Gupta, Chetan; Latypov, Rustam; Maus, Yannic; Pai, Shreyas; Särkkä, Simo; Studený, Jan; Suomela, Jukka; Uitto, Jara; Vahidi, Hossein

Fast Dynamic Programming in Trees in the MPC Model

Published in:
SPAA 2023 - Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures

DOI:
10.1145/3558481.3591098

Published: 17/06/2023

Document Version
Publisher's PDF, also known as Version of record

Fast Dynamic Programming in Trees in the MPC Model

Chetan Gupta
Aalto University
Espoo, Finland
chetan.gupta@aalto.fi

Rustam Latypov
Aalto University
Espoo, Finland
rustam.latypov@aalto.fi

Yannic Maus
TU Graz
Graz, Austria
yannic.maus@ist.tugraz.at

Shreyas Pai
Aalto University
Espoo, Finland
shreyas.pai@aalto.fi

Simo Särkkä
Aalto University
Espoo, Finland
simo.sarkka@aalto.fi

Jan Studený
Aalto University
Espoo, Finland
jan.studeny@aalto.fi

Jukka Suomela
Aalto University
Espoo, Finland
jukka.suomela@aalto.fi

Jara Uitto
Aalto University
Espoo, Finland
jara.uitto@aalto.fi

Hossein Vahidi
Aalto University
Espoo, Finland
hossein.vahidi@aalto.fi

ABSTRACT

We present a deterministic algorithm for solving a wide range of *dynamic programming problems* in trees in $O(\log D)$ rounds in the massively parallel computation model (MPC), with $O(n^\delta)$ words of local memory per machine, for any given constant $0 < \delta < 1$. Here D is the diameter of the tree and n is the number of nodes—we emphasize that our running time is independent of n.

Our algorithm can solve many classical *graph optimization problems* such as maximum weight independent set, maximum weight matching, minimum weight dominating set, and minimum weight vertex cover. It can also be used to solve many *accumulation* tasks in which some aggregate information is propagated upwards or downwards in the tree—this includes, for example, computing the sum, minimum, or maximum of the input labels in each subtree, as well as many inference tasks commonly solved with belief propagation. Our algorithm can also solve any *locally checkable labeling problem* (LCLs) in trees. Our algorithm works for any reasonable representation of the input tree; for example, the tree can be represented as a list of edges or as a string with nested parentheses or tags. The running time of $O(\log D)$ rounds is also known to be necessary, assuming the widely-believed 2-cycle conjecture.

Our algorithm strictly improves on two prior algorithms:

(1) Bateni, Behnezhad, Derakhshan, Hajiaghayi, and Mirrokni [ICALP'18] solve problems of these flavors in $O(\log n)$ rounds, while our algorithm is much faster in low-diameter trees. Furthermore, their algorithm also uses randomness, while our algorithm is deterministic.
(2) Balliu, Latypov, Maus, Olivetti, and Uitto [SODA'23] solve only locally checkable labeling problems in $O(\log D)$ rounds, while our algorithm can be applied to a much broader family of problems.

SPAA '23, June 17–19, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591098

CCS CONCEPTS

• **Theory of computation** → **Parallel computing models**; **Distributed computing models**; **Dynamic programming**; • **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

massively parallel model, MPC, trees, dynamic programming, accumulation, aggregation, locally checkable labeling, LCL, statistical inference, graphical models

ACM Reference Format:
Chetan Gupta, Rustam Latypov, Yannic Maus, Shreyas Pai, Simo Särkkä, Jan Studený, Jukka Suomela, Jara Uitto, and Hossein Vahidi. 2023. Fast Dynamic Programming in Trees in the MPC Model. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3558481.3591098

1 INTRODUCTION

In this work we present a general, unified algorithm framework for solving a very wide variety of computational problems related to *tree-structured data* in a massively parallel setting. Some examples of tasks that can be solved with our algorithm include:

- Solving traditional graph optimization problems in trees (e.g., finding a maximum-weight independent set or minimum-weight dominating set).
- Solving constraint-satisfaction problems in trees (e.g., finding a solution to any locally checkable labeling problem [23], as well as many generalizations of the theme).
- Analyzing large text documents with tree-structured data (e.g., processing large XML [10] documents).
- Aggregating information in trees (e.g., calculating the sum of inputs in each subtree [15]—this is a generalization of the classical prefix sum operation [22] from directed paths to rooted trees).
- Performing statistical inference in tree-structured graphical models (e.g., computations that are in the classical sequential setting commonly done with belief propagation [21]).

1.1 Setting: MPC Model

We work in the usual *massively parallel computation* model (MPC) [20]. The size of the input is n words—here n is much larger than what fits in the local memory of a single computer, and therefore the input is distributed among multiple computers. The local memory of each computer is $\Theta(n^\delta)$ words, for some constant $0 < \delta < 1$. We have got $\Theta(n^{1-\delta})$ computers that take part in the computation, and hence in total $\Theta(n)$ words of distributed memory.

We will assume that the key bottleneck is communication between computers, and hence the time complexity is measured in the *number of communication rounds*. We will assume that in one round each computer can send up to $\Theta(n^\delta)$ words to other computers and receive up to $\Theta(n^\delta)$ words from other computers. In essence, you can send everything you have in your local memory to someone else, and you can receive whatever fits in your local memory. When we refer to the *running time* in this work, we always refer to the number of communication rounds (but we point out already here that in our algorithms local computation will also be lightweight).

1.2 Prior Work: Solving LCL Problems Fast

In a recent work, Balliu, Latypov, Maus, Olivetti, and Uitto [4] presented efficient MPC algorithms for finding connected components, rooting trees, and solving so-called *locally checkable labeling problems* (LCLs) in forests. As we directly build on their work, we will first briefly discuss their contributions.

LCL problems were first formalized by Naor and Stockmeyer [23]. These are graph problems that can be specified by listing a *finite* set of feasible local neighborhoods. For example, "5-coloring a graph of maximum degree 4" is an example of an LCL problem; we can list all properly 5-colored neighborhoods that may occur in a graph of maximum degree 4. Typically, constraint satisfaction problems are LCLs (as long as we have bounded degrees and a finite label set), while global optimization problems like maximum-weight independent set are not LCLs.

The algorithms in [4] run in $O(\log D)$ rounds, where D is the diameter of the input graph, with no asymptotic global memory overhead. Finding connected components and rooting are their main contributions, but here we are primarily interested in the part that solves LCL problems.

The algorithm for solving LCL problems consists of phases that compress the input graph; there are $O(1)$ phases and each phase takes $O(\log D)$ rounds. After phase i, they define a new LCL problem on the compressed graph such that its solution can be expanded into a solution for the LCL problem defined on the graph of phase $i-1$. After performing $O(1)$ phases the graph is compressed into a single node (the root of the tree) for which any LCL problem is trivially solved. The algorithm then finishes off with $O(1)$ reversal phases that decompress all compressed parts while simultaneously spreading the correct LCL solution to the decompressed parts of the graph.

1.3 Key New Contributions: Unified Framework for Dynamic Programming Problems

We build on [4] and present a new algorithm framework, with the following main features:

Table 1: Examples of problems solved with our framework and the prior work [4].

Problem	Prior [4]	This work
Vertex coloring	✓	✓
Edge coloring	✓	✓
Maximal independent set	✓	✓
Maximum weight independent set	—	✓
Maximum weight matching	—	✓
Minimum weight dominating set	—	✓
Minimum weight vertex cover	—	✓
Weighted max-SAT problem	—	✓
Longest path problem	—	✓
Sum coloring problem	—	✓
Counting matchings modulo k	—	✓
Tree median problem	—	✓
Inference in Bayesian graphical models	—	✓
Evaluating arithmetic expressions	—	✓
Verifying the structure of XML-like documents	—	✓
Computing the sum, minimum, or maximum of the input labels in each subtree	—	✓

(1) We are able to solve a *much* broader family of problems in $O(\log D)$ time—instead of solving only LCL problems, we can solve a much more general family of so-called *dynamic programming problems* (see Definition 1). We refer to Table 1 for some examples of the applicability of our framework in comparison with [4].
(2) The prior algorithm [4] intermixes the tasks of compressing the tree and constructing the solution for an LCL. We show that it is possible to separate the concerns, as we will outline in Section 1.4. In particular, we can first use $O(\log D)$ rounds to construct a hierarchical clustering of the graph, and then with the help of the clustering, we can solve any dynamic programming problem in $O(1)$ rounds.

The fastest prior algorithm for dynamic programming in the MPC model was the algorithm by Bateni, Behnezhad, Derakhshan, Hajiaghayi, and Mirrokni [5, 6], but the running time of their algorithm is $O(\log n)$, which can be much worse than $O(\log D)$ in low-diameter trees, and moreover their algorithm is randomized while our algorithm is deterministic.

1.4 Simple Three-Step Approach

Our algorithm framework proceeds in three steps:

(1) We turn the input into a **standard representation**; the running time of this phase is $O(\log D)$ rounds. We work with tree-structured data, but such data can be represented in different forms: we might have e.g. an unrooted tree that is represented as a long list of undirected edges, or we might have a rooted tree that is represented as a very long string (e.g. a string with nested parentheses or nested pairs of opening and closing tags). We will turn any such representation

into a more convenient standard form: we will have a *rooted tree* that is represented as *list of directed edges*. We show that for a wide range of commonly-used representations of tree-structured data, this can be solved in $O(\log D)$ rounds. This is the only step that depends on the precise input representation. We will give the details in Section 3.

(2) We construct a **hierarchical clustering** of the tree; the running time of this phase is $O(\log D)$ rounds. We will introduce the properties of the hierarchical clustering in Section 1.5. We will show that such a clustering can be computed in $O(\log D)$ rounds. This step is fully generic—it depends neither on the input representation nor on the problem that we are solving. We will give the details in Section 4.

(3) We **solve the problem of interest**; the running time of this phase is $O(1)$ rounds. We show that we can solve a very wide variety of problems related to tree-structured data in $O(1)$ rounds, given the hierarchical clustering. We will give the details in Section 5.

Overall, this approach makes it possible to solve various computational problems in $O(\log D)$ rounds in trees. Furthermore, this results in algorithms that are *conditionally optimal*: many problems that can be solved with this framework require $\Omega(\log D)$ rounds, assuming the (widely-believed) two-cycle conjecture [1, 2, 14, 24]. The conjecture states that $\Omega(\log n)$ MPC-rounds are required to decide whether an input graph consists of a cycle of length n or two cycles of length $n/2$, even if a polynomial number of machines is available. It is known that this conjecture implies that finding connected components requires $\Omega(\log D)$ rounds [7, 11], which in turn can be used to show that solving a subset of dynamic programming problems on trees requires $\Omega(\log D)$ rounds [4].

The main conceptual message of our work is this:

> There exists a single, convenient, universal representation that one can use as a starting point for designing very efficient massively parallel algorithms for tree-structured data.

We emphasize that the hierarchical clustering needs to be computed only once for a given input topology, and it can be reused for any dynamic programming problem and any input values.

1.5 Hierarchical Clustering

Our hierarchical clustering is illustrated in Fig. 1. For convenience, we assume that all nodes of the tree have outdegree 1; to ensure this we add at the root an additional virtual edge pointing outside the tree—this edge will be ignored when solving the problem of interest.

To construct the hierarchical clustering, we start with the original tree (this is our layer 0). To obtain layer $i+1$, we contract a *cluster* of nodes into one node. The key properties that we ensure are:

- Each cluster contains only $O(n^\delta)$ nodes.
- Each cluster has outdegree 1.
- Each cluster has indegree 0 or 1.
- There are only $O(1)$ layers, and the topmost layer consists of only one cluster.

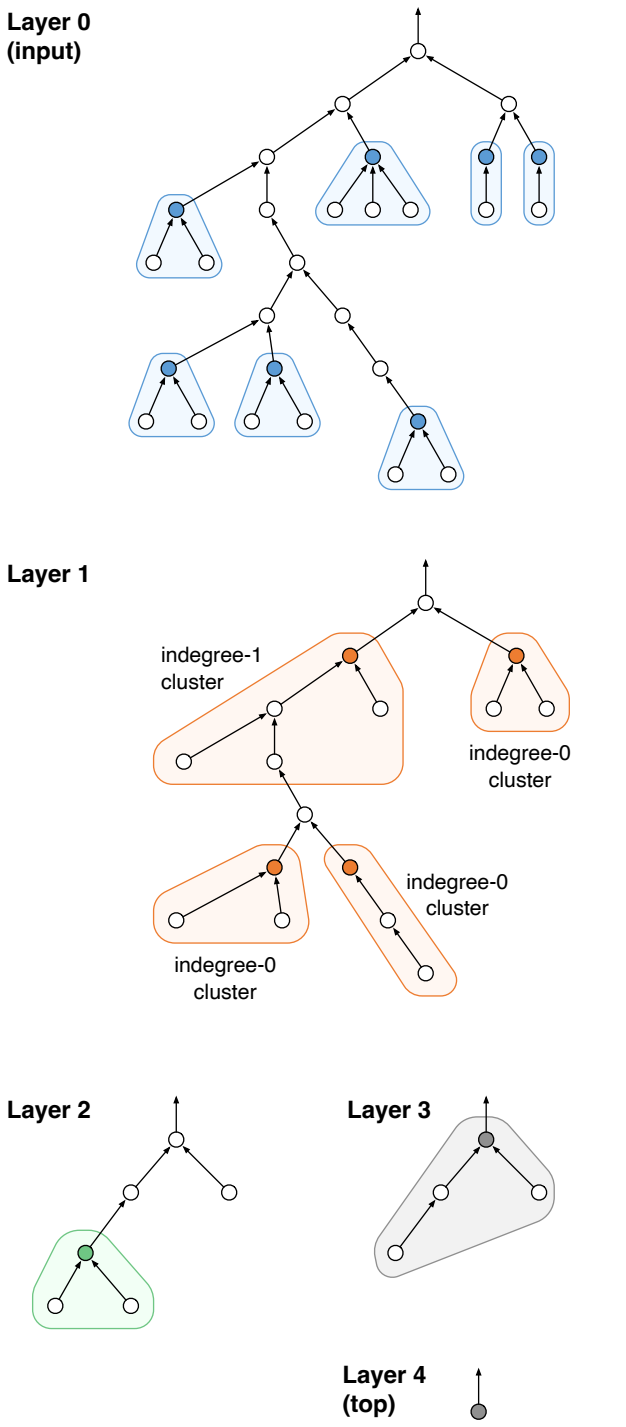

Figure 1: Our hierarchical clustering consists of constantly many layers. Layer 0 is the input tree. At each layer we compress some disjoint collection of clusters so that eventually we have got only one node left. Each cluster contains at most n^δ nodes, each cluster has got exactly one outgoing edge, and there are zero or one incoming edges.

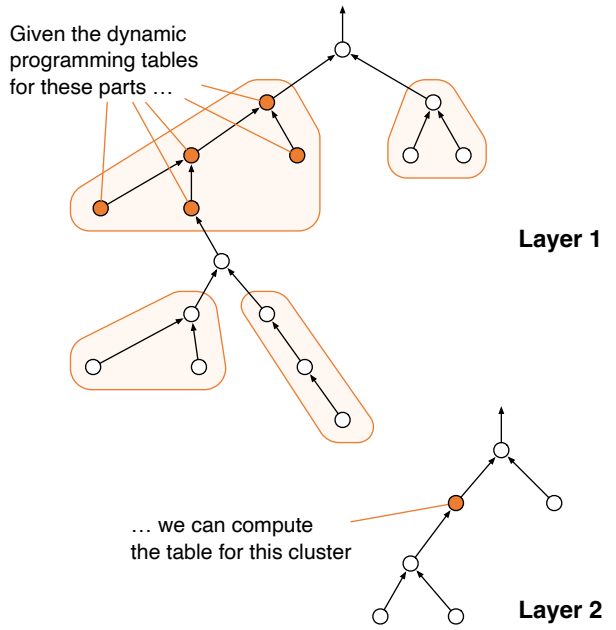

Figure 2: From bottom to top: given the summaries inside a cluster, we assume we can compute the summary for the entire cluster.

We formally define the hierarchical clustering in Section 4, and we further show that it not only exists, but can also be computed in $O(\log D)$ rounds in the MPC model.

1.6 Dynamic Programming Problems

Our main focus is on problems that we will call *dynamic programming problems*; as we will see in Section 1.6.1, it is straightforward to adapt many typical optimization problems into this framework:

Definition 1. A dynamic programming problem (DP problem) is a computational problem in trees with the following properties:

(1) The task is to compute a *label* for each edge.
(2) We can summarize each cluster C with a *dynamic programming table* $f(C)$ that can be represented with $O(1)$ words.
(3) Given such summaries for all nodes that form a cluster C, we can compute in the dynamic programming table $f(C)$, using only $O(|C|)$ words of additional space; see Fig. 2.
(4) We can compute the label for the outgoing edge of the top-level cluster C given $f(C)$.
(5) Assuming that we know the labels of the incoming and outgoing edges of a cluster C and the dynamic programming tables for each component of C, we can also compute the labels of all internal edges of cluster C, using only $O(|C|)$ words of additional space; see Fig. 3.

Here the labels of the edges are an abstraction of whatever is the specific task we are solving, while the dynamic programming tables are auxiliary data structures needed during the algorithm.

1.6.1 Example: Maximum-Weight Independent Set. We will use the maximum-weight independent set problem (MaxIS) as a running example: in our input, each node has a nonnegative weight, and the task is to find a maximum-weight subset of nodes $X \subseteq V$ such that there is no edge $(u, v) \in E$ with $u \in X$ and $v \in X$.

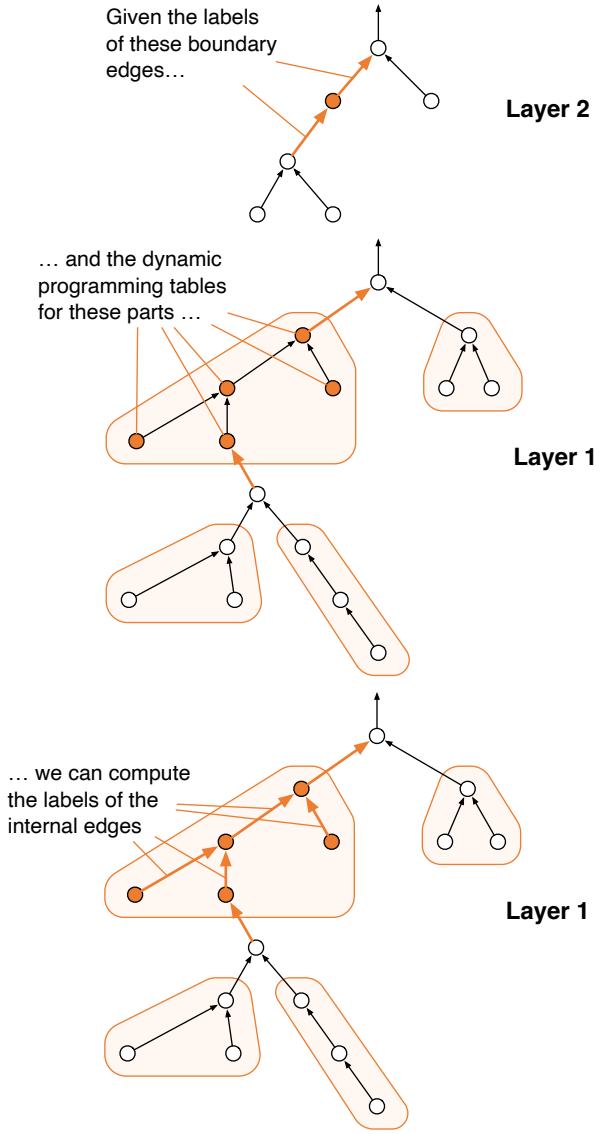

Figure 3: From top to bottom: given the solutions at the boundary edges, we assume we can compute the solution also for the internal edges.

Now the MaxIS problem is an example of a DP problem, with the following interpretation:

- The label of the edge (u, v) indicates whether $u \in X$.
- Let C be an indegree-0 cluster, where (u, v) is the outgoing edge. Then $f(C)$ is a table with two elements: (1) the weight of the heaviest independent set in C such that $u \in X$, and the (2) the weight of the heaviest independent set in C such that $u \notin X$.

- Let C be an indegree-1 cluster, where (u, v) is the outgoing edge and (s, t) is the incoming edge. Now $f(C)$ is a table with four elements: the weight of the heaviest independent set in C for all combinations of $u \in X$ vs. $u \notin X$ and $t \in X$ vs. $t \notin X$.

It is now easy to work out the details of the bottom-up and top-down phases. Note that the way we handle indegree-0 clusters is, in essence, identical to the classical centralized, sequential algorithm that solves MaxIS in trees (see e.g. [13, Sect. 6.7]). The way we handle indegree-1 clusters can be seen as a special case of the centralized, sequential algorithm that solves MaxIS in bounded-treewidth graphs [9]: we can summarize clusters with a constant number of interfaces to the rest of the graph, and we can merge such clusters.

1.6.2 Beyond Dynamic Programming. While we use the term *dynamic programming* here to capture the problem family of interest, we would like to emphasize that there is a broad range of problems that are compatible with this framework even if one does not usually think that they have got anything to do with dynamic programming (recall Table 1).

1.7 Technicality: Very High Degrees

So far we have ignored one technical difficulty: what if our input tree has nodes of degree more than n^δ. In such a case it is impossible to find small clusters, as the cluster that contains node v will also contain all of its children.

Fortunately, for many problems such as MaxIS, we can easily modify the input and the problem slightly, so that we replace each node v of degree more than $n^{\delta/2}$ with an $O(1)$-depth tree T_v. The new edges are equipped with additional labels so that we can handle them correctly in the dynamic programming algorithm and ensure that all nodes in T_v make the same consistent choice.

We discuss this in more detail in Sections 4.4 and 5.3. To summarize, we can solve in any DP problem (Definition 1), as long as we have degrees at most $n^{\delta/2}$ or we can reduce the degree as needed by replacing high-degree nodes with low-degree trees.

1.8 Further Discussion on Related Work

1.8.1 Bateni, Behnezhad, Derakhshan, Hajiaghayi, and Mirrokni. The prior work [5, 6] presents an MPC algorithm for dynamic programming in trees in $O(\log n)$ rounds in the MPC model. While the precise family of problems that they handle is phrased somewhat differently, the spirit is the same—they can also solve problems similar to the MaxIS problem.

Our work strictly improves on their work in two ways: our running time is $O(\log D)$, which is conditionally optimal, while their running time is $O(\log n)$, and our algorithm is deterministic, while their algorithm uses randomness.

In the full version of this work, we also show how to solve a problem called *tree median* using our framework. This is a problem engineered so that it does *not* satisfy the property of *binary adaptability*, which is a technical requirement used in [5, 6]. Informally, in binary adaptable problems one can replace high-degree nodes with binary trees, and hence it is sufficient to solve dynamic programming problems in bounded-degree trees; however, the tree median problem does not admit such a straightforward degree reduction. We hope this problem serves as a demonstration of the broad applicability of our framework, also beyond what was considered in prior work.

1.8.2 Balliu, Latypov, Maus, Olivetti, and Uitto. The prior work [4] presents an MPC algorithm for solving locally checkable labeling problems (LCLs) in trees in $O(\log D)$ rounds in the MPC model. Our running time is the same, but we solve a much broader family of problems (recall Table 1).

We make use of many subroutines and ideas developed in [4]. For example, we make use of their algorithm for rooting a tree, and the idea of the hierarchical clustering as well as its key properties are due to them.

From the conceptual perspective, the key difference is that their work presents a single (arguably rather complicated) algorithm that intermixes the tasks of clustering the tree and constructing the solution for an LCL. The hierarchical clustering is rather implicit, and it has got properties that make it not directly applicable for solving a broad variety of problems: for example, arbitrarily long paths are compressed into one cluster, which will then no longer fit in the memory of one computer, and leaf nodes are aggressively eliminated, which is not compatible with all dynamic programming problems. In our algorithm the hierarchical clustering is built first, explicitly, and our clustering has got convenient properties that allow us to do per-cluster computations locally inside one computer, and it also allows us to tackle a broad range of problems.

1.8.3 Other Related Work. While our technique is conditionally optimal for the *family* of dynamic programming problems, there are many problems that allow faster algorithms in certain cases. For example, Balliu, Brandt, Fischer, Latypov, Maus, Olivetti, and Uitto [3] consider classes of LCL problems that are local in nature, such as the MIS problem. For many classes of natural problems, they give MPC algorithms that are much more efficient than $\Theta(\log D)$ for high diameter graphs.

Im, Moseley, and Sun [19] consider dynamic programming in the MPC model for problems that are not directly related to tree-structured inputs.

There is a related yet more powerful model called AMPC in which machines, in addition to the regular MPC operations, can perform a sublinear number of (adaptive) queries to a distributed hash table per round. In the AMPC model, the problem of computing subtree sizes can be solved in $O(1)$ rounds [8].

In the classic PRAM model, problems of the same flavor have been studied already in the 1990s—for example, Gibbons, Cai, and Skillicorn [15] present an algorithm for upwards and downwards accumulation in trees that runs in $O(\log n)$ time. We emphasize that while $\Omega(\log n)$ is a natural lower bound for all such problems in the PRAM model, we can nevertheless achieve a running time of $O(\log D)$ in the MPC model.

2 PRELIMINARIES

We make use of the following primitives: *sorting* an array of n elements and computing *prefix sums* in an array of n elements. Both of these operations can be solved in the MPC model with a deterministic algorithm in $O(1)$ rounds, see [12, 16, 17].

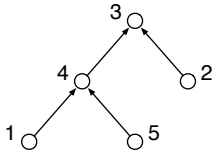

Figure 4: Tree T used as an example in Section 3.1.

3 INPUT REPRESENTATIONS

Our algorithm in Sections 4 and 5 will assume that the input tree is rooted and it is given as a set of directed edges such that each edge goes from a child to its parent node. However, in addition to this standard representation there are various other ways to represent a tree using an array. In this section, we define other commonly used representations and show that we can transform the input from any of these representations to an array of directed edges in $O(1)$ rounds—in the full version of this work we will also show how our algorithm framework makes it possible to turn the standard representation back to any of these representations.

3.1 Definitions

We consider tree-structured data represented in one of the following forms; we use the tree T illustrated in Fig. 4 as an example:

- **List-of-edges:** This is the representation that our algorithm works with. Each element in the input array contains a pair of integers that represents a directed edge in a tree going from a child to its parent. Tree T can be described as an array $[(1,4),(2,3),(5,4),(4,3)]$, if we use the labeling of the nodes given in Fig. 4.
- **String-of-parentheses:** In this representation, the tree is given as an array of properly nested parentheses or, equivalently, opening and closing tags. Each node in the tree is represented by two parentheses "(" and ")". We can interpret the array as a rooted tree in a bottom-up manner, with the leaf nodes represented as an empty pair of parentheses "()". The outermost pair of parentheses represents the root node. For example, T can be represented as an array $[(,(,),(,),),(,),)]$.
- **BFS-traversal:** The array represents the BFS-traversal of the tree: the indices of the array denote the nodes in the tree in the BFS order, and an array element contains the index of the parent node. Tree T can be represented as $[-,1,1,2,2]$.
- **DFS-traversal:** Similar to the above, the tree is given as an array that represents a DFS traversal of the tree. Tree T can be represented as $[-,1,2,2,1]$.
- **Pointers-to-parents:** Similar to the above, but the nodes are ordered arbitrarily. Tree T can be represented as $[4,3,-,3,4]$, if we order the nodes according to their labels in Fig. 4.

3.2 Normalizing the Representation

If the tree is originally given as a list of undirected edges, we can first root the tree at an arbitrary node and orient the edges in $O(\log D)$ rounds, using the algorithm from [4].

BFS-traversal, DFS-traversal, and pointers-to-parents already represent the input as a set of directed edges in different manners, and hence it is easy to turn them into a list-of-edges representation. The nontrivial part is to prove that we can obtain the **list-of-edges representation from string-of-parentheses** in $O(1)$ rounds in the MPC model.

For brevity, we will show how we can do this transformation for $\delta = 1/2$, i.e., assuming there are $m = \sqrt{n}$ computers each with $O(\sqrt{n})$ memory—in the full version of this work we will present the details of how to generalize the same strategy to any δ.

Let A be the array that contains properly nested parentheses. We assume that each opening parentheses "(" in A will represent a node in the tree. Now for each open parenthesis, we need to find its parent open parenthesis.

Initially, A is evenly distributed over $\sqrt{n}$ computers $N_0, \ldots N_{m-1}$ such that N_i contains the elements $A[i\sqrt{n}], \ldots, A[(i+1)\sqrt{n}-1]$. Let $A[i]$ and $A[j]$ be two opening parentheses such that $i < j$. We know that $A[i]$ is the parent of $A[j]$ if all the parentheses from $A[i+1]$ to $A[j-1]$ are properly nested. If $A[i]$ is the parent of $A[j]$ and both of them are stored in the same computer, then the computer can easily identify $A[i]$ as the parent of $A[j]$. The challenge is to identify the parent node if $A[i]$ and $A[j]$ are stored in different computers.

Notice that if $A[p]$ and $A[q]$ are a pair of opening and closing parentheses that denote the same node and both are stored in some computer N_i then $A[p]$ cannot be the parent of $A[k]$ if $A[k]$ is stored in some other computer. Thus, let us *cancel out* properly nested pairs of parentheses stored in a single computer. Now the remaining parentheses inside each computer N_i, will be nothing but a (possibly empty) sequence of closing parentheses followed by a (possibly empty) sequence of opening parentheses, for example, "))))))(((". Let S_i be the array of remaining parentheses in N_i.

Computer N_i computes a pair (c_i, o_i) where c_i and o_i is the number of closing and opening parentheses in S_i, and broadcasts it to all the other computers. Using this information, for each node we can identify the array S_i that contains its parent and also the index of the parent in S_i as follows. For each open parentheses $A[j]$ stored at N_i, N_i locally computes l_j and r_j that denote the number of closing and opening parentheses on the left and right side of $A[j]$, respectively, in S_i. Then $S[j]$ (stored in N_a) is the parent of $A[k]$ (stored in N_b) where $j < k$ and $a < b$, if a is the largest integer such that

$$r_j + \sum_{x=a+1}^{b} (o_x - c_x) - l_k = 0,$$

which can be computed in $O(1)$ rounds by N_b.

To identify the index of the parent of a node in A, we need to do some more calculations. For each node v we produce two tuples:

- Type 1: $[i, j, 1, v]$ denotes that node v is stored at the jth index of S_i—this information is readily available for the computer that holds node v.
- Type 2: $[i, j, 2, v]$ denotes that the *parent* of node v is stored at the jth index of S_i—this information can be computed as described above by the computer that holds node v.

This way we will have n tuples in total in the system, and we can sort them in $O(1)$ rounds. Once sorted, in the array there will always be one tuple of type 1, representing a node v, followed by zero or more tuples of type 2, representing the children of v. This way we can identify all parent–child edges in $O(1)$ rounds.

4 HIERARCHICAL CLUSTERING

In this section we present an $O(\log D)$-round algorithm that computes the hierarchical clustering required for our dynamic programming algorithm (see Section 5). Note that the clustering does not depend on the problem that we want to solve afterwards.

4.1 Definitions

We will now formalize the idea of hierarchical clustering that we introduced in Section 1.5; see Fig. 1 for an illustration.

Definition 2 (cluster)**.** A cluster C is a set such that each element is either a node u_i or another cluster C_i. We recursively define the set of nodes that participate in C as

$$V(C) = \bigcup_{C_i \in C} V(C_i) \cup \{u_i \mid u_i \in C\}.$$

We require that the cluster C contains at most n^δ elements, and the set of cut edges $(V(C), V \setminus V(C)) \subseteq E$ has exactly one outgoing edge and at most one incoming edge.

We classify clusters into two types based on the number of incoming edges: indegree-zero and indegree-one.

Definition 3 (hierarchical clustering)**.** A *hierarchical clustering* of a rooted tree $T = (V, E)$ is a collection of sets $S_0, S_1, \ldots, S_L$ called layers such that $L = O(1)$ and the following are satisfied

(1) each S_i consists of nodes or clusters,
(2) $S_0 = V$,
(3) For $i \geq 1$, (i) the nodes in S_i are also nodes in S_{i-1} and (ii) the clusters of S_i form a partition of the remaining elements of S_{i-1},
(4) S_L contains one element which is a cluster.

While it is easiest to grasp the clustering as a standalone *graph-theoretic concept* in order to use it algorithmically, we need to assign cluster IDs and store certain pointers between a cluster and its nodes/clusters, etc. More formally, we give each cluster $C \in S_i$ a unique cluster ID, and pointers to and from the clusters and nodes of S_{i-1} that are contained in C. Since a cluster has exactly one outgoing and at most one incoming edge, we can contract each cluster in S_i into a node, such that the resulting graph forms a tree T_i where each edge corresponds to an edge of the original tree.

4.2 Constructing the Clustering

As discussed in Section 3, we can without loss of generality assume that the input is a rooted tree $T = (V, E)$ with n nodes, represented as a list of edges. We will further assume that the maximum degree is $n^{\delta/2}$, but we will see how to overcome this limitation in Section 4.4. By sorting the edges, we can also assume that each node and its incident edges are hosted on the same machine. Our goal is to construct a hierarchical clustering as in Definition 3.

4.2.1 High-Level Idea. We will mostly follow the same ideas as what happens in the algorithm of [4]. However, there are two key differences that we will highlight in what follows, and we will also need to prove that the number of layers is still bounded by a constant.

We say that a subtree is a *caterpillar* if it is a tree containing a central path and all other nodes are within distance 1 from the path. We will alternate between two steps, for $O(1)$ iterations:

(1) Create indegree-zero clusters: we identify nodes v such that we can replace the entire subtree $T(v)$ rooted at v with a cluster.
(2) Create indegree-one clusters: we identify a disjoint set of caterpillars that we can replace with clusters.

In [4], they entirely removed what we call indegree-zero clusters, and then they only needed to contract long paths. Furthermore, they contracted arbitrarily long paths, while our clusters cannot be too large. Nevertheless, we can show that we make enough progress and we can finish after $O(1)$ pairs of such steps.

In our algorithm we will *color* the nodes that correspond to indegree-zero clusters instead of removing them. Then we can largely follow the process and the analysis of [4] for the uncolored parts of the tree. As the colored nodes are always leaf nodes, and as each node can have at most $n^{\delta/2}$ neighbors, if we put into each cluster up to $n^{\delta/2}$ uncolored nodes, together with their colored neighbors the size of a cluster will be bounded by n^δ, as needed.

4.2.2 Creating Indegree-Zero Clusters. Following [4], we define that a node v with more than $n^{\delta/2}$ uncolored nodes in its subtree $T(n)$ is called *heavy*, and the rest of the nodes are *light*.

We apply the following result from Lemma 6.13 of [4] to the uncolored subgraph (i.e., the subgraph induced by the uncolored nodes): there exists a deterministic optimal space $O(\log D)$-time MPC algorithm (CountSubtreeSizes) in which every node v learns either the exact size of $T(v)$ or that $|T(v)| > n^{\delta/2}$.

With this information, we can identify each node u such that u is light but its parent v is heavy. We apply Lemma 6.14 from [4]: there exists a deterministic optimal space $O(\log D)$-time MPC algorithm (GatherSubtrees) to collect $T(u)$ into the machine hosting u for each such node u. Then, we replace $T(u)$ with an indegree-zero cluster, which is then represented as a colored node—see Fig. 5. The overall running time is $O(\log D)$. The size of the cluster will be bounded by n^δ, as there were only $n^{\delta/2}$ uncolored nodes, each with at most $n^{\delta/2}$ colored leaf nodes attached to it.

4.2.3 Creating Indegree-One Clusters. Now we are ready to describe the second step: creating indegree-one clusters. The idea is to identify long paths in the uncolored subgraph. A long path in the uncolored subgraph corresponds to a caterpillar if we also take into account the colored nodes.

We apply Lemma 6.17 from [4] to the uncolored subgraph: there exists a deterministic $O(\log D)$-time MPC algorithm (CountDistances) in which each degree-2 node knows its distance to both endpoints of the path formed by degree-2 nodes—see Fig. 5.

Using the distances, we will split each path P formed by degree-2 nodes in the uncolored subgraph into sub-paths of length at most $n^{\delta/2}$ (i.e., nodes with distance value $1, \ldots, n^{\delta/2}$ form the first sub-path and so on). We call these sub-paths path fragment P'. We collect each fragment in a single machine and form a cluster C by including also all colored nodes connected to P'. This will result in a caterpillar C, and as the maximum degree of the graph was $n^{\delta/2}$, the size of the cluster is at most n^δ, as required. The overall running time of this step is $O(\log D)$.

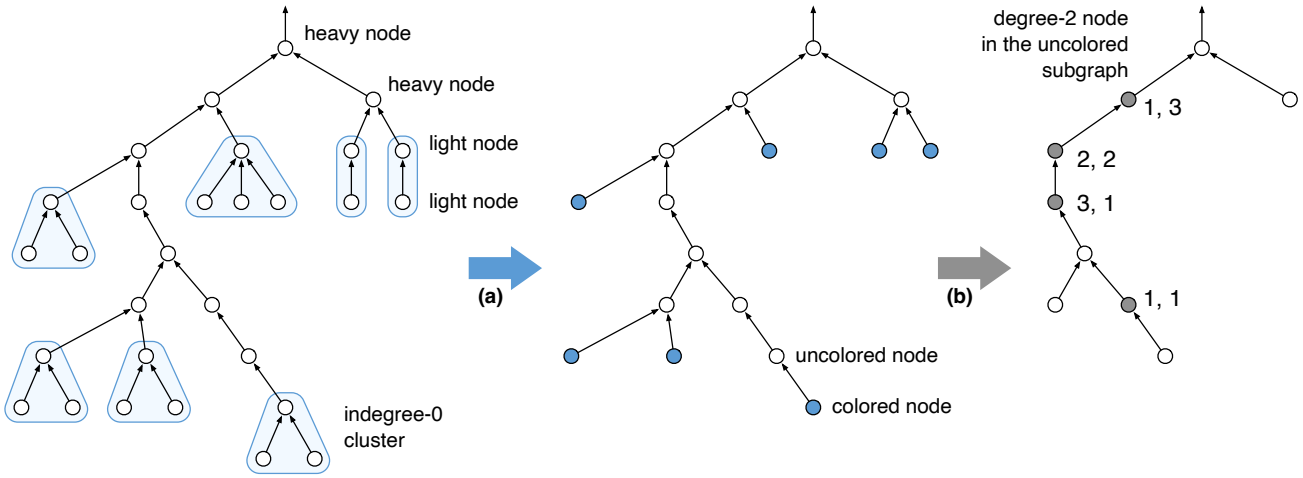

Figure 5: (a) Creating indegree-zero clusters. (b) Creating indegree-one clusters: we identify paths formed by degree-2 nodes in the subgraph induced by uncolored nodes and calculate their positions in the path both upwards and downwards.

4.3 Number of Layers

By construction, all clusters are sufficiently small. We still need to show that the number of layers is bounded by a constant:

Lemma 4. *The number of layers in the hierarchical clustering we created is* $O(1)$.

To prove Lemma 4, consider first an alternative process Π_1 where we delete indegree-zero clusters instead of marking them colored, and in which we replace arbitrarily long paths with one edge, similar to [4]. We can show:

Lemma 5. *Each iteration of process* Π_1 *makes the tree smaller by a factor of* $\Omega(n^{\delta/2})$.

Proof. Say we start with a tree T_0 with n_0 nodes. Let there be n_1 nodes in the tree T_1 obtained after we delete the indegree-zero clusters and replace all paths with a single indegree-one cluster. This means that all paths are of length at most 1. Consider a tree T_1', which is T_1 except all paths are replaces with an edge. Notice that $|T_1'| \geq n_1/2$, and T_1 has the same number of leaves as T_1'. Now, in T_1' there are no nodes with degree 2. And since any tree has at least as many leaves as nodes of degree 3 or more, T_1' has at least $|T_1'|/2$ leaves, which means that there are at least $n_1/4$ leaves in T_1.

Consider a leaf node v. Since v was not removed, it must have been heavy, and hence the subtree rooted at v has size $> n^{\delta/2}$. Hence, the number of nodes before we started our process clustered was $n_0 \geq (n_1/4) \cdot n^{\delta/2}$. Therefore, the number of nodes in each clustering step falls by a factor of $n^{\delta/2}$. □

Then slightly modify the process; let Π_2 be a process in which we still delete indegree-zero clusters instead of marking them colored, but we replace arbitrarily long paths with one node and two edges.

Lemma 6. *Each iteration of process* Π_2 *makes the tree smaller by a factor of* $\Omega(n^{\delta/2})$.

Proof. In essence, Π_2 behaves as if we first performed one iteration of Π_1 and then subdivided some edges. The subdivision only increases the number of nodes by a factor of two. □

Finally, let Π_3 be a process in which we still delete indegree-zero clusters instead of marking them colored, but we replace long paths with a sequence of clusters, each with at most $n^{\delta/2}$, similar to our real process. We can show:

Lemma 7. $O(1)$ *iterations of process* Π_3 *makes the tree smaller by a factor of* $\Omega(n^{\delta/2})$.

Proof. If we iterate Π_3 for more than $2/\delta$ iterations, each path gets contracted into a path with only one node. Hence, $2/\delta$ iterations of Π_3 makes at least as much progress as one iteration of Π_2. □

Lemma 4 now follows by observing that Π_3 describes accurately what happens in the uncolored subgraph in our real process:

Proof of Lemma 4. By applying Lemma 7 iteratively for $O(1)$ times to the uncolored subgraph, we can see that the uncolored part gets contracted into one node, and at that point the entire graph will fit in one indegree-zero cluster. □

4.4 Handling High-Degree Nodes

So far we have assumed that the tree that is given as input has degree at most $n^{\delta/2}$. The general solution to overcome this limitation is to replace high-degree nodes with $O(1)$-depth subtrees.

Let us now briefly describe how to implement it in $O(1)$ rounds in the MPC model. We can sort the original list of edges by the parent node identifier. Now whenever a single machine holds more than $n^{\delta/2}$ edges with the same parent u, it introduces new nodes whose parent is u and these new nodes become the new parent of $n^{\delta/2}$ children of u. We repeat this for $O(1)$ steps until all nodes have sufficiently low degrees. Throughout the process, we keep track of the *type* of the edge: whether it is an *original* edge or an *auxiliary* edge created while splitting high-degree nodes—this information is needed then later when we solve the DP problem (see Section 5.3).

This process will increase the number of nodes and the diameter by only a constant factor. Hence, if we now apply the clustering algorithm, the running time is still $O(\log D)$ rounds, where D is the diameter of the *original* tree.

5 SOLVING DP PROBLEMS

Now we will show how we can use the hierarchical clustering computed in Section 4 to solve dynamic programming problems (recall Definition 1).

5.1 From Bottom to Top

Let $L = O(1)$ be the number of layers in the hierarchical clustering. We fill in the dynamic programming tables in L iterations, by maintaining the following invariant:

Definition 8 (bottom-up invariant)**.** After iteration $i = 0, 1, \ldots, L$, each cluster C of layer i is labeled with its dynamic programming table $f(C)$, and all other nodes are labeled with their original inputs.

This invariant is trivial to satisfy in the beginning, as layer 0 is our input tree and there are no clusters yet.

Now assume that we satisfy the invariant before iteration $i > 0$. Now each node that still participates in the computation knows both its cluster identifier for layer i and either its input or its dynamic programming table. Furthermore, this information fits by assumption in $O(1)$ words. We can now sort the array of cluster identifiers and node labels and this way ensure that data related to one cluster is stored consecutively. Now one cluster spans at most two machines; with one additional routing step we can ensure that each cluster is fully contained inside one machine.

Now we can locally summarize each cluster C, by applying the sequential algorithm that we assumed exists. Finally, we have a summary $f(C)$ for each cluster. We can then apply sorting again to move the summary $f(C)$ back to the array location that we use to store information for cluster C. In essence, this enables us to solve the operation illustrated in Fig. 2 for each cluster in parallel.

Eventually, we have computed the dynamic programming tables for all clusters at all layers.

5.2 From Top to Bottom

Now we proceed to solve the problem, i.e., to fill in the labels of the edges. We proceed through the layers now in the reverse order, maintaining the following invariant:

Definition 9 (top-down invariant)**.** After iteration $i = L, L-1, \ldots, 0$, we have computed the labels of all edges (u, v) in the tree that corresponds to layer i, and this information is stored together with node u.

This invariant can be satisfied for $i = L$: there is only one edge in the tree, the outgoing edge of the topmost cluster C, and by assumption given $f(C)$ we can label this edge.

Now assume we satisfy the invariant before iteration $i < L$. Now if C is a cluster that appears in layer i, we can use sorting to ensure that the C is aware of both the label of its outgoing edge and the label of its incoming edge (if any). Then we again to reorganize data so that the nodes of layer $i - 1$ that form a cluster C at layer i are stored in the same computer. We can apply the sequential algorithm to now label all internal edges of C. In essence, this enables us to solve the operation illustrated in Fig. 3 for each cluster in parallel.

Eventually, we have computed the labels of all edges in layer 0, i.e., solved the original problem.

5.3 Handling High-Degree Nodes

In Section 4.4 we replaced high-degree nodes with $O(1)$-depth subtrees; we will have both *original* and *auxiliary* edges in the tree. In general, this will result in a new DP problem, with possibly different rules for different edges. For our running example, MaxIS, the rules can be specified as follows:

- Original edge (u, v): if we have $u \in X$, we must have $v \notin X$, and vice versa.
- Auxiliary edge (u, v): if we have $u \in X$, we must have $v \in X$, and vice versa.

In essence, this ensures that all new nodes that represent one original node make the same consistent choice. A similar strategy works for a wide range of graph problems.

6 APPLICATION: BAYESIAN TREE INFERENCE

Probabilistic or Bayesian graphical models are ubiquitous in machine learning and statistics [21]. A probabilistic graphical model is a graph, where the nodes (say, $x_i \in \mathbb{R}^{d_x}$) present hidden random variables with a conditional distribution structure defined by the vertices of the graph. We also get measurements of the graph (say, $y_i \in \mathbb{R}^{d_y}$) and an important problem of inference in graphical models is to compute the posterior distributions of the nodes, that is, $p(x_k \mid y_{1,\ldots,n})$ for some selected $k = 1, \ldots, n$.

We consider an important special case of a Bayesian graphical model, where the graph is a tree and the observations are conditionally independent observations obtained at each node from a given conditional distribution model $p(y_i \mid x_i)$. The conditional distributions of the nodes then take the form $p(x_i \mid x_{\gamma_i})$, where γ_i is the collection of child indices of the node x_i (a leaf j has $\gamma_j = \emptyset$). It now turns out that the algorithm framework presented in this paper allows us to compute $p(x_k \mid y_{1,\ldots,n})$ in $O(\log D)$ MPC rounds, at least in the Gaussian special case which we consider here. We assume that we have rooted the tree at x_k.

Let us denote the clique indices of the node i as $\alpha_i = \{i\} \cup \gamma_i$ and define clique potentials as

$$\psi_i(x_i, x_{\gamma_i}) = \psi_i(x_{\alpha_i}) = p(y_i \mid x_i)\, p(x_i \mid x_{\gamma_i}).$$

The computation of posterior probability density $p(x_k \mid y_{1,\ldots,n})$ then corresponds to computing the marginal of the product of the clique potentials:

$$p(x_k \mid y_{1,\ldots,n}) \propto \int \cdots \int \prod_{i=1}^{n} \psi_i(x_{\alpha_i})\, d(x_{1:n \setminus k}).$$

An efficient algorithm for solving this kind of problems on trees is called belief propagation [21]. In the case of path graphs (i.e., when each α_i is a pair of indices), the solution to the inference problem is given by Bayesian filters and smoothers [25], and belief propagation corresponds to so-called two-filter smoother. Parallel algorithms for the Bayesian filtering and smoothing problems (i.e., inference for probabilistic path graphs) have recently been developed in [18, 26], but not in the context of the MPC model. However, the associative formulations used in those algorithms provide practical means for path compression that we also need in Bayesian trees.

If we now think that the present tree is actually the subtree within the current cluster, then we have the following two possible cases to consider:

(1) *Indegree-zero cluster*, where we want to compute

$$\bar{\psi}_1(x_1) = \int \cdots \int \prod_{i=1}^{n} \psi_i(x_{\alpha_i})\, d(x_{2:n}),$$

where x_1 is the root. The potential $\bar{\psi}_1(x_1)$ then corresponds to compression of the indegree-zero cluster into a single node.

(2) *Indegree-one cluster*, where we want to compute

$$\bar{\psi}_{j\to 1}(x_1, x_j) = \int \cdots \int \prod_{i=1}^{n} \psi_i(x_{\alpha_i})\, d(x_{2:n\setminus j})$$

for some index $j \in \{2, \ldots, n\}$. Here $\bar{\psi}_{j\to 1}(x_1, x_j)$ corresponds to compression of the cluster into a node x_1 with an open child position x_j.

For concreteness, let us now take a look at a linear Gaussian graph in which we have (for $i = 1, \ldots, n$):

$$p(x_i \mid x_{\gamma_i}) = \mathcal{N}(x_i; \sum_{j\in\gamma_i} F_j\, x_j + c_i, Q_i),$$
$$p(y_i \mid x_i) = \mathcal{N}(y_i; H_i\, x_i + d_i, R_i),$$

that is,

$$\psi_i(x_{\alpha_i}) = \mathcal{N}(y_i; H_i\, x_i + d_i, R_i)\, \mathcal{N}(x_i; \sum_{j\in\gamma_i} F_j\, x_j + c_i, Q_i),$$

where $\mathcal{N}(x; \mu, \Sigma)$ denotes a multivariate Gaussian probability density with mean vector μ and covariance matrix Σ. The representation of a node thus consists of the $|\gamma_i|$ matrices $\{F_j : j \in \gamma_i\}$ along with c_i, Q_i, y_i, H_i, d_i, and R_i.

The implementation of the indegree-zero cluster operation (1) above requires just one primitive operation: the elimination of a leaf. We can repeat this operation until the whole tree is reduced into a single node. However, we need to ensure that we are able to do this operation in constant memory per node. Luckily, this is what happens in the Gaussian case.

Let us now consider a tree where we have an additional node x_{n+1} which is attached to the node j. What happens is that this adds a new child to node j:

$$\psi_j(x_{\alpha_j}) \to \tilde{\psi}_j(x_{\alpha_j}, x_{n+1}),$$

and we also need to multiply with the leaf potential $\psi_{n+1}(x_{n+1})$. Thus, the joint potential is $\psi(x_{1:n+1}) =$

$$\left[\prod_{i=1}^{j-1} \psi_i(x_{\alpha_i})\right] \tilde{\psi}_j(x_{\alpha_j}, x_{n+1})\, \psi_{n+1}(x_{n+1}) \left[\prod_{k=j+1}^{n} \psi_k(x_{\alpha_k})\right],$$

which we want to integrate over everything but x_1 in the present indegree-zero cluster case and over everything but x_1, x_j in the indegree-one cluster case below. The elimination of the leaf in both cases corresponds to integration over x_{n+1}.

The integration over x_{n+1} can be done in closed form in the Gaussian case. In practice, it consists of computing the posterior covariance and mean parameters of $\psi_{n+1}(x_{n+1})$ which are

$$\tilde{Q}_{n+1} = \left[Q_{n+1}^{-1} + H_{n+1}^\top R_{n+1}^{-1} H_{n+1}\right]^{-1},$$
$$\tilde{b}_{n+1} = \tilde{Q}_{n+1} \left[H_{n+1}^\top R_{n+1}^{-1} (y_{n+1} - d_{n+1}) + Q_{n+1}^{-1} c_{n+1},\right],$$

and then fusing them to the mean and covariance parameters of its parent node: $c_j \leftarrow F_{n+1}\, \tilde{b}_{n+1} + c_j$ and $Q_j \leftarrow Q_j + F_{n+1} \tilde{Q}_{n+1} F_{n+1}^\top$, which both are operations that can be done in constant memory.

For implementing the indegree-one cluster operation (2), we can first use the leaf elimination procedure above repeatedly to reduce the indegree-one cluster into a single indegree-one path. What we then have left is a path of the form (with re-indexed intermediate nodes)

$$\psi_1(x_1, x_2)\, \psi_2(x_2, x_3)\, \psi_3(x_3, x_4) \times \cdots \times \psi_{j-1}(x_{j-1}, x_j)$$

which we want to integrate over $x_{2:j-1}$. This can be implemented using pairwise combinations of the potentials, which can be done recursively as

$$\bar{\psi}_{m+1\to 1}(x_1, x_{m+1}) = \int \bar{\psi}_{m\to 1}(x_1, x_m)\, \psi_m(x_m, x_{m+1})\, dx_m$$

with initial condition $\bar{\psi}_{2\to 1}(x_1, x_2) = \psi_1(x_1, x_2)$. This is a special case of the Kalman filter's associative rule derived in [18, 26] (though backwards in time) and hence it can be implemented in constant additional memory for storing the temporary variables. The algorithm gives parameters (A, b, C, η, J) which define a factorization of the form:

$$\begin{aligned}\bar{\psi}_{j\to 1}(x_1, x_j) &\propto \mathcal{N}(x_1; A\, x_j + b, C)\, \mathcal{N}_I(x_j; \eta, J) \\ &= \mathcal{N}(x_1; A\, x_j + b, C)\, \mathcal{N}(x_j; J^{-1}\, \eta, J^{-1}).\end{aligned}$$

The term $\mathcal{N}(x_j; J^{-1}\, \eta, J^{-1})$ can now be fused to the measurement model at the node j by finding an artificial measurement z_j along with $\bar{H}_j$ and $\bar{R}_j$ such that $\mathcal{N}(x_j; J^{-1}\, \eta, J^{-1})\, N(y_j \mid H_j\, x_j + d_j, R_j) \propto N(z_j \mid \bar{H}_j\, x_j, \bar{R}_j)$. This can be done in constant memory by simple matrix and vector operations. In conclusion, the path compression just requires us to compute the parameters of the conditional distribution $\mathcal{N}(x_1; A\, x_j + b, C)$ and to form the artificial measurement model $N(z_j \mid \bar{H}_j\, x_j, \bar{R}_j)$ for the node x_j. This produces a new graph which we can continue to process recursively.

7 CONCLUSIONS

In this work, we showed how a broad class of *dynamic programming problems* can be solved in trees in the MPC model, with a relatively simple three-step approach: turn the input into a standard representation in $O(\log D)$ rounds, construct a hierarchical clustering in $O(\log D)$ rounds, and solve the problem of interest in $O(1)$ rounds. We expect that the hierarchical clustering will find applications also beyond the scope of dynamic programming problems.

One key open question is what happens once we step outside trees. The natural first step would be to consider bounded-treewidth graphs. Is it possible to find a similar hierarchical clustering efficiently also in bounded-treewidth graphs? And if so, does it still let us solve dynamic programming problems in constant time, given the hierarchical clustering?

ACKNOWLEDGMENTS

We are grateful to Alkida Balliu, Darya Melnyk, and Dennis Olivetti for several fruitful discussions, and to the anonymous reviewers for their helpful feedback on prior versions of this work. This work was supported in part by the Academy of Finland, Grants 321901 (Gupta and Vahidi) and 334238 (Latypov and Pai).

REFERENCES

[1] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. 2018. Parallel Graph Connectivity in Log Diameter Rounds. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, Mikkel Thorup (Ed.). IEEE Computer Society, 674–685. https://doi.org/10.1109/FOCS.2018.00070

[2] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. 2019. Massively Parallel Algorithms for Finding Well-Connected Components in Sparse Graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, Peter Robinson and Faith Ellen (Eds.). ACM, 461–470. https://doi.org/10.1145/3293611.3331596

[3] Alkida Balliu, Sebastian Brandt, Manuela Fischer, Rustam Latypov, Yannic Maus, Dennis Olivetti, and Jara Uitto. 2022. Exponential Speedup over Locality in MPC with Optimal Memory. In *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA (LIPIcs, Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:21. https://doi.org/10.4230/LIPIcs.DISC.2022.9

[4] Alkida Balliu, Rustam Latypov, Yannic Maus, Dennis Olivetti, and Jara Uitto. 2023. Optimal Deterministic Massively Parallel Connectivity on Forests. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, Nikhil Bansal and Viswanath Nagarajan (Eds.). SIAM, 2589–2631. https://doi.org/10.1137/1.9781611977554.ch99

[5] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. 2018. Brief Announcement: MapReduce Algorithms for Massive Trees. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic (LIPIcs, Vol. 107)*, Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 162:1–162:4. https://doi.org/10.4230/LIPIcs.ICALP.2018.162

[6] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. 2018. Massively Parallel Dynamic Programming on Trees. *CoRR* abs/1809.03685 (2018). arXiv:1809.03685 http://arxiv.org/abs/1809.03685

[7] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab S. Mirrokni. 2019. Near-Optimal Massively Parallel Graph Connectivity. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, David Zuckerman (Ed.). IEEE Computer Society, 1615–1636. https://doi.org/10.1109/FOCS.2019.00095

[8] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab S. Mirrokni, and Warren Schudy. 2021. Massively Parallel Computation via Remote Memory Access. *ACM Trans. Parallel Comput.* 8, 3 (2021), 13:1–13:25. https://doi.org/10.1145/3470631

[9] Hans L. Bodlaender. 1988. Dynamic Programming on Graphs with Bounded Treewidth. In *Automata, Languages and Programming, 15th International Colloquium, ICALP88, Tampere, Finland, July 11-15, 1988, Proceedings (Lecture Notes in Computer Science, Vol. 317)*, Timo Lepistö and Arto Salomaa (Eds.). Springer, 105–118. https://doi.org/10.1007/3-540-19488-6_110

[10] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau (Eds.). 2008. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. https://www.w3.org/TR/REC-xml/

[11] Sam Coy and Artur Czumaj. 2022. Deterministic massively parallel connectivity. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, Stefano Leonardi and Anupam Gupta (Eds.). ACM, 162–175. https://doi.org/10.1145/3519935.3520055

[12] Artur Czumaj, Peter Davies, and Merav Parter. 2021. Graph Sparsification for Derandomizing Massively Parallel Computation with Low Space. *ACM Trans. Algorithms* 17, 2 (2021), 16:1–16:27. https://doi.org/10.1145/3451992

[13] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. 2008. *Algorithms*. McGraw-Hill.

[14] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. 2019. Conditional Hardness Results for Massively Parallel Computation from Distributed Lower Bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, David Zuckerman (Ed.). IEEE Computer Society, 1650–1663. https://doi.org/10.1109/FOCS.2019.00097

[15] Jeremy Gibbons, Wentong Cai, and David B. Skillicorn. 1994. Efficient Parallel Algorithms for Tree Accumulations. *Sci. Comput. Program.* 23, 1 (1994), 1–18. https://doi.org/10.1016/0167-6423(94)00013-1

[16] Michael T. Goodrich. 1999. Communication-Efficient Parallel Sorting. *SIAM J. Comput.* 29, 2 (1999), 416–432. https://doi.org/10.1137/S0097539795294141

[17] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, Searching, and Simulation in the MapReduce Framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7074)*, Takao Asano, Shin-Ichi Nakano, Yoshio Okamoto, and Osamu Watanabe (Eds.). Springer, 374–383. https://doi.org/10.1007/978-3-642-25591-5_39

[18] Syeda Sakira Hassan, Simo Särkkä, and Ángel F. García-Fernández. 2021. Temporal Parallelization of Inference in Hidden Markov Models. *IEEE Trans. Signal Process.* 69 (2021), 4875–4887. https://doi.org/10.1109/TSP.2021.3103338

[19] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. 2017. Efficient massively parallel methods for dynamic programming. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, Hamed Hatami, Pierre McKenzie, and Valerie King (Eds.). ACM, 798–811. https://doi.org/10.1145/3055399.3055460

[20] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A Model of Computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, Moses Charikar (Ed.). SIAM, 938–948. https://doi.org/10.1137/1.9781611973075.76

[21] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press.

[22] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (1980), 831–838. https://doi.org/10.1145/322217.322232

[23] Moni Naor and Larry J. Stockmeyer. 1995. What Can be Computed Locally? *SIAM J. Comput.* 24, 6 (1995), 1259–1277. https://doi.org/10.1137/S0097539793254571

[24] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. 2018. Shuffles and Circuits (On Lower Bounds for Modern Parallel Computation). *J. ACM* 65, 6 (2018), 41:1–41:24. https://doi.org/10.1145/3232536

[25] Simo Särkkä. 2013. *Bayesian Filtering and Smoothing*. Cambridge University Press.

[26] Simo Särkkä and Ángel F. García-Fernández. 2021. Temporal Parallelization of Bayesian Smoothers. *IEEE Trans. Autom. Control.* 66, 1 (2021), 299–306. https://doi.org/10.1109/TAC.2020.2976316