution times of YAP$_{TD_1}$ and YAP$_{BU}$). For the parallel execution of the Knapsack problem, XSB shows no speedups and for the parallel execution of the LCS problem we have no results available (*n.a.*) since we got *segmentation fault* execution errors.

As we mentioned in Section 2.3, from our point of view, XSB's results are a consequence of the *usurpation operation* that restricts the potential of concurrency to non-mutually dependent sub-computations. As the parallel versions of the Knapsack and LCS problems create mutually dependent sub-computations, which can be executed in different threads, the XSB is actually unable to execute them in a parallel fashion. In other words, even if we launch an arbitrarily large number of threads on those programs, the system would tend to use only one thread at the end to evaluate most of the computations.

7. Related Work

Our framework provides a ground technology for multithreaded dynamic programming in Prolog. From the user's point of view, it can be enabled through the use of single instructions of the form '*:- table p/n*', meaning that common sub-computations for *p/n* will be synchronized and shared between threads at the engine level, i.e., at the level of the tables where the results for such sub-computations are stored. Nevertheless, the user still needs to explicitly implement the thread management and scheduler policy for task distribution, which is orthogonal to the focus of this work. In any case, high-level predicates or libraries, like the generic multithreaded scheduler presented in Fig. 7, can be easily developed on top of our framework to accomplish such tasks. To put our framework in perspective, we next briefly discuss and compare it with others available outside Prolog's world.

For functional programming languages, the Eden [21] and HDC [22] Haskell based frameworks allow the users to express their programs using polymorphic higher-order functions. Eden is a general-purpose parallel functional language suitable for developing sophisticated skeletons as well as for exploiting more irregular parallelism that cannot easily be captured by a predefined skeleton. HDC stands for *higher-order divide-and-conquer* and was originally developed for the parallelization of divide-and-conquer recursions, but is also appropriate for programming skeletons of any kind. Both frameworks showed the efficiency of these type of languages by presenting relevant speedups in benchmarks such as the Karatsuba algorithm, the N-Queens problem and the parallel computation of the Gröbner bases.

For object-oriented programming languages, the MALLBA [23] and DPSKEL [24] frameworks also showed relevant speedups in the parallel evaluation of combinatorial optimization benchmarks. MALLBA tackles the resolution of combinatorial optimization problems using algorithmic skeletons implemented in C++. Several skeletons are available, such as, divide-and-conquer, branch-and-bound, dynamic programming, hill climbing, among many others. DPSKEL is a skeleton tool for dynamic programming problems. In particular, DPSKEL used dynamic programming to solve the Knapsack and LCS problems in a *IBM RS-6000 SP - Nighthawk Power3* shared memory machine. The execution time obtained on the Knapsack benchmark with $1,600$ items and one thread was $2,379$ milliseconds and the best execution time was 359 milliseconds obtained with eight threads, giving a speedup of 6.63. The execution time obtained on the LCS benchmark with sequences of $3,000$ items and one thread was $10,049$ milliseconds and the best execution time was $1,233$ milliseconds obtained with eight threads, giving a speedup of 8.15. These speedups are in line with the speedups obtained with our approach.

Comparing our top-down results with Stivala et al.'s work [3], we can observe comparable results for the Knapsack problem and slight worst results for the LCS problem with YAP$_{TD_1}$, but significant better results with YAP$_{TD_2}$. For the Knapsack problem, Stivala et al.'s present speedups over the sequential time (time without the multithreaded support, i.e., same as column T_{seq} in Table 1) for 100 instances of uncorrelated, weakly correlated, strongly correlated, inverse strongly correlated and almost strongly correlated Knapsack problems, each with 500 items and weights in the interval $[1, 500]$. The best speedups obtained were 8.31 with 31 threads on a *UltraSPARC T1* architecture, 3.11 with 8 threads on a *IBM PowerPC* architecture and 3.21 with 8 threads on a *AMD Quad Core Opteron* architecture.

Regarding our bottom-up results, they are also quite relevant when compared with similar approaches in the literature. For example, for the Knapsack problem, our YAP$_{BU}$ bottom-up approach has similar speedups for 8 threads and better speedups for 16 threads if compared with a multithreaded implementation using the classic parallelization and the Morales parallelization of the Knapsack problem [25] in a *Intel Core 2 Duo - 2 cores* and *Intel Core 2 Quad - 4 cores* architectures. The classic parallelization was implemented using OpenMP and the Morales parallelization using Pthreads for a Knapsack problem with capacity $10,000$ and $10,000$ items generated using the procedure described in [26]. The best speedup obtained over the sequential execution was 7.80 for the classic parallelization and 5.10 for the Morales parallelization, both obtained for 8 threads. For the LCS problem, our bottom-up YAP$_{BU}$ approach shows similar base execution times (with one thread) for sequences of identical sizes, but far better speedups than parallel CUDA, OpenCL and OpenMP versions of the problem [27] running in a *Intel Core(TM) 2 Quad - 4 cores* with *Nvidia GT 430 - 96 cores* architecture, with parallelization based on [18] (same as our bottom-up approach). For two sequences with fixed sizes of $4,000$ symbols, the best results were obtained using CUDA with a speedup of about 13.80, while OpenCL and OpenMP had speedups of about 10.20 and 3.20, respectively.

8. Conclusions and Further Work

Our framework provides a ground technology for multithreaded dynamic programming in Prolog. From the user's point of view, it can be enabled through the use of single instructions of the form '*:- table p/n*'. 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 subproblem, if another thread has already computed them.

To show the potentiality of our multithreaded tabling Prolog engine, we have used two well-known dynamic programming problems, the Knapsack and the Longest Common Subsequence (LCS) problems, and we discussed how we were able to scale their execution. To do so, we have presented multithreaded tabled top-down and bottom-up approaches using, respectively, Yap's mode-directed tabling support and Yap's standard tabling support. Our experiments, on a 32-core AMD machine, showed that using either top-down or bottom-up techniques, we were able to scale the execution of both problems by taking advantage of the state-of-the-art 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. Regarding the particular comparison with XSB Prolog, Yap's results clearly outperform those of XSB for the execution time and for the speedups. For frameworks outside Prolog's world, our framework showed comparable or better speedup results than other parallel implementations of the same problems (based in results from the literature and not taking into account the different execution environments and specificities of each implementation).

Further work will include studying other dynamic programming problems and explore the impact of applying multithreaded tabling to other application domains.

Acknowledgments

This work is partially funded by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF) as part of project NanoSTIMA (NORTE-01-0145-FEDER-000016). Miguel Areias is funded by the FCT grant SFRH/BPD/108018/2015.

References

[1] R. Bellman, Dynamic Programming, Princeton University Press, 1957.

[2] H. Zhang, D. Liu, Y. Luo, D. Wang, Adaptive Dynamic Programming for Control - Algorithms and Stability, no. 1 in Communications and Control Engineering, Springer-Verlag London, 2013.

[3] A. Stivala, P. Stuckey, M. G. de la Banda, M. Hermenegildo, A. Wirth, Lock-Free Parallel Dynamic Programming, Journal of Parallel and Distributed Computing 70 (8) (2010) 839–848.

[4] W. Chen, D. S. Warren, Tabled Evaluation with Delaying for General Logic Programs, Journal of the ACM 43 (1) (1996) 20–74.

[5] R. Marques, T. Swift, Concurrent and Local Evaluation of Normal Programs, in: International Conference on Logic Programming, no. 5366 in LNCS, Springer, 2008, pp. 206–222.

[6] M. Areias, R. Rocha, Towards Multi-Threaded Local Tabling Using a Common Table Space, Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue 12 (4 & 5) (2012) 427–443.

[7] J. Santos, R. Rocha, On the Efficient Implementation of Mode-Directed Tabling, in: International Symposium on Practical Aspects of Declarative Languages, no. 7752 in LNCS, Springer, 2013, pp. 141–156.

[8] V. Santos Costa, R. Rocha, L. Damas, The YAP Prolog System, Journal of Theory and Practice of Logic Programming 12 (1 & 2) (2012) 5–34.

[9] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, D. S. Warren, Efficient Access Mechanisms for Tabled Logic Programs, Journal of Logic Programming 38 (1) (1999) 31–54.

[10] T. Swift, D. S. Warren, XSB: Extending Prolog with Tabled Logic Programming, Theory and Practice of Logic Programming 12 (1 & 2) (2012) 157–187.

[11] J. Freire, R. Hu, T. Swift, D. S. Warren, Exploiting Parallelism in Tabled Evaluations, in: International Symposium on Programming Languages: Implementations, Logics and Programs, no. 982 in LNCS, Springer, 1995, pp. 115–132.

[12] R. Marques, T. Swift, J. C. Cunha, A Simple and Efficient Implementation of Concurrent Local Tabling, in: International Symposium on Practical Aspects of Declarative Languages, no. 5937 in LNCS, Springer, 2010, pp. 264–278.

[13] T. Swift, D. S. Warren, Tabling with Answer Subsumption: Implementation, Applications and Performance, in: European Conference on Logics in Artificial Intelligence, no. 6341 in LNAI, Springer, 2010, pp. 300–312.

[14] J. Wielemaker, Native Preemptive Threads in SWI-Prolog, in: International Conference on Logic Programming, no. 2916 in LNCS, Springer, 2003, pp. 331–345.

[15] M. Areias, R. Rocha, A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs, in: International Symposium on High-level Parallel Programming and Applications, 2014, pp. 259–278.

[16] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley and Sons, 1990.

[17] H.-F. Guo, G. Gupta, Simplifying Dynamic Programming via Mode-directed Tabling, Software Practice and Experience 38 (1) (2008) 75–94.

[18] V. Kumar, Introduction to Parallel Computing, 2nd Edition, Addison-Wesley, 2002.

[19] S. Ghemawat, P. Menage, TCMalloc: Thread-Caching Malloc.
URL `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`

[20] M. Areias, R. Rocha, An Efficient and Scalable Memory Allocator for Multithreaded Tabled Evaluation of Logic Programs, in: International Conference on Parallel and Distributed Systems, IEEE Computer Society, 2012, pp. 636–643.

[21] R. Loogen, Y. Ortega-Mallén, R. Peña-Marí, Parallel functional programming in Eden, Journal of Functional Programming 15 (3) (2005) 431–475.

[22] C. A. Herrmann, C. Lengauer, HDC: A Higher-Order Language for Divide-and-Conquer, Parallel Processing Letters 10 (2/3) (2000) 239–250.

[23] E. Alba, F. Almeida, M. J. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. M. Moreno, C. Pablos, J. Petit, A. Rojas, F. Xhafa, MALLBA: A Library of Skeletons for Combinatorial Optimisation (Research Note), in: International Euro-Par Conference, no. 2400 in LNCS, Springer, 2002, pp. 927–932.

[24] I. Peláez, F. Almeida, F. Suárez, DPSKEL: A Skeleton Based Tool for Parallel Dynamic Programming, in: International Conference on Parallel Processing and Applied Mathematics, no. 4967 in LNCS, Springer, 2007, pp. 1104–1113.

[25] H. Rashid, C. Novoa, A. Qasem, An Evaluation of Parallel Knapsack Algorithms on Multicore Architectures, in: International Conference on Scientific Computing, CSREA Press, 2010, pp. 230–235.

[26] D. Pisinger, Core problems in knapsack algorithms., Operations Research 47 (1994) 570–575.

[27] A. Dhraief, R. Issaoui, A. Belghith, Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability, in: International Conference on Advanced Communications and Computation, 2011, pp. 143—-148.