J. Parallel Distrib. Comput. 70 (2010) 839–848

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Lock-free parallel dynamic programming

Alex Stivala [a,*], Peter J. Stuckey [a,b], Maria Garcia de la Banda [c], Manuel Hermenegildo [d,e], Anthony Wirth [a]

[a] Department of Computer Science and Software Engineering, University of Melbourne, 3010, Australia
[b] NICTA Victoria Research Laboratories, Australia
[c] School of Information Technology, Monash University, 6800, Australia
[d] IMDEA Software, Madrid, Spain
[e] Universidad Politecnica de Madrid (UPM), Madrid, Spain

ARTICLE INFO

Article history:
Received 13 September 2009
Received in revised form
7 January 2010
Accepted 16 January 2010
Available online 1 February 2010

Keywords:
Dynamic programming
Lock-free hash tables
Constraint programming
Multicores
Parallelism

ABSTRACT

We show a method for parallelizing top down dynamic programs in a straightforward way by a careful choice of a lock-free shared hash table implementation and randomization of the order in which the dynamic program computes its subproblems. This generic approach is applied to dynamic programs for knapsack, shortest paths, and RNA structure alignment, as well as to a state-of-the-art solution for minimizing the maximum number of open stacks. Experimental results are provided on three different modern multicore architectures which show that this parallelization is effective and reasonably scalable. In particular, we obtain over 10 times speedup for 32 threads on the open stacks problem.

1. Introduction

Dynamic programming [2] is a powerful technique for solving any optimization problem for which an optimal solution can be efficiently computed from optimal solutions to its subproblems. The idea is to avoid recomputing the optimal solution to these subproblems by reusing previously computed values. Thus, for dynamic programming to be useful, the same subproblems must be encountered often enough while solving the original problem.

Dynamic programming can be easily implemented using either a "bottom-up" or "top-down" approach. In the "bottom-up" approach, the solution to every single subproblem is computed and stored in the dynamic programming matrix, starting from the smallest subproblems until the solution to the entire problem is finally computed. This approach is particularly simple to implement; it requires no recursion and no data structure more sophisticated than an array. It is also efficient if (a) the problem is small enough for the entire matrix to be stored in memory, and (b) the computation of unnecessary cells does not introduce too much overhead. The classic bioinformatics sequence alignment algorithms of Needleman and Wunsch [22] and Smith and Waterman [30] are generally implemented in this way, for example.

In contrast, the "top-down" approach starts from the function call to compute the solution to the original problem, and uses recursion to only compute the solution to those subproblems that are actually encountered when solving the original problem. Previously computed values are reused by applying a technique called *memoization*. In this technique each computed value is stored in an associative array (implemented, for example, by a hash table). Then, the recursive function tests if the value it is called for has been previously computed (and therefore exists in the associative array) and, if so, simply reuses the value rather than recomputing it. This approach to implementing dynamic programming avoids the computation of unnecessary values and is particularly effective when combined with branch-and-bound techniques to further reduce unnecessary computations [24].

Previous efforts at parallelizing dynamic programming have focused on the "bottom-up" style dynamic programming matrix, by computing in parallel cells known to have no data dependencies. For example, the Smith–Waterman algorithm has been accelerated by the parallel computation of cells in the matrix that can be computed independently by the use of SIMD vector instructions [36,25,6], special-purpose hardware [23], general-purpose graphics processing units (*GPGPUs*) [14,16], or other parallel processors such as the Cell Broadband Engine [35]. More generally, Tan

* Corresponding author.
E-mail addresses: a.stivala@pgrad.unimelb.edu.au (A. Stivala), pjs@cs.mu.oz.au (P.J. Stuckey), mbanda@infotech.monash.edu.au (M. Garcia de la Banda), herme@fi.upm.es (M. Hermenegildo), awirth@csse.unimelb.edu.au (A. Wirth).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.01.004

et al. [31] describe a parallel pipelined algorithm to exploit fine-grained parallelism in dynamic programs, and apply it to Zuker's algorithm [40,15] for predicting RNA secondary structure. Subsequently, Xia et al. [37] implemented their own specific parallelization of the Zuker algorithm on FPGA hardware. Chowdhury and Ramachandran [5] describe tiling sequences (recursive decompositions) for several classes of dynamic programs for cache-efficient implementation on multicore architectures.

All these techniques require careful analysis of each particular algorithm to find the data dependencies in the dynamic programming matrix, resulting in a parallelization that is specific to each individual problem. Furthermore, they only work on the "bottom-up" approach and, therefore, can only be applied to problems for which computing every cell is feasible.

In this paper we describe a general technique for parallelizing dynamic programs in modern multicore processor architectures with shared memory. The contributions of our paper are:

- a generic approach to parallelizing "top-down" dynamic programming approach by using
  - a *lock-free* hash table for the memoization, where each thread computes the entire problem but shares results through the hash table;
  - the randomization of the order in which the dynamic program computes its subproblems to encourage divergence of the thread computations, so that fewer subproblems are computed by more than one thread simultaneously;
- an effective algorithm for a lock-free hash table supporting only insertions and lookups; and
- experimental results showing that this approach can produce substantial speedups on a variety of dynamic programs.

The remainder of the paper is organized as follows. In the next section we describe our approach to the parallelization of top-down dynamic programs. In Section 3 we define our hash table implementations, and show their effectiveness in the case where the ratio of inserts to lookups is quite high. In Section 4 we give the results of experiments on four different dynamic programs on three different architectures, illustrating the effectiveness of the parallelization. Finally, in Section 5 we conclude.

2. Parallelizing top down dynamic programs

Our approach to parallelizing top-down dynamic programs is simple. Each thread solves the entire dynamic program independently *except that* whenever it determines a result to a subproblem it places it in a shared hash table, and whenever it begins to compute the answer of a subproblem it checks whether the result already exists in the shared hash table. When one thread has found the solution to the entire dynamic program, we have the answer and simply terminate all other threads.

As previously mentioned, an advantage of the "top-down" versus the simpler "bottom-up" dynamic programming approach, is that the former might not need to compute a value for every subproblem. This opens up the question of the order in which to compute the subproblems, since this order can make a large difference to the number of cells computed [8,24].

In a serial (single-threaded) implementation, we are constrained to choosing a single order in which to compute the subproblems. However, now that we have multiple threads available and the means to safely share values between them, we can parallelize the dynamic program by simply starting several threads at the function call with a randomized ordering choice. That is, each thread runs exactly the same function, starting at the same point, but the randomization of the choice of subproblems results in the threads diverging to compute different subproblems, while still reusing any value that has already been computed by a thread.

```
f(x̄)                                    f(x̄)
  v ← lookup(x̄)                           v ← par_lookup(x̄)
  if v ≠ KEY_NOT_FOUND                    if v ≠ KEY_NOT_FOUND
    return v                                return v
  if b(x̄) then v ← g(x̄)                   if b(x̄) then v ← g(x̄)
  else                                    else
    for i ∈ 1..n                            for i ∈ 1..n in random order
      v[i] ← f(x̄_i)                           v[i] ← f(x̄_i)
    v ← F(v[1], ..., v[n])                  v ← F(v[1], ..., v[n])
  insert(x̄,v)                             par_insert(x̄,v)
  return v                                return v
```

Fig. 1. Generic top-down dynamic programming code on the left, and the parallelized version on the right.

In this way we take advantage of whatever parallel computing power is available to us to compute different subproblems simultaneously.

Throughout this paper we will use the 0/1 knapsack problem, a classic problem for dynamic programming, as an example to demonstrate our technique. In this problem, we are given the total weight (or capacity) W of the knapsack, and a set of n items $\{1, \ldots, n\}$, where each item i has been assigned a weight w_i and a profit p_i. The problem is to choose the subset of items $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} w_i \leq W$ and profit $\sum_{i \in I} p_i$ is maximized. The dynamic programming formulation computes the optimal profit $k(i, w)$ using only items in $1, \ldots, i$ with a weight limit w as:

$$k(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ k(i-1, w) & \text{if } w < w_i \\ \max\{k(i-1, w), k(i-1, w-w_i) + p_i\} & \text{otherwise.} \end{cases}$$

The above dynamic program is presented as a recurrence relation, where the order of computation of the two subproblems in the last case is not defined. However, an implementation using the "top-down" approach and memoization needs to determine an order. In the simple dynamic programming formulation of the 0/1 knapsack problem presented above, our technique randomly chooses, with equal probability, one of $k(i-1, w)$ and $k(i-1, w-w_i) + p_i$ to compute first.

In general, consider a dynamic program f defined as follows:

$$f(\bar{x}) \quad = \quad \textbf{if } b(\bar{x}) \textbf{ then } g(\bar{x}) \\ \textbf{else } F(f(\bar{x}_1), \ldots, f(\bar{x}_n))$$

where $b(\bar{x})$ holds for the base cases (which do not require considering subproblems), $g(\bar{x})$ is the result for base cases, and F is a function combining the optimal answers to a number of sub-problems $\bar{x}_1, \ldots, \bar{x}_n$. The pseudo-code to implement f as a recursive top-down dynamic program is shown on the left of Fig. 1, where insert $(\bar{x}, v)$ and lookup$(\bar{x})$ respectively insert a value v for key $\bar{x}$ in a hash table and look up the stored value.

The key insight of this paper is that we can run f in parallel simply by (a) using a shared parallel hash table and (b) randomizing the order in which we compute the subproblems. The resulting pseudo-code is shown on the right of Fig. 1 with (the very few) changes shown in *italics*. Each of the parallel threads executes this version of f.

Instead of this simple randomization technique, it could be possible to map particular subproblems to particular threads in advance, in order to ensure the divergence of paths through the subproblems from the beginning. For the knapsack problem, this is not particularly difficult, since there are exactly two choices of subproblem at each level, so we can simply fix the ordering for the first $\log_2(n)$ levels for each of n threads. However, this requires analysis of each particular dynamic program, including how many subproblems exist at each level and perhaps even which are likely to be easy and which hard. For more complex dynamic programs (such as the open stacks problem and RNA structural alignment, both considered in this paper), not only is there a variable number of subproblems at each level, but also a potentially large number