Parallel Batch-Dynamic Graphs: Algorithms and Lower Bounds

Laxman Dhulipala* David Durfee† Janardhan Kulkarni‡ Richard Peng§
Saurabh Sawlani¶ Xiaorui Sun‖

Abstract

In this paper we study the problem of dynamically maintaining graph properties under batches of edge insertions and deletions in the massively parallel model of computation. In this setting, the graph is stored on a number of machines, each having space strongly sublinear with respect to the number of vertices, that is, n^ϵ for some constant $0 < \epsilon < 1$. Our goal is to handle batches of updates and queries where the data for each batch fits onto one machine in constant rounds of parallel computation, as well as to reduce the total communication between the machines. This objective corresponds to the gradual buildup of databases over time, while the goal of obtaining constant rounds of communication for problems in the static setting has been elusive for problems as simple as undirected graph connectivity.

We give an algorithm for dynamic graph connectivity in this setting with constant communication rounds and communication cost almost linear in terms of the batch size. Our techniques combine a new graph contraction technique, an independent random sample extractor from correlated samples, as well as distributed data structures supporting parallel updates and queries in batches.

We also illustrate the power of dynamic algorithms in the MPC model by showing that the batched version of the adaptive connectivity problem is P-complete in the centralized setting, but sub-linear sized batches can be handled in a constant number of rounds. Due to the wide applicability of our approaches, we believe it represents a practically-motivated workaround to the current difficulties in designing more efficient massively parallel static graph algorithms.

1 Introduction

Parallel computation frameworks and storage systems, such as MapReduce, Hadoop and Spark, have been proven to be highly effective methods for representing and analyzing the massive datasets that appear in the world today. Due to the importance of this new class of systems, models of parallel computation capturing the power of such systems have been increasingly studied in recent years, with the Massively Parallel Computation (MPC) model [47] now serving as the canonical model. In recent years the MPC model has seen the development of algorithms for fundamental problems, including clustering [28, 16, 19, 69], connectivity problems [60, 48, 7, 14, 8], optimization [56, 29, 17], dynamic programming [43, 18], to name several as well as many other fundamental graph and optimization problems [15, 6, 51, 2, 13, 9, 7, 24, 26, 53, 59, 10, 11, 12, 21, 22, 33, 34, 35, 38]. Perhaps the main goal in these algorithms has been solving the problems in a constant number of communication rounds while minimizing the total communication in a round. Obtaining low round-complexity is well motivated due to the high cost of a communication round in practice, which is often between minutes and hours [47]. Furthermore, since communication between processors tends to be much more costly than local computation, ensuring low communication per-round is also an important criteria for evaluating algorithms in the MPC model [62, 20].

Perhaps surprisingly, many natural problems such as dynamic programming [43] and submodular maximization [17] can in fact be solved or approximated in a constant number of communication rounds in MPC model. However, despite considerable effort, we are still far from obtaining constant-round algorithms for many natural problems in the MPC setting where the space-per-machine is restricted to be sublinear in the number of vertices in the graph (this setting is arguably the most reasonable modeling choice, since real-world graphs can have trillions of vertices). For example, no constant round algorithms are known for a problem as simple as connectivity in an undirected graph, where the current best bound is $O(\log n)$ rounds in general [47, 60, 48, 7, 53, 14]. Other examples include a $O(\sqrt{\log n})$ round algorithm for approximate graph matching [59, 35], and a $O(\sqrt{\log \log n})$-round algorithm for $(\Delta + 1)$ vertex coloring [25]. Even distinguishing between a single cycle of size n and two cycles of size $n/2$ has been conjectured to require $\Omega(\log n)$ rounds [47, 60, 48, 61, 69, 34, 42]. Based on this conjecture, recent studies have shown that several other graph related problems, such as maximum matching, vertex cover, maximum independent set

*Carnegie Mellon University ldhulipa@cs.cmu.edu
†LinkedIn ddurfee@linkedin.com
‡Microsoft Research jakul@microsoft.com
§Georgia Tech richard.peng@gmail.com
¶Georgia Tech sawlani@gatech.edu
‖University of Illinois at Chicago xiaorui@uic.edu

and single-linkage clustering cannot be solved in a constant number of rounds [69, 34].

On the other hand, most large-scale databases are not formed by a single atomic snapshot, but form rather gradually through an accretion of updates. Real world examples of this include the construction of social networks [54], the accumulation of log files [40], or even the gradual change of the Internet itself [27, 45, 55]. In each of these examples, the database is gradually formed over a period of months, if not years, of updates, each of which is significantly smaller than the whole database. It is often the case that the updates are grouped together, and are periodically processed by the database as a batch. Furthermore, it is not uncommon to periodically re-index the data structure to handle a large number of queries between sets of updates.

In this paper, motivated by the gradual change in real-world datasets through batches of updates, we consider the problem of maintaining graph properties in dynamically changing graphs in the MPC model. Our objective is to maintain the graph property for *batches* of updates, while achieving a *constant number of rounds* of computation in addition to also minimizing the total *communication* between machines in a given round.

Specifically, we initiate the study of *parallel batch-dynamic graph problems* in MPC, in which an update contains a number of mixed edge insertions and deletions. We believe that batch-dynamic algorithms in MPC capture the aforementioned real world examples of gradually changing databases, and provide an efficient distributed solution when the size of the update is large compared to single update dynamic algorithms. We note that a similar model for dynamic graph problems in MPC was recently studied by Italiano et al. [44]. However, they focus on the scenario where every update only contains a single edge insertion or deletion. Parallel batch-dynamic algorithms were also recently studied in the shared-memory setting by Tseng et al. [67] for the forest-connectivity problem and Acar et al. [1] for dynamic graph connectivity. However, the depth of these algorithms is at least $\Omega(\log n)$, and it is not immediately clear whether these results can be extended to low (constant) round-complexity batch-dynamic algorithms in the MPC setting.

We also study the power of dynamic algorithms in the MPC setting by considering a natural "semi-online" version of the connectivity problem which we call *adaptive connectivity*. We show that the adaptive connectivity problem is P-complete, and therefore in some sense inherently sequential, at least in the centralized setting. In contrast to this lower bound in the centralized setting, we show that in the MPC model there is a batch-dynamic algorithm that can process adaptive batches with size proportional to the space per-machine in a constant number of rounds. Note that such an algorithm in the centralized setting (even one that ran in slightly sublinear depth per batch) would imply an algorithm for the Circuit Value Problem with polynomial speedup, thus solving a longstanding open problem in the parallel complexity landscape.

1.1 Our Results

Since graph connectivity proves to be an effective representative for the aforementioned difficulty of graph problems in the MPC model, the focus of this paper is studying graph connectivity and adaptive graph connectivity in the batch-dynamic MPC model.

Graph Connectivity The dynamic connectivity problem is to determine if a given pair of vertices belongs to same connected component in the graph as the graph undergoes (batches of) edge insertions and deletions. The dynamic connectivity algorithm developed in this paper is based on a hierarchical partitioning scheme that requires a more intricate incorporation of sketching based data structures for the sequential setting. Not only does our scheme allow us to achieve a constant number of rounds, but it also allows us to achieve a total communication bound that is linear with respect to the batch size with only an additional $n^{o(1)}$ factor.

THEOREM 1.1. *In the* MPC *model with memory per machine* $s = \widetilde{O}(n^{\epsilon})$ *we can maintain a dynamic undirected graph on* m *edges which, for constants* δ, α*, and integer* k *such that* $k \cdot n^{\alpha+\delta} \cdot \text{polylog}(n) \leq s$*, can handle the following operations with high probability:*

1. *A batch of up to* k *edge insertions/deletions, using* $O(1/(\delta\alpha))$ *rounds.*
2. *Query up to* k *pairs of vertices for 1-edge-connectivity, using* $O(1/\alpha)$ *rounds.*

Furthermore, the total communication for handling a batch of k *operations is* $\widetilde{O}(kn^{\alpha+\delta})$*, and the total space used across all machines is* $\widetilde{O}(m)$*.*

Adaptive Connectivity and Lower-Bounds in the Batch-Dynamic MPC Model In the *adaptive connectivity problem*, we are given a sequence of query/update pairs. The problem is to process each query/update pair in order, where each query determines whether or not a given pair of vertices belongs to the same connected component of the graph, and applies the corresponding dynamic update to the graph if the query succeeds. We obtain the following corollary by applying our batch-dynamic

connectivity algorithm, Theorem 1.1.

COROLLARY 1.1. *In the* MPC *model with memory per machine* $s = \widetilde{O}(n^\epsilon)$ *we can maintain a dynamic undirected graph on* m *edges which for constants* δ, α, *and integer* k *such that* $k \cdot n^{\alpha+\delta} \cdot \text{polylog}(n) \leq s$ *can handle the following operation with high probability:*

1. *An adaptive batch of up to* k *(query, edge insertions/deletions) pairs, using* $O(1/(\delta\alpha))$ *rounds.*

Furthermore, the total communication for handling a batch of k *operations is* $\widetilde{O}(kn^{\alpha+\delta})$, *and the total space used across all machines is* $\widetilde{O}(m)$.

We also provide a lower-bound for the adaptive connectivity problem in the centralized setting, showing that the problem is P-complete under NC^1 reduction. P-completeness is a standard notion of parallel hardness [50, 37, 23]. As a consequence of our reduction, we show that the adaptive connectivity algorithm does not admit a parallel algorithm in the centralized setting with polynomial speedup, unless the (Topologically-Ordered) Circuit Value Problem admits a parallel algorithm with polynomial speedup, which is a long-standing open problem in parallel complexity literature.

THEOREM 1.2. *The adaptive connectivity problem is* P*-complete under* NC^1 *reductions.*

By observing that our reduction, and the NC^1 reductions proving the hardness for the Circuit Value Problem can be done in $O(1)$ rounds of MPC, we have the following corollary in the MPC setting.

COROLLARY 1.2. *In the* MPC *model with memory per machine* $s = \widetilde{O}(n^\epsilon)$ *for some constant* ϵ, *if adaptive connectivity on a sequence of size* $O(n)$ *can be solved in* $O(k)$ *rounds, then every problem in* P *can be solved in* $O(k)$ *rounds.*

1.2 Batch-Dynamic MPC Model In this section, we first introduce the massively parallel computation (MPC) model, followed by the batch-dynamic MPC model which is the main focus of this paper.

Massively Parallel Computation (MPC) Model. The Massively Parallel Computation (MPC) model is a widely accepted theoretical model for parallel computation [47]. Here, the input graph G has n vertices and at most m edges at any given instant. We are given p processors/machines, each with local memory for storage $s = \widetilde{\Theta}(m/p)$.[1] Note that we usually assume $m^{1-\delta} \geq p \geq m^\delta$, for some $\delta > 0$. This is because the model is relevant only when the number of machines and the local memory per machine are significantly smaller than the size of the input.

The computation in the MPC model proceeds via rounds. Initially, the input data is distributed across the processors arbitrarily. During each round, each processor runs a polynomial-time algorithm on the data which it contains locally. Between rounds, each machine receives at most μ amount of data from other machines. The total data received by all machines between any two rounds is termed as the communication cost. Note that no computation can occur between rounds, and equivalently, no communication can occur during a round.

The aim for our algorithms in this model is twofold. Firstly and most importantly, we want to minimize the number of rounds required for our algorithm, since this cost is the major bottleneck of massively parallel algorithms in practice. Ideally, we would want this number to be as low as $O(1)$. Secondly, we want to decrease the maximum communication cost over all rounds, since the costs of communication between processors in practice are massive in comparison to local computation.

Batch-Dynamic MPC Model. At a high-level, our model works as follows. Similar to recent works by Acar et al. [1] and Tseng et al. [67], we assume that the graph undergoes batches of insertions and deletions, and in the initial round of each computation, an update or query batch is distributed to an arbitrary machine. The underlying computational model used is the MPC model, and assume that space per machine is strongly sublinear with respect to the number of vertices of the graph, that is, $O(n^\alpha)$ for some constant $0 < \alpha < 1$.

More formally, we assume there are two kinds of operations in a batch:

1. Update: A set of edge insertions/deletions of size up to k.
2. Query: A set of graph property queries of size up to k.

For every batch of updates, the algorithm needs to properly maintain the graph according to the edge insertions/deletions such that the algorithm can accurately answer a batch of queries at any instant. We believe that considering batches of updates and queries most closely relates to practice where often multiple updates occur in the examined network before another query is made. Furthermore, in the MPC model there is a distinction between a batch of updates and a single update, unlike the standard model, because it is possible for the batch

[1] Throughout this paper, $\widetilde{\Theta}$ and $\widetilde{O}$ hide polylogarithmic terms in the size of the input.

update to be made in parallel, and handling batch updates or queries is as efficient as handling a single update or query, especially in terms of the number of communication rounds.

We use two criteria to measure the efficiency of parallel dynamic algorithms: the number of communication rounds and the total communication between different machines. Note that massively parallel algorithms for static problems are often most concerned with communication rounds. In contrast, we also optimize the total communication in the dynamic setting, since the total communication becomes a bottleneck for practice when overall data size is very huge, especially when the update is much smaller than the total information of the graph. Ideally, we want to handle batches of updates and queries in constant communication rounds and sublinear total communication with respect to the number of vertices in the graph.

The key algorithmic difference between the dynamic model we introduce here and the MPC model is that we can decide how to partition the input into processors as updates occur to the graph.

Dynamic problems in the MPC model were studied in the very recent paper by Italiano et al. [44]. Their result only explicitly considers the single update case. In the batch-dynamic scenario, the result of [44] generalizes but has higher dependencies on batch sizes in both number of rounds and total communication.Our incorporation of graph sketching, fast contraction, and batch search trees are all critical for obtaining our optimized dependencies on batch sizes.

1.3 Our Techniques In this section we give in-depth discussion of the primary techniques used to achieve the results presented in the previous section.

Connectivity. Without loss of generality, we assume that the batch of updates is either only edge insertions or only edge deletions. For a mixed update batch with both insertions and deletions, we can simply handle the edge deletions first, and then the edge insertions. In case the same edge is being inserted and deleted, we simply eliminate both operations.

Similar to previous results on dynamic connectivity [31, 32, 39, 41, 3, 46, 36, 57, 68, 58], we maintain a maximal spanning forest. This forest encodes the connectivity information in the graph, and more importantly, undergoes few changes per update to the graph. Specifically:

1. An *insert* can cause at most two trees in F to be joined to form a single tree.
2. A *delete* may split a tree into two, but if there exists another edge between these two resulting trees, they should then be connected together to ensure that the forest is maximal.

Our dynamic trees data structure adapts the recently developed parallel batch-dynamic data structure for maintaining a maximal spanning forest in the shared-memory setting by Tseng et al. [67] to the MPC model. Specifically, [67] give a parallel batch-dynamic algorithm that runs in $O(\log n)$ depth w.h.p. to insert k new edges to the spanning forest, to remove k existing edges in the spanning forest, or to query the IDs of the spanning tree containing the given k vertices. We show that the data structure can be modified to achieve $O(1/\alpha)$ round-complexity and $O(k \cdot n^{\alpha})$ communication for any small constant α satisfying $k \cdot n^{\alpha} \cdot \text{polylog}(n) \leq s$ in the MPC setting. In addition, if we associate with each vertex a key of length ℓ_{key}, then we can query and update a batch of k key values in $O(1/\alpha)$ round-complexity and $O(k \cdot \ell_{key} \cdot)$ communication.

With a parallel batch-dynamic data structure to maintain a maximal spanning forest, a batch of edge insertions or edge queries for the dynamic connectivity problem can be handled in $O(1/\alpha)$ round-complexity and $O(k \cdot n^{\alpha})$ communication for any constant α. Our strategy for insertions and queries is similar to the dynamic connectivity algorithm of Italiano et al. [44]: A set of edge queries can be handled by querying the IDs of the spanning tree of all the vertices involved. Two vertices are in the same connected component if and only if their IDs are equal. To process a batch of edge insertions, we maintain the maximal spanning forest by first identifying the set of edges in the given batch that join different spanning trees without creating cycles using ID queries, and then inserting these edges to the spanning forest, by linking their respective trees. Handling a set of edge deletions, however, is more complex. This is because if some spanning forest edges are removed, then we need to find replacement edges which are in the graph, but previously not in the spanning forest, that can be added to the spanning forest without creating cycles. To facilitate this, we incorporate developments in sketching based sequential data structures for dynamic connectivity [3, 46].

To construct a sketch of parameter $0 < p < 1$ for a graph, we first independently sample every edge of the graph with probability p, and then set the sketch for each vertex to be the XOR of the IDs for all the sampled edges which are incident to the vertex. A sketch has the property that for any subset of vertices, the XOR of the sketches of these vertices equals to

the XOR of the IDs for all the sampled edges leaving the vertex subset. In particular, if there is only a single sampled edge leaving the vertex subset, then the XOR of the sketches of these vertices equals to the ID of the edge leaving the vertex subset.

The high level idea of [3, 46] is to use sketches for each current connected component to sample previous non-tree edges going out of the connected component using sketches with different parameters, and use these edges to merge connected components that are separated after deleting some tree edges. We visualize this process by representing each connected component as a vertex in a multigraph, and finding a replacement non-tree edge between the two components as the process of merging these two vertices. At first glance, it seems like we can translate this approach to the MPC model by storing all the sketches for each connected component in a single machine. However, directly translating such a data structure leads to either polylog(n) communication rounds or $\Omega(m)$ total communication per update batch. To see this, let us look at some intuitive ideas to adapt this data structure to the MPC model, and provide some insight into why they have certain limitations:

1. **Sketch on the original graph**: For this case, once we use the sketch to sample an edge going out of a given connected component, we only know the ID of the two vertices of the edge, but not the two connected components the edge connects. Obtaining the information about which connected components the endpoints belong to requires communication, because a single machine cannot store the connected component ID of each vertex in the graph. Hence, to contract all the connected components using sampled edges for each connected component, we need one round of communication. Since we may need to reconnect as many as k connected components (k is the number of deletions, i.e., the batch size), this approach could possibly require $\log k = \Theta(\log n)$ communication rounds.
2. **Sketch on the contracted graph where every connected component is contracted to a single vertex**: To do this, each edge needs to know which connected components its endpoints belong to. If we split a connected component into several new connected components after deleting some tree edges, the edges whose vertices previously belong to same connected component may now belong to different connected components. To let each edge know which connected components its endpoints belong to, we need to broadcast the mapping between vertices and connected components to all the related edges. Hence, the total communication can be as large as $\Omega(m)$. To further illustrate this difficulty via an example, consider the scenario that the current maximal spanning forest is a path of n vertices, and a batch of k edge deletions break the path into $k+1$ short paths. In this case, almost all the vertices change their connected component IDs. In order to find edges previously not in the maximal spanning forest to link these $k+1$ path, every edge needs to know if the two vertices of the edge belong to same connected component or not, and to do this, the update of connected component ID for vertices of every edge requires $\Omega(m)$ communication.

The high level idea of our solution is to speed up the "contraction" process such that constant iterations suffice to shrink all the connected components into a single vertex. To do this, sampling $\widetilde{O}(1)$ edges leaving each connected component in each iterations (as previous work) is not enough, because of the existence of low conductance graph. Hence, we need to sample a much larger number of edges leaving each connected component. Following this intuition, we prove a **fast contraction lemma** which shows that picking n^α edges out of each component finds all connecting non-tree edges between components within $O(1/\alpha)$ iterations.

However, a complication that arises with the aforementioned fast contraction lemma is that it requires the edges leaving a component to be *independently* sampled. But the edges sampled by a single sketch are correlated. This correlation comes from the fact that a sketch outputs an edge leaving a connected component if and only if there is only one sampled edge leaving that connected component. To address this issue, we construct an **independent sample extractor** to identify enough edges that are eventually sampled independently based on the sketches and show that these edges are enough to simulate the independent sampling process required by the fast contraction lemma.

We discuss these two ideas in depth below.

Fast Contraction Lemma. We first define a random process for edge sampling (which we term ContractionSampling) in Definition 1.1. The underlying motivation for such a definition is that the edges obtained from the sketch are not independently sampled. So, we tweak the sampling process via an independent sample extractor, which can then produce edges which obey the random process ContractionSampling . Before discussing this independent sample extractor, we will first outline how edges sampled using ContractionSampling suffice

for fast contraction.

DEFINITION 1.1. (CONTRACTIONSAMPLING PROCESS) *The random process* ContractionSampling *for a multigraph $G = (V, E)$ and an integer k is defined as follows: each vertex v independently draws t_v samples $S_{v,1}, S_{v,2}, \ldots S_{v,t_v}$ for some integer $t_v \geq k$ such that*

1. *the outcome of each $S_{v,i}$ can be an either an edge incident to v or $\perp$;*
2. *for every edge e incident to vertex v,*

$$\sum_{i=1}^{t_v} \Pr[S_{v,i} = e] \geq \Omega\left(\frac{k \log^2 n}{\boldsymbol{d}_G(v)}\right).$$

We show that in each connected component, if we contract edges sampled by the ContractionSampling process, the number of edges remaining reduces by a polynomial factor with high probability by taking $k = \text{poly}(n)$.

LEMMA 1.1. *Consider the following contraction scheme starting with a multigraph $G(V, E)$ on n vertices and $m < poly(n)$ (multi) edges: For a fixed integer k,*

1. *let E' be a set of edges sampled by the* ContractionSampling *process;*
2. *contract vertices belonging to same connected component of graph $G' = (V, E')$ into a new graph $G^\star = (V^\star, E^\star)$ as follows: each vertex of $V^\star$ represents a connected component in the sampled graph $G' = (V, E')$, and there is an edge between two vertices $x, y \in V^\star$ iff there is an edge in G between the components corresponding to x and y, with edge multiplicity equal to the sum of multiplicity of edges in G between the components corresponding to x and y.*

Then the resultant graph has at most $\widetilde{O}(mk^{-1/3})$ (multi) edges with high probability.

Based on Lemma 1.1, if we iteratively apply the ContractionSampling process with $k = n^\alpha$ and shrink connected components using sampled edges into a single vertex, then every connected component of the multigraph becomes a singleton vertex in $O(1/\alpha)$ rounds with high probability.

Lemma 1.1 can be shown using a straightforward argument for simple graphs. However, in the case of multigraphs (our graphs are multigraphs because there can be more than one edge between two components), this argument is not as easy. It is possible that for a connected component C_1, a large number of edges leaving C_1 will go to another connected component C_2. Hence, in one round, the sampled n^δ edges leaving C_1 may all go to C_2. From this perspective, we cannot use a simple degree-based counting argument to show that every connected component merges with at least n^δ other connected components if it connected to at least n^δ other connected components.

To deal with parallel edges, and to prove that the contraction occurs in constant, rather than $O(\log n)$ rounds, we make use of a more combinatorial analysis. Before giving some intuition about this proof, we define some useful terminology.

DEFINITION 1.2. (CONDUCTANCE) *Given a graph $G(V, E)$ and a subset of vertices $S \subseteq V$, the conductance of S w.r.t. G is defined as*

$$\phi_G(S) \stackrel{\text{def}}{=} \min_{S' \subseteq S} \frac{|E(S', S \setminus S')|}{\min\left\{\sum_{u \in S'} \boldsymbol{d}_G(u), \sum_{u \in S \setminus S'} \boldsymbol{d}_G(u)\right\}}.$$

The conductance of a graph is a measure of how "well-knit" a graph is. Such graphs are of consequence to us because the more well-knit the graph is, the faster it contracts into a singleton vertex. We use the expander decomposition lemma from [65], which says that any connected multigraph G can be partitioned into such subgraphs.

LEMMA 1.2. ([65], SECTION 7.1.) *Given a parameter $k > 0$, any graph G with n vertices and m edges can be partitioned into groups of vertices $S_1, S_2, \ldots$ such that*

- *the conductance of each S_i is at least $1/k$;*
- *the number of edges between the S_i's is at most $O(m \log n/k)$.*

For each such "well-knit" subgraph H to collapse in one round of sampling, the sampled edges in H must form a spanning subgraph of H. One way to achieve this is to generate a spectral sparsifier of H [64] - which can be obtained by sampling each edge with a probability at least $O(\log n)$ times its *effective resistance*. The effective resistance of an edge is the amount of current that would pass through it when unit voltage difference is applied across its end points, which is a measure of how important it is to the subgraph being well-knit.

As the last piece of the puzzle, we show that the edges sampled by the ContractionSampling process do satisfy the required sampling constraint to produce a spectral sparsifier of H. Since each such subgraph collapses, Lemma 1.2 also tells us that only a small fraction of edges are leftover in G, as claimed in Lemma 1.1.

It is important to note that although we introduce sophisticated tools such as expander partitioning

and spectral sparsifiers, these tools are only used in the proof and not in the actual algorithm to find replacement edges.

Independent Sample Extractor From Sketches. On a high level, our idea to achieve fast contraction is to use $O(k \cdot \text{polylog}(n))$ independent sketches to simulate the ContractionSampling process, and then apply Lemma 1.1. However, we cannot do this naively, because ContractionSampling requires the edges being sampled independently, whereas the sketch from Lemma 2.1 does not satisfy this property. Recall that the process of outputting edges by a sketch (with respect to a fixed partition of the graph) can be viewed as following two steps: (i) sample each edge independently with probability given by the parameter of the sketch; (ii) output all sampled edges leaving a subset of the partition if and only if there is only one sampled edge leaving that subset. This second step creates a correlation within the edge sampling process.

We would like to remark that this is not an issue for previous sketching based connectivity algorithms (e.g. [3, 46]), because in [3, 46], each time, any current connected component only needs to find an arbitrary edge leaving the connected component. In this way, if most current connected components find an arbitrary edge leaving the component, then after contracting connected components using sampled edges, the total number of connected components reduce by at least a constant factor. In this way, after $O(\log n)$ iterations, each connected component shrinks into a single vertex. But in our case the contraction lemma requires edges being sampled independently. Hence, we cannot directly apply Lemma 1.1 on sketches.

To get around this issue, we construct an independent edge sample extractor from the sketches and show that with high probability, this extractor will extract a set of independent edge samples that are equivalent to being sampled from a ContractionSampling random process, as required by Lemma 1.1. One key observation is that if the graph is bipartite, then sketch values on the vertices from one side of the bipartite graph are independent, because every edge sample is only related to one sketch value. The high level idea of our extractor is then to extract bipartite graphs from sketches, such that each edge appears in many bipartite graphs with high probability. For each sketch, consider the following random process:

1. For each vertex of the graph, randomly assign a color of red or yellow. Then we can construct a bipartite graph with red vertices on one side, yellow vertices on the other side, and an edge is in the bipartite graph if and only if the color of one endpoint is red, and the other endpoint is yellow. Note that this step is not related to the process of sketch construction.
2. Independently sample every edge not in the bipartite graph with probability same as the probability of sampling used in the sketch.
3. For each red vertex whose incident edges were not sampled in Step 2, independently sample every edge incident to the vertex in the bipartite graph with probability same as that used in the sketch.
4. Choose all edges sampled in Step 3 which do not share a red vertex with any other sampled edge.

We show that the edges obtained in Step 4 are sampled independently (conditioned on the outcome of Step 2). Another way to see this independence is to partition all the independent random variables in the process of generating all the sketches into two random processes R_1 and R_2 (based on the bipartite graph generated for each sketch) such that R_1 and R_2 are independent and simulate a required ContractionSampling process in the following sense:

1. After implementing the random process R_1 and based on the outcome of R_1, define a ContractionSampling process as required by Lemma 1.1.
2. The random process R_2 simulates the defined ContractionSampling process in the following sense: there is a partition of the independent random variables of random process R_2 into groups satisfying the following conditions:
   (a) There is a bijection between groups and random variables of the ContractionSampling process.
   (b) For each group, there exists a function of the random variables in the group such that the function is equivalent to the corresponding random variable of the ContractionSampling process.

Furthermore, all the edges sampled by the defined ContractionSampling process are generated by the sketches (meaning that there exist a vertex and a sketch such that sketch on the vertex is the ID of the sampled edge). In this way, we argue that the edges generated by all the sketches contains a set of edges generated by a ContractionSampling process so that we can apply Lemma 1.1.

LEMMA 1.3. *Given an integer k and a multigraph G of n vertices, $O(k \log^3 n)$ independent sketches simulates a ContractionSampling process. Furthermore, for every edge sampled by ContractionSampling , there exists a sketch and a vertex such that the value of the sketch on the vertex*

is exactly the ID of that edge.

Adaptive Connectivity and Lower-Bounds in the Batch-Dynamic MPC Model. The adaptive connectivity problem is the "semi-online" version of the connectivity problem where the entire adaptive batch of operations is given to the algorithm in advance, but the algorithm must apply the query/update pairs in the batch *in order*, that is each pair on the graph defined by applying the prefix of updates before it. We note that the problem is closely related to *offline* dynamic problems, for example for offline dynamic minimum spanning tree and connectivity [30]. The main difference is that in the offline problem the updates (edge insertions/deletions) are not adaptive, and are therefore not conditionally run based on the queries. We also note here that every problem that admits a static NC algorithm also admits an NC algorithm for the offline variant of the problem. The idea is to run, in parallel for each query, the static algorithm on the input graph unioned with the prefix of the updates occuring before the query. Assuming the static algorithm is in NC, this gives a NC offline algorithm (note that obtaining work-efficient parallel offline algorithms for problems like minimum spanning tree and connectivity is an interesting problem that we are not aware of any results for).

Compared to this positive result in the setting without adaptivity, the situation is very different once the updates are allowed to adaptively depend on the results of the previous query, since the simple black-box reduction given for the offline setting above is no longer possible. In particular, we show the following lower bound for the adaptive connectivity problem which holds in the centralized setting: the adaptive connectivity problem is P-complete, that is unless P = NC, there is no NC algorithm for the problem. The adaptive connectivity problem is clearly in P since we can just run a sequential dynamic connectivity algorithm to solve it. To prove the hardness result, we give a low-depth reduction from the Circuit Value Problem (CVP), one of the canonical P-complete problems. The idea is to take the gates in the circuit in some topological-order (note that the version of CVP where the gates are topologically ordered is also P-complete), and transform the evaluation of the circuit into the execution of an adaptive sequence of connectivity queries. We give an NC^1 reduction which evaluates a circuit using adaptive connectivity queries as follows. The reduction maintains that all gates that evaluate to true are contained in a single connected component connected to some root vertex, r. Then, to determine whether the next gate in the topological order, $g = g_a \wedge g_b$, evaluates to true the reduction runs a connectivity query testing whether the vertices corresponding to g_a and g_b are connected in the current graph, and adds an edge (g, r), thereby including it in the connected component of true gates if the query is true. Similarly, we reduce evaluating $g = g_a \vee g_b$ gates to two queries, which check whether g_a (g_b) is reachable and add an edge from (g, r) in either case if so. A $g = \neg g_a$ gate is handled almost similarly, except that the query checks whether g_a is disconnected from s. Given the topological ordering of the circuit, generating the sequence of adaptive queries can be done in $O(\log n)$ depth and therefore the reduction works in NC^1.

In contrast, in the MPC setting, we show that we can achieve $O(1)$ rounds for adaptive batches with size proportional to the space per machine. Our algorithm for adaptive connectivity follows naturally from our batch-dynamic connectivity algorithm based on the following idea: we assume that every edge deletion in the batch actually occurs, and compute a set of replacement edges in G for the (speculatively) deleted edges. Computing the replacement edges can be done in the same round-complexity and communication cost as a static batch of deletions using Theorem 1.1. Since the number of replacement edges is at most $O(k) = O(s)$, all of the replacements can be sent to a single machine, which then simulates the sequential adaptive algorithm on the graph induced by vertices affected by the batch in a single round. We note that the upper-bound in MPC does not contradict the P-completeness result, although achieving a similar result for the depth of adaptive connectivity in the centralized setting for batches of size $O(s) = O(n^\epsilon)$ would be extremely surprising since it would imply a polynomial-time algorithm for the (Topologically Ordered) Circuit Value Problem with sub-linear depth and therefore polynomial speedup.

1.4 Organization Section 2 describes the full version of the high level idea for graph connectivity. Section 3 contains a discussion of the data structure we used to handle batch-update in constant round. Section 4 gives a proof of our fast contraction lemma. A proof of our independent sample extractor from sketches is in the full version of the paper[2]. Section 5 presents the algorithm for graph connectivity and the correctness proof. Lastly, our lower and upper bounds for the adaptive connectivity problem can be found in the full version of the paper[2].

[2] https://arxiv.org/abs/1908.01956

2 1-Edge-Connectivity

In this section we prove our result for 1-edge-connectivity, restated here:

THEOREM 2.1. *In the* MPC *model with memory per machine* $s = \widetilde{O}(n^\epsilon)$ *we can maintain a dynamic undirected graph on* m *edges which, for constants* δ, α, *and integer* k *such that* $k \cdot n^{\alpha+\delta} \cdot \text{polylog}(n) \leq s$, *can handle the following operations with high probability:*

1. *A batch of up to* k *edge insertions/deletions, using* $O(1/(\delta\alpha))$ *rounds.*
2. *Query up to* k *pairs of vertices for 1-edge-connectivity, using* $O(1/\alpha)$ *rounds.*

Furthermore, the total communication for handling a batch of k *operations is* $\widetilde{O}(kn^{\alpha+\delta})$, *and the total space used across all machines is* $\widetilde{O}(m)$.

Parallel Batch-Dynamic Data Structure. Similar to previous results on dynamic connectivity [31, 32, 39, 41, 3, 46, 36, 57, 68, 58], our data structure is based on maintaining a maximal spanning forest, which we denote using F. Formally, we define it as follows.

DEFINITION 2.1. (MAXIMAL SPANNING FOREST) *Given a graph* G, *we call* F *a maximal spanning forest of* G *if* F *is a subgraph of* G *consisting of a spanning tree in every connected component of* G.

Note that this is more specific than a spanning forest, which is simply a spanning subgraph of G containing no cycles. This forest encodes the connectivity information in the graph, and more importantly, undergoes few changes per update to the graph. Specifically:

1. An *insert* can cause at most two trees in F to be joined to form a single tree.
2. A *delete* may split a tree into two, but if there exists another edge between these two resulting trees, they should then be connected together to ensure that the forest is maximal.

Note that aside from identifying an edge between two trees formed when deleting an edge from some tree, all other operations are tree operations. Specifically, in the static case, these operations can be entirely encapsulated via tree data structures such as dynamic trees [63] or Top-Trees [5]. We start by ensuring that such building blocks also exist in the MPC setting. In Section 3, we show that a forest can also be maintained efficiently in $O(1)$ rounds and low communication in the MPC model (Theorem 2.2). In this section, we build upon this data structure and show how to process updates and 1-edge-connectivity queries while maintaining a maximal spanning forest of G.

Let $T(v)$ indicate the tree (component) in F to which a vertex v belongs. We define the component ID of v as the as the ID of this $T(v)$. We represent the trees in the forest using the following data structure. We describe the data structure in more detail in Section 3.

THEOREM 2.2. *In the* MPC *model with memory per machine* $s = \widetilde{O}(n^\epsilon)$ *for some constant* ϵ, *for any constant* $0 < \alpha < 1$ *and a key length* ℓ_{key} *such that* $n^\alpha \cdot \ell_{key} \leq s$, *we can maintain a dynamic forest* F *in space* $\widetilde{O}(n)$, *with each vertex* v *augmented with a key* $\boldsymbol{x}_v$ *of length* ℓ_{key} *(*$\boldsymbol{x}_v$ *is a summable element from a semi-group),*

- LINK$(u_1v_1, \ldots, u_kv_k)$*: Insert* k *edges into* F.
- CUT$(u_1v_1, \ldots, u_kv_k)$*: Delete* k *edges from* F.
- ID$(v_1, \ldots, v_k)$*: Given a batch of* k *vertices, return their component IDs in* F.
- UPDATEKEY$((v_1, \widehat{\boldsymbol{x}}'_1), \ldots, (v_k, \widehat{\boldsymbol{x}}'_k))$*: For each* i, *update the value of* $\vec{\boldsymbol{x}}_{v_i}$ *to* $\vec{\boldsymbol{x}}'_i$.
- GETKEY$(v_1, \ldots, v_k)$*: For each* i, *return* $\vec{\boldsymbol{x}}_{v_i}$.
- COMPONENTSUM$(v_1 \ldots, v_k)$*: Given a set of* k *vertices, compute for each* v_i,

$$\sum_{w\,:\, w \in T(v_i)} \boldsymbol{x}_w$$

under the provided semi-group operation.

Moreover, all operations can be performed in $O(1/\alpha)$ *rounds and*

- LINK *and* CUT *operations can be performed in* $\widetilde{O}(k \cdot \ell_{key} \cdot n^\alpha)$ *communication per round,*
- ID *can be performed in* $\widetilde{O}(k)$ *communication per round,*
- UPDATEKEY, GETKEY *and* COMPONENTSUM *operations can be performed in* $\widetilde{O}(k \cdot \ell_{key} \cdot n^\alpha)$ *communication per round.*

Edge insertions and queries can be handled by above dynamic data structure: for a set of edge queries, we use the ID operation to query the ID of all the vertices. Two vertices are in the same connected component if and only if their IDs are same. For a batch of edge insertions, we maintain the spanning forest by first identifying all the inserted edges that join different connected components using ID operation, and then using the LINK operations to put these edges into the forest.

The process of handling a set of edge deletions is more complex. This is because, if some spanning forest edges are removed, then we need to find replacement edges in the graph which were previously not in the spanning forest, but can be added to maintain the

desired spanning forest. To do this, we use the the augmentation of tree nodes with $\boldsymbol{x}_u$ and the ComponentSum operation to accommodate each vertex storing "sketches" in order to find replacement edges upon deletions.

Sketching Based Approach Overview. At the core of the Delete operation is an adaptation of the sketching based approach for finding replacement edges by Ahn et al. [3] and Kapron et al. [46]. Since we rely on these sketches heavily, we go into some detail about the approach here. Without loss of generality, we assume every edge has a unique $O(\log n)$-bit ID, which is generated by a random function on the two vertices involved.

For a vertex v, this scheme sets $\boldsymbol{x}_v$ to the XOR of the edge IDs of all the edges incident to v (which we assume to be integers):

$$\boldsymbol{x}_v \stackrel{\text{def}}{=} \bigoplus_{e\colon e\sim v} e.$$

For a subset of vertices S, we define $\partial(S)$ as the set of edges with exactly one endpoint in S. Then, taking the total XOR over all the vertices in S gives (by associativity of XOR)

$$\bigoplus_{v\in S} \boldsymbol{x}_v = \bigoplus_{v\in S} \bigoplus_{e\colon e\sim v} e = \bigoplus_{e\in E} \left(\bigoplus_{v\colon v\in S, e\sim v} e \right) = \bigoplus_{e\in\partial(S)} e.$$

So if there is only one edge leaving S, this XOR over all vertices in S returns precisely the ID of this edge. To address the case with multiple edges crossing a cut, Ahn *et al.* [3] and Kapron *et al.* [46] sampled multiple subsets of edges at different rates to ensure that no matter how many edges are actually crossing, with high probability one sample picks only one of them. This redundancy does not cause issues because the edge query procedures also serve as a way to remove false positives.

We formally define the sketch as follows:

Definition 2.2. (Graph Sketch from [3, 46])
A sketch with parameter p of a graph $G = (V, E)$ is defined as follows:

1. *Every edge is sampled independently with probability p. Let E' be the set of sampled edges.*
2. *For every vertex $v \in V$, let*

$$\boldsymbol{x}_v \stackrel{\text{def}}{=} \bigoplus_{e\in E'\colon e\sim v} e.$$

We say a sketch generates edge e if there exists a vertex v such that $\boldsymbol{x}_v = e$. The variant of this sketching result that we will use is stated as follows in Lemma 2.1.

Lemma 2.1. (Graph Sketch from [3, 46])
Assume we maintain a sketch for each of $p \in \{1, 1/2, 1/4, \ldots, 1/2^{\lceil 2\ln\rceil - 1}\}$, and let $\vec{\boldsymbol{x}}_v$ denote the sketches on vertex v,

- *upon insertion/deletion of an edge, we can maintain all $\vec{\boldsymbol{x}}_v$'s in $O(\log^2 n)$ update time;*
- *for any subset of vertices S, from the value*

$$\bigoplus_{v\in S} \vec{\boldsymbol{x}}_v,$$

we can compute $O(\log n)$ edge IDs so that for any edge $e \in \partial(S)$, the probability that one of these IDs is e is at least $1/|\partial(S)|$.

Fast Contraction Lemma. As XOR is a semigroup operation, we can use these sketches in conjunction with the dynamic forest data structure given in Theorem 2.2 to check whether a tree resulting from an edge deletion has any outgoing edges. In particular, $O(\log n)$ copies of this sketch structure allow us to find a replacement edge with high probability after deleting a single edge in $O(1/\epsilon)$ rounds and $O(n^\epsilon)$ total communication. Our algorithm then essentially "contracts" these edges found, thus essentially reconnecting temporarily disconnected trees in F.

However, a straightforward generalization of the above method to deleting a batch of k edges results in an overhead of $\Theta(\log k)$, because it's possible that this random contraction process may take up to $\Theta(\log k)$ rounds. Consider for example a length k path: if we pick $O(1)$ random edges from each vertex, then each edge on the path is omitted by both of its endpoints with constant probability. So in the case of a path, we only reduce the number of remaining edges by a constant factor in expectation, leading to a total of about $\Theta(\log k)$ rounds. With our assumption of $s = O(n^\epsilon)$ and queries arriving in batches of $k \le s$, this will lead to a round count that's up to $\Theta(\log n)$. We address this with a natural modification motivated by the path example: instead of keeping $O(\log n)$ independent copies of the sketching data structures, we keep $\widetilde{O}(n^\delta)$ copies, for some small constant δ, which enables us to sample n^δ random edges leaving each connected component at any point. As this process only deals with edges leaving connected components, we can also view these connected components as individual vertices. The overall algorithm then becomes a repeated contraction process on a multi-graph: at each round, each vertex picks n^δ random edges incident to it, and contracts the graph along all picked edges. Our key structural result is a lemma that shows that this process terminates in $O(1/\delta)$ rounds with high

probability. To formally state the lemma, we first define a random process of sampling edges in a graph.

DEFINITION 1.1. (CONTRACTIONSAMPLING PROCESS) *The random process* ContractionSampling *for a multigraph $G = (V, E)$ and an integer k is defined as follows: each vertex v independently draws t_v samples $S_{v,1}, S_{v,2}, \ldots S_{v,t_v}$ for some integer $t_v \geq k$ such that*

1. *the outcome of each $S_{v,i}$ can be an either an edge incident to v or $\perp$;*
2. *for every edge e incident to vertex v,*

$$\sum_{i=1}^{t_v} \Pr[S_{v,i} = e] \geq \Omega\left(\frac{k \log^2 n}{\boldsymbol{d}_G(v)}\right).$$

Below is our structural lemma, which we prove in Section 4.

LEMMA 1.1. *Consider the following contraction scheme starting with a multigraph $G(V, E)$ on n vertices and $m < poly(n)$ (multi) edges: For a fixed integer k,*

1. *let E' be a set of edges sampled by the* ContractionSampling *process;*
2. *contract vertices belonging to same connected component of graph $G' = (V, E')$ into a new graph $G^\star = (V^\star, E^\star)$ as follows: each vertex of $V^\star$ represents a connected component in the sampled graph $G' = (V, E')$, and there is an edge between two vertices $x, y \in V^\star$ iff there is an edge in G between the components corresponding to x and y, with edge multiplicity equal to the sum of multiplicity of edges in G between the components corresponding to x and y.*

Then the resultant graph has at most $\widetilde{O}(mk^{-1/3})$ (multi) edges with high probability.

Independent Sample Extractor From Sketches. On a high level, our idea is to use $O(k \cdot \text{polylog}(n))$ independent sketches to simulate the required ContractionSampling process, and then apply Lemma 1.1. However, we cannot do this naively, because ContractionSampling requires the edges being sampled independently, whereas the sketch from Lemma 2.1 does not satisfy this property. Recall that the sketch generated at a vertex v can correspond to an edge (say uv) if no other edge adjacent to v was sampled in the same sketch. Consider an example where two edges uv and uw are sampled by the graph. This means that no other edge from v or w can be sampled in that same sketch, implying the sampling process is not independent.

We would like to remark that this is not an issue for previous sketching based connectivity algorithms (e.g. [3, 46]), because in [3, 46], each time, any current connected component only needs to find an arbitrary edge leaving the connected component. In this way, if most current connected components find an arbitrary edge leaving the component, then after contracting connected components using sampled edges, the total number of connected components reduce by at least a constant factor. In this way, after $O(\log n)$ iterations, each connected component shrinks into a single vertex. But in our case the contraction lemma requires edges being sampled independently. Hence, we cannot directly apply Lemma 1.1 on sketches.

To get around this issue, we construct an independent edge sample extractor from the sketches and show that with high probability, this extractor will extract a set of independent edge samples that are equivalent to being sampled from a ContractionSampling random process, as required by Lemma 1.1. One key observation is that if the graph is bipartite, then sketch values on the vertices from one side of the bipartite graph are independent, because every edge sample is only related to one sketch value. The high level idea of our extractor is then to extract bipartite graphs from sketches, such that each edge appears in many bipartite graphs with high probability. For each sketch, consider the following random process:

1. For each vertex of the graph, randomly assign a color of red or yellow. Then we can construct a bipartite graph with red vertices on one side, yellow vertices on the other side, and an edge is in the bipartite graph if and only if the color of one endpoint is red, and the other endpoint is yellow. Note that this step is not related to the process of sketch construction.
2. Independently sample every edge not in the bipartite graph with probability same as the probability of sampling used in the sketch.
3. For each red vertex whose incident edges were not sampled in Step 2, independently sample every edge incident to the vertex in the bipartite graph with probability same as that used in the sketch.
4. Choose all the edges sampled in Step 3 which do not share a red vertex with any other sampled edge.

We show that the edges obtained in Step 4 are sampled independently (conditioned on the outcome of Step 2). Another way to see this independence is to partition all the independent random variables in the process of generating all the sketches into two random processes R_1 and R_2 (based on the

bipartite graph generated for each sketch) such that R_1 and R_2 are independent and simulate a required ContractionSampling process in the following sense:

1. After implementing the random process R_1 and based on the outcome of R_1, define a ContractionSampling process as required by Lemma 1.1.
2. The random process R_2 simulates the defined ContractionSampling process in the following sense: there is a partition of the independent random variables of random process R_2 into groups satisfying the following conditions:
   (a) There is a bijection between groups and random variables of the ContractionSampling process.
   (b) For each group, there exists a function of the random variables in the group such that the function is equivalent to the corresponding random variable of the ContractionSampling process.

Furthermore, all the edges sampled by the defined ContractionSampling process are generated by the sketches (meaning that there exist a vertex and a sketch such that sketch on the vertex is the ID of the sampled edge). In this way, we argue that the edges generated by all the sketches contains a set of edges generated by a ContractionSampling process so that we can apply Lemma 1.1.
More formally, we define the simulation between two random processes as follows.

DEFINITION 2.3. *We say a set of independent random variables $E_1, E_2, \ldots, E_t$ simulates another set of independent random variables $F_1, F_2, \ldots, F_\ell$ if there exists a set of random variables $U \subseteq \{E_1, E_2, \ldots, E_t\}$ such that with constant probability, after fixing all the random variables of U, there are ℓ subsets $T_1, T_2, \ldots T_\ell \subseteq \{E_1, E_2, \ldots, E_t\} \setminus U$ (depending on the outcome of the random process for U) satisfying*

1. *$T_1, \ldots, T_\ell$ are mutually disjoint.*
2. *For every $i \in [\ell]$, there exist a random variable which is a function of random variables in T_i, denoted as $f_i(T_i)$, such that $f(T_i)$ is same to the random variable F_i.*

And we show that the process of generating $O(k \log^3 n)$ sketches simulates the random process in the contraction lemma.

LEMMA 1.3. *Given an integer k and a multigraph G of n vertices, $O(k \log^3 n)$ independent sketches simulates a ContractionSampling process. Furthermore, for every edge sampled by ContractionSampling , there exists a sketch and a vertex such that the value of the sketch on the vertex is exactly the ID of that edge.*

3 Batch-Dynamic Trees in MPC

In this section we describe a simple batch-dynamic tree data structure in the MPC setting. Our data structure is based on a recently developed parallel batch-dynamic data structure in the shared-memory setting [67]. Specifically, Tseng et al. give a parallel batch-dynamic tree that supports batches of k links, cuts, and queries for the representative of a vertex in $O(k \log(n/k+1))$ expected work and $O(\log n)$ depth w.h.p. Their batch-dynamic trees data structure represents each tree in the forest using an Euler-tour Tree (ETT) structure [39], in which each tree is represented as the cyclic sequence of its Euler tour, broken at an arbitrary point. The underlying sequence representation is a concurrent skip list implementation that supports batch *join* and *split* operations. Augmented trees are obtained by augmenting the underlying sequence representation. We show that the structure can be modified to achieve low *round-complexity* and *communication* in the MPC setting. We now define the batch-dynamic trees interface and describe how to extend the data structure into the MPC setting. The main difficulty encountered in the shared-memory setting is that nodes are stored in separate memory locations and refer to each other via pointers. Therefore, when traversing the skip list at some level i to find a node's ancestor at level $i+1$, it requires traversing all nodes that occur before (or after) it at level i. We show that by changing the sampling probability to $1/n^\epsilon$, we can ensure that each level has size $\tilde{O}(n^\epsilon)$, each level can be stored within a *single machine* and thus this search can be done within a *single round*. The new sampling probability also ensures that the number of levels is $O(1/\epsilon)$ w.h.p. which is important for achieving our bounds.

Batch-Dynamic Trees Interface. A batch-parallel dynamic trees data structure represents a forest $G = (V, E)$ as it undergoes batches of links, cuts, and connectivity queries. A *Link* links two trees in the forest. A *Cut* deletes an edge from the forest, breaking one tree into two trees. A *ID* query returns a unique representative for the tree containing a vertex. Formally the data structure supports the following operations:

- **Link**$(\{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\})$ takes an array of edges and adds them to the graph G. The input edges must not create a cycle in G.
- **Cut**$(\{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\})$ takes an array of

edges and removes them from the graph G.

- **ID**$(\{u_1, \ldots, u_k\})$ takes an array of vertex ids and returns an array containing the representative of each u_i. The *representative* of a node, $r(u)$ is a unique value s.t. $r(u) = r(v)$ iff u and v are in the same tree.

Furthermore, the trees can be augmented with values ranging over a domain D, and a commutative function $f : D^2 \to D$. The trees can be made to support queries for the sum according to f on arbitrary subtrees, but for the purposes of this paper queries over the entire tree suffice. The interface is extended with the following two primitives:

- **UpdateKey**$(\{\{u_1, \hat{x}_1\}, \ldots, \{u_k, \hat{x}_k\}\})$ takes an array of vertex id, value pairs and updates the value for u_i to $\hat{x}_i$.
- **GetKey**$(\{u_1, \ldots, u_k\})$ takes an array of vertex ids and returns an array containing the value of each u_i, $\hat{x}_i$.
- **ComponentSum**$(\{u_1, \ldots, u_k\})$ takes an array of vertex ids and returns an array containing $\sum_{w:w \in T(u_i)} \hat{x_w}$ where $T(u_i)$ is the tree containing u_i, $\hat{x_w}$ is the value for node w, and the sum is computed according to f.

We show the following theorem in this section. Let δ be a parameter controlling the size of the keys stored at each node and let α be a parameter controlling the size of the blocks stored internally within a single machine.

THEOREM 3.1. *Let δ be a parameter controlling the keysize, and α be a constant controlling the blocksize s.t. $\delta + \alpha < \epsilon$ and $0 < \alpha$. Then, in the* MPC *model with memory per machine $s = \tilde{O}(n^\epsilon)$ there is an augmented batch-dynamic tree data structure in* MPC *that supports batches of up to k* LINK, CUT, ID, UPDATEKEY, GETKEY, *and* COMPONENTSUM *operations in $O(1/\alpha)$ rounds per operation w.h.p. where $k = O(n^\alpha)$.*
Furthermore, the batch operations cost

- *$\tilde{O}(kn^\delta)$ communication per round w.h.p. for* UPDATEKEY, GETKEY, *and* COMPONENTSUM
- *$\tilde{O}(kn^\delta n^\alpha)$ communication per round w.h.p. for* LINK *and* CUT *and*
- *$O(k)$ communication per round for* ID.

3.1 Augmented Batch-Dynamic Sequences in MPC In order to obtain Theorem 3.1, we first show how to implement augmented batch-dynamic sequences in few rounds of MPC. In particular, we will show the following lemma. Note that achieving a similar bound on the round-complexity for large batches, e.g., batches of size $O(n)$, would disprove the 2-cycle conjecture. We refer to [67] for the precise definition of the sequence interface.

LEMMA 3.1. *Let δ be a parameter controlling the keysize, and α be a constant controlling the blocksize s.t. $\delta + \alpha < \epsilon$ and $0 < \alpha$. Then, in the* MPC *model with memory per machine $s = \tilde{O}(n^\epsilon)$ there is an augmented batch-dynamic sequence data structure in* MPC *that supports batches of up to k* SPLIT, JOIN, ID, UPDATEKEY, GETKEY, *and* SEQUENCESUM *operations in $O(1/\alpha)$ rounds per operation w.h.p. where $k = O(n^\alpha)$.*
Furthermore, the batch operations cost

- *$\tilde{O}(kn^\delta)$ communication per round w.h.p. for* UPDATEKEY, GETKEY, *and* SEQUENCESUM
- *$\tilde{O}(kn^\delta n^\alpha)$ communication per round w.h.p. for* SPLIT *and* JOIN *and*
- *$O(k)$ communication per round for* ID.

For the sake of simplicity we discuss the case where $\delta = 0$ and $0 < \alpha < \epsilon$ (i.e. values that fit within a constant number of machine words), and describe how to generalize the idea to larger values at the end of the sub-section.

Sequence Data Structure. As in Tseng et al. [67] we use a skip list as the underlying sequence data structure. Instead of sampling nodes with constant probability to join the next level, we sample them with probability $1/n^\alpha$. It is easy to see that this ensures that the number of levels in the list is $O(1/\alpha)$ w.h.p. since α is a constant greater than 0. Furthermore, the largest number of nodes in some level i that "see" a node at level $i+1$ as their left or right ancestor is $O(n^\alpha \log n)$ w.h.p. We say that the left (right) *block* of a node belonging to level i are all of its siblings to the left (right) before the next level $i+1$ node. As previously discussed, in the MPC setting we should intuitively try to exploit the locality afforded by the MPC model to store the blocks (contiguous segments of a level) on a single machine. Since each block fits within a single machine w.h.p., operations within a block can be done in 1 round, and since there are $O(1/\alpha)$ levels, the total round complexity will be $O(1/\alpha)$ as desired. Since the ideas and data structure are similar to Tseng et al. [67], we only provide the high-level details and refer the reader to their paper for pseudocode.

Join. The join operation takes a batch of pairs of sequence elements to join, where each pair contains the rightmost element of one sequence and the leftmost element of another sequence. We process the levels one by one. Consider a join of (r_i, l_i). We scan the blocks for r_i and l_i to find their left and right ancestors, and join them. In the

subsequent round, these ancestors take the place of (r_i, l_i) and we recursively continue until all levels are processed. Observe that at each level, for each join we process we may create a new block, with $\tilde{O}(n^\alpha)$ elements. In summary, the overall round-complexity of the operation is $O(1/\alpha)$ w.h.p., and the amount of communication needed is $\tilde{O}(kn^\alpha)$ w.h.p.

Split. The split operation takes a batch of sequence elements at which to split the sequences they belong to by deleting the edge to the right of the element. We process the levels one by one. Consider a split at a node e_i. On each level, we first find the left and right ancestors as in case of join. We then send all nodes splitting a given block to the machine storing that block, and split it in a single round. Then, we recurse on the next level. If the left and right ancestors of e_i were connected, we call split on the left right ancestor at the next level. The overall round-complexity is $O(1/\alpha)$ w.h.p., and the amount of communication needed is $\tilde{O}(kn^\alpha)$ w.h.p.

Augmentation and Other Operations. Each node in the skip list stores an augmented value which represents the sum of all augmented values of elements in the block for which it is a left ancestor. Note that these values are affected when performing splits and joins above, but are easily updated within the same round-complexity by computing the correct sum within any block that was modified and updating its left ancestor. SetKey operations, which take a batch of sequence elements and update the augmented values at these nodes can be handled similarly in the same round-complexity as join and split above. Note that this structure supports efficient range queries over the augmented value, but for the purposes of this paper, returning the augmented value for an entire sequence (SequenceSum) is sufficient, and this can clearly be done in $O(1/\alpha)$ rounds and $O(k)$ communication. Similarly, returning a representative node (ID) for the sequence can be done in $O(1/\alpha)$ rounds w.h.p. and $O(k)$ communication by finding the top-most level for the sequence containing the queried node, and returning the lexicographically first element in this block.

Handling Large Values. Note that if the values have super-constant size, i.e. size $O(n^\delta)$ for some δ s.t. $\delta + \alpha < \epsilon$ we can recover similar bounds as follows. Since the blocks have size $\tilde{O}(n^\alpha)$ and each value has size $O(n^\delta)$ the overall size of the block is $\tilde{O}(n^{\alpha+\delta}) = \tilde{O}(n^\epsilon)$. Therefore blocks can still be stored within a single machine without changing the sampling parameter. Storing large values affects the bounds as follows. First, the communication cost of performing splits and joins grows by a factor of $O(n^\delta)$ due to the increased block size. Second, the cost of getting, setting, and performing a component sum grows by a factor of $O(n^\delta)$ as well, since k values are returned, each of size $O(n^\delta)$. Therefore the communication cost of all operations other than finding a represntative increase by a multiplicative $O(n^\delta)$ factor. Finally, note that the bounds on round-complexity are not affected, since nodes are still sampled with probability $1/n^\alpha$.

3.2 Augmented Batch-Dynamic Trees in MPC

We now show how to implement augmented batch-dynamic trees in MPC, finishing the proof of Theorem 3.1. We focus on the case where $\delta = 0$ (we are storing constant size words) and explain how the bounds are affected for larger δ.

Forest Data Structure. We represent trees in the forest by storing the Euler tour of the tree in a sequence data structure. If the forest is augmented under some domain D and commutative function $f : D^2 \to D$, we apply this augmentation to the underlying sequences.

Link. Given a batch of link operations (which are guaranteed to be acyclic) we update the forest structure as follows. Consider a link (u_i, v_i). We first perform a batch split operation on the underlying sequences at all u_i, v_i for $1 \leq i \leq k$, which splits the Euler tours of the underlying trees at the nodes incident to a link. Next, we send all of the updates to a single machine to establish the order in which joins incident to a single vertex are carried out. Finally, we perform a batch join operation using the order found in the previous round to link together multiple joins incident to a single vertex. Since we perform a constant number of batch-sequence operations with batches of size $O(k)$, the overall round complexity is $O(1/\alpha)$ w.h.p. by our bounds on sequences, and the overall communication is $\tilde{O}(kn^\alpha)$ w.h.p.

Cut. Given a batch of cut operations, we update the forest structure as follows. Consider a cut (u_i, v_i). The idea is to splice this edge out of the Euler tour by splitting before and after (u_i, v_i) and (v_i, u_i) in the tour. The tour is then repaired by joining the neighbors of these nodes appropriately. In the case of batch cuts, we perform a batch split for the step above. For batch cuts, notice that many edges incident to a node could be deleted, and therefore we may need to traverse a sequence of deleted edges before finding the next neighbor to join. We handle this by sending all deleted edges and their neighbors to a single machine, which determines which nodes should be joined together to repair the tour. Finally,

we repair the tours by performing a batch join operation. Since we perform a constant number of batch-sequence operations with batches of size $O(k)$ the overall round complexity is $O(1/\alpha)$ w.h.p. by our bounds on sequences, and the overall communication is $\tilde{O}(kn^{\alpha})$ w.h.p.

Augmentation, Other Operations and Large Values. Note that the underlying sequences handle updating the augmented values, and that updating the augmented values at some nodes trivially maps to an set call on the underlying sequences. Therefore the bounds for GetKey and SetKey are identical to that of sequences. Similarly, the bounds for ID are identical to that of the sequence structure. For super-constant size values, the bounds are affected exactly as in the case for augmented sequences with large values. The communication costs for all operations other than ID grow by an $O(n^{\delta})$ factor and the round-complexity is unchanged. This completes the proof of Theorem 3.1.

4 Fast Contraction

The aim of this section is to prove Lemma 1.1, which is pivotal in proving the correctness of the main algorithm from Section 2.

Lemma 1.1 is important in proving that our algorithm can find replacement edges in the spanning forest quickly in the event of a batch of edges being deleted. The proof idea is as follows. We first show that there exists a partitioning of the vertices such that the edges within the partitions collapse in a single iteration.

To do this, we first need to define a few terms relating to expansion criteria of a graph. Let $\boldsymbol{d}_G(v)$ denote the degree of a vertex v in graph G. For edges in a partition to collapse in a single iteration, we need each partition to be sufficiently "well-knit". This property can be quantified using the notion of conductance.

Definition 1.2. (Conductance) *Given a graph $G(V, E)$ and a subset of vertices $S \subseteq V$, the conductance of S w.r.t. G is defined as*

$$\phi_G(S) \stackrel{\text{def}}{=} \min_{S' \subseteq S} \frac{|E(S', S \setminus S')|}{\min\left\{\sum_{u \in S'} \boldsymbol{d}_G(u), \sum_{u \in S \setminus S'} \boldsymbol{d}_G(u)\right\}}.$$

The following lemma proves the existence of a partitioning such that each partition has high conductance.

Lemma 1.2. ([65], Section 7.1.) *Given a parameter $k > 0$, any graph G with n vertices and m edges can be partitioned into groups of vertices $S_1, S_2, \ldots$ such that*

- *the conductance of each S_i is at least $1/k$;*
- *the number of edges between the S_i's is at most $O(m \log n/k)$.*

Now that we have a suitable partitioning, we want to find a strategy of picking edges in a decentralized fashion such that all edges within a partition collapse with high probability. One way to do this is to pick edges which form a spectral sparsifier of S_i. The following lemma by Spielman and Srivastava [64] helps in this regard: we use more recent interpretations of it that take sampling dependencies into account.

Lemma 4.1. ([64, 66, 49, 52]) *On a graph G, let $E_1 \ldots E_k$ be independent random distributions over edges such that the total probability of an edge e being picked is at least $\Omega(\log n)$ times its effective resistance, then a random sample from $H = E_1 + E_2 + \ldots + E_k$ is connected with high probability.*

Now we want to show that the random process ContractionSampling (Defintion 1.1) where each vertex draws $k \log^2 n$ samples actually satisfies the property mentioned in Lemma 4.1, i.e., all edges are picked with probability at least their effective resistance. To show this, we first need the following inequality given by Cheeger.

Lemma 4.2. ([4]) *Given a graph G, for any subset of vertices S with conductance ϕ, we have*

$$\lambda_2\left(\boldsymbol{D}_S^{-1/2} \boldsymbol{L}_S \boldsymbol{D}_S^{-1/2}\right) \geq \frac{1}{2}\phi^2,$$

where $\boldsymbol{L}_S$ is the Laplacian matrix of the subgraph of G induced by S. D_S is the diagonal matrix with degrees of vertices in S.

Lemma 4.3. *Let S be a subset of vertices of G such that $\phi_G(S) \geq 1/2\alpha^{1/3}$ for some $\alpha > 0$. For an edge $e = uv$, where $u, v \in S$, the effective resistance of e measured in S, $ER_S(e)$, satisfies*

$$ER_S(e) \leq \alpha\left(\frac{1}{\boldsymbol{d}_G(u)} + \frac{1}{\boldsymbol{d}_G(v)}\right).$$

Proof. From Lemma 4.2, we get that

$$\boldsymbol{L}_S \succeq \frac{1}{2}\left(\phi_G(S)\right)^2 \boldsymbol{\Pi}_{\perp \vec{1}_S} D_S \boldsymbol{\Pi}_{\perp \vec{1}_S}.$$

Using this, along with the definition $ER_S(u, v) \stackrel{\text{def}}{=} \chi_{uv}^T \boldsymbol{L}_S^{\dagger} \chi_{uv}$, gives us that

$$ER_S(u, v) \leq \frac{1}{2}\left(\phi_G(S)\right)^{-2}\left(\frac{1}{\boldsymbol{d}_S(u)} + \frac{1}{\boldsymbol{d}_S(v)}\right). \tag{4.1}$$

We have for any subset $S' \subseteq S$ that:

$$\frac{E(S', S \setminus S')}{\min\left\{\sum_{u \in S'} \boldsymbol{d}_G(u), \sum_{u \in S \setminus S'} \boldsymbol{d}_G(u)\right\}} \geq \phi_G(S).$$

Furthermore, for every vertex $v \in S$, we get $\boldsymbol{d}_S(v)/\boldsymbol{d}_G(v) \geq \phi_G(S)$, which when substituted into Equation 4.1 gives

$$ER_S(u, v) \leq \frac{1}{2} \left(\phi_G(S)\right)^{-3} \left(\frac{1}{\boldsymbol{d}_G(u)} + \frac{1}{\boldsymbol{d}_G(v)}\right).$$

Using $\phi_G(S) \geq 1/2\alpha^{1/3}$ completes the proof. ∎

Now, we have enough ammunition to prove Lemma 1.1.

Proof. [Proof of Lemma 1.1] From Lemma 1.2, we know that our graph can be partitioned into expanders with conductance at least $\Omega(k^{-1/3} \log^{1/3} n)$. Now, let S be one such partition and let $e = uv$ be an edge contained in S. From the definition of the random process in Definition 1.1, we know that for an edge uv, the probability that it is sampled by either u or v is at least

$$k \log^2 n \left(\frac{1}{\boldsymbol{d}_G(u)} + \frac{1}{\boldsymbol{d}_G(v)}\right) \geq ER_S(uv) \cdot \Omega(\log n),$$

where the inequality follows from Lemma 4.3. Since each such edge uv is chosen with probability greater than $\Omega(\log n)$ times its effective resistance w.r.t. S, from Lemma 4.1, we know that the edges chosen within S are connected with high probability.
Thus, we are left with the edges between the partitions, the number of which is bounded by $O(m \log^{4/3} n \cdot k^{-1/3})$ edges, ∎

5 Connectivity Algorithms and Correctness

We give the algorithms for batch edge queries, batch edge insertions, and batch edge deletions and prove the correctness in Section 5.1, Section 5.2 and Section 5.3 respectively. Putting together Lemmas 5.1, 5.3 and 5.2 then gives the overall result as stated in Theorem 1.1.
Throughout this section, we will use the batch-dynamic tree data structure discussed in Section 3 to maintain

1. a maximal spanning forest F of the graph,
2. a key $\vec{\boldsymbol{x}}_v$ for every vertex v, where $\vec{\boldsymbol{x}}_v$ is a vector of $\widetilde{O}(n^\delta)$ sketch values on vertex v,
3. an edge list data structure which can be used to check if an edge is in the graph given an edge ID.

5.1 Algorithm for Batch Edge Queries

Since F is a maximal spanning tree, the query operations are directly provided by calling ID on all involved vertices. Pseudocode of this routine is in Algorithm 5.1.

LEMMA 5.1. *The algorithm* QUERY *(Algorithm 5.1) correctly answers connectivity queries and takes $O(1/\alpha)$ rounds, each with total communication at most $\widetilde{O}(k)$.*

Proof. The correctness and performance bounds follow from the fact that F is a maximal spanning forest of F and from Theorem 2.2. ∎

5.2 Algorithm for Batch Edge Insertions

Given a batch of k edge insertions, we want to identify a subset of edges from the batch that are going to add to F to maintain the invariant that F is a maximal spanning forest. To do this, we use ID operation to find IDs of all the involved vertices in the edge insertion batch. Then we construct a graph G_{local} which initially contains all the edges in the edge insertion batch, and then contracts vertices from same connected component of F to a single vertex. Since this graph contains k edges, we can put this graph into a single machine, and compute a spanning forest F_{local} of G_{local}. We maintain the maximal spanning forest F by adding edges in F_{local} to F. We also maintain the edge list data structure by adding inserted edges to the list, and maintain the sketches for the involved vertices by the UPDATEKEY operation. Pseudocode of the batched insertion routine is in Algorithm 5.2.

LEMMA 5.2. *The algorithm* INSERT *in Algorithm 5.2 correctly maintains a maximal spanning forest of G and takes $O(1/\alpha)$ rounds, each with total communication at most $\widetilde{O}(kn^{\alpha+\delta})$.*

Proof. To show the correctness, notice that since we add only a forest on the components as a whole, there is never an edge added between two already connected components. Additionally, since the forest is spanning, we do not throw away any necessary edges.
From Theorem 2.2, using GETKEY, UPDATEKEY, ID and LINK falls under the claimed bound for rounds and communication, whereas the rest of the steps are performed only locally. ∎

5.3 Algorithm for Batch Edge Deletions

Pseudocode of the batched deletion routine is in Algorithm 5.3.

QUERY$((u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k))$
Input: Pairs of vertices $(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)$
Output: For each $1 \leq i \leq k$, **yes** if u_i and v_i are connected in G, and **no** otherwise.
1. Call ID$(u_1, v_1, u_2, v_2, \ldots, u_k, v_k)$.
2. For each i, output **yes** if u_i and v_i have the same component ID, and **no** otherwise.

Algorithm 5.1: Querying the connectivity between a batch of vertex pairs

INSERT$(u_1v_1, u_2v_2, \ldots, u_kv_k)$
Input: new edges $e_1 = u_1v_1, e_2 = u_2v_2, \ldots, e_k = u_kv_k$.
1. Add all k edges to the edge list data structure.
2. Run GETKEY$(u_1, v_1, \ldots, u_k, v_k)$.
3. For every sketch, sample every inserted edge with probability equal to the parameter of the sketch, and compute the updated key value for vertices $\vec{\boldsymbol{x}}'_{u_1}, \vec{\boldsymbol{x}}'_{v_1}, \ldots, \vec{\boldsymbol{x}}'_{u_k}, \vec{\boldsymbol{x}}'_{v_k}$.
4. Run UPDATEKEY$((u_1, \vec{\boldsymbol{x}}'_{u_1}), (v_1, \vec{\boldsymbol{x}}'_{v_1}), \ldots, (u_k, \vec{\boldsymbol{x}}'_{u_k}), (v_k, \vec{\boldsymbol{x}}'_{v_k}))$.
5. Run ID$(\{u_1, v_1, u_2, v_2 \ldots u_k, v_k\})$.
6. Using these IDs as vertex labels, construct a graph G_{local} among the inserted edges, on a local machine.
7. Find a maximal spanning forest F_{local} of G_{local} locally on this machine.
8. Run LINK$(E(F_{local}))$.

Algorithm 5.2: Pseudocode for maintaining the data structure upon a batch of insertions.

DELETE$(e_1, e_2, \ldots, e_k)$
Input: edges $e_1 = u_1v_1, e_2 = u_2v_2, \ldots, e_k = u_kv_k$ that are currently present in the graph.
1. Update the global edge index structure.
2. Run GETKEY$(u_1, v_1, \ldots, u_k, v_k)$.
3. For every sketch, compute the updated key value $\vec{\boldsymbol{x}}'_{u_1}, \vec{\boldsymbol{x}}'_{v_1}, \ldots, \vec{\boldsymbol{x}}'_{u_k}, \vec{\boldsymbol{x}}'_{v_k}$ for vertices $u_1, v_1, \ldots, u_k, v_k$ by removing the IDs of edges $e_1, \ldots, e_k$.
4. Run UPDATEKEY$((u_1, \vec{\boldsymbol{x}}'_{u_1}), (v_1, \vec{\boldsymbol{x}}'_{v_1}), \ldots, (u_k, \vec{\boldsymbol{x}}'_{u_k}), (v_k, \vec{\boldsymbol{x}}'_{v_k}))$.
5. Run CUT for all edges that are in the spanning forest. Let $u_1 \ldots u_t$ be representative vertices from the resulting trees
6. Run COMPONENTSUM$(\{u_1 \ldots u_t\})$ to extract the total XOR values from each of the trees.
7. Repeat $O(1/\delta)$ rounds:
   (a) From the XOR values of the current components, deduce a list of potential replacement edges, E_R
   (b) Identify the subset of edges with endpoints between current components given by ID$(u_1) \ldots$ ID(u_t) using a call to QUERY.
   (c) Find T_R, a maximal spanning forest of the valid replacement edges, via local computation.
   (d) LINK$(E(T_R))$.
   (e) Update $u_1 \ldots u_t$ and their XOR values, either using another batch of queries, or by a local computation.

Algorithm 5.3: Pseudocode for maintaining the data structure upon a batch of deletions.

LEMMA 5.3. *The algorithm* DELETE *(Algorithm 5.3) correctly maintains a maximal spanning forest of G and takes $O(1/\delta\alpha)$ rounds, each with total communication at most $\widetilde{O}(kn^{\alpha+\delta})$.*

Proof. Note that F remains a maximal spanning forest if the deleted edges are from outside of F. So, we only need to deal with the complementary case. Consider some tree $T \in F$, from which we deleted $\hat{k}-1$ edges. T is now separated into $\hat{k}$ trees, $T_1, T_2, \ldots, T_{\hat{k}}$. We need to show that the algorithm eventually contracts all T_i using the edges stored in the sketches. For this, note that the guarantees of Lemma 1.3 imply that from the $\widetilde{O}(n^{\delta})$ copies of sketches, we can sample edges leaving a group of T_is in ways that meet the requirements of Lemma 1.1. These trees will collapse into singleton vertices in $O(1/\delta)$ rounds with high probability by applying Lemma 1.1 iteratively. Thus the result is correct.
Steps 1-6 only require $O(1/\alpha)$ rounds of communication, from Theorem 2.2. Step 7 loops $O(1/\delta)$ times, and its bottleneck is step 7b, the verification of the locations of the endpoints in the trees. Once again by the guarantees of Theorem 2.2, this takes $O(1/\alpha)$ rounds for each iteration, and at most $\widetilde{O}(kn^{\delta+\alpha})$ communication per round.
Lastly, we call LINK on the edges in E_R across various iterations. Since at most k edges are deleted from F, there can only be at most k replacement edges, so the total communication caused by these is $\widetilde{O}(kn^{\alpha+\delta})$. ∎

References

[1] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 381–392, 2019.

[2] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. *ACM Transactions on Parallel Computing (TOPC)*, 4(4):17, 2018.

[3] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 459–467, 2012.

[4] Noga Alon and Vitali D Milman. λ1, isoperimetric inequalities for graphs, and superconcentrators. *Journal of Combinatorial Theory, Series B*, 38(1):73–88, 1985.

[5] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005.

[6] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 574–583, 2014.

[7] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 674–685, 2018.

[8] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Log diameter rounds algorithms for 2-vertex and 2-edge connectivity. *arXiv preprint arXiv:1905.00850*, 2019.

[9] Sepehr Assadi. Simple round compression for parallel vertex cover. *arXiv preprint arXiv:1709.04599*, 2017.

[10] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet edcs: algorithms for matching and vertex cover on massive graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1616–1635, 2019.

[11] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for (Δ+ 1) vertex coloring. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 767–786, 2019.

[12] Sepehr Assadi, Nikolai Karpov, and Qin Zhang. Distributed and streaming linear programming in low dimensions. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 236–253, 2019.

[13] Sepehr Assadi and Sanjeev Khanna. Randomized composable coresets for matching and vertex cover. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 3–12, 2017.

[14] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. *To appear in ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

[15] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5):454–465, 2012.

[16] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

[17] Rafael da Ponte Barbosa, Alina Ene, Huy L Nguyen, and Justin Ward. A new framework for distributed submodular maximization. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 645–654. Ieee, 2016.

[18] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab Mirrokni. Massively parallel

dynamic programming on trees. *arXiv preprint arXiv:1809.03685*, 2018.

[19] MohammadHossein Bateni, Aditya Bhaskara, Silvio Lattanzi, and Vahab Mirrokni. Distributed balanced clustering via mapping coresets. In *Advances in Neural Information Processing Systems*, pages 2591–2599, 2014.

[20] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 273–284, 2013.

[21] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Łącki, Vahab Mirrokni, and Warren Schudy. Massively parallel computation via remote memory access. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 59–68, 2019.

[22] Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G Harris. Exponentially faster massively parallel maximal matching. *arXiv preprint arXiv:1901.03744*, 2019.

[23] Guy E Blelloch and Bruce M Maggs. Parallel algorithms. *ACM Computing Surveys (CSUR)*, 28(1):51–54, 1996.

[24] Sebastian Brandt, Manuela Fischer, and Jara Uitto. Matching and MIS for uniformly sparse graphs in the low-memory MPC model. *arXiv preprint arXiv:1807.05374*, 2018.

[25] Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The complexity of $(\Delta+1)$ coloring in congested clique, massively parallel computation, and centralized local computation. In *PODC*, pages 471–480. ACM, 2019.

[26] Artur Czumaj, Jakub Łącki, Aleksander Mądry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *ACM Symposium on Theory of Computing (STOC)*, pages 471–484, 2018.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[28] Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using mapreduce. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 681–689, 2011.

[29] Alina Ene and Huy Nguyen. Random coordinate descent methods for minimizing decomposable submodular functions. In *International Conference on Machine Learning (ICML)*, pages 787–795, 2015.

[30] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *Journal of Algorithms*, 17(2):237–250, 1994.

[31] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[32] Z. Galil and G. Italiano. Fully dynamic algorithms for 2-edge connectivity. *SIAM Journal on Computing*, 21(6):1047–1069, 1992.

[33] Buddhima Gamlath, Sagar Kale, Slobodan Mitrović, and Ola Svensson. Weighted matchings via unweighted augmentations. *arXiv preprint arXiv:1811.02760*, 2018.

[34] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. *To appear in IEEE Symposium on Foundations of Computer Science (FOCS)*, 2019.

[35] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019.

[36] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015. Availabel at: http://arxiv.org/abs/1509.06464.

[37] Raymond Greenlaw, H James Hoover, Walter L Ruzzo, et al. *Limits to parallel computation: P-completeness theory*. Oxford University Press on Demand, 1995.

[38] MohammadTaghi Hajiaghayi, Saeed Seddighin, and Xiaorui Sun. Massively parallel approximation algorithms for edit distance and longest common subsequence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1654–1672, 2019.

[39] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[40] Hemant Hingave and Rasika Ingle. An approach for mapreduce based log analysis using hadoop. In *IEEE International Conference on Electronics and Communication Systems (ICECS)*, pages 1264–1268, 2015.

[41] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001. Announced at STOC'98.

[42] Sungjin Im and Benjamin Moseley. A conditional lower bound on graph connectivity in mapreduce. *CoRR*, abs/1904.08954, 2019.

[43] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *ACM Symposium on Theory of Computing (STOC)*, pages 798–811, 2017.

[44] Giuseppe F Italiano, Silvio Lattanzi, Vahab S Mirrokni, and Nikos Parotsidis. Dynamic algorithms for the massively parallel computation model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 49–58, 2019.

[45] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *IEEE International Conference on Data Mining (ICDM)*, pages 229–238, 2009.

[46] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2013.

[47] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[48] Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. Connected components in mapreduce and beyond. In *ACM Symposium on Cloud Computing (SOCC)*, pages 1–13, 2014.

[49] Ioannis Koutis, Alex Levin, and Richard Peng. Faster spectral sparsification and numerical algorithms for SDD matrices. *ACM Trans. Algorithms*, 12(2):17:1–17:16, 2016. Available at http://arxiv.org/abs/1209.5821.

[50] Clyde P Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, 1990.

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

[52] Rasmus Kyng, Jakub Pachocki, Richard Peng, and Sushant Sachdeva. A framework for analyzing resparsification algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2032–2043, 2017.

[53] Jakub Łącki, Vahab Mirrokni, and Michał Włodarczyk. Connected components at scale via local contractions. *arXiv preprint arXiv:1807.10727*, 2018.

[54] Guojun Liu, Ming Zhang, and Fei Yan. Large-scale social network analysis based on mapreduce. In *IEEE International Conference on Computational Aspects of Social Networks (CASoN)*, pages 487–490, 2010.

[55] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *ACM International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[56] Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *Advances in Neural Information Processing Systems*, pages 2049–2057, 2013.

[57] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $O(n^{1/2-\epsilon})$-time. In *Symposium on Theory of Computing (STOC)*, pages 1122–1129, 2017.

[58] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 950–961, 2017.

[59] Krzysztof Onak. Round compression for parallel graph algorithms in strongly sublinear space. *arXiv preprint arXiv:1807.08745*, 2018.

[60] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *IEEE Conference on Data Engineering (ICDE)*, pages 50–61, 2013.

[61] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *Journal of the ACM (JACM)*, 65(6):41, 2018.

[62] Anish Das Sarma, Foto N Afrati, Semih Salihoglu, and Jeffrey D Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *Proceedings of the VLDB Endowment*, volume 6, pages 277–288, 2013.

[63] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, June 1983.

[64] D. Spielman and N. Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011. Available at http://arxiv.org/abs/0803.0929.

[65] D. Spielman and S. Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011. Available at http://arxiv.org/abs/0808.4134.

[66] Joel A. Tropp. User-friendly tail bounds for sums of random matrices. *Found. Comput. Math.*, 12(4):389–434, August 2012. Available at http://arxiv.org/abs/1004.4389.

[67] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. Batch-parallel Euler tour trees. *Algorithm Engineering and Experiments (ALENEX)*, pages 92–106, 2019.

[68] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *ACM Symposium on Theory of Computing (STOC)*, pages 1130–1143, 2017.

[69] Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p-distances. In *International Conference on Machine Learning (ICML)*, 2018.