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*

A B S T R A C T

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

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

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

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

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

of subproblems (and hence a very large number of orderings). This not only makes it more complicated to try to fix the orderings in advance, but the large number of orderings means that it is very likely that simple randomization ensures divergence anyway. The simple randomization approach avoids all this complexity.

Another possible enhancement is, if some heuristic is available to choose a "good" order in which to compute subproblems, to use this heuristic to determine the order of computation in one thread, and only randomize the others. In this way, the non-randomized thread will use the heuristic ordering just as in the sequential case, but will be assisted by having available the results of any subproblems it requires that may have been computed by the other (randomized) threads.

3. Lock-free concurrent hash tables

The hash table needed to implement a top-down dynamic programming approach only requires two operations: insert a (key, value) pair, and look up a value given a key. Once a (key, value) pair has been inserted, any further insertions of the same key must have the same value (since any computed optimal value for a subproblem is the optimal value, and does not change). Hence, updates and deletions are not required.

Furthermore, to implement efficiently the dynamic programming approach previously described, we require a concurrent hash table that scales well in the number of threads inserting into and reading from it. Thus, the use of critical sections is undesirable, given they require access controlled by locks (mutexes). Also, we would like our hash table to be not only *non-blocking*, that is, an arbitrary delay of one thread cannot halt progress of others (thereby excluding the use of mutual exclusion), but also meet the stronger condition of being *lock-free*. Although lock-freedom has had several historical definitions, we use the definition of [10], that "[a] method is *lock-free* if it guarantees that infinitely often *some* method call finishes in a finite number of steps". [10, p. 60]. That is, there must be guaranteed system-wide progress.

Various implementations of hash tables are possible, including separate chaining (open hashing) and open addressing (closed hashing). In a separate chaining hash table, each entry contains the head of a linked list of elements with the same hash value; collisions are handled by adding elements to the linked list. An open addressing hash table stores elements directly in the table entries, resolving collisions by putting elements in some other free slot in the table. It was not clear *a priori* which implementation would work better for the associative array we require for parallel dynamic programming. Thus, we experimented with several implementations in three different processors: AMD Opteron (Intel IA-64), IBM PowerPC, and UltraSPARC T1.

3.1. Our implementations

In order to implement lock-free data structures and algorithms efficiently, hardware support in the form of atomic instructions is required. We only consider here the *compare-and-swap* (or CAS) operation, which we will denote as CompareAndSwap, with the semantics detailed in Fig. 2.

Support for the CompareAndSwap operation dates back to the IBM S/370 and it is still available on many modern processors including Intel IA-64 (*x*86) and Sun SPARC. Processors, like the IBM PowerPC, that do not support CompareAndSwap, often directly support Load-Linked and Store-Conditional (LL/SC) instead. Since CompareAndSwap can be implemented using LL/SC [19], it is sufficient to focus on CompareAndSwap. Use of CompareAndSwap can result in the *ABA problem*, whereby the value at an address is changed from *a* to *b* and then back to *a* after another thread, *P* has already read *a* from it. Then thread *P* can succeed in using

```
CompareAndSwap(address, expectedval, newval)
  atomically:
    load value_at(address) into oldval
    if oldval = expectedval then
      store newval at address
    return oldval
```

Fig. 2. Semantics of the CompareAndSwap operation. The new value *newval* is only stored at *address* if it contained the expected value *expectedval*. The value that was at *address* is returned, allowing the caller to detect failure of the operation, when *expectedval* is not returned.

CompareAndSwap on the same address with an expected value of *a*, as if the value at the address had never changed. This can result in incorrect semantics if, for example, the values are addresses, and the second use of *a* is due to reuse of a deleted node in a list. As we will show later, the ABA problem does not arise in our implementations.

As we require only insert and lookup, our implementations of the separate chaining and open addressing hash tables are quite simple.

Separate chaining: The lookup operation consists of simply hashing the key, and comparing the key with each element in the linked list anchored at the entry for that hash value. The insert operation requires the use of the CompareAndSwap operation to insert the newly allocated element at the head of the list anchored at the hash value for the new key; if the head of the list has changed during the insertion, we must retry it.

This is essentially a simplified version of the lock-free hash table described by Michael [20]. As we do not implement the delete operation, the ABA problem does not arise. To allocate new cells, our implementation uses the Streamflow scalable locality-conscious multithreaded allocator [27]. As Streamflow was not available for SPARC, on this platform we used a simple cellpool allocator, in which we allocated a large amount of memory at initialization time, and allocated cells in a lock-free manner by using the CompareAndSwap operation to advance the "next cell" pointer for each request.

Since the CompareAndSwap operation is used only on pointers (rather than the actual key), there is no limit on key size imposed by the maximum size of the operand for this instruction. Functions to compare keys and to copy keys and values can be provided as callback functions, allowing any data type to be used for keys and values.

If the allocator is lock-free then this hash table is lock-free. The only operation that could cause it not to be is the insert operation, which must be retried only if the CompareAndSwap operation fails (i.e., when another entry with the same hash value for the key has been inserted during the operation). But since the CompareAndSwap operation can only fail because it has succeeded in another thread, there is global progress and therefore the algorithm is lock-free.

Open addressing: We made the further simplifying assumption that the hash table does not need to be resized and simply allocate a very large (2^{26} entries) hash table at initialization. Hence, this implementation requires no thread-safe memory allocator and memory allocation is not required at all during its use, both simplifying its implementation and removing a possible impediment to scalability in the number of threads. Each entry in the hash table is a 64-bit key and a 64-bit value (thus, a key is always immediately followed in memory by its value). We use linear probing to resolve hash collisions. As with separate chaining, the lookup operation requires no special instructions, but the insert operation uses the CompareAndSwap operation to ensure the slot for the key it is about to insert did not have another key inserted during its operation. The algorithms for the par_insert and par_lookup operations in the open addressing hash table are shown in Fig. 3. The essential invariant in this concurrent data structure

```
par_insert(key,value)
    ent ← get_entry(key)
    if ent = NULL then
        error_exit("hash table full")
    if ent.key = NO_KEY then
        if CompareAndSwap(ent.key, NO_KEY, key)
                ≠ NO_KEY then
            return par_insert(key, value)
    ent.value ← value
    return TRUE

par_lookup(key)
    ent ← get_entry(key)
    if ent ≠ NULL ∧ ent.value ≠ NO_VALUE then
        return ent.value
    else
        return KEY_NOT_FOUND

get_entry(key)
    h ← hash(key)
    ent ← hashtable[h]
    probes ← 0
    while probes < TABLE_SIZE − 1 ∧ ent.key ≠ key
            ∧ ent.key ≠ NO_KEY do
        probes ← probes + 1
        h ← (h + 1) mod TABLE_SIZE
        ent ← hashtable[h]
    if probes ≥ TABLE_SIZE − 1 then
        return NULL
    else
        return ent
```

Fig. 3. Algorithms for the par_insert and par_lookup operations on the concurrent lock-free open addressing hash table.

is that once an entry has changed from the initial empty state (marked with the NO_KEY constant), to containing a key, the key field never changes: the entry is forever occupied by that key. The CompareAndSwap operation ensures that if another thread has taken the entry during the par_insert, then this attempt to use it is abandoned, and we restart the insertion operation. Because the key value can only change from NO_KEY to some other value, and never back again, the ABA problem cannot occur with this use of CompareAndSwap. Note also that (as is typically the case in hash tables) there can be no duplicate keys in the table; once an entry has been taken by a key, not only will it never again be empty, but any insertion of the same key will find that same entry.

Note that the key and the value fields are operated on separately. This allows a thread to look up a key while another thread is in the process of inserting that key and, hence, read the value field before it has been set by the par_insert function. This simply results in the par_lookup operation finding the NO_VALUE constant if the key did not previously exist in the table, since all entries are initialized to have NO_KEY and NO_VALUE in the key and value fields respectively. This is interpreted as the key not being found in the table, just as if the par_lookup operation had been serialized to occur prior to the par_insert operation that is in progress.

This algorithm is clearly non-blocking; an arbitrary delay of any thread will not halt the progress of any other threads. We claim that this open addressing hash table is also lock-free. To see that this is the case, we need only consider the par_insert operation, since par_lookup can clearly not be delayed by any other thread. Consider the case that two threads are simultaneously trying to insert the same key in an empty entry. Only one of them will execute CompareAndSwap successfully, as the other will find that the entry is no longer empty and fail. When this thread retries (the recursive call), it will find the entry no longer empty and complete. In general, we need to show that it is not possible for two (or more) threads to continually delay each other and cause livelock. To see that this cannot occur, observe that the CompareAndSwap operation only fails in one thread if it succeeds in another (because the contentious entry has been changed from NO_KEY to some key value). Hence, at least one thread has completed its operation (since if the CompareAndSwap succeeds, that par_insert operation will succeed), and the algorithm is lock-free.

Various optimizations are possible when implementing this algorithm. For example, we do not need to recursively call par_insert on failure of the CompareAndSwap operation, but could instead use a loop and not recompute the hash value for the key. In the interests of simplicity and clarity we do not show here or implement such optimizations, but use a straightforward implementation of the algorithm as shown.

3.2. Evaluation of the time performance of hash tables

We built and tested our two hash tables (separate chaining and open addressing) on three different processors: AMD Opteron (Intel IA-64) and IBM PowerPC processors running Linux, and on the UltraSPARC T1 processor running Solaris. These hash table implementations used the `gcc` (version 4.1.2) compiler builtins for atomic operations, except on Solaris where we used the Sun Studio 12 C compiler and the Solaris (version 5.10) atomic operation library functions.

For the AMD Opteron processor, we also compared our hash tables to a publicly available C implementation of a Java non-blocking data structure library called nbds [21], and the concurrent hash map with scalable allocator from the Intel Threading Building Blocks 2.1 (TBB) for Open Source [32]. The latter is unlike all other implementations in that it is not lock-free, and is in C++ rather than C. Note that TBB and nbds only work on Intel compatible processors.

In order to evaluate the time performance of each hash table, we used a test program that, for 10^7 uniformly distributed random 64-bit integer keys, first performs a lookup operation, then, if the key is not already in the table, performs a lookup on a new random key (also a 64-bit integer chosen uniformly at random), then inserts the original key, thereby generating on average twice as many lookup as insert operations. If the key is already in the table it simply asserts that the value is the correct one for the key (the values are inserted as the same as the key to make this correctness test straightforward). We use this ratio of inserts and lookups in order to model the dynamic programming usage we have observed in practice of twice as many lookups as inserts. Note that this ratio is quite different from those generally cited as typical for hash tables, such as 90% lookup, 9% insert, 1% delete [10, p. 300]. Thus, we require a hash table that is efficient for much more frequent insertions than typical usages (and which does not need to support a delete operation at all).

On the UltraSPARC T1 platform, Fig. 4 shows that the open addressing hash table scales quite well in the number of threads, up to a peak speedup of 32 times for 30 threads (note this is relative to the baseline sequential implementation of separate chaining, and even with 1 thread is faster). The separate chaining hash table, however, scales very poorly, with a peak speedup of less than 4 times, and no significant increase after 8 threads. This result was achieved with our own trivial cell pool allocator, as described in Section 3.1, as the Streamflow allocator [27] was not available for SPARC. We also experimented with the standard `malloc()` and the `umem_cache_*()` object cache allocator [3,4] but found that the trivial cell pool allocator was faster for this purpose (data not shown).

On the PowerPC platform, Fig. 5 shows that the separate chaining and open addressing hash tables achieve similarly good (linear) speedup. Note that the speedup actually appears to be superlinear; this is because the baseline uses the standard library `malloc()` for the memory allocator, while the lock-free implementation uses the Streamflow [27] allocator, which gives better performance for even a single thread on this platform.

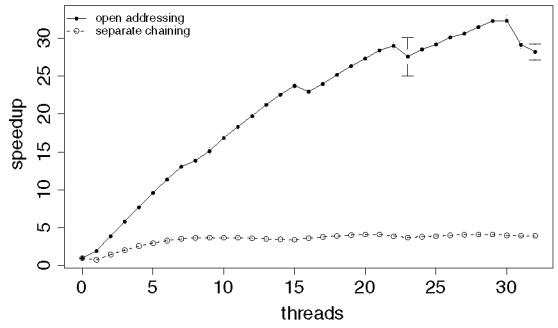

Fig. 4. Speedup for lock-free open addressing and separate chaining hash tables on UltraSPARC T1 (SunFire T2000, 1 GHz, 8 cores, 32 total concurrent threads, 16 GB RAM), 2/3 lookup and 1/3 insert. The baseline (0 threads value) is the separate chaining hash table implemented with no atomic instructions and compiled and linked with no threading support, and the standard library `malloc()` Each test was repeated 10 times and the average value used to compute the speedup; error bars show the 95% confidence interval, where this is large enough to show on the graph scale.

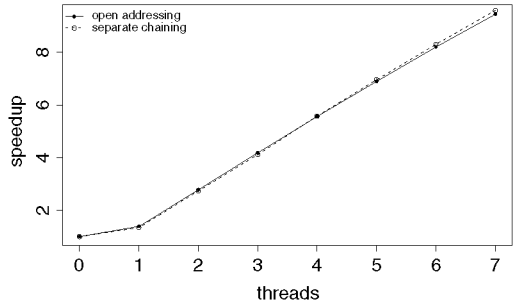

Fig. 5. Speedup for lock-free open addressing and separate chaining hash tables on PowerPC 64-bit (IBM eServer pSeries 650, 1.45 GHz, 8 cores, 16 GB RAM), 2/3 lookup and 1/3 insert. The baseline (0 threads value) is the separate chaining hash table implemented with no atomic instructions and compiled and linked with no threading support, and the standard library `malloc()`. Each test was repeated 10 times and the average value used to compute the speedup; error bars show the 95% confidence interval, where this is large enough to show on the graph scale.

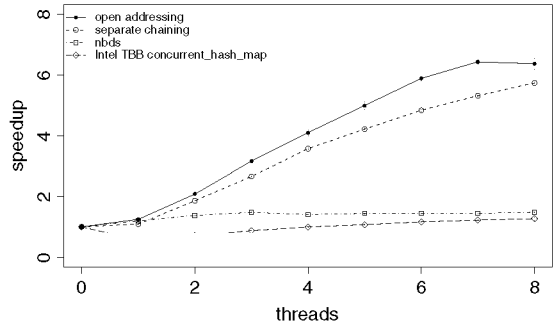

Fig. 6. Speedup of different concurrent hash table implementations on AMD Quad Core Opteron (2.3 GHz, 2 processors, total 8 cores, 32 GB RAM), 2/3 lookup and 1/3 insert. The baseline (0 threads value) is a simple separate chaining hash table implemented with no atomic instructions and compiled and linked with no threading support, and the standard library `malloc()`. Each test was repeated 10 times and the average value used to compute the speedup; error bars show the 95% confidence interval, where this is large enough to show on the graph scale.

Hence, even the multithreaded implementation with only one thread is faster than the baseline.

On the AMD Opteron platform, Fig. 6 shows the speedup of several concurrent hash table implementations. Speedup for the existing publicly available implementations is disappointing, with both the Intel TBB library (which uses locks) and nbds achieving no significant speedup with our high-insertion workload. Both the separate chaining and open addressing hash tables achieve quite good speedup on this platform, the former having a peak speedup of 5.7 times and the latter 6.4 times.

The results detailed in this section lead us to choose the lock-free open addressing hash table for implementing the parallel dynamic programming algorithm, since the same, quite simple, code achieves good scalable speedup on the three systems tested, and does not require the added complication of a scalable multithreaded allocator. The separate chaining hash table performs slightly better than the open addressing hash table on the PowerPC, but significantly worse on the other two platforms, although if Streamflow were available on the SPARC platform it may well change the results on that platform significantly. We note that the UltraSPARC T1 is particularly suitable for running scalable multithreading applications as it was designed specifically for scalable multithreaded performance rather than for optimal performance for single threads [29].

4. Experiments for parallelizing DP

We now show the results of using our method to parallelize the dynamic programming solutions to four very different problems. These dynamic programs were implemented in C with the same compilers described in Section 3.2. We note, however, that in testing the hash table implementation, as described in Section 3.2, we used uniformly randomly distributed keys. It is well known (see, e.g., Askitis [1] and citations therein) that uniform distribution of hash values is important in the performance of hash tables, in order to prevent too-frequent collisions. For the tests described in Section 3.2, the hash function was unimportant, since the keys were generated from a uniform random distribution. However, in implementing dynamic programs, this is certainly not the case. To obtain a reasonably uniform distribution of hash values, we have used a hash function that combines bit shifts and multiplication [13].

Table 1 shows the total elapsed times of the baseline (no threading) implementation for all of the problems on each of the platforms tested, in order to give some idea of the problem sizes and relative speed of the platforms for single core execution. These times are the sums of the times for each of the individual problem instances (as detailed for each problem in the following sections). The speedup calculations are relative to the baseline consisting of the mean of all the individual problem instances. Because our algorithm makes use of randomization, it is possible that there could be significant variation in the execution times over different runs of the same problem instance. Therefore, as well as averaging over the problem instances, we ran some of the tests with 10 iterations of each of the sets of problem instances, and found the standard deviation to be small enough to justify presenting the speedup figures to two decimal places.

4.1. Knapsack

We implemented in C the knapsack dynamic program formulation given in Section 2 using the top-down recursive approach. We then parallelized this program using randomization and ran it using our lock-free open addressing hash table described in Section 3.1. We used the test generator `gen2.c` used in Martello et al. [17] and available from http://www.diku.dk/~pisinger/gen2.c to create 100 instances each 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].

On the UltraSPARC T1 platform, Fig. 7 and Table 2 show that randomizing the subproblem ordering provides a speedup of 8.97 times (for 31 threads) for the knapsack dynamic program. Fig. 10 sheds some light on why this does not achieve the speedup of up to 19 times achieved in Fig. 4 for simple insertions and lookups:

Table 1
Total elapsed times for the baseline for each of the problems on each of the platforms tested.

	UltraSPARC	PowerPC	Opteron
Knapsack	4 h 12 min 25 s	2 h 08 min 43 s	0 h 44 min 32 s
Shortest paths	11 h 37 min 46 s	9 h 42 min 37 s	2 h 12 min 44 s
RNA struct. alignminent	1 h 12 min 34 s	0 h 30 min 22 s	0 h 10 min 04 s
Open stacks	0 h 53 min 45 s	0 h 24 min 01 s	0 h 07 min 47 s

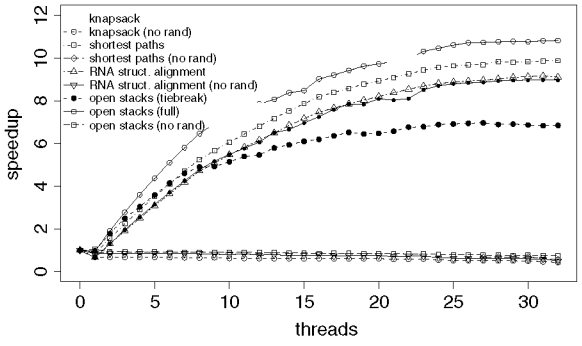

Fig. 7. Speedup for parallelized dynamic programs using the lock-free open addressing hash table on UltraSPARC T1. The baseline (0 threads value) for open stacks is the implementation described in Garcia de la Banda and Stuckey [8], and for the other problems are the same implementations as the parallelized versions, but without randomization of subproblem orderings and using a simple (not lock-free) hash table implementation. All baseline implementations use no atomic instructions and are compiled and linked with no threading support, and the standard library `malloc()`. Implementations labelled with "(no rand)" refer to implementations where no randomization is used, so all threads solve the same subproblems in the same order.

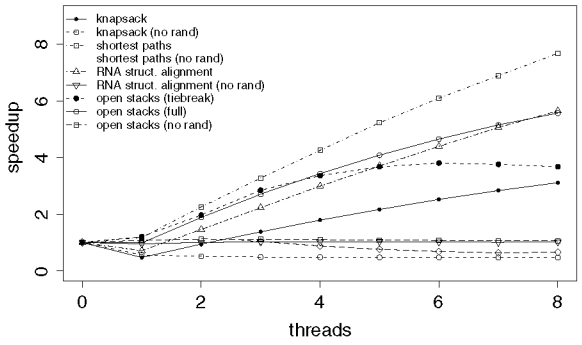

Fig. 8. Speedup for parallelized dynamic programs using the lock-free open addressing hash table on IBM PowerPC (8 cores). The baseline (0 threads value) for open stacks is the implementation described in Garcia de la Banda and Stuckey [8], and for the other problems are the same implementations as the parallelized versions, but without randomization of subproblem orderings and using a simple (not lock-free) hash table implementation. All baseline implementations use no atomic instructions and are compiled and linked with no threading support, and the standard library `malloc()`.

Table 2
Maximum speedup for parallelized dynamic programs using randomization and the lock-free open addressing hash table on UltraSPARC T1.

Problem	Speedup	Threads
Knapsack	8.97	31
Shortest paths	9.88	32
RNA struct. alignment	9.17	31
Open stacks (tiebreak)	6.96	27
Open stacks (full)	10.83	32

The total number of computations increases in approximately a straight line from 1.03×10^{10} for a single thread to 1.56×10^{10} for 32 threads. Thus, at 32 threads, 5.305×10^{9} more subproblems are computed in total, a 51% increase in total computations.

This shows that we cannot achieve indefinite linear scalability with our technique, even if we had a perfectly linearly scalable lock-free concurrent hash table. As we increase the number of threads, we inevitably recompute some subproblems, when two or more threads compute the same subproblem simultaneously. Hence, although we have more threads exploring the solution space, we also have in aggregate more computations than we would have with fewer threads.

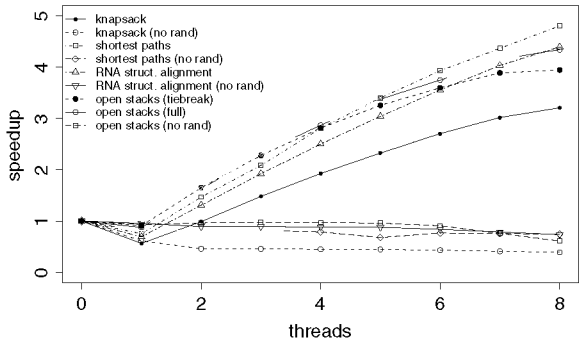

Fig. 9. Speedup for parallelized dynamic programs using the lock-free open addressing hash table on AMD Quad Core Opteron (2.3 GHz, 2 processors, total 8 cores, 32 GB RAM), The baseline (0 threads value) for open stacks is the implementation described in [8], and for the other problems are the same implementations as the parallelized versions, but without randomization of subproblem orderings and using a simple (not lock-free) hash table implementation. All baseline implementations use no atomic instructions and are compiled and linked with no threading support, and the standard library `malloc()`.

Table 3
Maximum speedup for parallelized dynamic programs using randomization and the lock-free open addressing hash table on IBM PowerPC (8 cores).

Problem	Speedup	Threads
Knapsack	3.11	8
Shortest paths	7.68	8
RNA struct. alignment	5.64	8
Open stacks (tiebreak)	3.81	6
Open stacks (full)	5.57	8

Table 4
Maximum speedup for parallelized dynamic programs using randomization and the lock-free open addressing hash table on AMD Quad Core Opteron (2.3 GHz, 2 processors, total 8 cores, 32 GB RAM).

Problem	Speedup	Threads
Knapsack	3.21	8
Shortest paths	4.81	8
RNA struct. alignment	4.40	8
Open stacks (tiebreak)	3.95	8
Open stacks (full)	4.34	8

On the PowerPC platform, Fig. 8 and Table 3 show that a speedup of 3.1 times is achieved for 8 threads, and for the AMD Opteron platform, Fig. 9 and Table 4 show a speedup of 3.2 times for 8 threads (as opposed to 4.2 times for 8 threads in the UltraSPARC T1). We note that the speedup appears to be nearly perfectly linear in Fig. 8, since the test system only has 8 cores. However, as seen for the UltraSPARC T1 platform (Fig. 7), and to a lesser extent (for the last data point only, for 8 threads) in Fig. 9, it is likely that the speedup will level out; its rate of increase will decline after 8 threads.

We also implemented the technique, mentioned in Section 2, of fixing the subproblem orderings so that each of n threads is

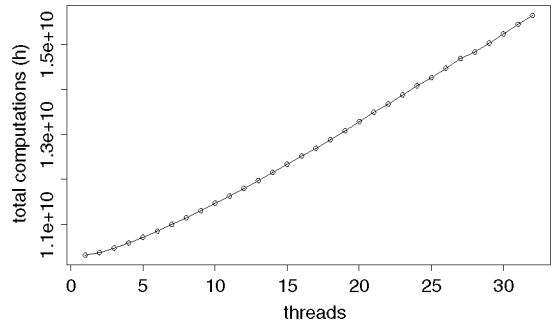

Fig. 10. Total computations (number of times the par_insert function is called, h) for the knapsack dynamic program using the lock-free open addressing hash table on UltraSPARC T1.

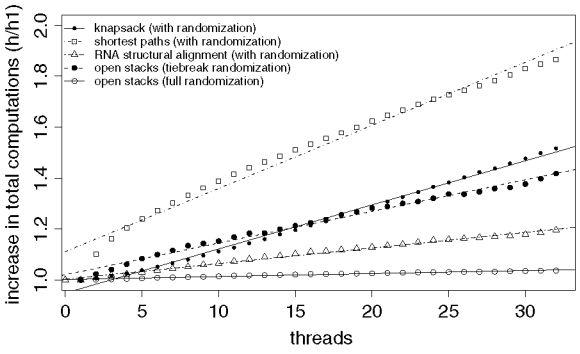

Fig. 11. Increase in total computations (number of times the par_insert function is called, h) relative to the number for a single thread (i.e., h/h_1) for the dynamic programs using the lock-free open addressing hash table on UltraSPARC T1. Straight lines were fitted to the data points by linear least squares regression with the `lm` function in *R*.

guaranteed to take a different path for the first $\log_2(n)$ levels, but found that it did not improve the results (data not shown).

4.2. Shortest paths

The shortest path between two nodes i and j in a directed graph can be found with the dynamic programming Floyd–Warshall algorithm [7,33]. This algorithm is based on the function $s(i, j, k)$, which returns the length of the shortest path from node i to node j using only $1, \ldots, k$ as intermediate nodes. This function is defined using dynamic programming as follows:

$$s(i,j,k) = \begin{cases} 0, & \text{if } i = j \\ d_{ij}, & \text{if } k = 0 \\ \min\{s(i,j,k-1), & \\ \quad s(i,k,k-1) + s(k,j,k-1)\} & \text{otherwise} \end{cases}$$

where d_{ij} is the weight (distance) of the directed edge from i to j. Once k reaches the total number of nodes in the graph, say n, the value returned by $s(i, j, n)$ represents the optimal solution to the initial problem.

As before, we implemented this dynamic program in C using the recursive top-down approach, parallelized it using randomization, and ran it using our lock-free open addressing hash table. To randomize we randomly choose, with equal probability, one of the $3! = 6$ possible orderings of the three subproblems in the recurrence relation just described. We derived the test data for this example from the 9th DIMACS Implementation Challenge — Shortest Paths data from http://www.dis.uniroma1.it/~challenge9/. We chose three 500 node slices of the New York City dataset, with 100 queries per instance.

Figs. 7–9 and Tables 2–4 show that the resulting program achieves a speedup of 9.88 times with 32 threads on the UltraSPARC T1 platform, 7.68 times with 8 threads on the IBM PowerPC platform, and 4.81 times with 8 threads on the AMD Opteron platform.

4.3. RNA base pairing probability matrix alignment

Finding an alignment of two RNA sequences that takes into account both the sequences and the secondary structures into which they fold, is another problem with a dynamic programming solution [26]. If we are given the base pairing probability matrices [18] for two RNA sequences, then $S(i, j, k, l)$, the best score of matching subsequences $i..j$ in sequence A and $k..l$ in sequence B can be computed by the following dynamic program [11]:

$$S(i,j,k,l) = \max \begin{cases} S(i+1,j,k,l) + \gamma, \\ S(i,j,k+1,l) + \gamma, \\ S(i+1,j,k+1,l) + \sigma(A_i, B_k), \\ \max\limits_{h \le j, q \le l} S^M(i,h,k,q) + S(h+1,j,q+1,l) \end{cases}$$

$$\begin{aligned} S^M(i,j,k,l) \;=\; & S(i+1,j-1,k+1,l-1) \\ & + \Psi^A_{ij} + \Psi^B_{kl} + \tau(A_i, A_j, B_k, B_l). \end{aligned}$$

In the above, $S(i, j, k, l)$ is the best score of matching subsequences $i..j$ in sequence A and $k..l$ in sequence B, $\gamma < 0$ is the gap penalty, σ and τ are score functions for unpaired bases and base pairs, respectively, and Ψ^A and Ψ^B are the base pairing log probability matrices for sequence A and sequence B, respectively.

`LocARNA` [34] is a recent implementation of this algorithm that significantly improves efficiency by carefully rearranging the dynamic programing equations, and by providing an efficient bottom-up implementation of the dynamic program. Its time complexity is effectively quadratic due to the filtering out of base pairs with probability less than a threshold. The sparseness of the probability matrices makes the number of remaining nonzero base pair probabilities effectively linear [34].

As before, we implemented the above dynamic program in C using the recursive top-down approach, parallelized it using randomization, and ran it using our lock-free open addressing hash table. To randomize we ordered the subproblem computations according to a random permutation of the entire list of subproblems in the $S(\cdot)$ and $S^M(\cdot)$ computation.

We used the probability threshold as described by Will et al. [34] to reduce the computational complexity of the problem, and, as does `LocARNA` [34], normalized the base pairing log probabilities and converted them to integers in order to avoid floating point computations. This is particularly important on the UltraSPARC T1 processor, since it has a single floating point unit per chip, shared by all cores [29], which limits scalability of programs using floating point operations.

We run the parallelized program with the BRAliBase II [9] pairwise RNA structural alignment benchmark data set, which consists of 118 pairs of RNA sequences (from a total of 78 individual sequences) to be aligned. We used the `RNAfold` program in the Vienna RNA Package [40,12] to generate the base pairing probability matrix for each sequence, which is then processed by a script into a form used as input to our program.

Figs. 7–9 and Tables 2–4 show that the top-down implementation with the concurrent lock-free open addressing hash table achieves a speedup of 9.17 times with 31 threads on the UltraSPARC T1 platform, 5.64 times with 8 threads on the IBM PowerPC platform, and 4.40 times with 8 threads on the AMD Opteron platform for this implementation.

We note that this problem is different from the ones previously shown, in that it is perfectly feasible to compute and store every possible subproblem (i.e., use the bottom-up technique) for realistic problem instances. Hence, a hash table is not necessary. Furthermore, a careful arrangement of the computation order and efficient implementation of the bottom-up technique, such as

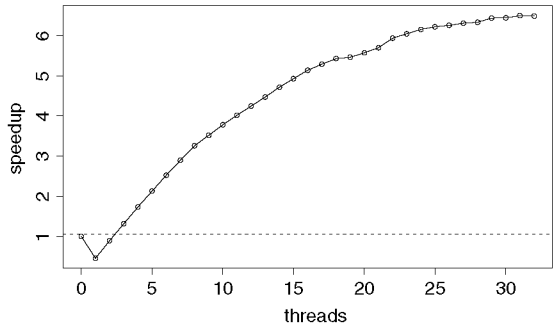

Fig. 12. Speedup for parallelized RNA base pairing matrix alignment using an array to store computed values on the UltraSPARC T1 (32 cores). The baseline (0 threads value) is a top-down implementation of the dynamic program using an array to store computed values, with no randomization, and compiled and linked with no threading support. The dashed horizontal line represents the speedup relative to the same baseline of a bottom-up implementation of the dynamic program, showing that our parallelized implementation is faster than the bottom-up implementation on this platform when 3 or more threads are used.

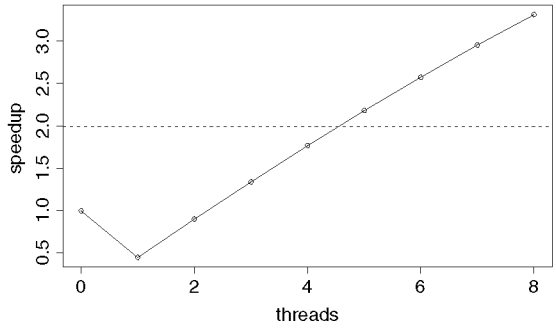

Fig. 13. Speedup for parallelized RNA base pairing matrix alignment using an array to store computed values on the IBM PowerPC (8 cores). The baseline (0 threads value) is a top-down implementation of the dynamic program using an array to store computed values, with no randomization, and compiled and linked with no threading support. The dashed horizontal line represents the speedup relative to the same baseline of a bottom-up implementation of the dynamic program, showing that our parallelized implementation is faster than the bottom-up implementation on this platform when 5 or more threads are used.

LocARNA [34], is considerably faster than the top-down technique (certainly for a single thread). Therefore, we also implemented a bottom-up version of the algorithm, as well as a top-down version that uses an array rather than the concurrent hash table to store results. Fig. 12 shows that the top-down parallelization using an array rather than a hash table achieves a maximum speedup of 6.50 times on the UltraSPARC T1 processor and is faster than the bottom-up implementation for 3 or more threads. Figs. 13 and 14 show that the corresponding results on the PowerPC and AMD Opteron processors respectively are a maximum speedup of 3.31 times and 2.78 times, and that on the IBM PowerPC and AMD Opteron, 5 and 4 or more threads,respectively, are required in order to be faster than the bottom-up implementation.

This example demonstrates that our technique can be used to accelerate any top-down implementation of a dynamic program (whether storing the results in a hash table or an array), without having to find an efficient ordering of the (bottom-up) computation.

4.4. Minimization of Open Stacks

The Minimization of Open Stacks Problem [38,39] aims at finding a sequence in which to manufacture a set of products so that the maximum number of *active customers* is minimized. A client c is said to be active from the time the first product ordered by c starts to be manufactured, until the last product ordered by c is manufactured. Traditionally, the already manufactured products ordered by a client are stored in a stack, until the order was completed and the stack was closed. Thus, the number of active customers corresponds to the number of open stacks (thus the name).

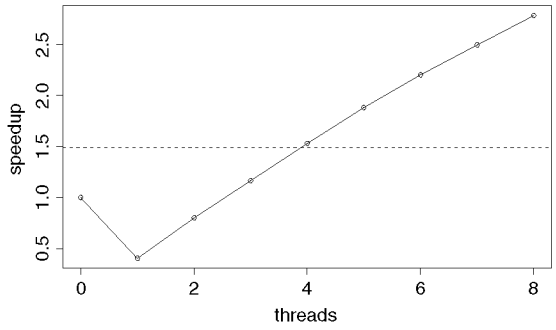

Fig. 14. Speedup for parallelized RNA base pairing matrix alignment using an array to store computed values on the AMD Opteron (8 cores). The baseline (0 threads value) is a top-down implementation of the dynamic program using an array to store computed values, with no randomization, and compiled and linked with no threading support. The dashed horizontal line represents the speedup relative to the same baseline of a bottom-up implementation of the dynamic program, showing that our parallelized implementation is faster than the bottom-up implementation on this platform when 4 or more threads are used.

A highly effective dynamic programming solution to this problem is described by Garcia de la Banda and Stuckey [8] in terms of a set P of products, a set C of customers, a function $c(p)$ that returns the set of customers who have ordered product $p \in P$, and its extension $c(S) = \cup_{p \in S} c(p)$ which returns the set of customers ordering products from the set $S \subseteq P$. If $A \subset P$ is the set of all products scheduled to be manufactured after product p, then the set of active customers at the time p is manufactured is

$$a(p, A) = c(p) \cup (c(A) \cap c(P - A - \{p\}))$$

that is, those who ordered p, plus those whose orders include some products scheduled after p and some scheduled before p. Let $S \subseteq P$ denote the set of products that still need to be manufactured and let $stacks_P(S)$ be the minimum number of stacks required to schedule the products in S. Then, the dynamic programming recurrence relation to compute $stacks_P(S)$ is:

$$\min_{p \in S} \max\{a(p, S - \{p\}), stacks_P(S - \{p\})\}$$

which computes, for each product p, the maximum number of open stacks needed if p was scheduled first, and then takes the minimum of this over all products.

In the dynamic programming formulation of the open stacks problem described by Garcia de la Banda and Stuckey [8], the order in which the products are tried significantly affects the amount of work performed, and a heuristic is used which selects the product with the least number of active customers (its pseudo code is given in Fig. 15). We demonstrate two different approaches to randomization. Our first approach, stacks_tiebreak, retains the heuristic and only introduces randomization for breaking ties. This is achieved by changing the third line of the **while** loop in Fig. 15 to

$$p \leftarrow \text{RandomElementOf(Q)}.$$

Our second approach, stacks_full, discards the heuristic and randomly selects any product that has the same or fewer number of active customers than the currently known minimum. In addition to the change above, the 2nd line in the **while** loop is replaced by

$$Q \leftarrow \{p \mid p \in T, a(p, S - \{p\}) \leq \min\}.$$

We used the same C code described in Garcia de la Banda and Stuckey [8], adding the randomization techniques just

```
stacks(S, L, U)
  if S = ∅ then
    return 0
  min ← lookup(S)
  if min ≠ KEY_NOT_FOUND then
    return min
  min ← U + 1
  T ← S
  if ∃p ∈ S.c(p) ⊆ c(P − S) ∩ c(S) then
    return stacks(S − {p}, L, U)
  while min > L ∧ T ≠ ∅ do
    min_now ← min{a(p, S − {p}) | p ∈ T}
    Q ← {p | p ∈ T, a(p, S − {p}) =min_now }
    p ← FirstElementOf(Q)
    T ← T − {p}
    if a(p, S − {p}) ≥ min then
      break
    sp ← max{a(p, S − {p}), stacks(S − {p}, L, U)}
    if sp < min then
      min ← sp
  insert(S,min)
  return min
```

Fig. 15. The A^* dynamic programming algorithm for the open stacks problem [8]. stacks(S, L, U) returns the minimal number of open stacks required for scheduling the set of products S given a lower bound on the number of stacks L and an upper bound U. A heuristic choice of the order in which to try products is made by selecting a product p which results in the least number of active customers if scheduled immediately.

described, and replacing the simple separate chaining hash table of Garcia de la Banda and Stuckey [8] with our lock-free open addressing hash table described in Section 3.1. We use the same problem instances from the Constraint Modelling Challenge 2005 (http://www.dcs.st-and.ac.uk/~ipg/challenge), omitting the two instances that exceeded the search limit in Garcia de la Banda and Stuckey [8], as well as a single instance that requires a set of more than 64 products after preprocessing (since their implementation represents the product set as a bit set, and is currently restricted to a single (64 bit) machine word as the hash table key).

Figs. 7–9 and Tables 2–4 show that the tiebreak randomization achieves a speedup of up to 6.96 times (for 27 threads) on the UltraSPARC T1 platform, 3.81 times (for 6 threads) on the PowerPC platform, and 3.95 (for 8 threads) on the AMD Opteron platform. For the full randomization (stacks_full) the corresponding speedups are 10.83, 5.57, and 4.34 times for 32, 8 and 8 threads, respectively.

Note that, for comparison, not randomizing the choice so that each thread performs the same computation results in no speedup and, in fact, the addition of more threads causes a slight slowdown.

Just as for the knapsack problem, we can see from Fig. 11 that increasing the number of threads also linearly increases the total number of subproblems computed. At the maximum value of 32 threads, this is a 51% increase for knapsack and a 43% increase for open stacks with the tiebreak randomization, but only a 3.7% increase for open stacks with the full randomization.

The difference in speedup between stacks_tiebreak and stacks_full illustrates the importance of ensuring that the path threads take through the set of subproblems diverges as much as possible. In the extreme case of no randomization (so that each thread computes exactly the same subproblems in exactly the same order), we clearly obtain no speedup. In general, the more divergent between threads that the path through the set of possible subproblems of a particular dynamic program becomes, the better the speedup will be. This is because more divergent paths cause a slower growth (lesser slope of the h/h_1 lines as plotted in Fig. 11) in the number of duplicated computations (when two more more threads compute the same subproblem). In the case of the open stacks problem, this reduction in duplicated computation more than makes up for the loss of the heuristic which selects the product with the least number of open stacks if selected first. We also implemented the technique, mentioned in Section 2, of having one thread always use the heuristic ordering rather than randomizing, but the results are not significantly improved in this case (data not shown).

5. Conclusions

We have described a technique for parallelizing dynamic programs on shared memory multiprocessors, and demonstrated its application to dynamic programming formulations of the well-known knapsack and shortest paths problems, as well as the bioinformatics problem of RNA structural alignment, and the problem of minimizing the maximum number of open stacks. Our technique is applicable to any dynamic program, since it operates on the top-down (i.e., recursive) implementation of the dynamic program, which is a direct implementation of the recurrence relation (Bellman equation) expression of the problem. This is in contrast to previous work on parallelizing dynamic programs, which focuses on vectorizing the operations in filling in the dynamic programming matrix in the bottom-up technique.

Much greater speedups (orders of magnitude) can be achieved for specific dynamic programming problems by careful analysis of the problem structure and properties of optimal solutions, in order to apply, for example, bounding techniques. Although the parallelization technique we have described results in much more modest speedups, it can be applied immediately to any dynamic program, without the need for further analysis.

For dynamic programs that are too large to implement in the bottom-up manner (filling in every entry of a dynamic programming matrix), such as open stacks, vectorization approaches are inapplicable and a method, such as the one presented here, that is applicable to the top-down implementation is required. We have shown a speedup greater than 10 times (for 32 threads) by applying this method to a state of the art dynamic programming algorithm for the minimization of the maximum number of open stacks problem.

For problems that can be practically implemented with the bottom-up technique, such as sequence alignment and RNA structure prediction and alignment, vectorization techniques have been successful. However, these techniques require careful analysis of the data dependencies in the particular problem being parallelized and result in increased complexity of the implementation. Our method, in contrast, can be applied directly to a simple implementation of the recurrence relation defining the problem as a recursive function, without any analysis of the particular problem. In these cases an array can be used to store results, without even the need for a lock-free hash table.

In order to simplify our algorithm and implementation we made the assumption that dynamic resizing of the hash table is unnecessary, and simply allocated a very large hash table at initialization. This is clearly wasteful for dynamic programming problem instances that do not require a large number of entries. A more sophisticated implementation would use a dynamically resizable lock-free hash table, such as that provided by split-ordered lists [28]. Further experiments would have to be carried out to determine if the increase in complexity and use of indirection (resulting in reduced cache efficiency) is a winning trade-off for the advantage of not allocating unnecessarily large amounts of memory.

Acknowledgments

This research made use of the Victorian Partnership for Advanced Computing HPC facility and support services. The first author is funded by an Australian Postgraduate Award. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT