15-418

Parallel Top Down Dynamic Programming through Lock-Free Hash Tables

Adrian Abedon (aabedon), Nikita Basu (nbasu)

May 5, 2022

Contents

1 Summary

We implemented a lock-free hash table and parallelized two top-down dynamic programming problems, Bellman-Ford and Knapsack. On the PSC machines, we see speedups of up to 78.17× with Bellman Ford and up to 5.95× with Knapsack.

2 Background

2.1 Dynamic programming

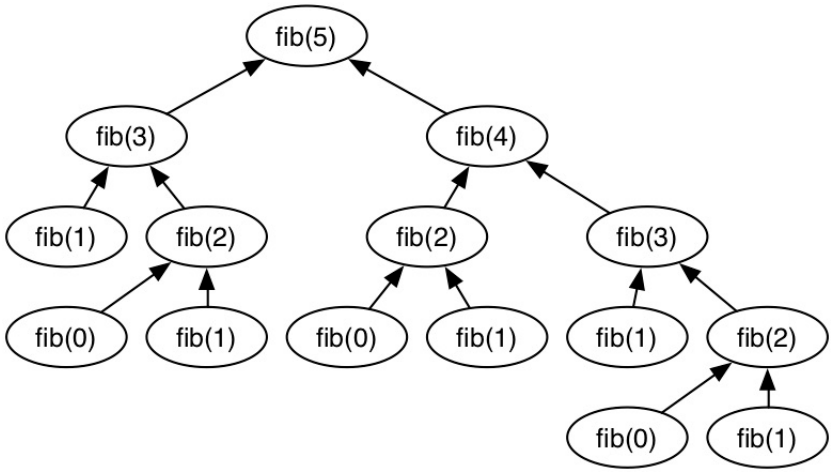

Figure 1: Top-down Fibonacci recurrence graph

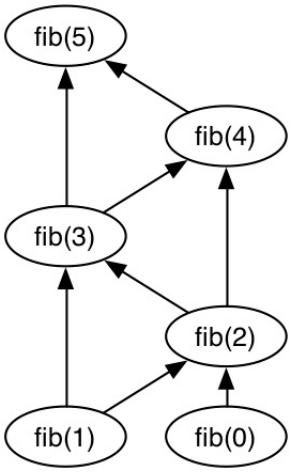

Figure 2: Fibonacci dynamic programming graph

Dynamic programming is a method for solving problems that are easily divisible into smaller subproblems. Oftentimes, many of these problems will need the result of a subproblem multiple

times throughout the course of the algorithm. Dynamic programming allows for subproblems to be computed once by storing the results in a dictionary so they can be reused later. Figure 1 shows the recurrence structure for computing the 5th fibonacci number, and Figure 2 shows how dynamic programming can reduce how many times each subproblem is computed by storing the results in a dictionary.

Dynamic programming can be implemented in two different ways. The first is a top down method which follows a mathematical recurrence to compute the whole problem starting from the root. Subproblem solutions can be stored through a technique called memoization, in which solutions are placed in a dictionary for later use so that they do not need to be recalculated. The other method for dynamic programming is the bottom up method. With this technique, the problem is computed starting from the subproblems lowest on the computation tree, and then using their result to compute the next level of subproblems, all the way up to the root. So for example, the top down approach for computing fib(5) would compute fib(4) and fib(3) first and so on. The bottom up would compute fib(0), fib(1), fib(2) and so on until fib(5) can be computed.

Top down dynamic programming recurrences are often simpler to derive but often don't present a very clear way to parallelization. This is due to the fact that subproblems are often not independent, and the recurrences do not provide any sort of indication to subproblem independence. Bottom up techniques, on the other hand, require an deep understanding of the problem to know the relationships between subproblems. Despite the extra complexity, bottom up techniques typically have a clearer path to parallelization since independent subproblems can be easily identifies and computed in parallel.

2.2 Parallelization with Lock Free Hash Tables

Hash tables are one possible data structure for the implementation of the dictionary used in dynamic programming. Most hash table implementations do not support concurrent operations. Therefore, one of our project deliverables is a concurrent hash table.

With a concurrent hash table, dynamic programming can be parallelized by having multiple threads compute subproblems in parallel, and once they have the result, they store it in the hash table. If a thread needs to compute a subproblem where the result is in the hash table,

it will simply read the result and continue, reducing the number of subproblems that will be computed. While there is a chance that some subproblems may be computed multiple times if two threads work on the subproblem at the same time, through randomization of subproblems, duplicate computations can be mostly eliminated.

3 Approach

3.1 Top Down Dynamic Programming

Top down dynamic dynamic programming is typically defined by recurrences. This makes the resulting algorithm naturally recursive. Using this approach, however, especially on very large problem inputs, can lead to stack overflows. Therefore, a non-recursive approach was devised that kept the basic stack layout of a recursive solution but entirely in the heap. Dynamically allocating memory ensured that stack size did not limit the problem size.

Figure 3 outlines the general dynamic programming algorithm with pseudocode. To keep track of which subproblems were needed, a job queue is maintained that defines the subproblem as well as the memory location where the result of the job should be written to. If a job is popped off the queue for the first time, then it is re-added to the queue along with all of the subproblems that it needs the results of. When a job is popped off the queue for the second time, all of subproblem results should be loaded in and it can compute its own result. It then writes its result out to which ever job added it to the queue.

When a subproblem is being computed, a table is first checked to see whether the subproblem was previously computed. If the subproblem's entry already exists, then it just returns the value in that entry. Otherwise, it will compute the subproblem and add the entry to the table with the resulting value. This technique is called memoization and means subproblems are not computed more than once.

3.2 Parallelizing Top Down Dynamic Programming

Top down dynamic programming does not have an obvious path to parallelization. The recurrences that usually define a top down dynamic programming algorithm do not specify which problems are independent. Therefore, another approach to parallelization is needed.

```
subproblem_result dp_algo(table H, problem p)
{
    subproblem_result final_solution;
    // compute problem p and have result written to final_solution
    add_job(p, &final_solution);

    while(!job_queue_empty())
    {
        problem p, subproblem_result *output = job_peek();
        k = EMPTY;
        if(second_time_popped(p))
        {
            // subproblems have been calculated and written to rs
            subproblem_result rs[] = get_results(p);
            k = compute(p, rs);
            insert(H, p, k);
        }
        else if (base_case(p))
        {
            k = compute(p);
            insert(H, p, k);
        }
        else
        {
            k = lookup(H, key);
        }

        if(k != EMPTY) {
            // write output to parent
            *output = k;
            job_pop();
            continue;
        }

        problem ps[] = get_subproblems(p);
        subproblem_result rs[];
        /* compute subproblems in random order and have results
         * written to entries in rs */
        add_jobs_randomly(ps, rs);
    }

    return final_solution;
}
```

Figure 3: Non-recursive Parallel top down dynamic programming pseudocode

Our approach, inspired by a previous paper, is to have multiple threads compute the same recurrence starting from the root [4]. So in Figure 3, all of the threads would call `dp_algo` at the same time and they all share the same table `H`. If a step requires the value of subproblems, the subproblems are added to the job queue in a random order. The threads then share a single lock free hash table where the result of the subproblems are added. This approach greatly simplifies parallelizing top down dynamic programming. While some subproblems are computed by multiple threads, the randomization ensures that most subproblems are computed once and the result is shared among all the threads so that it does not have to be computed again.

3.3 Lock Free Open-Addressing Linear Probing Hash Table

3.3.1 Motivation & Design

We utilize a hash table to perform the memoization that's required within dynamic programming problems. Since multiple threads are concurrently accessing the hash table to insert/lookup results for their subproblems, it is necessary to have a thread-safe hash table. One way of implementing this would be to lock/unlock the hash table each time a thread tries to access it. However, as threads will be simultaneously attempting to access the hash table, the lock for the hash table will have high contention which leads to poor performance. Thus, an implementation of a lock free hash table would be better performance-wise.

There are multiple implementations of hash tables to choose from, such as open-addressing with linear/quadratic probing, or open-addressing with double hashing, or separate chaining implemented through various data structures such as self-balancing binary search trees or linked lists. We chose to implement an open-addressing hash table with linear probing as we can ensure all allocation occurs before the execution of the dynamic programming problem and that probing is optimized for caching.

Hash tables often include a resizing/rehashing feature where when the load factor exceeds the desired load factor, the table resizes to minimize the probability of collision and ensure that all keys can fit into the table. However, this operation is very expensive as it requires memory allocation/initialization, and rehashing all keys from the previous hash table. We did not want this operation to occur in the middle of the execution the dynamic programming problem as the resizing/rehashing would pause each thread's execution until the new hash table is ready.

Thus, we decided to initialize the table with a large enough capacity so that resizing/rehashing would be unnecessary.

3.3.2 Implementation Details

For our implementation of a lock free hash table, we utilized C++'s atomic library. More specifically, we used the `atomic` type for the keys in the hash table to atomically perform a Compare-And-Swap with the key with `compare_exchange_strong`. The table was implemented as an array of entries, structs of type `entry_t`. An `entry_t` struct contains a key of type `atomic<int>` and a value of type `int`.

The hash table interface supports initialization of a hash table, looking up the value of a key, inserting a key-value pair, and freeing the table. Of the interface functions, the most interesting function is `ht_insert` for which the pseudocode is presented in Figure 4. The `insert` operation in particular was challenging to program correctly in a concurrent setting. Ensuring the two conditions listed below is necessary for correctness.

1. **Two threads cannot insert into the same entry in the table simultaneously.**

   To prevent this, a `compare_exchange_strong` is used for storing a key in the table where each thread tries to insert a key and expects that `EMPTY` was stored. If the `compare_exchange_strong` fails, i.e. the entry was not `EMPTY`, then another thread succeeded and wrote their key to that entry before. The thread that failed to insert will linearly probe to find the next empty slot. The linear probing occurs in the function `get_index_entry`.

2. **Two threads inserting the same key at the same time should not create duplicate entries in the table.**

   To avoid this, when `compare_exchange_strong` inevitably fails for one thread, `get_index_entry` linearly probes from the original hashed index and checks if any of the keys in the entries match the key that's being inserted. If so, then that entry is returned to `ht_insert` and the value is updated accordingly.

```
int get_index_entry(H, key)
    // Get entry at original hashed index
    i = hash_key(H, k, H->capacity)
    entry = H->table[i]
    probes = 0

    // Linearly probe until an entry has our key or is EMPTY
    key_in_table = entry->key
    while (probes < H->capacity - 1 && key_in_table != key
                                    && key_in_table != EMPTY)
        probes += 1
        i = (i + 1) & (H->capacity-1)
        entry = H->table[i]
        key_in_table = entry->key

    if (probes >= H->capacity-1)
        return TABLE_FULL
    return i

void ht_insert(H, key, value)
    while (true)
        // Get entry that has our key or is EMPTY
        i = get_index_entry(H, key)
        entry = H->table[i]
        key_in_table = entry->key

        // Entry has our key --> update value
        if (key_in_table == key)
            entry->value = value
            return

        // Entry is EMPTY --> try to store key
        if (key_in_table == EMPTY)
            if (atomic_compare_exchange_strong(&(entry->key), EMPTY, key))
                entry->value = value
                return
        // Swap failed --> try again with a new entry
```

Figure 4: Pseudocode for insertion into a hash table

3.3.3 Miscellaneous Performance Details

The hash table is an expensive data structure to utilize as it is extremely cache unfriendly. With a good hash function, the entries in the table will be relatively spread out which leads cache misses for each call to `ht_insert` and `ht_lookup`.

To optimize the hash table, we utilized a MurmurHash function, which is optimized for speed and distribution of hash values [1]. Additionally, we established the capacity to be a power of two which allows us to compute `hash_value % capacity` by performing `hash_value & (capacity-1)`. This is beneficial as the `&` operation is much faster than the `%` operation, and this computation is performed every time a hash value is computed, i.e. for all lookups and insertions.

Moreover, if the key already exists in the hash table, insertion is cheap as we directly modify the value in the table without a `compare_exchange_strong`. This is still correct as once a key is inserted into the table with `compare_exchange_strong`, the key is never modified. Thus, an entry with a matching key can be freely and cheaply modified.

3.4 Bellman-Ford Shortest Path

3.4.1 Definition

Given a directed graph G with weighted edges, determine the minimum length path from the source vertex s to the destination vertex v.

3.4.2 Recurrence

Let $D(v, k)$ be the minimum length path from the source vertex s to the destination vertex v in k edges. Let $N(v)$ be the set of neighbors for the vertex v. The Bellman-Ford Shortest Path problem can be solved by the following recurrence [3].

$$D(v,k) = \begin{cases} 0 & \text{if } k = 0 \text{ and } v = s \\ \infty & \text{if } k = 0 \text{ and } v \neq s \\ \min\{D(v,k-1), \min\limits_{x \in N(v)} D(x,k-1) + len(x,v)\} & \text{otherwise} \end{cases}$$

If there are 0 edges left to use, then if the current vertex is the source vertex, $v = s$, then the minimum length path is 0. If there are 0 edges left to use, and the current vertex is not the source vertex, then there is no applicable path and the length is infinite. Otherwise, the shortest length path is the minimum of the shortest length path to v with $k-1$ edges and the minimum of the shortest length paths to $x + len(x, v)$ with $k-1$ edges, where x is a neighbor of v.

3.5 Knapsack Problem

3.5.1 Definition

Given a knapsack with a finite capacity, and a set of items with weights and values, determine the maximum total value that the knapsack can hold such that the sum of weights of the items is less than or equal to the capacity.

3.5.2 Recurrence

Let $V(k, B)$ represent the maximum total value given a set of k items and a knapsack with capacity B. w_i and v_i represent the weight and value of item i respectively. The Knapsack problem can be solved by the following recurrence [2].

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } w_k > B \\ \max\{V(k-1,B), V(k-1,B-w_k) + v_k\} & \text{otherwise} \end{cases}$$

If there are no items, when $k = 0$, then the maximum total value that we can attain (regardless of capacity) is 0. If the weight of the k-th item is too heavy for the knapsack, when $w_k > B$, then we cannot carry the k-th item and the maximum total value is equivalent to the maximum total value of the set without k, $V(k-1, B)$. If we are able to carry the k-th item, then the maximum total value of this set of k items is equivalent to the maximum of the largest total value with the k-th item in the knapsack and without the k-th item in the knapsack, $\max\{V(k-1,B), V(k-1,B-w_k) + v_k\}$.

3.6 Generating Test Cases & Ensuring Correctness

Test cases are randomly generated. For the Knapsack problem, the test cases can be scaled either by increasing the capacity of the sack or increasing the number of objects that are considered. Object values and weights are randomized for a given capacity and object count.

Bellman-Ford test cases can vary both the number of vertices as well as the number of edges. We generated random graphs of a given vertex count where 75% of the edges are filled. This ensures a path between any two vertices is likely to exist. The weights on the edges are randomized as well.

For both the Knapsack and Bellman-Ford problems, a sequential reference algorithm is used to check the correctness of our parallel solution.

4 Results

To measure the performance of our parallel algorithm, we measured the speedup on different problem sizes on the PSC machines relatively to the single-threaded performance of our application. This allowed us to determine how well our solution scaled with up to 128 CPU cores, as well as see how problem size affected speedup. We were able to reach our 100% goal of having benchmarks for Bellman-Ford and surpassed our goal of 15× speedup with 32 cores.

4.1 Analysis of Bellman-Ford

We measured Bellman-Ford's performance against 4 different workloads, where each workload was a randomly generated graph. These graphs can be found in the folder `tests/bellman`. The sizes of each workload are described below.

Number of Vertices	Number of Edges	Reference Time Single Thread
200	29850	1.80 s
400	119700	14.53 s
1000	748966	219.75 s
2000	2997650	1,825.73 s

Figure 5: Bellman-Ford test sizes and times on a single processor.

Below, Figure 6 demonstrates the speedup we observed on the different workloads. Figure 5 denotes the single-threaded performance for each input.

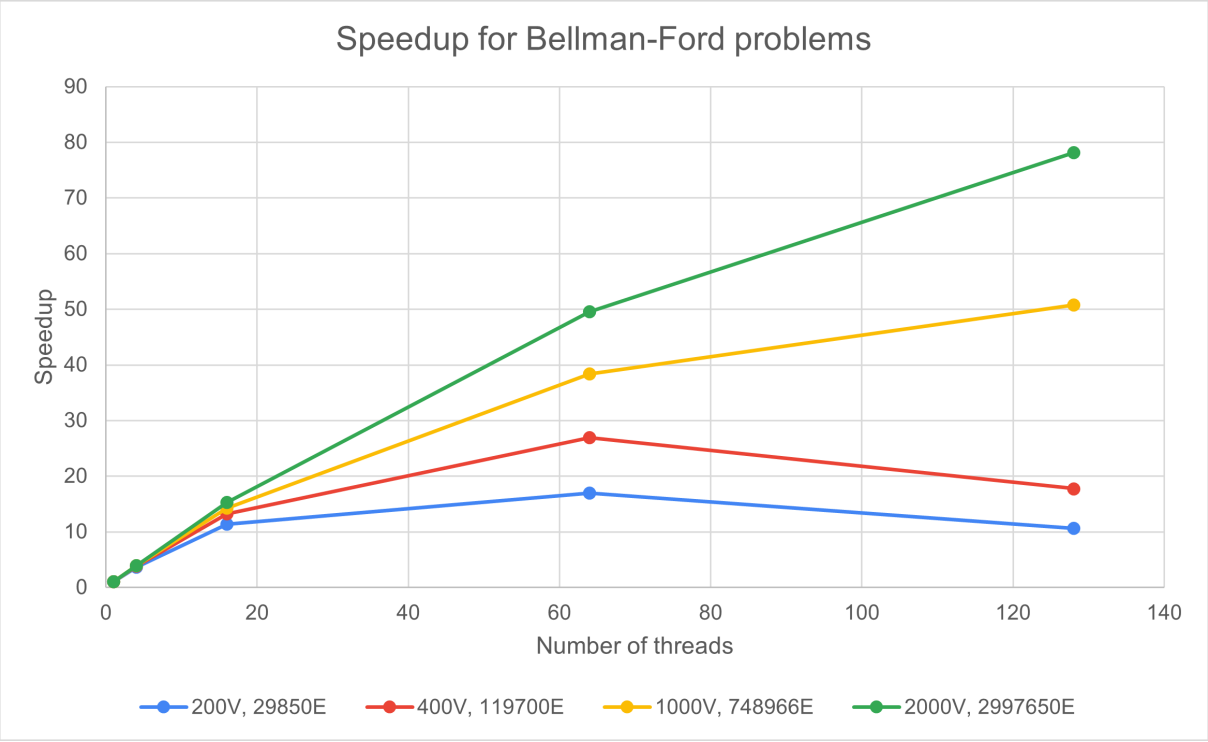

Figure 6: Speedup for Bellman-Ford Problems

The speed up that we observed for the Bellman-Ford dynamic programming algorithm improved across all thread counts as the problem size increased. Each subproblem in Bellman-Ford has $O(n)$ subproblems where n is the number of vertices. Therefore, as the number of vertices increased, the complexity of each subproblem increased. Thus, the overhead of the job system and the hash table was less compared to the complexity of the actual computation of each subproblem. With each thread spending more time spent doing useful work computing the subproblems, the observed speedup improved.

4.2 Analysis of Knapsack

We measured Knapsack's performance against 3 different workloads, where each workload had a different number of objects and capacity, and included a randomly generated set of items with weights and values. These collections can be found in the folder `tests/knapsack`. The sizes of each workload are described below.

Number of Objects	Capacity	Reference Time Single Thread
100	400	0.02s
1000	4000	3.39s
2000	8000	14.04s

Figure 7: Knapsack test sizes and times on a single processor.

Below, Figure 8 demonstrates the speedup we observed on the different workloads. Figure 7 denotes the single-threaded performance for each input.

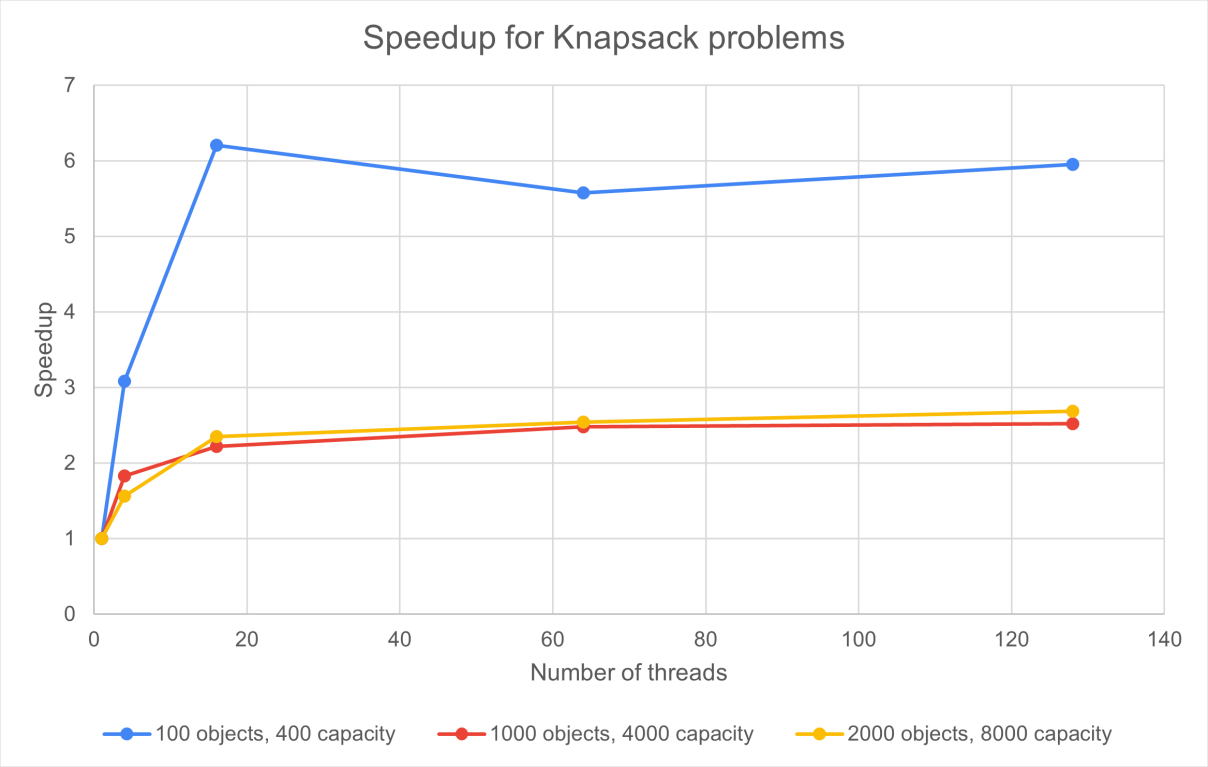

Figure 8: Speedup for Knapsack Problems

We observed generally decreasing speedup with larger workloads and stagnating speedups with higher thread counts. For the smallest workload, we initialized the capacity of the hash table such that it would fit into the 256MB cache the PSC machines have. For the larger workloads, we were not able to do this since the working set was larger than the cache. This is why we see the best speedups for the smallest workload. Additionally, as each subproblem has only O(1) complexity, the overhead of the job system and complexity of the hash table likely outweighed the computational requirements.

4.3 Comparison between Knapsack and Bellman-Ford

We observed strikingly different speedups with Knapsack and Bellman-Ford. For Knapsack, we observed speedups up to 5.95× with 128 threads, and for Bellman-Ford, up to 78.17× with 128 threads. We theorize that this is due to the difference in computational intensity between the two problems. For Knapsack, the complexity of the recursive case is $O(1)$ whereas in Bellman-Ford, the complexity of the recursive case is $O(n)$ where n is the number of vertices in the graph. This means that Bellman-Ford spends more time computing the subproblems compared to the Knapsack algorithm. A larger fraction of time spent computing each subproblem compared to operations in the hash table and the job queue leads to better speedup for Bellman-Ford.

5 Future Work

Possible extensions on this work would be to analyze different implementations of hash tables. We chose a lock free hash table, but other concurrent implementations such as fine-grained locking or coarse-grained locking may be faster due to the sparsely located entries within the table. Additionally, a different scheduling of jobs for threads, particularly, one that accounts for cache locality, may be better than a random assignment. Furthermore, additional dynamic programming problems, such as Traveling Salesperson or Floyd-Warshall, may be analyzed to fully understand the benefits that our parallel dynamic programming application provides.

6 Distribution of Work

There was an even 50-50 split of the work.

1. Project Proposal - Adrian and Nikita
2. Hash Table Implementation - Nikita
3. Dynamic Programming Application - Adrian
4. Generating Test Cases - Adrian and Nikita
5. Benchmarking Workloads - Adrian and Nikita
6. Project Report & Video - Adrian and Nikita

References

[1] Jeff Preshing. The world's simplest lock-free hash table.

[2] Danny Sleator. Dynamic programming 1 - 15-451 notes.

[3] Danny Sleator. Dynamic programming 2 - 15-451 notes.

[4] Alex Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing*, 70(8):839–848, 2010.