A Lattice Linear Predicate Parallel Algorithm for the Dynamic Programming Problems

Vijay K. Garg
The University of Texas at Austin
Austin, Texas, USA
garg@ece.utexas.edu

ABSTRACT

It has been shown that the parallel Lattice Linear Predicate (LLP) algorithm solves many combinatorial optimization problems such as the shortest path problem, the stable marriage problem and the market clearing price problem. In this paper, we give the parallel LLP algorithm for many dynamic programming problems. In particular, we show that the LLP algorithm solves the longest subsequence problem, the optimal binary search tree problem, and the knapsack problem. Furthermore, the algorithm can be used to solve the constrained versions of these problems so long as the constraints are lattice linear. The parallel LLP algorithm works correctly in a distributed system setting where a processor may use an older value of a variable stored at a different processor.

CCS CONCEPTS

• **Theory of computation → Parallel algorithms**;

KEYWORDS

distributive lattices; predicate detection; optimization problems

ACM Reference Format:
Vijay K. Garg. 2022. A Lattice Linear Predicate Parallel Algorithm for the Dynamic Programming Problems. In *23rd International Conference on Distributed Computing and Networking (ICDCN 2022), January 4–7, 2022, Delhi, AA, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3491003.3491019

1 INTRODUCTION

It has been shown that the Lattice Linear Predicate (LLP) algorithm solves many combinatorial optimization problems such as the shortest path problem, the stable marriage problem and the market clearing price problem [8]. In this paper, we build on the work in [8] to show that many problems that can be solved using dynamic programming [2] can also be solved in *parallel* and *distributed* fashion using the LLP algorithm. Dynamic programming is applicable to problems where it is easy to set up a recurrence relation such that the solution of the problem can be derived from the solutions of problems with smaller sizes. One can solve the problem using recursion; however, recursion may result in many duplicate computations. By using memoization, we can avoid recomputing previously computed values. We assume that the problem is solved using dynamic programming with such bottom-up approach in this paper.

The LLP algorithm views solving a problem as searching for an element in a finite distributive lattice [3, 4, 7] that satisfies a given predicate B. The predicate is required to be closed under the operation of meet (or, equivalently lattice-linear). For all the problems considered in the paper, the longest subsequence problem, the optimal binary search tree problem and the Knapsack problem, the predicate is indeed closed under meets. Any finite distributive lattice can be equivalently characterized by a finite poset of its join-irreducibles from Birkhoff's theorem [3, 4]. The LLP algorithm works on the finite poset in parallel to find the least element in the distributive lattice that satisfies the given predicate. It starts with the bottom element of the lattice and marches towards the top element of the lattice in a *parallel* fashion by advancing on any chain of the poset for which the current element is *forbidden*. An advantage of the LLP algorithm is that the algorithm stays correct even when a processor ends up using an older value of the variable stored at another processor. The only requirement is that the value updated at any processor is eventually communicated at other processors.

There are also some key differences between dynamic programming (the bottom-up approach) and the LLP algorithm. The usual dynamic programming problem seeks a structure that minimizes (or maximizes) some scalar. For example, the longest subsequence problem asks for the subsequence in an array $A[1..n]$ that maximizes the sum. In contrast, the LLP algorithm seeks to minimize or maximize a *vector*. In the longest subsequence problem with the LLP approach, we are interested in the longest subsequence in the array $A[1..i]$ for each $i \leq n$ that ends at index i. Thus, instead of asking for a scalar, we ask for the vector of size n. We get an array $G[1..n]$ and the solution to the original problem is just the maximum value in the array G. Similarly, the optimal binary search tree problem [13] asks for the construction of an optimal binary search tree on n symbols such that each symbol i has probability p_i of being searched. Our goal is to find the binary search tree that minimizes the expected cost of search in the tree. The LLP problem seeks the optimal binary search tree for all ranges $i \dots j$ instead of just one range $1..n$. Finally, the knapsack problem [9, 11] asks for the maximum valued subset of items that can be fit in a knapsack such that the profit is maximized and the total weight of the knapsack is at most W. The LLP problem seeks the maximum profit obtained by choosing items from $1..i$ and the total weight from $1..W$. In all these problems, traditionally we are seeking a single structure that optimizes a single scalar; whereas the LLP algorithm asks for a vector. It turns out that that in asking for an optimal *vector* instead of an optimal *scalar*, we do not lose much since the existing solutions also end up finding the optimal solutions for the subproblems. The LLP algorithm returns a vector G such that $G[i]$ is optimal for i.

The second difference between dynamic programming and the LLP algorithm is in terms of parallelism. The dynamic programming solution does not explicitly refers to parallelism in the problem. The LLP algorithm has an explicit notion of parallelism. The solution uses an array G for all problems and the algorithm requires the components of G to be advanced whenever they are found to be *forbidden.* If $G[i]$ is forbidden for multiple values of i, then $G[i]$ can be advanced for all those values in parallel.

The third difference between dynamic programming and the LLP algorithm is in terms of synchronization required during parallel and distributed execution of the algorithm. In case of dynamic programming, if the recursive formulas are evaluated in parallel it is assumed that the values used are correct. In contrast, suppose that we check for $G[i]$ and $G[j]$ to be forbidden concurrently such that $G[i]$ ends up using an old value of $G[j]$, the LLP algorithm is still correct. The only requirement we have for parallelism is that when $G[i]$ uses a value of $G[j]$, it should either be the most recent value of $G[j]$ or some prior value. A processor that is responsible for keeping $G[i]$ may get an old value from $G[j]$ in a parallel setting when it gets this value from a cache. In a message passing system, it may get the old value of $G[j]$ if the message to update $G[j]$ has not yet arrived at the processor with $G[i]$. Thus, LLP algorithms are naturally distributed with little synchronization overhead.

The fourth difference between dynamic programming and the LLP algorithm is that we can use the LLP algorithm to solve a constrained version of the problem, so long as the constraint itself is lattice-linear. Suppose that we are interested in the longest subsequence such that successive elements differ by at least 2. It can be (easily) shown that this constraint is lattice-linear. Hence, the LLP algorithm is applicable because we are searching for an element that satisfies a conjunction of two lattice-linear predicates. Since the set of lattice-linear predicates is closed under conjunction, the resulting predicate is also lattice-linear and the LLP algorithm is applicable. Similarly, the predicate that the symbol i is not a parent of symbol j is lattice-linear and the constrained optimal binary search tree algorithm returns the optimal tree that satisfies the given constraint. In the Knapsack problem, it is easy to solve the problem with the additional constraint that if the item x is included in the Knapsack, then the item y is also included. The constrained versions of these problems are not discussed in the literature and are of independent interest.

We note here that our goal is not to improve the time or work complexity of the algorithms, but to provide a single parallel (and distributed) algorithm that solve all of these problems and their constrained versions. For the increasing subsequence problem, the standard dynamic programming approach takes $O(n^2)$ time where n is the size of the array. The parallel LLP algorithm takes $O(\Delta \log n)$ time where Δ is the longest increasing subsequence in the array. The parallel algorithms specific to the longest increasing subsequence problem are discussed in [6, 14, 16]. The optimal binary search tree problem takes $O(n^3)$ time using the dynamic programming approach where n is the number of symbols. The parallel LLP algorithm takes $O(n \log n)$ with $O(n^2)$ processors. The parallel algorithms for the optimal binary search tree are described in [1, 10]. The knapsack problem takes $O(nW)$ time using the dynamic programming approach where n is the number of items and W is the maximum weight of any item. The parallel LLP algorithm takes $O(n)$ time with $O(W)$ processors. Parallel algorithms specific to the knapsack problem are described in [12, 15, 18]. We note here that each of the problems has multiple parallel algorithms. Our goal is to give a *single* algorithm for all these problems. This is the first parallel and distributed algorithm that solves all these problems. Furthermore, the parallel algorithm we propose has no synchronization overhead. The algorithm works correctly in a distributed setting where a processor may use an older value of a variable stored at other nodes. We are unaware of such algorithms for these problems.

The reader is referred to [8] for the background information on the LLP Algorithm.

2 LONGEST INCREASING SUBSEQUENCES

We are given an integer array as input. For simplicity, we assume that all entries are distinct. Our goal is find for each index i the length of the longest increasing sequence that ends at i. For example, suppose the array A is `{35 38 27 45 32}`. Then, the desired output is `{1 2 1 3 2}`. The corresponding longest increasing subsequences are: `(35), (35, 38), (27), (35, 38, 45), (27, 32).`

We can define a graph H with indices as vertices. For this example, we have five vertices numbered v_1 to v_5. We draw an edge from v_i to v_j if i is less than j and $A[i]$ is also less than $A[j]$. This graph is clearly acyclic as an edge can only go from a lower index to a higher index. We use $pre(j)$ to be the set of indices which have an incoming edge to j. The length of the longest increasing subsequence ending at index j is identical to the length of the longest path ending at j.

To solve the problem using LLP, we model it as a search for the smallest vector G that satisfies the constraint $B \equiv \forall j : G[j] \geq 1 \wedge \forall j : G[j] \geq \max\{G[i] + 1 \mid i \in pre(j)\}$. To understand B, we first consider a stronger predicate $B_* = (G[1] = 1) \wedge \forall j : G[j] = \max\{1, \max\{G[i] + 1 \mid i \in pre(j)\}\}$. The interpretation of $G[j]$ in B_* is that it is the length of the longest path that ends in j. Thus, in the longest increasing subsequence problem we are searching for the vector that satisfies the predicate B_*. Instead of searching for an element in the lattice that satisfies B_*, we search for the least element in the lattice that satisfies B. This allows us to solve for the constrained version of the problem in which we are searching for an element that satisfies an additional lattice-linear constraint.

The underlying lattice we consider is that of all vectors of natural numbers less than or equal to the maximum element in the lattice. A vector in this lattice is *feasible* if it satisfies B. We first show that the constraint B is lattice-linear.

Lemma 2.1. *The constraint* $B \equiv (\forall j : G[j] \geq 1) \wedge (\forall j : G[j] \geq \max\{G[i] + 1 \mid i \in pre(j)\})$ *is lattice-linear.*

Proof. Since the predicate B is a conjunction of two predicates, it is sufficient to show that each of them is lattice-linear. The first conjunct is lattice linear because the constant function 1 is monotone. The second conjunct can be viewed as a conjunction over all

j. For a fixed j, the predicate $G[j] \geq \max\{G[i]+1 \mid i \in pre(j)$ is lattice-linear because it is a monotonic function. □

Our goal is to find the smallest vector in the lattice that satisfies B. Now, LLP algorithm can be formulated as LLP-Increasing-Subsequence.

ALGORITHM LLP-Increasing-Subsequence: Finding the Longest Increasing Subsequence.

```
$P_j$: Code for thread $j$
**input**: $A$:array of int;
**var** $G$: array[$1 \ldots n$] of int;
**init**: $G[j] = 1; pre(j) := \{i \in 1..j-1 | A[i] < A[j]\}$;
**ensure**: $G[j] \geq \max\{G[i]+1 \mid i \in pre(j)\}$;
```

This algorithm starts with all values as 1 and increases the G vector till it satisfies the constraint $G[j] \geq \max\{G[i]+1 \mid i \in pre(j)\}$.

The above algorithm, although correct, does not preclude $G[j]$ from getting updated multiple times. To ensure that no $G[j]$ is updated more than once, we introduce a boolean $fixed$ for each index such that we update $G[j]$ only when it is not fixed and all its predecessors are fixed. With this change, our algorithm becomes LLP2-Increasing-Subsequence.

ALGORITHM LLP2-Increasing-Subsequence: Finding the Longest Increasing Subsequence.

```
$P_j$: Code for thread $j$
**input**: $A$:array of int;
**var** $G$: array[$1 \ldots n$] of int; $fixed$: array[$1 \ldots n$] of boolean;
**init**: $G[j] = 1; fixed[j] := false$;
    $pre(j) := \{i \in 1..j-1 | A[i] < A[j]\}$;
**forbidden**: $\neg fixed[j] \wedge (\forall i \in pre(j) : fixed[i])$;
    **advance**: $G[j] := \max\{G[i]+1 \mid i \in pre(j)\}$;
        $fixed[j] := true$;
```

Let us now analyze the complexity of the algorithm. The sequential complexity is simple because we can maintain the list of all vertices that are forbidden because all its predecessors are fixed. Once we have processed a vertex, we never process it again. This is similar to a sequential algorithm of topological sort. In this case, we examine a vertex exactly once only after all its predecessors are fixed. The time complexity of this algorithm is $O(n^2)$.

For the parallel time complexity, assume that we have n^2 processors available. Then, in time $O(\log n)$, one can determine whether the vertex is forbidden and advance it to the correct value if it is forbidden. This is because for every j, we simply need to check that all vertices in $pre(j)$ are fixed and j is not fixed. By using a parallel *reduce* operation, we can check in $O(\log n)$ time whether j is forbidden. If the longest path in the graph H is Δ, then the algorithm takes $O(\Delta \log n)$ time.

Now, let us consider the situation where each thread j writes the value of $fixed[j]$ and $G[j]$ without using any synchronization. If any thread j reads the old value of $fixed[i]$ for some i in $pre(j)$, it will not update $fixed[j]$ at that point. Eventually, it will read the correct value of $fixed[i]$, and perform *advance*. We do assume in this version that if a process reads $fixed[i]$ as true, then it reads the correct value of $G[i]$, because $fixed[i]$ is updated after $G[i]$. Consequently, we get the following result.

Lemma 2.2. *There exists a parallel algorithm for the longest increasing subsequence problem which uses just read-write atomicity and solves the problem in $O(\Delta \log n)$ time.*

We note here that the longest increasing subsequence problem is not known to be in class NC or P-complete. The best sequential complexity of the problem is $O(n \log n)$. The parallel complexity is shown to be $O(n \log n/p)$ with p processors when $1 < p < n/m^2$ where m is the number of decreasing sequences in the array[16].

We now add lattice-linear constraints to the problem. Instead of the longest increasing subsequence, we may be interested in the longest increasing subsequence that satisfies an additional predicate.

Lemma 2.3. *All the following predicates are lattice linear.*

(1) *For any j, $G[j]$ is greater than or equal to the longest increasing subsequence of odd integers ending at j.*
(2) *$G[j]$ is greater than or equal to the longest increasing subsequence such that j^{th} element in the subsequence exceeds $(j-1)^{th}$ element by at least k.*

Proof. (1) Since lattice-linear predicates are closed under conjunction, it is sufficient to focus on a fixed j. If $G[j]$ is less than the length of the longest increasing subsequence of odd integers ending at j, then the index j is forbidden. Unless j is increased the predicate can never become true.
(2) We view this predicate as redrawing the directed graph H such that we draw an edge from v_i to v_j if i is less than j and $A[i]+k$ is less than or equal to $A[j]$.

□

3 OPTIMAL BINARY SEARCH TREE

Suppose that we have a fixed set of n symbols called *keys* with some associated information called *values*. Our goal is to build a dictionary based on binary search tree out of these symbols. The dictionary supports a single operation search which returns the value associated with the the given key. We are also given the frequency of each symbol as the argument for the search query. The cost of any search for a given key is given by the length of the path from the root of the binary search tree to the node containing that key. Given any binary search tree, we can compute the total cost of the tree for all searches. We would like to build the binary search tree with the least cost.

Let the frequency of key i being searched is p_i. We assume that the keys are sorted in increasing order of p_i. Our algorithm is based on building progressively bigger binary search trees. The main idea is as follows. Suppose symbol k is the root of an optimal binary search tree for symbols in the range $[i..j]$. The root k divides the range into three parts – the range of indices strictly less than k, the index k, and the range of indices strictly greater than k. The left or the right range may be empty. Then, the left subtree and the right subtree must themselves be optimal for their respective ranges. Let $G[i,j]$ denote the least cost of any binary search tree built from symbols in the range $i..j$. We use the symbol $s(i,j)$ as the sum of all frequencies from the symbol i to j, i.e., $s(i,j) = \sum_{k=i}^{j} p_k$. For convenience, we let $s(i,j)$ equal to 0 whenever $i > j$, i.e., the range is empty.

We now define a lattice linear constraint on $G[i,j]$. Let $i \le k \le j$. Consider the cost of the optimal tree such that symbol k is at the root. The cost has three components: the cost of the left subtree if any, the cost of the search ending at this node itself and the cost of search in the right subtree. The cost of the left subtree is $G[i,k-1]+s(i,k-1)$ whenever $i < k$. The cost of the node itself is $s(k,k)$. The cost of the right subtree is $G[k+1,j]+s(k+1,j)$. Combining these expressions, we get $G[i,j] = \min_{i \le k \le j}(G[i,k-1] + s(i,j) + G[k+1,j])$.

This is also the least value of $G[i,j]$ such that
$G[i,j] \ge \min_{i \le k \le j}(G[i,k-1] + s(i,j) + G[k+1,j])$.

We now show that the above predicate is lattice-linear.

Lemma 3.1. *The constraint $B \equiv \forall i,j : G[i,j] \ge \min_{i \le k \le j}(G[i,k-1] + s(i,j) + G[k+1,j])$ is lattice-linear.*

Proof. Suppose that B is false, i.e.,
$\exists i,j : G[i,j] < \min_{i \le k \le j}(G[i,k-1] + s(i,j) + G[k+1,j])$. This means that there exists i,j,k with $i \le k \le j$ such that $G[i,j] < (G[i,k-1] + s(i,j) + G[k+1,j])$. This means the the index (i,j) is forbidden and unless $G[i,j]$ is increased, the predicate B can never become true irrespective of how other components of G are increased. □

We now have our LLP-based algorithm for Optimal Binary Search Tree as Algorithm LLP-OptimalBinarySearchTree. The program has a single variable G. It is initialized so that $G[i,i]$ equals $p[i]$ and $G[i,j]$ equals zero whenever i is not equal to j. The algorithm advances $G[i,j]$ whenever it is smaller than $\min_{i \le k \le j} G[i,k-1] + s(i,j) + G[k+1,j]$. In Algorithm LLP-OptimalBinarySearchTree, we have used the **always** clause as a macro that uses $s(i,j)$ as a short form for $\sum_{k=i}^{j} p[k]$.

ALGORITHM LLP-OptimalBinarySearchTree: Finding An Optimal Binary Search Tree

```
$P_{i,j}$: Code for thread $(i,j)$
input: $p$:array of real;// frequency of each symbol
init: $G[i,j] = 0 \ \forall i \ne j$; $G[i,i] = p[i]$;
always: $s(i,j) = \sum_{k=i}^{j} p[k]$
ensure: $G[i,j] \ge \min_{i \le k < j} G[i,k-1] + s(i,j) + G[k+1,j]$
priority: $(j - i)$
```

Although, the above algorithm will give us correct answers, it is not efficient as it may update $G[i,j]$ before $G[i,k]$ and $G[k,j]$ for $i \le k \le j$ have stabilized. However, the following scheduling strategy ensures that we update $G[i,j]$ at most once. We check for whether $G[i,j]$ is forbidden in the order of $j - i$. Hence, initially all $G[i,j]$ such that $j = i + 1$ are updated. This is followed by all $G[i,j]$ such that $j = i + 2$, and so on. We capture this scheduling strategy with the **priority** statement. We pick $G[i,j]$ to update such that $(j - i)$ have minimal values. Of course, our goal is to compute $G[1,n]$. With the above strategy of updating $G[i,j]$, we get that $G[i,j]$ is updated at most once. Since there are $O(n^2)$ possible values of $G[i,j]$ and each takes $O(n)$ work to update, we get the work complexity of $O(n^3)$. On a CREW PRAM, we can compute all i,j with the fixed difference in parallel. By using $O(\log n)$ span algorithm to compute min, we get the parallel time complexity as $O(n \log n)$. Thus, we have the following result.

Lemma 3.2. *There exists a parallel algorithm for the optimal binary search tree problem which uses just read-write atomicity and solves it in $O(n \log n)$ parallel time using $O(n^2)$ processors.*

We note here that [10] gives an algorithm that takes $O(\sqrt{n} \log n)$ time; however, it uses $O(n^{3.5}/\log n)$ processors on a CREW PRAM.

We now consider the constrained versions of the problem.

Lemma 3.3. *All the following predicates are lattice linear.*

1. *Key x is not a parent for any key.*
2. *The difference in the sizes of the left subtree and the right subtree is at most* 1.

Proof.
1. This requirement changes the ensure predicate to $G[i,j] \ge \min_{i \le k \le j, k \ne x} G[i,k-1] + s(i,j) + G[k+1,j]$. The right hand side of the constraint continues to be monotonic and therefore it is lattice linear.
2. This requirement changes the ensure predicate to $G[i,j] \ge \min_{i \le k \le j, |k-1-i, j-k-1| \le 1} G[i,k-1] + s(i,j) + G[k+1,j]$. This change simply restricts the values of k, and the right hand side continues to be monotonic.

□

Remark: A problem very similar to the optimal Binary Search tree problem is that of constructing an optimal way of multiplying a chain of matrices. Since matrix multiplication is associative, the product of matrices $(M_1 * M_2) * M_3$ is equal to $M_1 * (M_2 * M_3)$. However, depending upon the dimensions of the matrices, the computational effort may be different. We can view any evaluation of a chain as a binary tree where the intermediate notes are the multiplication operation and the leaves are the matrices themselves.

4 KNAPSACK PROBLEM

We are given n items with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$. We are also given a knapsack that has a capacity of W. Our goal is to determine the subset of items that can be carried in the knapsack and that maximizes the total value. The standard dynamic programming solution is based on memoization of the following dynamic programming formulation [5, 17]. Let $G[i,w]$ be the maximum value that can be obtained by picking items from $1..i$ with the capacity constraint of w. Then, $G[i,w] = max(G[i-1, w-w_i] + v_i, G[i-1,w])$. The first argument of the max function corresponds to the case when the item i is included in the optimal set from $1..i$, and the second argument corresponds to the case when the item i is not included and hence the entire capacity can be used for the items from $1..i-1$. If $w_i > w$, then the item i can never be in the knapsack and can be skipped. The base cases are simple. The value of $G[0,w]$ and $G[i,0]$ is zero for all w and i. Our goal is to find $G[n,W]$. By filling up the two dimensional array G for all values of $0 \le i \le n$ and $0 \le w \le W$, we get an algorithm with time complexity $O(nW)$.

We can model this problem using lattice-linear predicates as follows. We model the feasibility as $G[i,w] \ge \max(G[i-1, w-w_i] + v_i, G[i-1,w])$ for all $i, w > 0$ and $w_i \le w$. Also, $G[i,w] = 0$

if $i = 0$ or $w = 0$. Our goal is to find the minimum vector G that satisfies feasibility.

Lemma 4.1. *The constraint $B \equiv \forall i, w : G[i,w] \geq \max(G[i-1, w-w_i]+v_i, G[i-1,w])$ for $w_i \leq w$ is lattice-linear.*

Proof. If the predicate B is false, there exists i and w such that $G[i,w] < \max(G[i-1,w-w_i]+v_i, G[i-1,w])$. The value $G[i,w]$ is forbidden; unless $G[i,w]$ is increased the predicate can never become true. □

ALGORITHM LLP-Knapsack: Finding An Optimal Solution to the Knapsack Problem

```
$P_{i,j}$: Code for thread $(i,j)$
input: $w, v$:array[1..$n$] of int;// weight and value of items
var: $G$:array[$0 \ldots n, 0 \ldots W$] of int;
init: $G[i,j] = 0\ if\ (i=0) \vee (j=0)$;
ensure:
$G[i,j] \geq \max\{G[i-1,j-w_i]+v_i, G[i-1,j]\}$ if $j \geq w_i$
        $\geq G[i-1,j]$, otherwise.
```

Algorithm LLP-Knapsack updates the value of $G[i,j]$ based only on the values of $G[i-1,.]$. Furthermore, $G[i,j]$ is always at least $G[i-1,j]$. Based on this observation, we can simplify the algorithm as follows. We consider the problem of adding just one item to the knapsack given the constraint that the total weight does not exceed W. We maintain the list of all optimal configurations for each weight less than W.

ALGORITHM LLP-IncrKnapsack2: Finding An Optimal Solution to the Incremental Knapsack Problem

```
$P_j$: Code for thread $j$
input: $w, v$: int;// weight and value of the next item
    $C$: array[$0 \ldots W$] of int;
var: $G$:array[$0 \ldots W$] of int;
init: $\forall j : G[j] = C[j]$;
ensure: $G[j] \geq C[j-w]+v$ if $j \geq w$
```

The incremental algorithm can be implemented in $O(1)$ parallel time using $O(W)$ processors as shown in Fig. LLP-IncrKnapsack2. Each processor j can check whether $G[j]$ needs to be advanced.

We can now invoke the incremental Knapsack algorithm by simply looping over all items. If we had W cores, then computing $G[i,.]$ from $G[i-1,.]$ can be done in $O(1)$ giving us the span of $O(n)$.

We now add some lattice-linear constraints to the Knapsack problem. In many applications, some items may be related and the constraint $x_a \Rightarrow x_b$ means that if the item x_a is included in the Knapsack then the item x_b must also be included. Thus, the item x_a has profit of zero if x_b is not included. The item x_b has utility even without x_a but not vice-versa. Without loss of generality, we assume that all weights are strictly positive, and that index $b < a$. In the following Lemma, we use an auxiliary variable $S[i,j]$ that keeps the set of items included in $G[i,j]$ and not just the profit from those items.

Lemma 4.2. *First assume that $(i \neq a)$. Let $B(i,w) \equiv G[i,w] \geq \max(G[i-1,w-w_i]+v_i, G[i-1,w])$ for $(w_a \leq w)$ and $G[i,w] \geq G[i-1,w]$, otherwise. This predicate corresponds to any item i different from a. The value with a bag of capacity w is always greater than or equal to the choice of picking the item or not picking the item.*

Let $B(a,w) \equiv G[a,w] \geq \max(G[a-1,w-w_a]+v_a, G[a-1,w])$ if $b \in S[a-1,w-w_a] \wedge (w_a \leq w)$ and $B(a,w) \equiv G[a,w] \geq G[a-1,w]$, otherwise. Then, $B(i,w)$ is lattice-linear for all i and w.

Proof. Suppose that $B(i,w)$ is false for some i and w. Unless $G[i,w]$ is increased, it can never become true. □

5 CONCLUSIONS

In this paper, we have shown that many dynamic programming problems can be solved using a single *parallel* Lattice-Linear Predicate algorithm. The parallel algorithms described in the paper work correctly with read-write atomicity of variables without any use of *locks.*

REFERENCES

[1] Mikhail J Atallah, S Rao Kosaraju, Lawrence L Larmore, Gary L Miller, and S-H Teng. 1989. Constructing trees in parallel. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures.* 421–431.

[2] Richard Bellman. 1952. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America* 38, 8 (1952), 716.

[3] G. Birkhoff. 1967. *Lattice Theory.* Providence, R.I. third edition.

[4] B. A. Davey and H. A. Priestley. 1990. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, UK.

[5] David B Shmoys David P Williamson. 2010. *The Design of Approximation Algorithms.* Cambridge University Press.

[6] Thierry Garcia, Jean-Fréedéeric Myoupo, and David Semé. 2001. A work-optimal cgm algorithm for the longest increasing subsequence problem. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01),* Vol. 2. 563–569.

[7] Vijay K Garg. 2015. *Lattice Theory with Computer Science Applications.* Wiley, New York, NY.

[8] Vijay K. Garg. 2020. Predicate Detection to Solve Combinatorial Optimization Problems. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020,* Christian Scheideler and Michael Spear (Eds.). ACM, 235–245. https://doi.org/10.1145/3350755.3400235

[9] Ellis Horowitz and Sartaj Sahni. 1974. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)* 21, 2 (1974), 277–292.

[10] Shou-Hsuan Stephen Huang, Hongfei Liu, and Venkatraman Viswanathan. 1992. A sublinear parallel algorithm for some dynamic programming problems. *Theoretical Computer Science* 106, 2 (1992), 361–371.

[11] Oscar H Ibarra and Chul E Kim. 1975. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)* 22, 4 (1975), 463–468.

[12] Ehud D. Karnin. 1984. A parallel algorithm for the knapsack problem. *IEEE Trans. Comput.* 33, 05 (1984), 404–408.

[13] Donald E. Knuth. 1971. Optimum binary search trees. *Acta informatica* 1, 1 (1971), 14–25.

[14] Peter Krusche and Alexander Tiskin. 2010. Parallel Longest Increasing Subsequences in Scalable Time and Memory. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–185.

[15] KL Li, QH Li, Sheng-Yi Jiang, and Wei ZHANG. 2003. An optimal parallel algorithm for the knapsack problem. *Journal of software* 14, 5 (2003), 891–896.

[16] Takaaki Nakashima and Akihiro Fujiwara. 2002. Parallel algorithms for patience sorting and longest increasing subsequence. In *International Conference in Networks, Parallel and Distributed Processing and Applications.* 7–12.

[17] Vijay V. Vazirani. 2001. *Approximation Algorithms.* Springer-Verlag, Berlin, Germany.

[18] Andrew Chi-Chih Yao. 1981. On the parallel computation for the knapsack problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing.* 123–127.