Efficient Massively Parallel Methods for Dynamic Programming

Sungjin Im*
Electrical Engineering and Computer Science,
University of California
5200 N. Lake Road
Merced, California 95344, USA
sim3@ucmerced.edu

Benjamin Moseley†
Department of Computer Science and Engineering,
Washington University in St. Louis
1 Brookings Drive
St. Louis, Missouri 63130, USA
bmoseley@wustl.edu

Xiaorui Sun‡
Simons Institute for the Theory of Computing,
University of California
Berkeley, California 94702, USA
sunsirius@gmail.com

ABSTRACT

Modern science and engineering is driven by massively large data sets and its advance heavily relies on massively parallel computing platforms such as Spark, MapReduce, and Hadoop. Theoretical models have been proposed to understand the power and limitations of such platforms. Recent study of developed theoretical models has led to the discovery of new algorithms that are fast and efficient in both theory and practice, thereby beginning to unlock their underlying power. Given recent promising results, the area has turned its focus on discovering widely applicable algorithmic techniques for solving problems efficiently.

In this paper we make progress towards this goal by giving principles for simulating a large class of sequential dynamic programs in the distributed setting. In particular, we identify two key properties, *monotonicity* and *decomposibility*, which allow us to derive efficient distributed algorithms for problems possessing the properties. We showcase our framework by considering several core dynamic programming applications, Longest Increasing Subsequence, Optimal Binary Search Tree, and Weighted Interval Selection. For these problems, we derive algorithms yielding solutions that are arbitrarily close to the optimum, using $O(1)$ rounds and $\tilde{O}(n/m)$ memory on each machine where n is the input size and m is the number of machines available.

CCS CONCEPTS

• **Theory of computation → Approximation algorithms analysis**; **MapReduce algorithms**; **Massively parallel algorithms**;

KEYWORDS

Massively parallel computing, Dynamic programming

ACM Reference format:
Sungjin Im, Benjamin Moseley, and Xiaorui Sun. 2017. Efficient Massively Parallel Methods for Dynamic Programming. In *Proceedings of 49th Annual ACM SIGACT Symposium on the Theory of Computing, Montreal, Canada, June 2017 (STOC'17),* 14 pages.
DOI: 10.1145/3055399.3055460

1 INTRODUCTION

The modern era is witnessing a revolution in the ability to scale problems to massively large data sets. There have been several innovations that have enabled computation to scale to this unprecedented level. A key breakthrough in scalability was the introduction of fast, efficient, easy-to-use massively parallel distributed computing frameworks such as MapReduce [20], Hadoop [1] and Spark [2] and these frameworks have been critical to data-driven science, engineering, and industry. The use of these frameworks have mostly been developed by practitioners and applied researches with little influence form the theoretical computer science community. This is unlike other large data settings, such as parallel computing [24] and streaming algorithms [37], where the algorithms community has had a large impact on algorithms in production today.

These massively parallel frameworks have the potential to be more useful than originally thought. Modeling and studying these frameworks algorithmically has a potential to unlock their true underlying power and expand their use to a rich class of applications. Towards this end, there has been an effort by the theoretical computer science community to model the key underlying features and constraints of these frameworks and discover algorithms using the developed models.

The first models of massively parallel computation were introduced for the MapReduce framework and several models have been proposed [5, 12, 23, 26, 27, 31, 39, 40]. The theoretical study of these frameworks started to increase rapidly after the introduction of the MapReduce model of computation given in [31]. The model in [31] has been refined and extended afterwards subsequently [5, 12, 40]. Perhaps the main advantage of the model in [31] was its simplicity relative to others. The model identified several key features which could be obstructed otherwise in a plethora of system parameters. The area has gained growing attention as the proposed theoretical model given in [31] and its refinements have led to the discovery of novel algorithms which translated into efficient algorithms in practice [9, 10, 15, 17, 21–23, 31, 34, 36, 38, 42, 45].

*Supported in part by NSF grants CCF-1409130 and CCF-1617653. Work partially done while the author was visiting the Simons Institute.
†Supported in part by a Google Research Award, a Yahoo Research Award and NSF Grant CCF-1617724. Work partially done while the author was visiting the Simons Institute.
‡This work was done while the author was a research fellow at Simons Institute for the Theory of Computing.

STOC'17, Montreal, Canada

DOI: 10.1145/3055399.3055460

Theoretical Model. The model we consider in this paper, which is arguably the most popular, is the following. Let n be the input size and m be the number of machines, which is a part of the input. It is assumed that the *local* memory on each machine should be at most $\tilde{\Theta}(n/m)$ and the algorithm should allow for $n^{\epsilon} \leq m \leq n^{1-\epsilon}$ for some constant $\epsilon > 0$. The motivation for the memory and machine constraints is that the number of machines, m, and local memory size, M, on each machine should be much smaller than the input size to the problem since these frameworks are used to process large data sets. Further, the total memory available should not be much larger than the input size, thus is restricted to $\tilde{\Theta}(n)$.

In this model, computation proceeds in rounds. During a round each machine runs a polynomial time algorithm on the data assigned to the machine. No communication between machines is allowed during a round. Between rounds, machines are allowed to communicate so long as each machine receives no more communication than its memory. Any data output from a machine must be computed locally from the data residing on the machine and initially the input data is distributed across machines arbitrarily. The high level goal is to minimize the total number of rounds required, as this captures the typical main bottleneck of network communication. The ultimate goal is developing $O(1)$-rounds algorithms, which are highly desirable in practice, but today with the advent of more efficient distributed framework like Spark, $O(\log n)$-rounds algorithms are also applicable. The model is a special case of the Bulk-Synchronous-Parallel (BSP) model [43], but has the advantage of not having many parameters. This makes the algorithm design clearer and streamlines the search for efficient algorithms. As mentioned, this model has become popular due to its connection to practice. See [5, 31] for comparison of this model to other computing models such as PRAM and streaming models.

Many problems have been studied in this setting such as clustering [10, 11, 29], submodular function optimization [22, 34, 38], graph analysis [3, 9], and query optimization [12]. Two of the major algorithmic techniques used are sketching [28, 30] and the sample-and-prune technique [34]. Using these techniques, several sequential algorithms have been efficiently simulated in the massively distributed setting. For example, the work of [34] showed that the sequential algorithm for submodular optimization, which appears to have little use in parallel models, can actually be efficiently simulated in the parallel setting.

These results have begun to explore the so-called holy-grail question in parallel and distributed computing:

> *Is there a meta-algorithm that takes as input a sequential algorithm and outputs an efficient distributed algorithm? What kinds of problems admit such a meta-algorithm?*

Past work has shown that a large class of sequential greedy algorithms can be efficiently simulated in the distributed setting [34]. The looming question is, what other algorithms can be efficiently simulated in the massively distributed setting? How close can we come to achieving the ideal meta-algorithm that can efficiently scale any sequential algorithm? Resolving these questions has the potential for having impact in both theory and practice. Towards reaching this goal it is of additional interest to introduce widely applicable techniques that can be used to solve problems in the massively parallel setting.

Simulating Dynamic Programming: Efficiently simulating dynamic programming algorithms is a clear target in the area. Dynamic programming is one of the most fundamental techniques for solving problems sequentially and is an essential part of the standard CS curriculum. The technique is of central use in various fields such as bioinformatics, economics, and operations research. While this method has been a core algorithmic technique in the past, as data sets become larger, the technique is being rendered ineffective and considered obsolete by practitioners. There is a pressing need to develop techniques for running dynamic programming methods efficiently in the massively parallel environment.We make progress to answering the holy-grail question with respect to dynamic programming, focusing on the following question:

> *Are there principled guidlines for converting a sequential dynamic program into an efficient distributed algorithm? The algorithm should run in $O(\log n)$ rounds, ideally in $O(1)$ rounds. What are the common features of the problems that will make themselves subject to such a framework?*

These are challenging questions. Dynamic programs typically have large space requirements and run in a sequential manner. It is non-trivial to adapt such dynamic programs to the massively parallel setting where each machine is constrained by a sublinear memory size. Further, the number of rounds should be small for the algorithms to be adopted in practice.

Our Results and Contributions: Our main contribution is in identifying two key properties, *monotonicity* and *decomposability* needed to simulate various dynamic programs in the massively parallel model of computation and give *principled guides* for converting a dynamic program into an efficient distributed algorithm when the underlying problem satisfies the two properties. The key properties and the guides can be found in Section 2. To demonstrate the effectiveness of our principled guides, we consider three canonical dynamic program applications: Optimal Binary Search Tree (OBST), Weighted Interval Selection (IS), and Longest Increasing Subsequence (LIS). These problems will be formally defined in Sections 4, 5, 3, respectively.

More specifically, our results are as follows:

- Guided by the monotonicity and decomposability properties, we give $O(1)$ round algorithms. For any constant $0 < \epsilon < 1$, all algorithms are $(1+\epsilon)$-approximations and use $\tilde{O}(n)$ aggregate memory. Let $M = \tilde{O}(n^{\delta})$ be the memory usage on each machine for some constant $\delta > 0$. Specifically, the number of rounds and local memory used by each algorithm is as follows.
  - The algorithm for Optimal Binary Search Tree runs in $O(\frac{1}{\delta})$ rounds for any $\delta > 0$ (Section 3).
  - The algorithm for Longest Increasing Subsequence runs in $O(\frac{1}{\epsilon^2})$ rounds for any $\delta > 3/4$ (Section 5).
  - The algorithm for Weighted Interval Selection runs in $O(\frac{1}{\delta}(\log \frac{1}{\epsilon} + \log \frac{1}{\delta}))$ rounds for any $\delta > 0$ (Section 4).

The above algorithmic results are involved. To showcase our main algorithmic principles, we derive $O(\log n)$ rounds algorithms

for interval selection and longest increasing subsequence. All algorithms have an approximation factor $1+\epsilon$, and use a near-linear aggregate memory $\tilde{O}(n)$ for an arbitrary constant $0 < \epsilon < 1$.

There have been a few previous works on individual problems subject to dynamic programming in other models which readily translate into results in the theoretical model we consider in this paper. For example, the work in the BSP model in [33] implies a $O(\log n)$-rounds algorithm for the LIS problem in our model. To the best of our knowledge, there has been no attempt to give principled guides for deriving efficient distributed algorithms from sequential dynamic programming. Further, with the aid of the two key properties, we give the *first* $O(1)$-rounds algorithms for all the aforementioned problems.

Our Related Work: Dynamic programming was extensively studied in the standard PRAM model [6, 25]. For example, the optimal order of matrix multiplications problem [13, 14, 18, 19, 41, 44], the string editing problem [7], the longest increasing subsequence and longest common subsequence problems [16, 35], and the tree dynamic programming problems [8, 32, 41]. All these works require $O(\log n)$ rounds in our massively parallel model. Dynamic programming was studied also in the BSP model, e.g. [4, 33]. All the previous work requires $O(\log n)$ rounds if the number of machine is sublinear in the input size, which is a main constraint of our massively parallel model, and we do not know how to extend the previous work to obtain $O(1)$-round algorithms.

2 KEY PRINCIPLES

In this section we present two key properties that lead to efficient distributed algorithms when the problem in consideration possesses them. The two key properties we identify are *monotonicity* and *decomposibility*. The first property alone is sufficient for the existence of algorithms running in $O(\log n)$ rounds for some problems, but reducing the number of rounds to $O(1)$ additionally requires the second property. Throughout this section, we will focus on computing a near optimum value since one can find an actual solution achieving the near optimum using a standard trace-back method.

Key Properties

(1) **Monotonicity**: A sub-problem has no greater (smaller, resp.) optimum if the objective is to be maximized (minimized, resp.). (2) **Decomposibility**: The input can be decomposed into a two-level laminar family of partial input, where an upper level partial input is called a group and a lower level partial input is called a block, such that:

- A nearly optimal solution (for the whole input) can be constructed by concatenating solutions for groups.
- A nearly optimal solution for each group can be constructed from $O(1)$ blocks.

We illustrate how we use these properties to derive an efficient distributed algorithm from a sequential dynamic programming using the (Weighted) Interval Selection problem (IS) as an example. Our main results are for the LIS and OBST problems, but here we have chosen the simplest problem, IS, only for the purpose of illustration to clearly explain the high-level ideas without introducing heavy notations. The $O(1)$-round algorithms for other two problems we consider, LIS and OBST, require extra non-trivial observations about the optimal solution's structural properties; although, the derivation of their distributed algorithms use the key properties in similar manners. We also give an example that doesn't satisfy the monotonicity property, for which we believe no $O(1)$ approximate $O(1)$-round algorithm exists.

2.1 An Example Satisfying the Key Properties: Weighted Interval Selection

Let's first formally define the Weighted Interval Seletion (IS) problem. The input to IS is a collection of intervals $\{I_i = (a_i, b_i)\}_{i\in[n]}$ with their respective weights $\{w_i\}_{i\in[n]}$. The goal is to choose a subset of disjoint intervals with the maximum weight. It is well-known that this problem can be solved in polynomial time using DP.

Standard Sequential DP. We start by reviewing a simple DP for IS. Assume that intervals are ordered in increasing order of their start points; sorting by end points will be symmetric but we choose to sort by start points since it will simplify notation when we derive distributed algorithms. Let $\mathrm{OPT}(S)$ denote the optimal solution to the instance consisting of a subset S of intervals, or the optimal value depending on the context. Let $A(i)$ denote $\mathrm{OPT}(\{I_i, I_{i+1}, I_{i+2}, ..., I_n\})$. Our goal is to compute $A(1)$ which can be computed using the following recursion:

$$A(i) = \max\{A(i+1), w_i + A(j)\}$$

where $j = \arg\min_{j'}(b_i < a_{j'})$ when $i \le n$; and $A(n+1) = 0$.

An $O(\log n)$-Round Distributed Algorithm using Monotonicity. The problem of the above recursion is that it is too sequential to be directly translated into one working in the distributed setting. We distribute the input intervals equally to machines keeping the sorted order; for example, the first machine gets n/m intervals with the earliest start points[1]. Since each machine can only see a subset of intervals, it is natural to define a subproblem that each machine faces. To be able to append a solution to a subproblem to other solutions we obtain from other machines, we define a subproblem by adding another parameter. Let $B(i, j)$ denote the maximum weight subset of intervals that start no earlier than I_i but end before I_j starts; note that intervals corresponding to $B(i, j)$ end before intervals corresponding to $B(j, j')$ start, hence can be chosen simultaneously. However, the subproblems require a quadratic memory usage, which is not affordable in our distributed setting.

We now observe that the subproblem $B(i, j)$ satisfies the monotonicity property: for any $i' \le i$ and $j' \ge j$, $B(i', j') \ge B(i, j)$. Using this property, *we define a similar entry $C(i, w)$ by swapping j with the target value*: $C(i, w) = \min_{j':B(i,j')\ge w} j'$; if no such j' exists, then set $C(i, w) = \infty$. At first sight, this seems worse since j can have at most n values while w can have a much wider range of values. Hence, we discretize the second parameter w. In this case, it turns out that we only need $O(\log n)$ different values of w. What the monotonicity property guarantees is the following: *if we have a feasible solution including the subset of intervals corresponding to $C(i, w)$, then we still get a feasible schedule when replacing it with that corresponding to $C(i, w')$ for $w' \le w$*. If w' is close to w, we lose

[1] Note that sorting can be done in the massively parallel (distributed) setting in $O(1)$ rounds so long as the machines have memory at least n^δ for constant $\delta > 0$. This follows by adapting sample sort.

only a small weight. Note that the number of entries $\{C(i,w)\}_{i,w}$ is almost linear in n.

We compute entries $\{C(i,w)\}_{i,w}$, particularly $C(1,w)$, in $O(\log n)$ rounds – $\arg\max_w(C(1,w) < \infty)$ will be within $(1+\epsilon)$ factor of the optimum for a fixed constant $\epsilon > 0$. For the purpose of illustration, we focus on the first machine and consider an interval I_i assigned to the machine. To simplify the argument, assume that the k_{th} machine has a copy of the partial input the $k+1_{th}$ machine has for all k. Let $C'(i,w)$ be the DP entry corresponding to $C(i,w)$. In the beginning, if $C(i,w) \le n/m$, then it means that the machine has all the information needed to compute $C(i,w)$, so we have $C'(i,w) = C(i,w)$; note that the first machine has $2n/m$ intervals with the earliest start times. Otherwise, we have $C'(i,w) = \infty$. In the subsequent rounds, we consider all pairs of (w_1, w_2) such that $w_1 + w_2 \simeq w$. Here we use $\simeq$ to mean that the quantities are within a $1+\epsilon$ factor of each other. If $C'(i,w_1) = j_1$, then the machine gets $C'(j_1,w_2) = j_2$ from the machine having the interval I_{j_1}. This means that $C'(i,w') \le j_2$ where $w' \simeq w$. By taking the minimum of j_2 over all pairs (this can be done in parallel), the one-round update of the DP entries completes. Here we approximate w by a slightly smaller w', which keeps the feasibility of the final solution due to the monotonicity property. In the l_{th} iteration, $C'(i,w)$ is computed based on the partial input on 2^l machines. Hence $O(\log n)$ rounds suffice to complete the computation of $\{C'(i,w)\}$.

$O(1)$ **Rounds using Monotonicity and Decomposability.** What the previous $O(\log n)$-round algorithm essentially does is approximately computing $C(1,w)$ by considering partial inputs from a exponentially increasing number of machines simultaneously. We can speed up this process in terms of the number of rounds by doubling the number of 'crossing' intervals considered in the current solution. A crossing interval is an interval that ends later than all other intervals on the same machine end. Intuitively, crossing intervals create dependencies across machines. We observe that *the problem satisfies the decomposibility property*. Although *the property itself is defined standalone*, we explain it in the context of the distributed setting for a more intuitive explanation. We observe that there is a nearly optimal solution that is decomposed into groups of intervals where each group includes at most $O(1)$ crossing intervals – further, each of the groups consists of intervals from a disjoint set of machines. Thus, once we compute a (nearly optimal) solution to each group in parallel, we only need to concatenate solutions from groups that are disjoint over machines. Computing the solution to each group can be done in $O(1)$ rounds since it has at most $O(1)$ crossing intervals; here a block is the partial input on each machine, thus the solution comes from $O(1)$ machines. Since we don't know how to partition the input into groups a priori, we will create groups that are potentially useful but not necessarily pairwise disjoint over machines. However, at this point, the solutions to groups can be further simplified by machine indices since a nearly optimal solution can be constructed from groups that are disjoint over machines. This reduces the input size, and we recurse until the input becomes sufficiently small to fit into a single machine.

2.2 An Example Not Satisfying the Monotonicity Property

Consider a DAG $G = (V, E)$ consisting of $\sqrt{n}$ levels of nodes where each level has exactly $\sqrt{n}$ nodes. Each node i has weight w_i. Arcs exist only from each level l to the next level $l+1$. Assume that the DAG is sparse. The goal is to find a maximum weight path. Let $A(j)$ denote the weight of the maximum weight path ending with vertex j. If we know $A(i)$ for all nodes i in level l, we can compute $A(j)$ for all j in the next level $l+1$: $A(j) := \max_{(i,j)\in E} A(i) + w_j$. A natural sub-problem we can consider for this problem is $B(i,j)$, the maximum weight of all paths starting and ending with nodes i and j, respectively. This problem does not satisfy the monotonicity property, meaning that $B(i,j)$ can increase or decrease when we change the value of j. Storing $B(i,j)$ for all pairs (i,j) consumes too much memory. We are not aware of any algorithm that uses a sublinear memory while running in $O(\log n)$ rounds.

3 OPTIMAL BINARY SEARCH TREE

In this section, we consider the Optimal Binary Search Tree problem (OBST) and give a $(1+\epsilon)$-approximation running in $O(1/\delta)$ rounds. The input to OBST consists of n elements, $1, 2, 3, \cdots, n$ with their respective weights/probabilities, $w(1), w(2), w(3), \cdots, w(n)$, where $\sum_{i=1}^n w(i) = 1$. For simplicity, we assume w.l.o.g. that element i has key value i. The goal of the OBST problem is to find a binary search tree (BST) for elements $[n]$ so as to minimize the expected search cost, i.e., $\sum_{i=1}^n w(i) \cdot \mathrm{dep}(i)$, where $\mathrm{dep}(i)$ is the depth of element i in the binary search tree; the root has depth 1.

Standard Sequential DP. Let's first review a standard sequential DP for OBST. Define $w(i,j) := \sum_{k=i}^{j} w(k)$. Let $T_{i,j}$ denote the optimal binary search subtree for elements, $i, i+1, \ldots, j$, and $\mathrm{OPT}(i,j)$ the cost of $T_{i,j}$. We observe the follow recursion:

$$\mathrm{OPT}(i,j) = \min_{i \le k \le j} \Big(\mathrm{OPT}(i,k-1) + \mathrm{OPT}(k+1,j)\Big) + w(i,j) \quad (1)$$

for all $1 \le i \le j \le n$, and $\mathrm{OPT}(i,j) = 0$ when $i > j$.

3.1 A High-level Overview of a $O(1/\delta)$-round Algorithm

For the OBST problem, we do not give a separate overview of a $O(\log n)$-round algorithm since it is very similar to the $O(1/\delta)$-round algorithm we will discuss. In the following, we will give a high-level overview of a $O(1/\delta)$-round algorithm explaining how the two key properties are used in deriving the algorithm.

As before, we swap the target cost/value v with j. Let $\mathrm{last}^*(i,v)$ denote the maximum j such that there is a BST of cost at most v for elements $i, i+1, \cdots, j$. Since the objective is to be minimized, the monotonicity property is similarly defined but with the inequality flipped. Note that if $v \ge v'$, then $\mathrm{last}^*(i,v) \ge \mathrm{last}^*(i,v')$. Let $\mathrm{last}(i,v)$ be the entry corresponding to $\mathrm{last}^*(i,v)$, which we would like to compute. By preprocessing the input and discretizing the target values, we only need to keep $O(\log n)$ entries, $\mathrm{last}(i,v)$, for each element i. Our goal is to obtain $\mathrm{last}(\cdot,\cdot)$ such that $\mathrm{last}(i,v') \ge \mathrm{last}^*(i,v)$ for $v' \simeq v$. What this means is quite intuitive: We can construct a binary search subtree, with a comparable cost, that starts from each element i and includes as many elements as the

corresponding optimal subtree. Hence, we will be done if we can compute last$(\cdot,\cdot)$ in a distributed fashion.

Unfortunately, computing last$(\cdot,\cdot)$ in a distributed manner is a non-trivial task since computing an entry last(i,v) may require access to too many elements which cannot fit into a single machine. One natural approach would be exponentially increasing the number of elements to be considered; let's call this quantity 'width.' Let's take a close look at this computation process to see the hidden challenges. Suppose we would like to approximate OPT(i,j), the cost of $T_{i,j}$, when the number of elements to be considered, $j-i+1$, is much larger than a machine's local memory size. Let k be the root of $T_{i,j}$. If we already know approximate values of OPT$(i,k-1)$ and OPT$(k+1,j)$, we will be able to compute OPT(i,j) approximately. If k is not too close to i or j, by appropriately overlapping two sets of elements, $\{i,i+1,\cdots,k_1\}$ and $\{k_2,k_2+1,\cdots,j\}$, we can increase the number of elements in consideration by a constant factor: Say $v_1 \simeq$ OPT$(i,k-1)$ and $v_2 \simeq$ OPT$(k+1,j)$. Intuitively, approximate costs of $T_{i,k-1}$ and $T_{k+1,j}$ can be recovered from last(i,v_1) and last$(k+1,v_2)$. Hence if we know that k is roughly in the 'middle' of the elements $i,i+1,\cdots,j$, then by querying last(k',v_2) for a smaller number of candidate elements k' for the root node of $T_{i,j}$ that can be put into a single machine, we will be able to effectively increase the width over iterations.

However, this may not be the case when elements have very different weights. To overcome this challenge we use another trick of grouping elements into blocks of similar weights. Fortunately, we can show that k is roughly in the 'weighted' middle of elements $i,i+1,\cdots,j$. Further, we show that we can add or ignore some elements in approximately computing last(i,v). Thus, we recursively build a hierarchical/laminar block system where the upper level is an abstraction of the lower level, and we gradually compress last$(\cdot,\cdot)$ according to the set of blocks in each level. Here we have to be careful with grouping elements since some elements can have huge weights, so must remain as singletons in some cases. The block system can be viewed as an application of the decomposibility property.

Overall, our dynamic program uses two different types of sketches: (i) last$(\cdot,\cdot)$ to simplify the DP entries by discretizing the target values based on the monotonicity property, and (ii) a hierarchical block system of elements where last$(\cdot,\cdot)$ is projected on blocks in each level to be compressed based on the decomposibility property.

3.2 Useful Properties of Optimal Subtrees

We observe that the OBST problem satisfies the monotonicity property.

Proposition 3.1 (Monotonicity). *For any $1 \le i' \le i \le j \le j' \le n$, $OPT(i,j) \le OPT(i',j')$.*

The following claim asserts that the cost of an optimal subtree is within factor $O(\log n)$ of the total weight of elements in the subtree.

Claim 3.2. *$w(i,j) \le OPT(i,j) \le \lceil \log n \rceil w(i,j)$.*

Proof. The lower bound is straightforward to see since every node has depth at least 1. The upper bound follows since one can pack all elements into a tree of depth $\lceil \log n \rceil$. □

We first show that one can assume w.l.o.g. that all elements have weights at least $\frac{\epsilon}{10n^2}$. We say that such elements are non-negligible and the other elements are negligible.

Lemma 3.3. *Let $T := T_{1,n}$ be the optimal tree for the entire set of elements. Let T' be the optimal tree for the non-negligible elements. Then, we have $(1-\epsilon/10)\text{cost}(T) \le \text{cost}(T') \le \text{cost}(T)$.*

Proof. The upper bound immediately follows from Proposition 3.1; although the proposition is stated only for sets of consecutive elements, it is straightforward to see that monotonicity holds for any two sets of elements where one set is a subset of the other. To see that the lower bound holds, iteratively add each negligible element to T' as a new leaf node, which increases the cost by at most $n \cdot \frac{\epsilon}{10n^2} = \frac{\epsilon}{10n}$. Hence we have $\text{cost}(T) \le \text{cost}(T') + \epsilon/10$. By Claim 3.2, we have $\text{cost}(T) \ge w(1,n) = 1$. Thus, $\text{cost}(T') \ge \text{cost}(T) - \epsilon/10 \ge (1-\epsilon/10)\text{cost}(T)$. □

Since the proof of the lower bound in Lemma 3.3 only uses the obvious fact that any node has depth at most n, we can add the negligible elements later only ensuring that the resulting tree is a valid binary search tree. Since it can be done in a straightforward manner using $O(1/\delta)$ additional rounds, henceforth we assume that all elements are non-negligible.

Next, we observe that any optimal subtree must be 'balanced' in terms of weights.

Claim 3.4. *Let $i \le k \le j$ be the element associated with the root of $T_{i,j}$. We have $w(i,k-1) \le 2w(i,j)/3$ and $w(k+1,j) \le 2w(i,j)/3$.*

Proof. Assume w.l.o.g. that $w(i,j) = 1$ by scaling elements' weights. We only prove the first inequality since the other can be proved symmetrically. For the sake of contradiction assume $w(i,k-1) > 2/3$. Let $T = T_{i,j}$ and ℓ be the root of the left subtree. From the recursion (1) on the optimal subtrees, we have

$$\begin{aligned}\text{OPT}(i,j) = \text{cost}(T) = \text{OPT}(i,\ell-1) + \text{OPT}(\ell+1,k-1) \\ + \text{OPT}(k+1,j) + w(i,k-1) + w(i,j)\end{aligned} \quad (2)$$

We consider two cases depending on the value of $w(i,\ell)$.

Case (i). $w(i,\ell) > 1/3$. In this case, we draw a contradiction by constructing a new tree T' with a smaller cost for $i,i+1,\cdots,j$ that has ℓ as root, and $T_{i,\ell-1}$ and $T_{\ell+1,j}$ as the root's left and right subtrees, respectively. We have

$$\begin{aligned}\text{cost}(T') =& \text{OPT}(i,\ell-1) + \text{OPT}(\ell+1,j) + w(i,j) \\ \le& \text{OPT}(i,\ell-1) + \text{OPT}(\ell+1,k-1) \\ &+ \text{OPT}(k+1,j) + w(\ell+1,j) + w(i,j)\end{aligned}$$

Knowing that $i < \ell < k < j$ and $w(k,j) \le 1 - w(i,k-1) < 1/3$, we have $\text{OPT}(i,j) - \text{cost}(T') \ge w(i,\ell) - w(k,j) > 0$. This contradicts to T's optimality.

Case (ii). $w(i,\ell) \le 1/3$. Let r be the root of $T_{\ell+1,k-1}$. Expand Eq. (2) by plugging in $\text{OPT}(\ell+1,k-1) = \text{OPT}(\ell+1,r-1) + \text{OPT}(r+1,k-1) + w(\ell+1,k-1)$:

$$\begin{aligned}\text{OPT}(i,j) = \text{OPT}(i,\ell-1) + \text{OPT}(\ell+1,r-1) + \text{OPT}(r+1,k-1) \\ +\text{OPT}(k+1,j) + w(\ell+1,k-1) + w(i,k-1) + w(i,j).\end{aligned} \quad (3)$$

Consider a new tree T'' for elements $i, i+1, \cdots, j$ having r as root, and $T_{i,r-1}$ and $T_{r+1,j}$ as the left and right subtrees, respectively. We have

$$\begin{aligned}\text{cost}(T'') &= \text{OPT}(i, r-1) + \text{OPT}(r+1, j) + w(i,j) \\ &\le \text{OPT}(i, \ell-1) + \text{OPT}(\ell+1, r-1) + \text{OPT}(r+1, k-1) \\ &\quad + \text{OPT}(k+1, j) + w(i, r-1) + w(r+1, j) + w(i,j)\end{aligned}$$

where the inequality follows by making ℓ (k, resp.) as the root of the subtree for elements $i, i+1, \cdots, r-1$ (elements, $r+1, r+2, \cdots, j$, resp.).

Knowing that $\text{OPT}(\ell+1, k-1) \le \text{OPT}(\ell+1, r-1) + \text{OPT}(r+1, k-1) + w(\ell+1, k-1)$, we have,

$$\begin{aligned}&\text{OPT}(i,j) - \text{cost}(T'') \\ &\ge w(i, k-1) - w(r+1, j) + w(\ell+1, k-1) - w(i, r-1) \\ &= w(i,r) - w(k,j) + w(\ell+1, k-1) - w(i, r-1) \\ &\ge w(\ell+1, k-1) - w(k,j) > 0\end{aligned}$$

The last inequality follows from the fact that $w(\ell+1, k-1) = w(i, k-1) - w(i, \ell) > 1/3$ and $w(k,j) = w(i,j) - w(i, k-1) < 1/3$. This contradicts to the fact that T is the optimal tree for $i, i+1, \cdots, j$, proving $w(i, k-1) \le 2w(i,j)/3$. □

The following claim states that the cost of an optimal subtree changes only little when some small weight elements are added or ignored. The claim will allow us to consider subtrees over 'blocks' of elements rather than elements and define a hierarchical block system so as to compress the cost information over iterations. Intuitively, the claim holds since Claim 3.4 implies that the tree (consisting of non-negligible elements) has depth $O(\log n)$, thus an element i can contribute to the cost by $O(\log n)w(i)$.

Claim 3.5. *Assume $1 \le i' \le i \le j \le j' \le n$ such that $w(i,j) \ge (1-\alpha)w(i', j')$ for some $0 < \alpha < \frac{1}{12 \log n}$. Then $OPT(i,j) \ge (1 - 12\alpha \log n) OPT(i', j')$.*

Proof. We show that $w(i,j) \ge (1-\alpha)w(i', j)$ implies $\text{OPT}(i,j) \ge (1 - 6\alpha \log n)\text{OPT}(i', j)$. Then the lemma is obtained by applying this argument twice. We first construct a binary search tree for $\{v_{i'}, \ldots, v_j\}$ based on $T_{i,j}$. We start from root of $T_{i,j}$ and go left until the current subtree has total weight less than $w(i', i-1)$. Let k denote the root of the current subtree and r be the largest index of the current subtree (current subtree is $T_{i,r}$). Let $d = \text{dep}(k)$. We replace the current subtree by a tree with i as the root, $T_{i', i-1}$ as the left subtree and $T_{i,r} \setminus i$ be the right subtree, and denote this new tree as T'. We have $\text{OPT}(T') - \text{OPT}(T) < \text{OPT}(i', i-1) + w(i', i-1) \cdot d + w(i,r) \le w(i', i-1)(\lceil \log n \rceil + d + 1)$. If $d \le 4 \log n$, then $\text{OPT}(T) > \text{OPT}(T') - 6 \log n \cdot w(i', i-1) \ge \text{OPT}(T') - 6 \log n \alpha w(i,j) \ge (1 - 6\alpha \log n)\text{OPT}(T') \ge (1 - 6\alpha \log n)\text{OPT}(i', j)$. Otherwise, we have $d > 4 \log n$. By Claim 3.4, $\text{OPT}(T) \ge w(i', i-1) \cdot \left(\frac{3}{2}\right)^{d-2} > w(i', i-1) \cdot n^2$. Hence

$$\begin{aligned}\text{OPT}(T) &> \text{OPT}(T') - w(i', i-1)(\lceil \log n \rceil + d + 1) \\ &> \text{OPT}(T') \left(1 - \frac{\lceil \log n \rceil + d + 1}{n^2}\right) \\ &> (1 - 6\alpha \log n)\text{OPT}(T')\end{aligned}$$

□

3.3 $O(1/\delta)$-round Algorithm

This section is devoted to giving a $O(1/\delta)$-round algorithm for OBST. Recall that we assume w.l.o.g. that all elements have weight at least $\frac{\epsilon}{10n^2}$; see Lemma 3.3. Extending the notation $w(\cdot)$, let $w(S) := \sum_{i \in S} w(i)$.

Definition 3.6 (Block System). A *block* $B = (s_B, t_B)$ is a consecutive sequence of elements $\{s_B, s_B + 1, \cdots, t_B\}$. Let $\mathbb{B} = \{B_1, B_2, \ldots\}$ be an ordered set of blocks. Given a *weight width* parameter ℓ, we say $\mathbb{B}$ is an ℓ-block system if

(1) $\mathbb{B}$ is a partition of the elements, $[n]$, i.e., every node $i \in [n]$ belongs to one block; and no two distinct blocks have common elements.
(2) Blocks are ordered in increasing order of their smallest (starting) elements, i.e., $s_{B_a} < s_{B_b}$ if $a < b$.
(3) For any two consecutive blocks, $B_a, B_{a+1} \in \mathbb{B}$, $w(B_a) + w(B_{a+1}) > \ell$.

Let $|B|$ be the number of elements in block B. The first (last, resp.) element in B is called B's starting (ending, resp.) element, and denoted as s_B (t_B, resp.). To avoid double subscript, for an indexed block B_a, we may use s_a and t_b for s_{B_a} and t_{B_a} respectively if the notation is clear from context. Let $|\mathbb{B}|$ be the number of blocks in $\mathbb{B}$. Given a block system $\mathbb{B}$, let $w(B_a, B_b) := \sum_{c=a}^{b} w(B_c) = \sum_{i=s_{B_a}}^{t_{B_b}} w(i)$. A block's weight is defined as the total weight of the elements in the block.

Recursively Constructing a Hierarchical Block System. From the definition of block system, note that a block system is 'coarser' when the weight width parameter ℓ is larger. Intuitively, the bottom level of our DP is defined over a block system with the smallest width parameter and the top level over that with the largest width parameter. We recursively build a hierarchical block system so that $\mathbb{B}_h$ is a refinement of $\mathbb{B}_{h+1}$. The initial system is $\mathbb{B}_0 := \{\{1\}, \{2\}, \{3\}, \cdots, \{n\}\}$. Let $\alpha := \delta/4$. The block system $\mathbb{B}_h$ has width parameter $\ell_h := \frac{\epsilon}{10n^2} n^{(\alpha+2)h} = \ell_0 n^{\alpha h}$. Given $\mathbb{B}_h$, one can construct $\mathbb{B}_{h+1}$ by considering blocks in $\mathbb{B}_h$ in the given order and aggregating a maximal collection of blocks that does not exceed weight ℓ_{h+1}; if a block in $\mathbb{B}_h$ has weight greater than ℓ_{h+1}, it is added to $\mathbb{B}_{h+1}$ without being merged with other blocks in $\mathbb{B}_h$. To parallelize this, we can consider in parallel each maximal collection of blocks in $\mathbb{B}_h$ with weight less than ℓ_{h+1} and partition it into blocks of aggregate weight roughly ℓ_{h+1}. It is easy to see that this only needs $O(1)$ rounds. Our algorithm and analysis still work as long as $w(B_a) + w(B_{a+1}) = \Omega(\ell)$ in the third requirement in the definition of block system. To make our presentation more transparent, we assume that $\mathbb{B}_{h+1}$ is constructed from $\mathbb{B}_h$ as described above.

Fact 3.7. *For any h,*

- *$\mathbb{B}_h$ is an ℓ_h-block system.*
- *Any block in $\mathbb{B}_h$ with weight greater than ℓ_h has only one element.*
- $|\mathbb{B}_h| \le \frac{2}{\ell_h} + 1$.

Efficient Dynamic Programming over the Hierarchical Block System. We define $\text{last}_h(a, q)$ over each block system $\mathbb{B}_h$, which has the following meaning: if $\text{last}_h(a, q) = b$, then we

know how to construct a BST of cost no greater than $v_q := \frac{\epsilon}{10n^2}(1+\epsilon_0)^q$ (q is a non-negative integer) for elements in the blocks $B_a, B_{a+1}, \cdots, B_b$ in the block system $\mathbb{B}_h$ where $\epsilon_0 = \frac{\epsilon}{100 \log n}$. Further, we will ensure that the BST is almost optimal when $\ell_h n^\alpha \le w(B_a, B_b) \le \ell_h n^{2\alpha}$. In other words, $\text{last}_h(a,q)$ offers a compact description of the cost of every binary search tree when the aggregate weight of the elements in the tree is considerably larger than but not so far from ℓ_h. Since ℓ_h increases by a factor of n^α in each iteration, we will only need $O(1/\alpha) = O(1/\delta)$ iterations to approximate the cost of the BST for every subset of consecutive elements. Notice that there are two types of 'sketches' here: First, the cost $v_q := \frac{\epsilon}{10n^2}(1+\epsilon_0)^q$ is an exponentially discretized value; and second, subtrees are parameterized by blocks B_a, B_b rather than all possible pairs of elements. The first sketch is enabled by the monotonicity property. The second sketch is justified since we showed in Claim 3.5 that adding elements of small aggregate weights to the front or back changes the cost very little, and each individual block has small weight relative to the total weight of elements in the tree we need to consider in each iteration; this is closely tied to the decomposibility property.

We formally describe how we compute $\text{last}_h(a,q)$ over iterations, which is defined over $\mathbb{B}_h, h \in \{0,1,2,\cdots,h_1\}$ where h_1 is defined as the smallest h such that $\ell_h n^{2\alpha} \ge 1$; recall that the total weight of all elements is 1. When $h = 0$, recall that $\mathbb{B}_0 := \{\{1\},\{2\},\{3\},\cdots,\{n\}\}$. Set $\text{last}_0(a,q)$ as the minimum $b \ge a$ such that the BST for $a, a+1, \cdots, b$ has cost at most v_q for all q such that $\ell_0 n^\alpha \le v_q \le \ell_0 n^{2\alpha}$; note that for such q, $b - a + 1 < n^\delta$ since every element has weight at least $\ell_0 = \frac{\epsilon}{10n^2}$. (We can compute $\text{last}_0(a,q)$ for larger values of q, but doing so up to $v_q \le \ell_0 n^{2\alpha}$ will suffice for the recursion to work). The computation can be done in parallel by assigning elements 1 to $2n^\delta$ to the first machine, elements $n^\delta + 1$ to $3n^\delta$ to the second machine and so on.

In the following we define $\text{cost}'_h(B_a, B_b)$ which can be viewed as an approximate cost of the optimal BST for the elements in the blocks in $\mathbb{B}_h$ from B_a to B_b. It means that we can construct a BST of cost at most $\text{cost}'_h(B_a, B_b)$ for those elements. The discretized cost is derived from $\text{last}_h(a,r)$. Thus, cost'_h and last_h are equivalent in terms of determining the cost of a subtree starting and ending with blocks in $\mathbb{B}_h$ – the only difference is that last_h is more compact than cost'_h.

$$\text{cost}'(B_a, B_b) = \begin{cases} 0 & \text{if } a > b \\ v_r := \frac{\epsilon}{10n^2}(1+\epsilon_0)^r & \text{otherwise} \end{cases}$$

where r is the smallest integer such that $\text{last}_h(a,r) \ge b$.

We obtain last_{h+1} (thus cost'_{h+1}) from last_h using the following two steps. We will often call the total weight of elements in a subtree as weight width. Recall that $v_q := \ell_0(1+\epsilon_0)^q$. We repeat for $0 \le h \le h_1$ – intuitively, h_1 is a sufficiently large weight width (close to 1) that allows us to see the entire set of elements in a hierarchically compressed fashion.

(1) Computing cost'_h to a higher weight width.
Given $\text{cost}'_h(a,v)$ for all blocks $B_a \in \mathbb{B}_h$ and v such that $\ell_h n^{\alpha h} \le v_q \le \ell_h n^{2\alpha h}$, we further compute $\text{cost}'_h(a,v)$ for v such that $\ell_h n^{\alpha h} \le v_q \le \ell_h n^{3\alpha h}$ using the following recursion:
(a) Let $\text{cost}(B_a, B_b) =$

$$\min_{\substack{a \le b_0 \le b: \\ w(B_a, B_{b_0-1}) < 3w(B_a,B_b)/4, \\ w(B_{b_0+1}, B_b) < 3w(B_a,B_b)/4}} \begin{cases} \text{cost}'(B_a, B_{b_0}) + \text{cost}'(B_{b_0}, B_b) + w(B_a, B_b) \\ \qquad \text{if } |B_{b_0}| > 1 \\ \text{cost}'(B_a, B_{b_0-1}) + \text{cost}'(B_{b_0+1}, B_b) + w(B_a, B_b) \\ \qquad \text{if } |B_{b_0}| = 1 \end{cases} \tag{4}$$

and $\text{root}(B_a, B_b)$ be the b_0 that minimizes Eq. (4).
(b) Set $\text{last}_h(a,q)$ as the minimum b such that $\text{cost}_h(B_a, B_{b'}) \le v_q$ for all $b' \le b$.

(2) Compressing cost'_h via projection into a higher level block system, $\mathbb{B}_{h+1}$:
For every $B_x = (s_x, t_x) \in \mathbb{B}_{h+1}$, denote $B_a \in \mathbb{B}_h$ such that $s_a = s_x$. For any integer q such that $\ell_h n^{2\alpha} \le v_q \le \ell_h n^{3\alpha}$, let B_y be the block in $\mathbb{B}_{h+1}$ containing $B_{\text{last}(a,q)} \in \mathbb{B}_h$.

$$\text{last}_{h+1}(x,q) = \begin{cases} y & \text{if } t_y = t_{\text{last}_h(a,q)} \\ y-1 & \text{if } t_y \ne t_{\text{last}_h(a,q)} \end{cases}$$

From the previous step in the recursion, we have computed $\text{cost}'_h(B_a, B_b)$ (more precisely, the corresponding last_h) for some blocks in $B_a, B_b \in \mathbb{B}_h$. We keep the value of $\text{cost}'_h(B_a, B_b)$ only up to a certain weight width since it is a good approximate estimate only when the weight width, i.e. the aggregate weight of the elements in the blocks in $\mathbb{B}_h$ from B_a through B_b, is in a certain range depending on ℓ_h. Recall that in the ℓ_h-block system $\mathbb{B}_h$, each block in $\mathbb{B}_h$ has weight about ℓ_h. [2] Since each machine has memory $\tilde{\Theta}(n^\delta)$, each machine can see elements of aggregate weight $\ell_h n^\delta = \ell_h n^{4\alpha}$. By placing the starting elements of the first $2n^\delta$ blocks in $\mathbb{B}_h$ on the first machine, and those of the blocks from the $(n^\delta+1)$th to the $3n^\delta$th on the second machine, and so on, we can surely extend $\text{cost}'_h(B_a, B_b)$ up to a weight width of $\ell_h n^{3\alpha}$ in distributed fashion. This requires only $O(1)$ rounds.

In the first step, we derive cost'_h from last_h by slightly over-estimating the cost. The sketch last_h only tells us what blocks we can include in a BST without exceeding a discretized cost; discretized costs are powers of $(1+\epsilon_0)$. By over-estimating the cost by a factor of $(1+\epsilon_0)$, we can safely estimate the cost. There are two cases we consider in computing $\text{cost}(B_a, B_b)$. If a block has a huge weight (it only happens when the block is a singleton element set; see Fact 3.7) and is chosen as the root (the second case), we have to explicitly compute the additional cost incurred by doing so. Otherwise, the block can be either in the left subtree or in the right subtree (the first case). In this case, we even let both the left and right subtrees include the block B_{b_0}. This doesn't increase the cost too much since B_{b_0}'s weight is small compared to the total weight of elements in the tree (see Claim 3.5). Further, having redundant elements is not an issue since we can remove some of them without

[2] This is not exactly true since some blocks can have huge weights and we can only say that any two consecutive blocks have aggregate weight at least ℓ_h, but this is for an intuitive explanation.

increasing the cost. Finally, the new cost_h values are stored in the compact form of last_h.

In the next step, we further compress cost'_h via 'projection' into the higher level of block system, $\mathbb{B}_{h+1}$. Recall that $\mathbb{B}_h$ is a refinement of $\mathbb{B}_{h+1}$. In this step, we compress last_h by essentially keeping the information w.r.t. starting blocks in the upper level $\mathbb{B}_{h+1}$. Here we go conservative again. We let $\text{last}_{h+1}(a,q) = b$ only when there is a pair of blocks a', b' in $\mathbb{B}_h$ such that $\text{last}_h(a',q) = b'$ and the blocks from $B_{a'}$ to $B_{b'}$ include all elements in the blocks from B_a to B_b. Still, we have to be careful if the ending block $B_{\text{last}(a,q)}$ has a huge weight, in which case the block has only one element. This is why we distinguish the two cases in setting $\text{last}_{h+1}(x,q)$.

Finally, we note that recovering an actual nearly optimal BST is straightforward via a standard trace-back method, hence is omitted.

3.4 Analysis

Definition 3.8. We say that cost'_h is updated up to weight width v if $\text{cost}'_h(B_a, B_b) \le (1+\epsilon_0)^{10h\alpha \log n} \cdot \text{OPT}(s_{B_a}, t_{B_b})$ for all blocks $B_a, B_b \in \mathbb{B}_h$ such that $w(B_a, B_b) \le v$.

Recall that we only keep cost'_h for all blocks $B_a, B_b \in \mathbb{B}_h$ such that $w(B_a, B_b) \ge \ell_h n^\alpha$ and it is not stated in the definition for simplicity. We will show the following main lemma.

Lemma 3.9. *The recursion ensures that* cost'_h *is updated up to weight width* $\ell_h n^{2\alpha}$ *for all* $h \le h_1$.

Before proving the lemma, we observe that it immediately implies the desired result. By definition of h_1, when $h = h_1$, we have $w(B_a, B_b) \le \ell_h n^{2\alpha}$ is satisfied for all $B_a, B_b \in \mathbb{B}_h$. For the last block system $\mathbb{B}_{h_1}$, consider the blocks B_a and B_b including the first and last elements (1 and n), respectively. Then, we have that $\text{cost}'_h(B_a, B_b) \le (1+\epsilon_0)^{10h\alpha \log n} \cdot \text{OPT}(s_{B_a}, t_{B_b}) \le (1+\epsilon)\text{OPT}(1,n)$; recall that $\epsilon_0 = \frac{\epsilon}{100 \log n}$. As mentioned before, we can reconstruct a BST with cost at most $(1+\epsilon)\text{OPT}(1,n)$, i.e. a $(1+\epsilon)$-approximate optimal BST.

It now remains to show Lemma 3.9. To understand and analyze the effect of the first step, we need the following lemma. For notational convenience, we may use s_a and t_a in place of s_{B_a} and t_{B_a} for an indexed block B_a.

Lemma 3.10. *Consider an* ℓ_h*-block system* $\mathbb{B}_h$ *for some* $h > 0$. *For any* $B_a, B_b \in \mathbb{B}_h$ *such that* $w(B_a, B_b) \ge \ell_h n^{2\alpha}$, *if there exists a* $0 < \gamma < 1$ *such that*

1. $\text{cost}'(B_a, B_{b_0}) \le (1+\gamma)OPT(s_a, t_{b_0})$ *for any* b_0 *such that* $w(B_a, B_{b_0}) < 3w(B_a, B_b)/4$,
2. $\text{cost}'(B_{b_0}, B_b) \le (1+\gamma)OPT(s_{b_0}, t_b)$ *for any* b_0 *such that* $w(B_{b_0}, B_b) < 3w(B_a, B_b)/4$,

then $\text{cost}(B_a, B_b) \le (1+\gamma)(1+\epsilon_0)OPT(s_a, t_b)$ *and* $\text{cost}'(B_a, B_b) \le (1+\gamma)(1+\epsilon_0)^2 OPT(s_a, t_b)$.

Proof. Let k be the root of T_{s_a, t_b}, and B_u be the block in $\mathbb{B}_h$ containing element/node k. If B_u is a single-element block, then

$$\begin{aligned}\text{cost}(B_a, B_b) &\le \text{cost}'(B_a, B_{u-1}) + \text{cost}'(B_{u+1}, B_b) + w(B_a, B_b)\\ &\le (1+\gamma)\left(\text{OPT}(s_a, k-1) + \text{OPT}(k+1, t_b)\right) + w(s_a, t_b)\\ &\le (1+\gamma)\text{OPT}(s_a, t_b).\end{aligned}$$

If B_u contains more than one element, then $w(B_u) \le \ell_h$ (see Fact 3.7). From the precondition of the lemma, we have $w(B_a, B_b) = w(s_a, t_b) \ge \ell_h \cdot n^{2\alpha}$. Thus, by Lemma 3.4, we have that $w(s_a, k-1) \le 2w(s_a, t_b)/3$ and $w(k+1, t_b) \le 2w(s_a, t_b)/3$, which imply that

$$w(s_a, s_u - 1) \ge \frac{w(s_a, s_b)}{3} - \ell_h \ge \left(\frac{1}{3} - \frac{1}{n^{2\alpha}}\right) w(B_a, B_b) \ge \frac{1}{4} w(B_a, B_b)$$

and similarly $w(t_u + 1, t_b) \ge w(B_a, B_b)/4$. Hence $\frac{w(k, t_u)}{w(s_a, t_u)} \le \frac{\ell_h}{w(s_a, t_b)/4} \le \frac{4}{n^{2\alpha}}$ and $\frac{w(s_u, k)}{w(s_u, t_b)} \le \frac{\ell_h}{w(s_a, t_b)/4} \le \frac{4}{n^{2\alpha}}$. By Claim 3.5,

$$\begin{aligned}&\text{cost}(B_a, B_b)\\ &\le \text{cost}'(B_a, B_u) + \text{cost}'(B_u, B_b) + w(B_a, B_b)\\ &\le (1+\gamma)\left(\text{OPT}(s_a, t_u) + \text{OPT}(s_u, t_b)\right) + w(s_a, t_b)\\ &\le \frac{1+\gamma}{1 - 48\log n / n^{2\alpha}}\left(\text{OPT}(s_a, k-1) + \text{OPT}(k+1, t_b)\right) + w(s_a, t_b)\\ &\le (1+\gamma)(1+\epsilon_0)\text{OPT}(s_a, t_b).\end{aligned}$$

From the definition of $\text{cost}'(B_a, B_b)$, we have

$$\text{cost}'(B_a, B_b) \le (1+\epsilon_0)\text{cost}(B_a, B_b) \le (1+\gamma)(1+\epsilon_0)^2 \text{OPT}(s_a, t_b).$$

□

Recall that cost'_h is currently updated up to weight width $\ell_h n^{2\alpha}$. Hence, if $\ell_h n^\alpha w(B_a, B_b) \le \frac{4}{3}\ell_h n^{2\alpha}$, the condition stated in Lemma 3.10 is satisfied, and cost'_h is updated up to weight width $\frac{4}{3}\ell_h n^{2\alpha}$. Hence, after $\lceil \log_{4/3} n^\alpha \rceil \le 3\alpha \log n$ iterations of the recursion, Eq. (4), cost'_h is updated up to weight width $\ell_h n^{3\alpha}$. By Lemma 3.10, we know that cost'_h is $(1+\epsilon_0)^{(10\alpha h + 6\alpha)\log n}$-approximate up to weight width $\ell_h n^{3\alpha}$.

Finally, we now consider the effect of the second step. In this step, $\ell_h n^{2\alpha} \le v_q \le \ell_h n^{3\alpha}$, which implies that $\ell_h n^{2\alpha} \le w(B_a, B_{\text{last}_h(a,q)}) \le \ell_h n^{3\alpha}$. In the first case we set $\text{last}_{h+1}(x,q) = y$ and there is no loss since the blocks in $\mathbb{B}_{h+1}$ from B_x to B_y has exactly the same set of elements as those in $\mathbb{B}_h$ from B_a to $\text{last}_h(a,q)$. In the second case that we set $\text{last}_{h+1}(x,q) = y$, we know that B_y includes more than one element, hence we have $w(B_y) \le \ell_{h+1}$ by Fact 3.7. Since $w(B_y)/w(B_a, B_{\text{last}_h(a,q)}) \le \ell_{h+1}/(\ell_h n^{2\alpha}) \le 1/n^\alpha$, we have

$$\begin{aligned}&\text{cost}'_{h+1}(B_x, B_y) = \text{cost}'_h(B_a, B_{\text{last}_h(a,q)})\\ &\le (1+\epsilon_0)^{(10\alpha h + 6\alpha)\log n}\text{OPT}(s_{B_a}, t_{B_{\text{last}_h(a,q)}})\\ &\le (1+\epsilon_0)^{(10\alpha h + 6\alpha)\log n}(1+\epsilon_0)\text{OPT}(s_{B_x}, t_{B_y})\\ &\le (1+\epsilon_0)^{10(h+1)\alpha \log n}\text{OPT}(s_{B_x}, t_{B_y}).\end{aligned}$$

The penultimate inequality follows from Claim 3.5 with the fact $w(B_y)/w(B_a, B_{\text{last}_h(a,q)}) \le 1/n^\alpha$. This completes the proof of Lemma 3.9.

4 WEIGHTED INTERVAL SELECTION

In this section, we consider the Weighted Interval Selection problem (IS) and give a $(1-\epsilon)$-approximation algorithm running in $O(1)$ rounds. For the problem definition, see Section 2.

4.1 $O(1)$-round Algorithm

This section is devoted to proving the following theorem.

Theorem 4.1. *For any $\epsilon, \delta > 0$, there exists a $(1-\epsilon)$-approximation for the Weighted Interval Selection problem running in $O(\frac{1}{\delta}(\log\frac{1}{\epsilon}+\log\frac{1}{\delta}))$ rounds when each machine has memory $\tilde{O}(n^{\delta})$.*

We perform a simple preprocessing before we introduce some definitions. For simplicity, assume w.l.o.g. that intervals start and end times are all distinct and not integers. We can have this property by perturbing those times without changing the feasibility of any subset of intervals. Here start (end, resp.) times are increased (decreased, resp.) by a infinitesimal amount. Order intervals in increasing order of their start times. Equally distribute intervals to machines – the first n/m intervals go to machine 1, the next n/m machines go to machine 2, and so on. For notational simplicity, we assume that intervals on machine k start at times between k and $k+1$. This is w.l.o.g. that since the feasibility of the solution only depends on the relative ordering of intervals start and end times, and we can easily transform the given instance to satisfy this property in $O(1)$ rounds. Note that machine k can only see intervals starting at times between k and $k+1$.

When we say an interval (a,b) starts on machine k, we mean that the interval is on machine k or equivalently $k < a < k+1$. Similarly, we say an interval (a,b) ends on machine k if $k < b < k+1$. The interval is said to be *local* if $k < a, b < k+1$ for some integer k, otherwise *crossing*. We may call a subset of disjoint intervals, D, as a *block*. We can define D's start (end, resp.) time as the start (end, resp.) time of the earliest starting (ending, resp.) interval in D. We say that a block D spans machines $k, k+1, \cdots, k'$ if the earliest starting interval in D starts on machine k and the latest ending interval in D ends on machine k'. We say that two blocks are (pair-wise) independent if they span disjoint sets of machines. If a block contains *at most* ℓ crossing intervals, we say that the block is a ℓ-block. Let $w(D) := \sum_{I_i \in D} w_i$.

Lemma 4.2. *For any even integer $L \geq 2$, there exists a $(1-2/L)$-approximate solution consisting of (pair-wise) independent L-blocks.*

Proof. Consider a fixed optimal solution, from which we remove some crossing intervals as follows, without losing more than $1/L$ times the optimum. Partition the crossing intervals into groups so that each group contains L consecutive crossing intervals; the last group may contain less than L crossing intervals. We remove the lightest interval from every group except the last. Clearly we lose at most $1/L$ times the optimum. It is easy to see that the resulting solution consists of $2L$-blocks that are pairwise independent. By scaling L, we obtain the lemma. □

Overview of the Algorithm. Let's take a moment to understand what Lemma 4.2 implies. If there is a nearly optimal solution consisting only of 0-blocks, then we can let each machine compute the best 'local' solution from the local intervals on the machine, and simply aggregate the weights of the local solutions from all machines. The aggregation can be done in $O(1/\delta)$ rounds; recall that each machine has $\tilde{O}(n^{\delta})$ memory. Thus, we will be able to compute a nearly optimal solution in $O(1/\delta)$ rounds. However, the optimum solution may have lots of heavy crossing intervals, which can't be ignored to obtain a nearly optimal solution. By Lemma 4.2, we know that we can construct a $(1-\epsilon)$-approximate solution from $2/\epsilon$-blocks that are pair-wise independent. We compute such blocks that start no earlier than I_i for each interval I_i. Here by using the monotonicity property and the swapping argument, we compute them approximately and memory-efficiently in $O(1_{\epsilon,\delta})$ rounds. Once we have such blocks, we view each block as an 'consolidated' interval. More precisely, we zoom out the picture by approximating each interval's start and end times by the machines (indices) where they fall. This is justified since the blocks we will choose are pairwise independent. By taking this simplified yet equivalent view, we get a new instance whose size is almost linear in the number of machines. Hence we can recurse on the new instance. We will only need to recurse $O(1/\delta)$ times since the number of intervals decreases by a factor of $\tilde{\Omega}(n^{\delta})$. Thus, this whole procedure requires only $O(1_{\epsilon,\delta})$ rounds. To derive Theorem 4.1, we need to set parameters more carefully, but this is a high-level overview.

We assume w.l.o.g. that the heaviest interval has weight 1 by scaling and $w_i \geq \epsilon/n$ for all i since intervals with weight less than ϵ/n contribute to the optimum by at most ϵ.

Definition 4.3. For each interval I_i and a target weight μ, let $\mathsf{OPT}(i,\mu,L)$ be the smallest interval index $j \geq i$ such that there is a L-block of weight at least μ that is a subset of $\{I_i, I_{i+1}, I_{i+2}, \cdots I_n\}$ and is disjoint from $\{I_j, I_{j+1}, \cdots, I_n\}$. Let $W := \{0, \frac{1}{n^2}, \frac{(1+\eta)}{n^2}, \frac{(1+\eta)^2}{n^2}, \cdots, n\}$ for η to be determined. A family $\mathcal{F} = \{D(i,\mu,L)\}_{i\in[n],\mu\in W}$ of blocks is said to be a $1-\gamma$-approximate compact family of L-blocks if $D(i,\mu,L)$ has weight at least $(1-\gamma)\mu$ and $D(i,\mu,L) \leq \mathsf{OPT}(i,\mu,L)$. Here, $D(i,\mu,L)$ is analogously defined as $\mathsf{OPT}(i,\mu,L)$.

What the compact family means is the following. In Lemma 4.2, we observed that there is a $(1-2/L)$-approximate solution consisting only of L-blocks, i.e. with at most L crossing intervals. For each of those blocks, D^*, the compact family contains almost as a good block D whose weight is within factor $(1-\gamma)$ of D^*'s weight and that ends no later than D^*. Therefore, if we replace each block in the nearly optimal solution with the corresponding block in the family, we can still get a feasible solution only losing $(1-2/L)$ factor in the approximation ratio. By Lemma 4.2, this implies that we can construct a $(1-2/L)(1-\gamma)$-approximate solution from the compact family of L-blocks.

The following lemma states the concrete goal we aim to achieve in each iteration of the recursion.

Lemma 4.4. *If we can construct a $(1-O(\epsilon\delta))$-approximate compact family of L-blocks in $O(\log\frac{1}{\epsilon}+\log\frac{1}{\delta})$ rounds where $L = \frac{2}{\epsilon\delta}$ and $\eta = \frac{\epsilon\delta}{10(\log(1/\epsilon)+\log(1/\delta))}$, then we can obtain a $(1-O(\epsilon))$-approximate solution for the IS problem in $O(\frac{1}{\delta}\cdot(\log\frac{1}{\epsilon}+\log\frac{1}{\delta}))$ rounds.*

Proof. Note that the family has size $n|W| = n \cdot O(\log_{1+\eta} n)$. Knowing that we can find a $(1-2/L)$-approximate solution consisting of independent optimal L-blocks, we can construct a $(1-O(\epsilon\delta))\cdot(1-2/L) = 1-O(\epsilon\delta)$ approximate solution from the given family. As discussed before, we only need to use pair-wise independent blocks in the family meaning that we only need blocks corresponding to $D(1,\cdot,L)$, $D(1+n/m,\cdot,L)$, $D(1+2n/m,\cdot,L)$, $\cdots$. We can view each of these blocks as a new interval where each interval's start and end times are replaced with the appropriate

machine indices. So the new instance has size $n^{1-\delta} \cdot O(\log_{1+\eta} n)$; later we will use $O(\log^2_{1+\eta} n)$ factor more memory in the actual DP which doesn't affect this lemma. We need only $O(1/\delta)$ iterations of the recursion to reduce the instance size to fit it into a single machine's memory, $\tilde{\Theta}(n^\delta)$. The claimed approximation ratio and number of rounds follow immediately. □

It now remains to show Lemma 4.4.

Constructing the Desired Compact Family using DP. We will find a desired compact family, $\mathcal{F}(\ell) := \{D(i, \mu, 2^\ell - 1)\}_{i\in[n], \mu\in W}$, as stated in Lemma 4.4. Towards this end, we find such a family for $\ell = 0, 1, 2, \cdots, \ell_1 = \lceil \log 2/(\epsilon\delta)\rceil$ in this order until we have $2^\ell - 1 \geq L = 2/(\epsilon\delta)$. Since we will use an induction on the value of ℓ, we refine and strengthen the property that $\mathcal{F}(\ell)$ satisfies: our goal is to find $\mathcal{F}(\ell)$ that is $(1-\eta)^\ell$-approximate in increasing order of $\ell = 0, 1, 2, \cdots \ell_1$ in $O(\ell)$ rounds. Since $L = \frac{2}{\epsilon\delta}$, we only need $O(\log 1/\epsilon + \log 1/\delta)$ rounds, and we have $(1-\eta)^{\ell_1} = 1 - O(\epsilon\delta)$ since $\eta = \frac{\epsilon\delta}{10(\log(1/\epsilon)+\log(1/\delta))}$.

Our remaining goal is to find $\mathcal{F}(\ell)$ that is $(1-\eta)^\ell$-approximate in increasing order of $\ell = 0, 1, 2, 3, \cdots \ell_1$ in $O(\ell)$ rounds. In our DP, each machine k needs to know in $O(1)$ rounds the earliest ending crossing interval of weight at least $\mu \in W$ that starts on machine $k+1$ or later; the exact definition is deferred to Corollary 4.6. The following claim will be useful towards this end.

Claim 4.5. *Suppose n numbers, $u_1, u_2, \cdots, u_n$, are stored across m machines; the first machine has $u_1, \cdots, u_{n/m}$, the second has $u_{n/m+1}, \cdots, u_{2n/m}$, and so on. Let $S(k)$ be the smallest among the numbers assigned to machines $k, k+1, \cdots, m$. Then, every machine k can compute $S(k)$ simultaneously in $O(1/\delta)$ rounds.*

Proof. Let $S(k, \Delta)$ be the smallest among the numbers assigned to machines $k, k+1, \cdots, k+\Delta-1$. It is obvious that we can compute $S(k, 1)$ in round 1 and store it on machine k. In round 2, machine k computes $S(k, n^\delta)$ by getting $S(k, 1), S(k+1, 1), \cdots, S(k+n^\delta-1, 1)$ from machines k through $k+n^\delta-1$ and taking the minimum. Similarly, in the next round, machine k computes $S(k, n^{2\delta})$ by getting $S(k, n^\delta)$, $S(k+n^\delta, n^\delta)$, $\cdots$, $S(k+n^\delta(n^\delta-1), n^\delta)$ and taking the minimum. The claim follows by repeating this process. □

Corollary 4.6. *For any interval I_i, let $C(i, \mu)$ denote the index of the earliest starting crossing interval of weight at least μ that starts after I_i ends. We can compute $C(i, \mu)$ for every i and $\mu \in W$ in $O(1/\delta)$ rounds and store it on the machine $\lfloor b_i \rfloor$ where the interval I_i ends. Further, each machine uses $\tilde{O}(n/m)$ memory.*

Proof. Fix μ. The interval $I_{C(i,\mu)}$ can start on machine $\lfloor b_i \rfloor$ or later. If it starts on machine $\lfloor b_i \rfloor$, the machine has direct access to the interval. Otherwise, the machine computes $I_{C(i,\mu)}$ by setting $S(k)$ in Claim 4.5 to the earliest ending crossing interval that starts on machine k or later, which can be done in $O(1/\delta)$ rounds by Claim 4.5. We copy $S(\lfloor b_i \rfloor + 1)$ to machine $\lfloor b_i \rfloor$. It is easy to see that each machine only uses $O(|W|)$ extra memory. □

Clearly we can compute $\mathcal{F}(0)$ in 1 round with no communication across machines since the blocks in $\mathcal{F}(0)$ include no crossing intervals. We now show that we can find $\mathcal{F}(\ell+1)$ from $\mathcal{F}(\ell)$ in $O(1)$ rounds. Fix i, μ, and ℓ. We show how to compute $D(i, \mu, 2^{\ell+1} - 1)$. Consider any triplet $\mu_1, \mu_2, \mu_3 \in W$ such that $(1-\eta)\mu \leq \mu_1 + \mu_2 + \mu_3 \leq \mu$ which we call a candidate triplet. Let $j_1 = D(i, \mu_1, 2^\ell - 1)$. The machine $\lfloor a_i \rfloor$ having I_i asks machine $\lfloor a_{j_1} \rfloor$ (where I_{j_1} starts) for the earliest ending crossing interval of weight at least μ_2 starting no earlier than I_{j_1}, which can be done in $O(1)$ round by Corollary 4.6. Let C denote the crossing interval. Then, the machine also gets the information $j_3 = D(j_2, \mu_3, 2^\ell - 1)$ where I_{j_2} is the earliest starting interval after the interval C ends, from the machine where C ends. We take the minimum j_3 over all possible triplets. The value j_3 can be obtained for different triplets in parallel. Hence this requires only $O(1)$ rounds. Further, there are at most $|W|^3 = O(\log^3_{1+\eta} n)$ triplets to be considered for each i. Thus, each machine uses memory at most $(n/m)O(\log^3_{1+\eta} n) = \tilde{O}(n/m)$.

To show this process correctly computes $D(i, \mu, 2^{\ell+1} - 1)$ that is $(1-\eta)^{\ell+1}$-approximate, assume the block corresponding to $\mathsf{OPT}(i, \mu, 2^{\ell+1}-1)$ includes at least 2^ℓ crossing intervals since otherwise we have $\mathsf{OPT}(i, \mu, 2^\ell - 1) = \mathsf{OPT}(i, \mu, 2^{\ell+1} - 1)$, thus by induction hypothesis, we can use the block corresponding to $D(i, \mu, 2^\ell - 1)$ as the desired block corresponding to $D(i, \mu, 2^{\ell+1} - 1)$. Say the block corresponding to $\mathsf{OPT}(i, \mu, 2^{\ell+1} - 1)$ consists of 2^ℓ - 1-block B^*_1 starting with interval $I_{i^*_1}$, a 'middle' crossing interval C^*, and a 2^ℓ - 1-block B^*_3 starting with $I_{i^*_3}$; the block B^*_3 could be empty. Let μ^*_1 and μ^*_3 be the weights of the two blocks, B^*_1 and B^*_3, respectively. Also let μ^*_2 be the interval C^*'s weight. Note that $\mu \leq \mu^*_1 + \mu^*_2 + \mu^*_3$. Let μ'_1, μ'_2, μ'_3 denote the largest values in W that are no greater than $\mu^*_1, \mu^*_2, \mu^*_3$, respectively. Note that $(1-\eta)\mu \leq \mu'_1 + \mu'_2 + \mu'_3 \leq \mu$.

We now show that we can find as good blocks B_1 and B_3 as B^*_1 and B^*_3, and as good crossing interval C as C^*. Suppose the algorithm now considers the triplet (μ'_1, μ'_2, μ'_3).

(1) Since $i \leq i^*_1$ from the definition of $\mathsf{OPT}(i, \mu, 2^{\ell+1} - 1)$, it must be the case that $j_1 = D(i, \mu'_1, 2^\ell - 1) \leq D(i^*_1, \mu'_1, 2^\ell - 1)$. Let B_1 be the block corresponding to $D(i, \mu'_1, 2^\ell - 1)$. By induction hypothesis, we have $w(B_1) \geq (1-\eta)^\ell \mu'_1$.
(2) The algorithm looks for a crossing interval of weight at least μ'_2 that starts no earlier than I_{j_1} and ends the earliest. Clearly, C^* is a candidate. Hence $w(C) \geq \mu'_2$.
(3) The interval C ends no later than C^*. Let B_3 be the block corresponding to $D(j_2, \mu'_3, 2^\ell - 1)$. Since B^*_3 is considered as a candidate for B_3, we have $w(B_3) \geq (1-\eta)^\ell \mu'_3$ by induction hypothesis; and B_3 ends no later than B^*_3.

Hence, the $2^{\ell+1} - 1$-block consisting of B_1, B_3 bridged by C is considered for a valid triplet (μ'_1, μ'_2, μ'_3), and the block has weight at least $(1-\eta)^\ell \mu'_1 + \mu'_2 + (1-\eta)^\ell \mu'_3 \geq (1-\eta)^{\ell+1}\mu$. Further, the block ends before that corresponding to $\mathsf{OPT}(i, \mu, 2^{\ell+1} - 1)$. This completes the proof of Theorem 4.1.

5 LONGEST INCREASING SUBSEQUENCE

This section focuses on the longest increasing subsequence (LIS) problem. A *subsequence* of a string I is a string obtained by deleting entries of I. A subsequence is increasing if each position is strictly greater than the one before. We assume the input is a string of length n consisting of integers between 1 and n. Duplicates are allowed in the input. The section gives a $O(\log n)$ round algorithm. This algorithm is presented because it gives a streamlined view of

how the principals of monotonicity and decomposability can be used to design dynamic programs in the distributed setting. Indeed, this algorithm follows similar a development as was used for the logarithmic round interval selection problem. Due to the space constraints, we only give an overview of our $O(1)$ round algorithm, deferring details to the full version of this paper.

5.1 A Second Example of the Properties: LIS in $O(\log n)$ rounds

Consider an instance of the longest increasing subsequence (LIS) problem where I is an input string of length n. Define the **position** x of a string I to be the xth entry in I and the **value** to be the number stored in this position. Our goal is to design a efficient distributed $1-\epsilon$ approximation algorithm for the problem when the number of machines is $\Theta(m)$ and each of the machines has memory $\tilde{\Theta}(n/m)$ for any constant $0 < \epsilon < 1$. The memory is assumed to be at least n^{δ} for some constant $\delta > 0$.

Standard Sequential DP. We begin by considering a simple DP for the LIS problem in the sequential setting. Let $B(v, i)$ denote the length of the LIS of I that only uses elements in positions $i, i+1, \ldots, n$ in I and includes no element of value less than v. The longest increasing subsequence can be computed as follows where $I[i]$ denotes the value stored in the ith position of I.

$$B(v,i) = \begin{cases} \max\{B(v,i+1), 1 + B(I[i]+1, i+1)\} & \text{if } I[i] \geq v \\ B(v,i+1) & \text{otherwise} \end{cases}$$

The final value is stored in $B(-\infty, 1)$. It is assumed that all entries are initially 0 and then they are filled in recursively.

An $O(\log n)$-Round Distributed Algorithm Using Monotonicity and Decomposability. Like in the interval selection problem, the above recurrence is too sequential to be directly of use in the distributed setting. Further, the above recurrence requires too many entries to be stored. Indeed, there are $\Theta(n^2)$ possible entires required to compute L, yet we aim to use $\tilde{O}(n)$ aggregate memory. Both of the challenges will need to be overcome to have an efficient distributed algorithm.

The first insight used will be *decomposability*. Consider a substring I^* of I. Let I' and I'' be two substrings of I^* such that their concatenation $I' \cup I''$ equals I^*. We say a string I' is a *substring* of a string I^* if I' is obtained from I^* by cutting out some prefix and suffix of I^*. Consider extending the definition of B to the following. Let $D_{I^*}(v, v', i)$ be the length of the LIS of a string I^* that only uses positions $i, i+1, \ldots, |I^*|$ of I^* such that the elements used have value at least v and at most v'. Then, D_{I^*} can be decomposed and computed using $D_{I'}$ and $D_{I''}$. Indeed, if $|I^*| - i + 1 \leq |I''|$ then $D_{I^*}(v, v', i) = D_{I''}(v, v', i - |I'|)$. Otherwise,

$$D_{I^*}(v,v',i) = \max\{D_{I'}(v,v',i), D_{I''}(v,v',1), \max_{v''}\{D_{I'}(v,v'',i) + D_{I''}(v''+1, v', 1)\}\}.$$

The first term in the maximum states the subsequence corresponding to $D_{I^*}(v, v', i)$ is completely contained in I' and the second term captures the case where the subsequence is in I''. The third captures the case where the subsequence spans both I' and I''. This decomposition will be critical for our algorithm design. Note though that so far we have only increased the memory usage.

The next insight required is used to reduce the memory required to compute D. Like in the interval scheduling case, the key is *monotonicity*. Notice that by definition $B(v', i') \geq B(v, i)$ for any $i' \leq i$ and $v' \leq v$ and similarly for D. With monotonicity, the recurrence D can be replaced by a similar recurrence just as in the interval selection problem. Define a similar entry $P_{I^*}(v, \ell, i)$ by swapping v' with ℓ where ℓ represents the target length: $P_{I^*}(v, \ell, i) = \min_{v': D_{I^*}(v,v',i) \geq \ell} v'$. For an optimal algorithm, this does not change the amount of space required. However, later it will be established that storing only a small poly-logarithmic number of entries for ℓ will be sufficient.

The last insight is to drop i from P to get the following recurrence L. By the way we will later combine entries for L, we will show that i is not needed due to the decomposability of the problem. This gives a recurrence of the desired size $\tilde{O}(n)$ assuming only a logarithm number of entries for ℓ is required. For any string I^*, let $L_{I^*}(v, \ell) = P_{I^*}(v, \ell, 1)$ be the smallest value v' such that there is an increasing subsequence of I^* of length at least ℓ such that only values between v and v' are used. It is assumed that if there is no increasing subsequence for I^* of length at least ℓ using values at least v then $L_{I^*}(v, \ell) = \infty$.

We now show how to define L recursively in a similar manner as was established for D. As before, let I', I'' be two substrings of I such that the concatenation of I' and I'' forms a substring of the input I. Let I^* be this substring. Given $L_{I'}$ and $L_{I''}$, we can compute L_{I^*} as follows. By the notation $v \in I^*$, we mean that the value v appears in the string I^*.

$$L_{I^*}(v,\ell) = \min\left\{ \min_{v' \geq v : v' \in I'} L_{I'}(v', \ell), \min_{v' \geq v : v' \in I''} L_{I''}(v', \ell), \min_{\substack{\ell', \ell'', v', v'' : v' \in I', v'' \in I'', v'' > v' \geq v, \\ L_{I'}(v', \ell') < v'', \ell' + \ell'' \geq \ell}} L_{I''}(v'', \ell'') \right\} \quad (5)$$

The first term in the minimum says that the LIS is completely stored in I' and the second term considers when the LIS is completely in I''. The third term combines subsequences bridging between I' and I''.

Before giving the details of the algorithm, first it is established that storing only a small number of entries for ℓ will be sufficient to approximate L. To do so, first consider the following fact.

Fact 5.1. *The length of the longest increasing sequence of I is the largest ℓ such that there is an interger v satisfying $L_I(v, \ell) < \infty$.*

Let $0 < \epsilon < 1$ be a constant. Let $\mathcal{L}_I$ be an estimation of L_I. The purpose of $\mathcal{L}_I$ is to behave similarly to L but is only an approximation due to only considering values of ℓ of the form $(1 + \frac{\epsilon}{10 \log n})^k$ for integer $k \geq 0$. We will say that ℓ takes *geometric* values. We say $\mathcal{L}_I$ $(1-\gamma)$-approximates L_I if for every v, ℓ such that $L(v, \ell) < \infty$, there is a $\ell' \geq (1-\gamma)\ell$ such that $\mathcal{L}_I(v, \ell') \leq L_I(v, \ell)$ and ℓ is geometric. We assume that $\mathcal{L}$ is computed in the same way as L in the recurrence given in Eq. 5, but restricting ℓ to be geometric.

The following lemma bounds the amount the approximation degrades when computing $\mathcal{L}_I$ from $\mathcal{L}_{I''}$ and $\mathcal{L}_{I'}$.

Lemma 5.2. *Let I', I'' be two substrings of I such that $I' \cup I''$ forms a substring of I. Denote $I^* = I' \cup I''$. If $\mathcal{L}_{I'}$ $(1-\gamma)$-approximates $L_{I'}$ and $\mathcal{L}_{I''}$ $(1-\gamma)$-approximates $L_{I''}$, then $\mathcal{L}_{I^*}$ $(1 - \frac{\epsilon}{10 \log n})(1-\gamma)$-approximates L_{I^*}.*

Proof. Fix any v, ℓ such that $L_{I^*}(v,\ell) < \infty$. Let $v' = L_{I^*}(v,\ell)$. Let A be the increasing sequence corresponding to the entry in $L_{I^*}(v,\ell)$. The analysis is broken into several cases. If $A \subseteq I'$, then $L_{I'}(v,\ell) = v'$. Since $\mathcal{L}_{I'}$ $(1-\gamma)$-approximates $L_{I'}$, there is a geometric $\ell' \geq (1-\gamma)\ell$ such that $\mathcal{L}_{I'}(v,\ell') \leq v'$. Hence $\mathcal{L}_{I^*}(v,\ell') \leq v'$. A similar argument holds if $A \subseteq I''$.

The interesting case is when A is not a subset of I' nor I'. In this case, let $A' = A \cap I'$ and $A'' = A \cap I''$. Let t be the value of the last element of A' and s be the value of the first element of A''. Let $\ell_1 = |A'|$ and $\ell_2 = |A''|$. Notice that $L_{I'}(v,\ell_1) \leq t$ and $L_{I''}(s,\ell_2) \leq v'$. Since $\mathcal{L}_{I'}$ $(1-\gamma)$-approximates $L_{I'}$ and $\mathcal{L}_{I''}$ $(1-\gamma)$-approximates $L_{I''}$, there are $\ell_1^\diamond \geq (1-\gamma)\ell_1$ and $\ell_2^\diamond \geq (1-\gamma)\ell_2$ such that $\mathcal{L}_{I'}(v,\ell_1^\diamond) \leq t$ and $\mathcal{L}_{I'}(s,\ell_2^\diamond) \leq v'$. Consider rounding $\ell_1^\diamond + \ell_2^\diamond$ down to the closest geometric value ℓ^*. This rounding ensures that $\ell^* \geq (\ell_1^\diamond + \ell_2^\diamond)\frac{1}{(1+\frac{\epsilon}{10\log n})} \geq (1-\gamma)\frac{1}{(1+\frac{\epsilon}{10\log n})}(\ell_1+\ell_2) \geq (1-\gamma)(1-\frac{\epsilon}{10\log n})(\ell_1+\ell_2)$. Further, $\mathcal{L}_{I^*}(v,\ell^*) \leq \mathcal{L}_{I''}(s,\ell_2^\diamond) \leq v'$ completing the proof. □

The previous lemma will allow us to focus on geometric values of ℓ to ensure memory efficiency.

Sequential Algorithm. With the above ideas in place, we are ready to describe the algorithm. We first describe the algorithm without details on how to make it distributed. That is, as if it were sequential. Later it is discussed how the algorithm fits in the distributed setting.

The algorithm works as follows.

(1) Initially let $\mathcal{I}_1 = \{I_i : 1 \leq i \leq n\}$ where I_i is the substring of I only containing i-th element of I. For any $1 \leq i \leq n$, set $\mathcal{L}_{I_i}(v,1) = v$ for $v \in I_i$, and ∞ for all the other entries.
(2) Set $k = 1$.
(3) Repeat the following process $\lceil \log n \rceil$ times until there is only one interval in $\mathcal{I}_k$
   (a) Increment k. For every $i > 0$, let I_i be the concatenation of $(2i-1)$-th and $(2i)$-th interval in $\mathcal{I}_{k-1}$. Put all the new intervals into $\mathcal{I}_k$.
   (b) For every $I^* \in \mathcal{I}_k$, denote I' and I'' be the two intervals in $\mathcal{I}_{k-1}$ such that $I^* = I' \cup I''$. For any $v \in I$, let $\mathcal{L}_{I^*}(v,\ell)$ be the value of the same entry for the interval in $\mathcal{I}_{k-1}$.
      (i) For any $v \in I'$ and any geometric ℓ', let $\tau(v,\ell')$ be the smallest value v'' in I'' such that $v'' > \mathcal{L}_{I'}(v,\ell')$. Update $\mathcal{L}_{I^*}(v,\ell)$ for every geometric ℓ to be $\min\{\mathcal{L}_{I^*}(v,\ell), \min_{\substack{\ell',\ell'':\\ \ell'+\ell'' \geq \ell}} \mathcal{L}_{I''}(\tau(v,\ell'),\ell'')\}$.
      (ii) For every $v \in I$ and geometric ℓ, update $\mathcal{L}_{I^*}(v,\ell)$ to be $\min_{v' \geq v} \mathcal{L}_{I^*}(v',\ell)$.

The following lemma establishes that the algorithm computes the desired solution.

Lemma 5.3. *Fix any k and a string $I \in \mathcal{I}_k$. Let $I = I' \cup I''$ where $I', I'' \in \mathcal{I}_{k-1}$. Given $\mathcal{L}_{I'}$ and $\mathcal{L}_{I''}$, the values the algorithm computes for $\mathcal{L}_{I^*}$ is the same as the recurrence in Eq. 5 for geometric values of target lengths.*

Proof. For any v and geometric ℓ. Consider the recurrence in Eq. 5 for $\mathcal{L}_{I^*}(v,\ell)$. Value is computed from the minimum of three terms. In order, we call these the first, second and third cases. If value stored in $\mathcal{L}_{I^*}(v,\ell)$ is computed from the first or the second case in Eq. 5, then clearly $\mathcal{L}_{I^*}(v,\ell)$ is correctly computed by the algorithm in Step 3(b)ii.

Consider the case where $\mathcal{L}_{I^*}(v,\ell)$ is computed from the third case. In this case, $\mathcal{L}_{I^*}(v,\ell)$ is obtained from combining both $\mathcal{L}_{I'}(v',\ell')$ and $\mathcal{L}_{I''}(v'',\ell'')$. Without loss of generality assume v'' is the smallest value in I'' that is greater than $\mathcal{L}_{I'}(v',\ell')$. After Step 3(b)i, $\mathcal{L}_{I^*}(v',\ell) \leq \mathcal{L}_{I''}(v'',\ell'')$, and after 3(b)ii, $\mathcal{L}_{I^*}(v,\ell) \leq \mathcal{L}_{I''}(v'',\ell'')$. Thus, the algorithm computes the desired solution. □

Knowing that the algorithm runs in $\lceil \log n \rceil$ iterations, we have the following lemma bounding the approximation guarantee of the algorithm to be $1-\epsilon$ using Lemma 5.2.

Corollary 5.4. *Let v be the smallest value in input string I. Let ℓ be the largest value such that $\mathcal{L}_I(v,\ell) < \infty$, then the length of the LIS of I is at least $\ell / \left(1 - \frac{\epsilon}{10\log n}\right)^{\lceil \log n \rceil} \geq \ell/(1-\epsilon)$.*

Making the Algorithm Distributed. Now we discuss how to parallelize the algorithm. Assume that machine i stores the positions $i \cdot \lfloor \frac{n}{m} \rfloor + 1$ to $(i+1) \cdot \lfloor \frac{n}{m} \rfloor$ for $i = 0, 1, \ldots, m$. This remains on the machine throughout the computation of the algorithm.

Step 1, 2, 3(a) are all easily adapted to the distributed setting. Consider step 3(b)i. Fix the strings I^*, I', and I'' as in this step of the algorithm. Let M'' be the set of machines that I'' is stored on. These machines sort the elements in I'' by their values. This can be done in $O(1)$ rounds. Then, the machines M'' perform a binary search for each $v \in I'$ and geometric ℓ to find $\tau(v,\ell)$. This takes $O(\frac{1}{\delta})$ rounds if the machines have memory n^δ by performing a B-tree search. This is sufficient to compute the minimum in this step.

For 3(b)ii, we first sort all the elements in I^* by the values. We view all the elements of I^* as the leaves of a binary tree. We compute the smallest value of $\mathcal{L}_{I^*}(v,\ell)$ for every internal node of the tree for every geometric ℓ in a bottom up way. Then we can find the smallest $\mathcal{L}_{I^*}(v',\ell)$ for every $v' \geq v$ for every v, ℓ by binary search. This also takes $O(\log n)$ rounds.

Overall algorithm takes $O(\log n)$ rounds. There are $O(\log n)$ steps in the sequential algorithm given. Each step can be implemented in $O(1)$ rounds.

5.2 LIS in Constant Rounds

Consider an instance of the longest increasing subsequence (LIS) problem where I is an input string of length n. We assume that I is a string of integers. Let S be the sorted version of the string[3].

[3]Note that sorting can be done in the massively parallel (distributed) setting in $O(1)$ rounds so long as the machines have memory at least n^δ for constant $\delta > 0$. This follows by adapting sample sort [21].

Throughout the section, I is referred to as the *input string* and S the *sorted string*. Our goal is to design a efficient distributed $(1-\epsilon)$-approximate algorithm for the problem. Let the number of machines available be $\Theta(m)$ and assume that each of the machines has memory $\tilde{\Theta}(n/m)$. This section describes an algorithm for LIS when the memory on each machine is at least $\tilde{\Omega}(n^{3/4})$.

As in the previous section, the position x of a string I' to be the xth entry in I' and the value to be the number stored in this position. Whenever a position of I is stored on a machine it is assumed that both the position and value are stored on the machine. Let I_i (respectfully, S_i) be the substring of I (resp. S) consisting of the positions $i \cdot \left\lfloor \frac{n}{m} \right\rfloor + 1$ to $(i+1) \cdot \left\lfloor \frac{n}{m} \right\rfloor$ for $i = 0, 1, \ldots, m$.

The following definition will be crucial for the design of the algorithm.

Definition 5.5. An interval S_i is called a **crossing** interval with respect to an increasing subsequence A if $A \cap S_i$ includes element from more than one interval I_j.

The design of the algorithm is based on the following property. This property captures the **decomposability** of the optimal solution.

Lemma 5.6. *For any $0 < \epsilon < 1$ there exists an $1-\epsilon$ approximate solution A such that the sorted intervals can be partitioned into sets $P_1, P_2, \ldots P_k$ where each P_j has the following properties.*

- *P_j includes a continuos set of intervals. That is, there is values a and b such that $P_j = \{S_a, S_{a+1}, S_{a+2}, \ldots, S_b\}$.*
- *P_j contains at most $\frac{10}{\epsilon^2}$ crossing intervals*
- *Consider any input interval I_j. If it is the case that $A \cap I_j$ includes elements in two different sorted intervals S_a and S_b then both S_a and S_b must be in the same partition.*

Overview of the Algorithm: The previous lemma captures the decomposability of the optimal solution that will enable our algorithm to run in $O(1)$ rounds. Intuitively, the lemma states that there is a near optimal solution that allows the sorted intervals S_i to be partitioned in a way that there is no interaction between the partitions. Call any $1-\epsilon$ approximate increasing subsequence with the properties given in the lemma $(1-\epsilon)$-**breakable**. Each partition is contiguous, has few crossing intervals and, further, the intersection of the optimal solution and a fixed input interval can be associated with a *single* partition. The last property ensures that there is no interaction between the increasing subsequences in the different partitions.

Fix a $(1-\epsilon)$-breakable solution. Assume the algorithm knows the partitioning given in the lemma and the input intervals associated with each partition. In this case, the algorithm focuses on constructing the LIS for each partition separately. Since there is no overlap between the partitions for sorted intervals or input intervals, a concatenation of the LIS for each subproblem gives the overall optimal solution.

When focusing on an algorithm for each partition, the second property given in the lemma states that the solution is not too complicated. In particular, the lemma states that there is little interaction between the sorted intervals. Indeed, handling crossing intervals is challenging, but we are guaranteed that there is a small number of them. This can be leveraged in the algorithm as follows. The algorithm computes a solution with β crossing intervals assuming that each machine knows the best solution for the subproblem of only having $\beta - 1$ crossing intervals. The idea is that communication between machines is required to coordinate the solution when there are crossing intervals. However, by enforcing there to be few crossing intervals, we can handle one additional crossing interval per round.

If the algorithm knew the partitioning, then this idea can be used to construct the solution. Unfortunately, it is non-obvious how to determine the partitioning. To do this, the algorithm computes the LIS for all pairs i, j, defining sequences of sorted intervals $S_i, S_{i+1}, \ldots S_j$ and all pairs a, b defining a sequence of input intervals $I_a, I_{a+1}, \ldots, I_b$. This LIS only allows elements in the intersection of these two groups and additionally only allows for some small number β of crossing intervals. The solution for $\beta + 1$ can be computed using the information for β over all i, j, a, b. The solutions are computed for all i, j, a, b and allows for up to $\frac{10}{\epsilon^2}$ crossing intervals. The lengths of all such solutions are communicated to a single machine. This machine uses dynamic programming to determine the partition, which can be found since the machine knows the LIS for each possible partition with a small number of crossing intervals. Key is that all entries can fit onto a single machine, which will be the case assuming the memory is at least $\tilde{\Omega}(n^{3/4})$ on each machine and by using monotonicity to remove the need to store information for one of the indices (in particular b is not required).

Using the above high-level ideas, we obtain the following result. The proof is deferred to the full version of this paper.

Theorem 5.7. *There is a $1-\epsilon$ approximation algorithm for longest increasing subsequence on an input of length n that uses m machines, $\tilde{O}(\frac{1}{\epsilon^4} n/m)$ memory and $O(1)$ rounds when the machines $m \leq n^{1/4}$ and fixed constant $\epsilon > 0$.*

REFERENCES

[1] http://hadoop.apache.org.
[2] https://spark.apache.org/.
[3] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 202–211, 2015.
[4] C. E. R. Alves, E. N. Cáceres, and F. Dehne. Parallel dynamic programming for solving the string editing problem on a cgm/bsp. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 275–281, New York, NY, USA, 2002. ACM.
[5] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *STOC*, 2014.
[6] Rumen Andonov, Frédéric Raimbault, and Patrice Quinton. *Dynamic programming parallel implementations for the knapsack problem.* PhD thesis, INRIA, 1993.
[7] Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990.
[8] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S.-H. Teng. Constructing trees in parallel. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 421–431, New York, NY, USA, 1989. ACM.
[9] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454–465, 2012.
[10] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *PVLDB*, 5(7):622–633, 2012.
[11] MohammadHossein Bateni, Aditya Bhaskara, Silvio Lattanzi, and Vahab S. Mirrokni. Distributed balanced clustering via mapping coresets. In *NIPS*, pages 2591–2599, 2014.
[12] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32Nd Symposium on Principles of Database*

Systems, PODS '13, pages 273–284, 2013.

[13] Phillip G. Bradford, Gregory J. E. Rawlins, and Gregory E. Shannon. Efficient matrix chain ordering in polylog time. *SIAM Journal on Computing*, 27(2):466–490, 1998.

[14] Phillip Gnassi Bradford. *Parallel dynamic programming*. PhD thesis, Indiana University, 1994.

[15] Andrei Z. Broder, Lluis Garcia Pueyo, Vanja Josifovski, Sergei Vassilvitskii, and Srihari Venkatesan. Scalable k-means by ranked retrieval. In *WSDM*, pages 233–242, 2014.

[16] C. Cerin, C. Dufourd, and J. F. Myoupo. An efficient parallel solution for the longest increasing subsequence problem. In *Computing and Information, 1993. Proceedings ICCI '93., Fifth International Conference on*, pages 220–224, May 1993.

[17] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *WWW*, pages 231–240, 2010.

[18] Artur Czumaj. An optimal parallel algorithm for computing a near-optimal order of matrix multiplications. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, SWAT '92, pages 62–72, London, UK, UK, 1992. Springer-Verlag.

[19] Artur Czumaj. *Parallel algorithm for the matrix chain product and the optimal triangulation problems (extended abstract)*, pages 294–305. Springer Berlin Heidelberg, 1993.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

[21] Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using mapreduce. In *KDD*, pages 681–689, 2011.

[22] Alina Ene and Huy L. Nguyen. Random coordinate descent methods for minimizing decomposable submodular functions. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 787–795, 2015.

[23] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Trans. Algorithms*, 6(4), 2010.

[24] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4, 2012.

[25] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than o(1) dependency. *Journal of Parallel and Distributed Computing*, 21(2):213 – 222, 1994.

[26] Ashish Goel and Kamesh Munagala. Complexity measures for map-reduce, and comparison to parallel computing. *CoRR*, abs/1211.6526, 2012.

[27] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, pages 374–383, 2011.

[28] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Eng.*, 15(3):515–528, 2003.

[29] Sungjin Im and Benjamin Moseley. Brief announcement: Fast and better distributed mapreduce algorithms for k-center clustering. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 65–67, 2015.

[30] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 41–52, 2010.

[31] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.

[32] D. G. Kirkpatrick and T. Przytycka. Parallel construction of near optimal binary trees. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '90, pages 234–243, New York, NY, USA, 1990. ACM.

[33] Peter Krusche and Alexander Tiskin. New algorithms for efficient parallel string comparison. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 209–216, New York, NY, USA, 2010. ACM.

[34] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in mapreduce and streaming. *TOPC*, 2(3):14, 2015.

[35] M. Lu and H. Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):835–848, August 1994.

[36] Gustavo Malkomes, Matt Kusner, Wenlin Chen, Kilian Weinberger, and Benjamin Moseley. Fast distributed k-center clustering with outliers on massive data. In *NIPS*, 2015.

[37] Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.

[38] Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *NIPS*, pages 2049–2057, 2013.

[39] Andrea Pietracaprina, Geppino Pucci, Matteo Riondato, Francesco Silvestri, and Eli Upfal. Space-round tradeoffs for mapreduce computations. In *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*, pages 235–244, 2012.

[40] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits: (on lower bounds for modern parallel computation). In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 1–12, 2016.

[41] Wojciech Rytter. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science*, 59(3):297 – 307, 1988.

[42] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[43] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[44] V. Viswanathan, S. H. S. Huang, and Hongfei Liu. Parallel dynamic programming. In *Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on*, pages 497–500, Dec 1990.

[45] Zhao Zhao, Guanying Wang, A.R. Butt, M. Khan, V.S.A. Kumar, and M.V. Marathe. Sahad: Subgraph analysis in massive networks using hadoop. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 390–401, May 2012.