2018 IEEE 59th Annual Symposium on Foundations of Computer Science

Parallel Graph Connectivity in Log Diameter Rounds

Alexandr Andoni*, Zhao Song†, Clifford Stein*, Zhengyu Wang† and Peilin Zhong*
* *Department of Computer Science*
Columbia University, New York, NY, USA
{andoni,cliff}@cs.columbia.edu ,peilin.zhong@columbia.edu
† *School of Engineering And Applied Sciences*
Harvard University, Cambridge, MA, USA
{zhaos,zhengyuwang}@g.harvard.edu

***Abstract*—Many modern parallel systems, such as MapReduce, Hadoop and Spark, can be modeled well by the MPC model. The MPC model captures well coarse-grained computation on large data — data is distributed to processors, each of which has a sublinear (in the input data) amount of memory and we alternate between rounds of computation and rounds of communication, where each machine can communicate an amount of data as large as the size of its memory. This model is stronger than the classical PRAM model, and it is an intriguing question to design algorithms whose running time is smaller than in the PRAM model.**

One fundamental graph problem is connectivity. On an undirected graph with n nodes and m edges, $O(\log n)$ round connectivity algorithms have been known for over 35 years. However, no algorithms with better complexity bounds were known. In this work, we give fully scalable, faster algorithms for the connectivity problem, by parameterizing the time complexity as a function of the *diameter* of the graph. Our main result is a $O(\log D \log\log_{m/n} n)$ time connectivity algorithm for diameter-D graphs, using $\Theta(m)$ total memory. If our algorithm can use more memory, it can terminate in fewer rounds, and there is no lower bound on the memory per processor.

We extend our results to related graph problems such as spanning forest, finding a DFS sequence, exact/approximate minimum spanning forest, and bottleneck spanning forest. We also show that achieving similar bounds for reachability in *directed graphs* would imply faster boolean matrix multiplication algorithms.

We introduce several new algorithmic ideas. We describe a general technique called *double exponential speed problem size reduction* which roughly means that if we can use total memory N to reduce a problem from size n to n/k, for $k = (N/n)^{\Theta(1)}$ in one phase, then we can solve the problem in $O(\log\log_{N/n} n)$ phases. In order to achieve this fast reduction for graph connectivity, we use a multistep algorithm. One key step is a carefully constructed truncated broadcasting scheme where each node broadcasts neighbor sets to its neighbors in a way that limits the size of the resulting neighbor sets. Another key step is *random leader contraction*, where we choose a smaller set of leaders than many previous works do.

***Keywords*-MPC model; MapReduce; graph connectivity; parallel algorithm; diameter;**

I. Introduction

Recently, several parallel systems, including MapReduce [DG04], [DG08], Hadoop [Whi12], Dryad [IBY+07], Spark [ZCF+10], and others, have become successful in practice. This success has sparked a renewed interest in algorithmic ideas for these parallel systems.

One important theoretical direction has been to develop good models of these modern systems and to relate them to classic models such as PRAM. The work of [FMS+10], [KSV10], [GSZ11], [BKS13], [ANOY14] have led to the model of *Massive Parallel Computing* (MPC) that balances accurate modeling with theoretical elegance. MPC is a variant of the Bulk Synchronous Parallel (BSP) model [Val90]. In particular, MPC allows N^δ space per machine (processor), where $\delta \in (0,1)$ and N is the input size, with alternating rounds of unlimited local computation, and communication of up to N^δ data per processor. An MPC algorithm can equivalently be seen as a small circuit, with arbitrary, N^δ-fan-in gates; the depth of the circuit is the parallel time. Any PRAM algorithm can be simulated on MPC in the same parallel time [KSV10], [GSZ11]. However, MPC is in fact more powerful than the PRAM: even computing the XOR of N bits requires near-logarithmic parallel-time on the most powerful CRCW PRAMs [BH89], whereas it takes constant, $O(1/\delta)$, parallel time on the MPC model.

The main algorithmic question of this area is then: for which problems can we design MPC algorithms that are *faster* than the best PRAM algorithms? Indeed, this question has been the focus of several recent papers, see, e.g., [KSV10], [LMSV11], [EIM11], [ANOY14], [AG18], [AK17], [IMS17], [CLM+18]. Graph problems have been particularly well studied and one fundamental problem is *connectivity in a graph*. While this problem has a standard logarithmic time PRAM algorithm [SV82], we do not know whether we can solve it faster in the MPC model.

While we would like *fully scalable* algorithms—which work for any value of $\delta > 0$—there have been graph algorithms that use space close to the number of vertices n of the graph. In particular, the result of [LMSV11] showed a faster algorithm for the setting when the space per machine is polynomially larger than the number of vertices, i.e., $s \geq n^{1+\Omega(1)}$, and hence the number of edges is necessarily $m \geq n^{1+\Omega(1)}$. In fact, similar space restrictions are pervasive for all known sub-logarithmic time graph algo-

2575-8454/18/$31.00 ©2018 IEEE
DOI 10.1109/FOCS.2018.00070

rithms, which require $s = \Omega(\frac{n}{\log^{O(1)} n})$ [LMSV11], [AG18], [AK17], [CLM+18] (the only exception is [ANOY14] who consider geometric graphs). We highlight the work of [CLM+18], who manage to obtain *slightly sublinear* space of $n/\log^{\Omega(1)} n$ in $\log^{O(1)} \log n$ parallel time, for the approximate matching problem and [ABB+17] who obtain *slightly sublinear* space of $n/\log^{\Omega(1)} n$ in $O(\log \log n)$ parallel time. We note that the space of $\sim n$ also coincides with the space barrier of the semi-streaming model: essentially no graph problems are solvable in less than n space in the streaming model, unless we have many more passes; see e.g. the survey [McG09].

It remains a major open question whether there exist fully scalable connectivity MPC algorithms with sub-logarithmic time (e.g., for sparse graphs). There are strong indications that such algorithms do *not* exist: [BKS13] show logarithmic lower bounds for restricted algorithms. Alas, showing an unconditional lower bound may be hard to prove, as that would imply circuit lower bounds [RVW16].

In this work, we show **faster, fully scalable algorithms for the connectivity problem**, by parameterizing the time complexity as a function of the *diameter* of the graph. The diameter of the graph is the largest diameter of its connected components. Our main result is an $O(\log D \log \log_{m/n} n)$ time connectivity algorithm for diameter-D graphs with m edges. Parameterizing as a function of D is standard, say, in the distributed computing literature [PRS16], [HHW18]. In fact, some previous MPC algorithms for connectivity in the applied communities have been *conjectured* to obtain $O(\log D)$ time [RMCS13]; alas, we show that the algorithm of [RMCS13] has a lower bound of $\Omega(\log n)$ time (see the full version [ASS+18]).

Our algorithms exhibit a tradeoff between the total amount of memory available and the number of rounds of computation needed. For example, if the total space is $\Omega(n^{1+\gamma'})$ for some constant $\gamma' > 0$, then our algorithms run in $O(\log D)$ rounds only.

A. The MPC model

Before stating our full results, we briefly recall the MPC model [BKS13]. A detailed discussion appears in Section IV, along with some core primitives implementable in the MPC model.

Definition I.1 ($(\gamma, \delta) -$ MPC model)**.** *Fix parameters $\gamma \geq 0, \delta > 0$, and suppose $N \geq 1$ is the input size. There are $p \geq 1$ machines (processors) each with local memory size $s = \Theta(N^\delta)$, such that $p \cdot s = O(N^{1+\gamma})$. The space size is measured by words, each of $\Theta(\log(s \cdot p))$ bits. The input is distributed on the local memory of $\Theta(N/s)$ input machines. The computation proceeds in rounds. In each round, each machine performs computation on the data in its local memory, and sends messages to other machines at the end of the round. The total size of messages sent or received by a machine in a round is bounded by s. In the next round, each machine only holds the received messages in its local memory. At the end of the computation, the output is distributed on the output machines. Input/output machines and other machines are identical except that input/output machine can hold a part of the input/output. The parallel time of an algorithm is the number of rounds needed to finish the computation.*

In this model, the space per machine is sublinear in N, and the total space is only an $O(N^\gamma)$ factor more than the input size N. In this paper, we consider the case when δ is an arbitrary constant in $(0, 1)$. Our results are for both the most restrictive case of $\gamma = 0$ (total space is linear in the input size), as well as $\gamma > 0$ (for which our algorithms are a bit faster). The model from Definition I.1 matches the model MPC(ϵ) from [BKS13] with $\epsilon = \gamma/(1 + \gamma - \delta)$ and the number of machines $p = O(N^{1+\gamma-\delta})$.

B. Our Results

While our main result is a $\sim \log D$ time connectivity MPC algorithm, our techniques extend to related graph problems, such as spanning forest, finding a DFS sequence, and exact/approximate minimum spanning forest. We also prove a lower bound showing that, achieving similar bounds for reachability in *directed graphs* would imply faster boolean matrix multiplication algorithms.

We now state our results formally. For all results below, consider an input graph $G = (V, E)$, with $n = |V|$, $N = |V| + |E|$, and D being the upper bound on the diameter of any connected component of G.

Connectivity: In the connectivity problem, the goal is to output the connected components of an input graph G, i.e. at the end of the computation, $\forall v \in V$, there is a unique tuple (x, y) with $x = v$ stored on an output machine, where y is called the color of v. Any two vertices u, v have the same color if and only if they are in the same connected component.

Theorem I.2 (Connectivity in MPC model)**.** *For any $\gamma \in [0, 2]$ and any constant $\delta \in (0, 1)$, there is a randomized $(\gamma, \delta) -$ MPC algorithm which outputs the connected components of the graph G in $O(\min(\log D \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}, \log n))$ parallel time. The success probability is at least 0.98. In addition, if the algorithm fails, then it returns* FAIL.

Notice that in the most restrictive case of $\gamma = 0$ and $m = n$, we obtain $O(\min(\log D \cdot \log \log n, \log n))$ time. When the total space is slightly larger, or the graph is slightly denser—i.e. $\gamma > c$ or $\log_n m > c$, where $c > 0$ is an arbitrarily small constant—then we obtain $O(\log D)$ time.

Remark I.3. *We note the concurrent and independent work of [ASW18], who also give a connectivity algorithm in the* MPC *model but with different guarantees. In particular,*

their runtime is parameterized as a function of λ, which is a lower bound on the spectral gap[1] of the connected components of G. For a graph G with n vertices and $m = \widetilde{O}(n)$ edges, their algorithm runs in $O(\log\log n + \log(1/\lambda))$ parallel time and uses $\widetilde{O}(n/\lambda^2)$ total space. In contrast, our algorithm has a runtime of $O(\log D \cdot \log\log_{N/n} n)$, where D is the largest diameter of a connected component of G, and $N = \Omega(m)$ is the total space available. To compare the two runtimes, we note that: 1) $D \leq O(\frac{\log n}{\lambda})$ for any undirected graph G; and 2) there exist sparse graphs G[2] with n vertices and $O(n)$ edges such that $\frac{1}{\lambda} \geq D \cdot n^{\Omega(1)}$ and $D \leq O(\log n)$. Thus, our results subsume [ASW18] in the case when total space is $N = n^{1+\Omega(1)}$, but are incomparable otherwise.

We note another concurrent and independent work of [ŁMW18], who also give a graph connectivity algorithm in the MPC *model and with different guarantees. 1. Our algorithm has $O(\log D \cdot \log\log_{N/n} n)$ parallel running time for general undirected graphs while the analysis of their algorithm can only achieve an $O(\log n)$ parallel time. 2. Their algorithm has an $O(\log\log n)$ parallel time for random graphs while the analysis of ours can only achieve an $O((\log\log n)^2)$ bound. However, we conjecture that our algorithm can also achieve $O(\log\log n)$ parallel time for such random graphs.*

Spanning forest problem: In the spanning forest problem, the goal is to output a subset of edges of an input graph G such that the output edges together with the vertices of G form a spanning forest of the graph G. In the rooted spanning forest problem, in addition to the edges of the spanning forest, we are also required to orient the edge from child to parent, so that the parent-child pairs form a rooted spanning forest of the input graph G.

Theorem I.4 (Spanning Forest, restatement of Theorem V.5)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$*MPC *algorithm which outputs the rooted spanning forest of the graph G in $O(\min(\log D \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}, \log n))$ parallel time. The success probability is at least 0.98. In addition, if the algorithm fails, then it returns* FAIL.

Our spanning forest algorithm can also output an approximation to the diameter, as follows.

Theorem I.5 (Diameter Estimator, restatement of Theorem V.6)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$* MPC *algorithm which outputs a diameter estimator D' of the input graph G in $O(\min(\log D \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}, \log n))$ parallel time such that $D \leq D' \leq D^{O(\log(1/\gamma'))}$, where $\gamma' = \frac{\log(N^{1+\gamma}/n)}{\log n}$.*

[1]The spectral gap of a graph G is the second smallest eigenvalue of the normalized Laplacian of G.

[2]We can construct G as the following: a bridge connects two 3-regular expanders where each expander has $n/2$ vertices.

The success probability is at least 0.98. In addition, if the algorithm fails, then it returns FAIL.

Depth-First-Search sequence: If the input graph G is a tree, then we are able to output a Depth-First-Search sequence of that tree in $O(\log D) + T$ parallel time, where T is parallel time to compute a rooted tree (see Theorem I.4 for our upper bound of T) for G.

Theorem I.6 (DFS Sequence of a Tree, restatement of Theorem V.7)**.** *Suppose the graph G is a tree. For any $\gamma \in [\beta, 2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$* MPC *algorithm that outputs a Depth-First-Search sequence for the input graph G in $O(\min(\log D \cdot \log(1/\gamma), \log n))$ parallel time, where $\beta = \Theta(\log\log n/\log n)$. The success probability is at least 0.98. In addition, if the algorithm fails, then it returns* FAIL.

Applications of DFS sequence of a tree include lowest common ancestor, tree distance oracle, the size of every subtree, and others.

Minimum Spanning Forest: In the minimum spanning forest problem, the goal is to compute the minimum spanning forest of a weighted graph G.

Theorem I.7 (Minimum Spanning Forest, restatement of Theorem VI.2)**.** *Consider a weighted graph G with weights $w : E \to \mathbb{Z}$ such that $\forall e \in E, |w(e)| \leq \text{poly}(n)$. For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$*MPC *algorithm which outputs a minimum spanning forest of G in $O(\min(\log D_{MSF} \cdot \log(\frac{\log n}{1+\gamma\log n}), \log n) \cdot \frac{\log n}{1+\gamma\log n})$ parallel time, where D_{MSF} is the diameter (with respect to the number of edges/hops) of a minimum spanning forest of G. The success probability is at least 0.98. In addition, if the algorithm fails, then it returns* FAIL.

We note that we require the bounded weights condition merely to ensure that each weight is described by one word.

Theorem I.8 (Approximate Minimum Spanning Forest, restatement of Theorem VI.3)**.** *Consider a weighted graph G with weights $w : E \to \mathbb{Z}_{\geq 0}$ such that $\forall e \in E, |w(e)| \leq \text{poly}(n)$. For any $\epsilon \in (0,1)$, $\gamma \in [\beta, 2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$* MPC *algorithm which can output a $(1+\epsilon)$ approximate minimum spanning forest for G in $O(\min(\log D_{MSF} \cdot \log(\frac{\log n}{\log(N^{1+\gamma}/(\epsilon^{-1} n \log n))}), \log n))$ parallel time, where $\beta = \Theta(\log(\epsilon^{-1}\log n)/\log n)$, and D_{MSF} is the diameter (with respect to the number of edges/hops) of a minimum spanning forest of G. The success probability is at least 0.98. In addition, if the algorithm fails, then it returns* FAIL.

Theorem I.9 (Bottleneck Spanning Forest, restatement of Theorem VI.4)**.** *Consider a weighted graph G with weights $w : E \to \mathbb{Z}$ such that $\forall e \in E, |w(e)| \leq \text{poly}(n)$. For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$* MPC *algorithm which can output a*

bottleneck spanning forest for G in $O(\min(\log D_{MSF} \cdot \log(\frac{\log n}{1+\gamma \log n}), \log n) \cdot \log(\frac{\log n}{1+\gamma \log n}))$ parallel time, where D_{MSF} is the diameter (with respect to the number of edges/hops) of a minimum spanning forest of G. The success probability is at least 0.98. In addition, if the algorithm fails, then it returns FAIL.

Conditional hardness for directed reachability. We also consider the reachability question in the directed graphs, for which we show similar to the above results are unlikely. In particular, we show that if there is a fully scalable multi-query directed reachability $(0, \delta) - \text{MPC}$ algorithm with $n^{o(1)}$ parallel time and polynomial local running time, then we can compute the Boolean Matrix Multiplication in $n^{2+\epsilon+o(1)}$ time for arbitrarily small constant $\epsilon > 0$. We note that the equivalent problem for *undirected graphs* can be solved in $O(\log D \log \log n)$ parallel time via Theorem I.2.

Theorem I.10 (Directed Reachability vs. Boolean Matrix Multiplication, restatement of Theorem VII.1)**.** *Consider a directed graph $G = (V, E)$. If there is a polynomial local running time, fully scalable $(\gamma, \delta)-\text{MPC}$ algorithm that can answer $|V|+|E|$ pairs of reachability queries simultaneously for G in $O(|V|^{\alpha})$ parallel time, then there is a sequential algorithm which can compute the multiplication of two $n \times n$ boolean matrices in $O(n^2 \cdot n^{2\gamma+\alpha+\epsilon})$ time, where $\epsilon > 0$ is a constant which can be arbitrarily small.*

C. Our Techniques

In this section, we give an overview of the various techniques that we use in our algorithms. More details, as well as some of the low level details of the implementation in the MPC model, are defered to later sections.

Before getting into our techniques, we mention two standard tools to help us build our MPC subroutines. The first one is sorting: while in the PRAM model it takes $\sim \log N$ parallel time, sorting takes only constant parallel time in the MPC model [Goo99], [GSZ11]. The second tool is indexing/predecessor search [GSZ11], which also has a constant parallel time in MPC model. Furthermore, these two tools are fully scalable, and hence all the subroutines built on these two tools are also fully scalable. See Section IV for how to use these two tools to implement the MPC operations needed for our algorithms.

Graph Connectivity: A natural approach to the graph connectivity problem is via the classic primitive of contracting to leaders: select a number of leader verteces, and contract every vertex (or most vertices) to a leader from its connected component (this is usually implemented by labeling the vertex by the corresponding leader). Indeed, many previous works (see e.g. [KSV10], [RMCS13], [KLM+14]) are based on this approach. There are two general questions to address in this approach: 1) how to choose leader vertices, and 2) how to label each vertex by its leader. For example, the algorithm in [KSV10] randomly chooses half of the vertices as leaders, and then contracts each non-leader vertex to one of its neighbor leader vertex. Thus, in each round of their algorithm, the number of vertices drops by a constant fraction. At the same time, half of the vertices are leaders, and hence their algorithm still needs at least $\Omega(\log n)$ rounds to contract all the vertices to one leader. Note that a constant fraction of leaders is needed to ensure that there is a constant fraction of non-leader vertices who are adjacent to at least one leader vertex and hence are contracted. This leader selection method appears optimal for some graphs, e.g. path graphs.

To improve the runtime to $\ll \log n$, one would have to choose a much smaller fraction of the vertices to be leaders. Indeed, for a graph where every vertex has a large degree, say at least $d \gg \log n$, we can choose fewer leaders: namely, we can choose each vertex to be a leader with probability $p = \Theta((\log n)/d)$. Then the number of leaders will be about $\widetilde{O}(n/d)$, while each non-leader vertex has at least one leader neighbor with high probability. After contracting non-leader vertices to leader vertices, the number of remaining vertices is only a $1/d$ fraction of original number of vertices.

By the above discussion, the goal would now be to modify our input graph G so that every vertex has a uniformly large degree, without affecting the connectivity of the graph. An obvious such modification is to add edges between pairs of vertices that are already in the same connected component. In particular, if a vertex v learns of a large number of vertices which are in the same connected component as v, then we can add edges between v and those vertices to increase the degree of v. A naïve way to implement the latter is via broadcasting: each vertex v first initializes a set S_v which contains all the neighbors of v, and then, in each *round*, every vertex v updates the set S_v by adding the union of the sets S_u over all neighbors u of v (old and new). This approach takes log-diameter number of rounds, and each vertex learns all vertices which are in the same connected component at the end of the procedure. However, in a single round, the total communication needed may be as huge as $\Omega(n^3)$ since each of n vertices may have $\Omega(n)$ neighbors, each with a set of size $\Omega(n)$.

Since our goal of each vertex v is to learn only d vertices in the same component (not necessarily the entire component), we can therefore implement a "truncated" version of the above broadcasting procedure:

1) If S_v already had size d, then we do not need any further operation for S_v.
2) If u is in S_v, and S_u already has d vertices, then we can just put all the elements from S_u into S_v and thus S_v becomes of size d.
3) If $|S_v| < d$, and for every $u \in S_v$, the set S_u is also smaller than d, then we can implement one step of the broadcasting — add the union of S_u's, for all neighbors $u \in S_v$, to S_v.

In the above procedure, if the number of vertices in S_v is smaller than d after the i^{th} round, then we expect S_v to contain all the vertices whose distance to v is at most 2^i.

Thus, the above procedure also takes at most log-diameter rounds. Furthermore, the total communication needed is at most $O(n \cdot d^2)$.

Our full graph connectivity algorithm implements the above "truncated broadcasting" procedure iteratively, for values d that follow a certain "schedule", depending on the available space. At the beginning of the algorithm, we have an n vertex graph G with diameter D, and a total of $\Omega(m)$ space. The algorithm proceeds in phases, where each phase takes $O(\log D)$ rounds of communication. In the first phase, the starting number of vertices is $n_1 = n$. We implement a truncated broadcasting procedure where the target degree d is $d_1 = (m/n_1)^{1/2}$, using $O(\log D)$ rounds and $O(m)$ total space. Then we can randomly select $\widetilde{O}(n_1/d_1)$ leaders, and contract all the non-leader vertices to leader vertices. At the end of the first phase, the total number of remaining vertices is at most $n_2 = \widetilde{O}(n_1/d_1) = \widetilde{O}(n_1^{1.5}/m^{0.5})$. In general, suppose, at the beginning of the i^{th} phase, the number of remaining vertices is n_i. Then we use the truncated broadcasting procedure for value d set to $d_i = (m/n_i)^{1/2}$, thus making each vertex have degree at least $d_i = (m/n_i)^{1/2}$ in $O(\log D)$ number of communication rounds and $O(m)$ total space. Then we choose $\widetilde{O}(n_i/d_i)$ leaders, and, after contracting non-leaders, the number n_{i+1} of remaining vertices is at most $\widetilde{O}(n_i^{1.5}/m^{0.5})$. Let us look at the progress of the value d_i. We have that $d_{i+1} = \widetilde{\Omega}((m/n_{i+1})^{1/2}) = \widetilde{\Omega}((m^{1.5}/n_i^{1.5})^{1/2}) = \widetilde{\Omega}(d_i^{1.5})$. Thus, we are making double exponential progress on d_i, which implies that the total number of phases needed is at most $O(\log \log_{m/n} n)$, and the total parallel time is thus $O(\log D \cdot \log \log_{m/n} n)$.

This technique of double-exponential progress is more general and extends to other problems beyond connectivity. In particular, for a problem, suppose its size is characterized by a parameter n (not necessarily the input size—e.g. in connectivity problem, n is the number of vertices). When n is a constant, the problem can be solved in $O(1)$ parallel time. If there is a procedure that uses total space $\Theta(m)$ to reduce the problem size to at most n/k for $k = (m/n)^c, c = \Omega(1)$, then we can repeat the procedure $O(\log \log_{m/n} n)$ times to solve the overall problem. In particular, after repeating the procedure i times, the problem size is $n_i \le n_{i-1}/(m/n_{i-1})^c \le n \cdot (n/m)^{(1+c)^i - 1}$. We call this technique *double-exponential speed problem size reduction*.

Remark I.11. *For any problem characterized by a size parameter n, if we can use parallel time T and total space $\Theta(m)$ to reduce the problem size such that the reduced problem size is n/k for $k = (m/n)^{\Omega(1)}$, then we can solve the problem in $O(m)$ total space and $O(T \cdot \log \log_{m/n} n)$ parallel time.*

Spanning Forest and Diameter Estimator: Extending a connectivity algorithm to a spanning forest algorithm is usually straightforward. For example, in [KSV10], they only contract a non-leader vertex to an adjacent leader vertex, thus their algorithm can also give a spanning forest, using the contracted edges. Here however, extending our connectivity algorithm to a spanning forest algorithm requires several new ideas. In our connectivity algorithm, because of the added edges, we only ensure that when a vertex u is contracted to a vertex v, u and v must be in the same connected component; but u and v may not be adjacent in the original graph. Thus, we need to record more information to help us build a spanning forest.

We can represent a forest as a collection of parent pointers $\mathrm{par}(v)$, one for each vertex $v \in V$. If v is a root in the forest, then we let $\mathrm{par}(v) = v$. We use $\mathrm{dep}_{\mathrm{par}}(v)$ to denote the depth of v in the forest, i.e. $\mathrm{dep}_{\mathrm{par}}(v)$ is the distance from v to its root. Let $\mathrm{dist}_G(u, v)$ denote the distance between two vertices u and v in a graph G.

Our connectivity algorithm uses the "neighbor increment" procedure described above. We observed that if the set S_v has fewer than d vertices after the i^{th} round, then S_v should contain all the vertices with distance at most 2^i to v. This motivates us to maintain a shortest path tree for S_v, with root v. In the i^{th} round, if we need to update S_v to be $\bigcup_{u \in S_v} S_u$, then we can update the shortest path tree of S_v in the following way:

1) For each $x \in S_u$ for some $u \in S_v$, we can create a tuple (x, u).
2) Then, for each $x \in \left(\bigcup_{u \in S_v} S_u\right) \setminus S_v$, we can sort all the tuples $(x, u_1), (x, u_2), \cdots, (x, u_k)$ such that u_1 minimizes $\min_{u \in S_v} \mathrm{dist}_G(v, u) + \mathrm{dist}_G(u, x)$. Since u is in S_v, x is in S_u, it is easy to get the value of $\mathrm{dist}_G(v, u), \mathrm{dist}_G(u, x)$ by the information of shortest path tree for S_v and S_u. Then we set the new parent of x in the shortest path tree for S_v to be the parent of x in the shortest path tree for S_{u_1}.

Since S_v before the update contains all the vertices which have distance to v at most 2^{i-1}, the union of the shortest path from x to u_1 and the shortest path from u_1 to v must be the shortest path from x to v. Then by induction, we can show that the parent of x in the shortest path tree for S_{u_1} is also the parent of x in the shortest path tree for updated S_v. Thus, this modified "neighbor increment" procedure can find n local shortest path trees where there is a tree with root v for each vertex v. Furthermore, the procedure still takes $O(\log D)$ rounds. And we can still use $O(nd^2)$ total space to make each shortest path tree have size at least d. Next, we show how to use these n local shortest path trees to construct a forest with the roots in the forest being the leaders.

As discussed in the connectivity algorithm, if every local shortest path tree has size at least d, we can choose each vertex as a leader with probability $p = \Theta((\log n)/d)$ and then every tree will contain at least one leader with high probability. Let L be the set of sampled leaders, and let $\mathrm{dist}_G(v, L)$ be defined as $\min_{u \in L} \mathrm{dist}_G(v, u)$. Let v be a non-leader vertex, i.e. $v \in V \setminus L$. According to the shortest

path tree for S_v [3], since $L \cap S_v \neq \emptyset$, we can find a child u of the root v such that $\mathrm{dist}_G(v, L) > \mathrm{dist}_G(u, L)$; in this case we set $\mathrm{par}(v) = u$. For vertex $v \in L$, we can set $\mathrm{par}(v) = v$. We can see now that par denotes a rooted forest where the roots are sampled leaders. Furthermore, since $\forall v \notin L$, $(v, \mathrm{par}(v))$ is from the shortest path tree for S_v, we know that v and $\mathrm{par}(v)$ are adjacent in the original graph G. After doing the above for all nodes v, the forest denoted by the resulting vector par must be a subgraph of the spanning forest of G. We then apply the standard doubling algorithm to contract all the vertices to their leaders (roots), in $O(\log D)$ rounds. Therefore, the problem is reduced to finding a spanning forest in the contracted graph. The number of vertices remaining in the contracted graph is at most $\widetilde{O}(n/d)$, where $d = (m/n)^{\Theta(1)}$. By Remark I.11, we can output a spanning forest in $O(\log D \cdot \log \log_{m/n} n)$ parallel time.

Although the above algorithm can output the edges of a spanning forest, it cannot output a rooted spanning forest. To output a rooted spanning forest, we follow a top-down construction. Suppose now we have a rooted spanning forest of the contracted graph. Since we have all the information of how vertices were contracted, we know the contraction trees in the original graph. To merge these contraction trees into the rooted spanning forest of the contracted graph, we only need to change the root of each contraction tree to a proper vertex in that tree. This changing root operation can be implemented by the doubling algorithm via a divide-and-conquer approach.

Since the spanning forest algorithm needs $O(\log \log_{m/n} n)$ phases to contract all vertices to a single vertex, the total parallel time to compute a rooted spanning forest is $O(\log D \cdot \log \log_{m/n} n)$. Furthermore, the depth of the rooted spanning forest will be at most $O(D^{O(\log \log_{m/n} n)})$. Thus, we can use the doubling algorithm to calculate the depth of the tree, and output this depth as an estimator of the diameter of the input graph.

Depth-First-Search Sequence: Here, when the input graph G is a tree, our goal is to output a DFS sequence for this tree. Once we have this sequence, it is easy to output a rooted tree. Thus, computing a DFS sequence is at least as hard as computing a rooted tree, and all the previous algorithms need $\Omega(\log n)$ parallel time to do so.

First of all, we use our spanning forest algorithm to compute a rooted tree, reducing the problem to computing a DFS sequence for a rooted tree. The idea is motivated by TeraSort [O'M08]. If the size of the tree is small enough such that it can be handled by a single machine, then we can just use a single machine to generate its DFS sequence. Otherwise, our algorithm can be roughly described as follows. (Recall that δ is the parameter such that each machine has $\Theta(n^\delta)$ local memory.)

1) Sample $n^{\delta/2}$ leaves $l_1, l_2, \cdots, l_s$.
2) Determine the order of sampled leaves in the DFS sequence.
3) Compute the DFS sequence $\widetilde{A}$ of the tree which only consists of sampled leaves and their ancestors.
4) Compute the DFS sequence A_v of every root-v subtree which does not contain any sampled leaf.
5) Merge $\widetilde{A}$ and all the A_v.

The first and second steps go as follows. Since we only sample $n^{\delta/2}$ leaves, we can send them to a single machine. We generate queries for every pair of sampled leaves where each query (l_i, l_j) queries the lowest common ancestor of (l_i, l_j). We have n^δ such queries in total. Since the input tree is rooted, we can use a doubling algorithm to preprocess a data structure in $O(\log D)$ parallel time and answer all the queries simultaneously in $O(\log D)$ parallel time. Thus, we know the lowest common ancestor of any pair of sampled leaves, and we can store this all on a single machine. Based on the information of lowest common ancestors of each pair of sampled leaves, we are able to determine the order of the leaves.

For the third step, suppose the sampled leaves have order $l_1, l_2, \cdots, l_s$. Let v be the root of the tree. Then the DFS sequence $\widetilde{A}$ should be: the path from v to l_1, the path from l_1 to the lowest common ancestor of (l_1, l_2), the path from the lowest common ancestor of (l_1, l_2) to l_2, the path from l_2 to the lowest common ancestor of (l_2, l_3), ..., the path from l_s to v. We can find these paths simultaneously by a doubling algorithm together with a divide-and-conquer algorithm in $O(\log D)$ parallel time.

In the fourth step, we apply the procedure recursively. Suppose the total number of leaves in the tree is $q \leq n$. Since we randomly sampled $n^{\delta/2}$ number of leaves, with high probability, each subtree which does not contain a sampled leaf will have at most $O(q/n^{\delta/2})$ number of leaves. Thus, the depth of the recursion will be at most a constant, $O(1/\delta)$.

Minimum Spanning Forest and Bottleneck Spanning Forest: Recall that the input is a graph $G = (V, E = (e_1, e_2, \cdots, e_m))$ together with a weight function w on E. Without loss of generality, we only consider the case when all the weights of edges are different, i.e. $w(e_1) < w(e_2) < \cdots < w(e_m)$. Since the weights of edges are different, the minimum spanning forest of the graph is unique. By Kruskal's algorithm, the diameter of the graph induced by the first i edges for any $i \in [m]$ is at most the depth of the minimum spanning forest. Now, let us use D to denote the depth of the minimum spanning forest.

We first discuss the minimum spanning forest algorithm. A crucial observation of Kruskal's algorithm is: if we want to determine which edges in $e_i, e_{i+1}, \cdots, e_j$ are in the minimum spanning forest, we can always contract the first $i-1$ edges to obtain a graph G', run a minimum spanning forest algorithm on the contracted graph G', and observe whether an edge is included in the spanning forest of G'.

[3] The construction of S_v for spanning forest algorithm is slightly different from that described in the connectivity algorithm. S_v in spanning forest algorithm has a stronger property: $\forall u \in V \setminus S_v$, $\mathrm{dist}_G(u, v)$ must be at least $\mathrm{dist}_G(u', v)$ for any $u' \in S_v$.

Thus, if the total space is $\Theta(m^{1+\gamma})$, we can have m^γ copies of the graph, where the i^{th} copy contracts the first $(i-1)\cdot m^{1-\gamma}$ edges. Thus, we are able to divide the edges into m^γ groups where each group has $m^{1-\gamma}$ number of edges. We only need to solve the minimum spanning forest problem for each group. Then in the second phase, we can divide the edges into $m^{2\gamma}$ groups where each group has $m^{1-2\gamma}$ number of edges. Thus, the total number of phases needed is at most $O(1/\gamma)$. In each phase, we just need to run our connectivity algorithm to contract the graph.

For the approximate minimum spanning forest algorithm, we use a similar idea. If we want a $(1+\epsilon)$ approximation, then we round each weight to the closest value $(1+\epsilon)^i$ for some integer i. After rounding, there are only $O(1/\epsilon\cdot\log n)$ edge groups. Since our total space is at least $\Omega(m\log(n)/\epsilon)$, we can make $O(1/\epsilon\cdot\log n)$ copies of the graph. The i^{th} copy of the graph contracts all the edges in group $1,2,\cdots,i-1$. Then, we only need to run our spanning forest algorithm on each copy to determine which edges should be chosen in each group.

Another application of our *double exponential speed problem size reduction* technique is bottleneck spanning forest. For the bottleneck spanning forest, suppose we have $\Theta(km)$ total space. We can have k copies of the graph where the i^{th} copy contracts the first $(i-1)\cdot m/k$ number of edges. We can determine the group of $O(m/k)$ edges which contains the bottleneck edge. Thus, we reduce the problem to $O(m/k)$. According to Remark I.11, the number of phases is at most $O(\log\log_k m)$, and each phase needs T parallel time, where T is the parallel time for spanning forest.

Directed Reachability vs. Boolean Matrix Multiplication: If there is a fully scalable multi-query directed reachability MPC algorithm with almost linear total space, we can simulate the algorithm in sequential model. Thus, it will imply a good sequential multi-query directed reachability algorithm which implies a good sequential Boolean Matrix Multiplication algorithm.

II. Notations

$[n]$ denotes the set $\{1,2,\cdots,n\}$. Let G be an undirected graph with vertex set V and edge set E. For $v\in V$, $\Gamma_G(v)$ denotes the set of neighbors of v in G, i.e. $\Gamma_G(v)=\{u\in V\mid (v,u)\in E\}$. For any $u,v\in V$, $\mathrm{dist}_G(u,v)$ denotes the distance between u,v in graph G. If u,v are not in the same connected component, then $\mathrm{dist}_G(u,v)=\infty$. If u,v are in the same connected component, then $\mathrm{dist}_G(u,v)<\infty$. For $v\in V$, $\{u\in V\mid \mathrm{dist}_G(u,v)<\infty\}$ is the set of all the vertices in the same connected component as v. The diameter $\mathrm{diam}(G)$ of G is the largest diameter of its components, i.e. $\mathrm{diam}(G)=\max_{u,v\in V:\mathrm{dist}_G(u,v)<\infty}\mathrm{dist}_G(u,v)$.

III. Graph Connectivity

In this section, we describe a batch version of a graph connectivity algorithm which can be easily implemented in the MPC model.

Algorithm 1 Neighbor Increment Operation

procedure NeighborIncrement($m, G=(V,E)$)
  Initially, $n=|V|, E'=\emptyset$ and $S_v^{(0)}=\{v\}$ for all $v\in V$.
  for $v\in V, u\in\Gamma_G(v)$ **do** ▷ Initialize $S_v^{(0)}$ to be the set (or a subset) of direct neighbors of v
    If $|S_v^{(0)}|<\lceil(m/n)^{1/2}\rceil$, then let $S_v^{(0)}\leftarrow S_v^{(0)}\cup\{u\}$.
  end for
  $r\leftarrow 1$. ▷ r denotes the number of iterations.
  for true do
    for $v\in V$ **do**
      if $\exists u\in S_v^{(r-1)}, |S_u^{(r-1)}|\geq\lceil(m/n)^{1/2}\rceil$ **then**
        $S_v^{(r)}=S_u^{(r-1)}\cup\{v\}$.
        If $|S_v^{(r)}|>|S_u^{(r-1)}|$, then $S_v^{(r)}\leftarrow S_v^{(r)}\setminus\{u\}$.
      else
        $S_v^{(r)}=\bigcup_{u\in S_v^{(r-1)}}S_u^{(r-1)}$.
      end if
    end for
    if $\forall v\in V$, either $|S_v^{(r)}|\geq\lceil(m/n)^{1/2}\rceil$ or $|S_v^{(r)}|=|S_v^{(r-1)}|$ **then** Let $E'=E\cup\bigcup_{v\in V}\{(v,u)\mid v\in S_u^{(r)}$ or $u\in S_v^{(r)}, u\neq v\}$. **return** $G'=(V,E')$.
    else
      $r\leftarrow r+1$.
    end if
  end for
end procedure

A. Neighbor Increment Operation

In this section, we describe a procedure (see Algorithm 1) which can increase the number of neighbors of every vertex and preserve the connectivity at the same time. The input of the procedure is an undirected graph $G=(V,E)$ and a parameter m which is larger than $|V|$. The output is a graph $G'=(V,E')$ such that for each vertex v, either the connected component which contains v is a clique or v has at least $\lceil(m/|V|)^{1/2}\rceil-1$ neighbors. Furthermore, the total space of the procedure is $O(m)$, and the number of iterations is at most $\min(\lceil\log(\mathrm{diam}(G))\rceil,\lceil\log(m/n)\rceil)+1$.

B. Random Leader Selection

Given an undirected graph $G=(V,E)$, to design a connected component algorithm, a natural way is constantly contracting the vertices in the same component. One way to do the contraction is that we randomly choose some vertices as leaders, then contract non-leader vertices to the neighbor leader vertices. In this section, we show that if $\forall v\in V$, the number of neighbors of v is large enough, then we can just sample a small number of leaders such that for each non-leader vertex $v\in V$, there is at least one neighbor of v which is chosen as a leader. A more generalized statement is stated in the following lemma.

Lemma III.1. *Let V be a vertex set with n vertices. Let $0<\gamma\leq n, \delta\in(0,1)$. For each $v\in V$, let S_v be a subset of $V\setminus\{v\}$ with size at least $\gamma-1$. Let $l: V\to\{0,1\}$ be a random hash function such that $\forall v\in V, l(v)$ are i.i.d. Bernoulli*

random variables, i.e. $l(v) = \begin{cases} 1 & \text{with probability } p; \\ 0 & \text{otherwise.} \end{cases}$ *If* $p \geq \min((10\log(2n/\delta))/\gamma, 1)$, *then, with probability at least* $1-\delta$,

1) $\sum_{v\in V} l(v) \leq \frac{3}{2}pn$;
2) $\forall v \in V, \exists u \in S_v \cup \{v\}$ *such that* $l(u) = 1$.

If the number of neighbors of each vertex is not large, then we can still have a constant fraction of vertices which can contract to a leader.

Lemma III.2. *Let* V *be a vertex set with* n *vertices. Let* S_v *be a subset of* $V \setminus \{v\}$ *with size at least* 1. *Let* $l : V \to \{0,1\}$ *be a random hash function such that* $\forall v \in V, l(v)$ *are i.i.d. Bernoulli random variables, i.e.* $l(v) = \begin{cases} 1 & \text{with probability } \frac{1}{2}; \\ 0 & \text{otherwise.} \end{cases}$ *Let* $L = \{v \in V \mid l(v) = 1\} \cup \{v \in V \mid \forall u \in S_v \cup \{v\}, l(u) = 0\}$. $\mathbf{E}(L) \leq 0.75n$.

C. Contraction Operation

In this section, we introduce the contraction operation. Firstly, let us introduce the concept of the parent pointers which can define a rooted forest.

Definition III.3. *Given a set of vertices* V, *let* $\mathrm{par} : V \to V$ *satisfy that* $\forall v \in V, \exists i > 0$ *such that* $\mathrm{par}^{(i)}(v) = \mathrm{par}^{(i+1)}(v)$, *where* $\forall v \in V, j > 0, \mathrm{par}^{(j)}(v)$ *is defined as* $\mathrm{par}(\mathrm{par}^{(j-1)}(v))$, *and* $\mathrm{par}^{(0)}(v) = v$. *Then, we call such* par *a set of parent pointers on* V. *For* $v \in V$, *if* $\mathrm{par}(v) = v$, *then we say* v *is a root of* par. par *can have more than one root. The depth of* $v \in V$, $\mathrm{dep}_{\mathrm{par}}(v)$ *is the smallest* $i \in \mathbb{Z}_{\geq 0}$ *such that* $\mathrm{par}^{(i)}(v) = \mathrm{par}^{(i+1)}(v)$. *The root of* $v \in V$, $\mathrm{par}^{(\infty)}(v)$ *is defined as* $\mathrm{par}^{(\mathrm{dep}_{\mathrm{par}}(v))}(v)$. *The depth of* par, $\mathrm{dep}(\mathrm{par})$ *is defined as* $\max_{v\in V} \mathrm{dep}_{\mathrm{par}}(v)$.

It is easy to see that a set of parent pointers par on V formed a rooted forest on V. For a vertex $v \in V$, if $\mathrm{par}(v) = v$, then v is a root in the forest. Otherwise $\mathrm{par}(v)$ is the parent of v in the forest.

Now we focus on the parent pointers which can preserve the connectivity of the graph.

Definition III.4. *Given a graph* $G = (V, E)$ *and a set of parent pointers* par *on* V, *if* $\forall v \in V$, *we have* $\mathrm{dist}_G(v, \mathrm{par}(v)) < \infty$, *then* par *is compatible with* G.

It is easy to show the following fact:

Fact III.5. *Given a graph* $G = (V, E)$ *and a set of parent pointers* par *which is compatible with* G, *then* $\forall u, v \in V$ *with* $\mathrm{par}^{(\infty)}(u) = \mathrm{par}^{(\infty)}(v)$, *we have* $\mathrm{dist}_G(u, v) < \infty$.

Now, we describe a procedure (see Algorithm 2) which can contract vertices to reduce the number of vertices. The input of the procedure is an undirected graph $G = (V, E)$ and a set of parent pointers $\mathrm{par} : V \to V$, where par is compatible with G. The output of the procedure will be the root of each vertex in V and an undirected graph $G' = (V', E')$ which satisfies $V' = \{v \in V \mid \mathrm{par}(v) = v\}$, $E' = \{(u, v) \in V' \times V' \mid u \neq v, \exists (p, q) \in E, \mathrm{par}^{(\infty)}(p) = u, \mathrm{par}^{(\infty)}(q) = v\}$. Notice that V' only contains all the roots in the forest induced by par, and $|E'| \leq |E|$. Furthermore, the number of iterations is at most $\lceil \log \mathrm{dep}(\mathrm{par}) \rceil$, the total space is linear in the input size, and $\mathrm{diam}(G') \leq \mathrm{diam}(G)$.

Algorithm 2 Tree Contraction Operation

```
1:  procedure TreeContraction(G = (V, E), par : V → V)
2:      Initially, ∀v ∈ V, g^(0)(v) ← par(v). Let V' = ∅, E' = ∅.
3:      l ← 0.
4:      for ∃v ∈ V, par(g^(l)(v)) ≠ g^(l)(v) do
5:          l ← l + 1.
6:          For v ∈ V, compute g^(l)(v) = g^(l−1)(g^(l−1)(v)).
7:      end for
8:      r ← l.                          ▷ r denotes the number of iterations.
9:      For v ∈ V, if par(v) = v, let V' ← V' ∪ {v}.
10:     For (u, v) ∈ E, if g^(r)(u) ≠ g^(r)(v), let E' ← E' ∪
        {(g^(r)(u), g^(r)(v))}.
11:     return g^(r)(v) as par^(∞)(v) for v ∈ V, and G' =
        (V', E')
12: end procedure
```

D. Connectivity Algorithm

In this section, we described a batch algorithm for graph connectivity/connected components problem. The input is an undirected graph $G = (V, E)$, a space/rounds trade-off parameter m, and the rounds parameter $r \leq |V|$. The output is a function $\mathrm{col} : V \to V$ such that $\forall u, v \in V, \mathrm{dist}_G(u, v) < \infty \Leftrightarrow \mathrm{col}(u) = \mathrm{col}(v)$.

The algorithm is described in Algorithm 3. The following theorem shows the correctness of Algorithm 3.

Theorem III.6 (Correctness of Algorithm 3)**.** *Let* $G = (V, E)$ *be an undirected graph,* $m \geq 4|V|$, *and* $r \leq |V|$ *be the rounds parameter. If* Connectivity(G, m, r) *(Algorithm 3) does not output FAIL, then* $\forall u, v \in V$, *we have* $\mathrm{dist}_G(u, v) < \infty \Leftrightarrow \mathrm{col}(u) = \mathrm{col}(v)$.

Now let us consider the number of iterations of Algorithm 3 and the success probability.

Definition III.7 (Total iterations)**.** *Let* $G = (V, E)$ *be an undirected graph,* $\mathrm{poly}(n) \geq m > 4n$, *and* $r \leq n$ *be the rounds parameter where* n *is the number of vertices in* G. *The total number of iterations of* Connectivity(G, m, r) *(Algorithm 3) is defined as* $\sum_{i=1}^{r}(k_i + r'_i)$, *where* k_i *denotes the number of iterations of* NeighborIncrement(m, G_{i-1}) *(see line 10), and* r'_i *denotes the number of iterations of* TreeContraction(G''_i, par_i) *(see line 21).*

Theorem III.8 (Success probability and total iterations)**.** *Let* $G = (V, E)$ *be an undirected graph,* $\mathrm{poly}(n) \geq m > 4n$, *and* $r \leq n$ *be the rounds parameter where* $n = |V|$. *Let* $c > 0$ *be a sufficiently large constant. If* $r \geq c \log\log_{m/n}(n)$, *then with probability at least* 0.98, Connectivity(G, m, r)

Algorithm 3 Graph Connectivity

procedure CONNECTIVITY($G = (V, E), m, r$)
  Output: FAIL or col : $V \to V$.
  $n \leftarrow |V|$
  $\forall v \in V$, $h_0(v) \leftarrow$ null.
  $G_0 = (V_0, E_0) = G$, i.e. $V_0 = V, E_0 = E$.
  $n_0 = n$.
  for $i = 1 \to r$ **do**
    $\forall v \in V$, $h_i(v) \leftarrow$ null.
    //Neighbor Increment:
    $G'_i = (V'_i, E'_i) =$ NEIGHBORINCREMENT(m, G_{i-1}). ▷ Algorithm 1
    $V''_i = \{v \in V'_i \mid |\Gamma_{G'_i}(v)| \geq \lceil (m/n_{i-1})^{1/2} \rceil - 1\}$.
    $E''_i = \{(u, v) \in E_{i-1} \mid u \in V''_i, v \in V''_i\}$.
    $G''_i = (V''_i, E''_i)$.
    //Random Leader Selection:
    Set $\gamma_i = \lceil (m/n_{i-1})^{1/2} \rceil, p_i = \min((30 \log(n) + 100)/\gamma_i, 1/2)$.
    Let $l_i : V''_i \to \{0, 1\}$ be a random hash function such that $\forall v \in V''_i, l_i(v)$ are i.i.d. Bernoulli random variables, and $\Pr(l_i(v) = 1) = p_i$.
    Let $L_i = \{v \in V''_i \mid l_i(v) = 1\} \cup \{v \in V''_i \mid \forall u \in \Gamma_{G'_i}(v) \cup \{v\}, l_i(u) = 0\}$. ▷ L_i is a set of all the leaders
    $\forall v \in V''_i$ with $v \in L_i$, let $\mathrm{par}_i(v) = v$.
    $\forall v \in V''_i$ with $v \notin L_i$, let $\mathrm{par}_i(v) = \min_{u \in L_i \cap (\Gamma_{G'_i}(v) \cup \{v\})} u$. ▷ Non-leader finds a leader.
    //Vertices Contraction:
    $((V_i, E_i), g_i^{(r'_i)}) =$ TREECONTRACTION(G''_i, par_i). ▷ Algorithm 2
    $G_i = (V_i, E_i)$.
    $n_i = |V_i|$.
    For $v \in V'_i \setminus V''_i$, $h_i(v) \leftarrow \min_{u \in \Gamma_{G'_i}(v) \cup \{v\}} u$.
    For $v \in V''_i \setminus V_i$, $h_i(v) \leftarrow g_i^{(r'_i)}(v)$.
    For $v \in V$, if $h_{i-1}(v) \neq$ null, $h_i(v) \to h_{i-1}(v)$.
  end for
  If $n_r \neq 0$, **return** FAIL.
  $((\widehat{V}, \widehat{E}), \mathrm{col}) =$ TREECONTRACTION(G, h_r). ▷ Algorithm 2
  return col.
end procedure

(Algorithm 3) will not return FAIL. *Furthermore, with probability at least* 0.98, *the total number of iterations (see Definition III.7) of* CONNECTIVITY(G, m, r) *is* $O(\min(r \log(\mathrm{diam}(G)), \log n))$.

IV. THE MPC MODEL

In this section, let us introduce the computational model studied in this paper. Suppose we have p machines indexed from 1 to p each with memory size s words, where n is the number of words of the input and $p \cdot s = O(n^{1+\gamma}), s = \Theta(n^{\delta})$. Here $\delta \in (0, 1)$ is a constant, $\gamma \in \mathbb{R}_{\geq 0}$, and a word has $\Theta(\log(s \cdot p))$ bits. Thus, the total space in the system is only $O(n^{\gamma})$ factor more than the input size n, and each machine has local memory size sublinear in n. When $0 \leq \gamma \leq O(1/\log n)$, the total space is just linear in the input size. The computation proceeds in rounds. At the beginning of the computation, the input is distributed on the local memory of $\Theta(n/s)$ input machines. Input machines and other machines are identical except that input machine can hold a part of the input in its local memory at the beginning of the computation while each of other machines initially holds nothing. In each round, each machine performs computation on the data in its local memory, and sends messages to other machines (including the sender itself when it wants to keep the data) at the end of the round. Although any two machines can communicate directly in any round, the total size of messages (including the self-sent messages) sent or received of a machine in a round is bounded by s, its local memory size. In the next round, each machine only holds the received messages in its local memory. At the end of the computation, the output is distributed on the output machines. Output machines and other machines are identical except that output machine can hold a part of the output in its local memory at the end of the computation while each of other machines holds nothing. We call the above model $(\gamma, \delta) - \mathrm{MPC}$ model. The model is exactly the same as the model $\mathrm{MPC}(\epsilon)$ defined by [BKS13] with $\epsilon = \gamma/(1 + \gamma - \delta)$ and the number of machines $p = O(n^{1+\gamma-\delta})$. Since we care more about the total space used by the algorithm, we use (γ, δ) to characterize the model, while in [BKS13] they use parameter ϵ to characterize the repetition of the data. The main complexity measure is the number of rounds R required to solve the problem.

V. IMPLEMENTATIONS IN MPC MODEL

In this section, we show the theoretical guarantees of implementations of previous batch algorithms in MPC model.

A. Graph Connectivity

The following lemma shows the number of rounds needed to implement Algorithm 1 in MPC model.

Lemma V.1. *Let graph* $G = (V, E), n = |V|, N = |V| + |E|$ *and* $m = \Theta(N^{\gamma})$ *for some arbitrary* $\gamma \in [0, 2]$. NEIGHBORINCREMENT(m, G) *(Algorithm 1) can be implemented in* $(\gamma, \delta) - \mathrm{MPC}$ *model for any constant* $\delta \in (0, 1)$. *Furthermore, the parallel running time is* $O(r)$, *where* r *is the number of iterations (see line 6 of Algorithm 1) of* NEIGHBORINCREMENT(m, G).

As mentioned in Section III-A, the number of iterations of NEIGHBORINCREMENT(m, G) is at most $\min(\lceil \log(\mathrm{diam}(G)) \rceil, \lceil \log(m/n) \rceil) + 1$. Thus, the number of rounds needed to implement the procedure NEIGHBORINCREMENT(m, G) in MPC model is at most $O(\min(\lceil \log(\mathrm{diam}(G)) \rceil, \lceil \log(m/n) \rceil))$.

The following lemma shows the number of rounds needed to implement Algorithm 2 in MPC model.

Lemma V.2. *Let graph* $G = (V, E)$ *and* $\mathrm{par} : V \to V$ *be a set of parent points (see Definition III.3) on the vertex set* V. TREECONTRACTION(G, par) *(Algorithm 2) can be implemented in* $(0, \delta) - \mathrm{MPC}$ *model for any constant* $\delta \in$

$(0,1)$. *Furthermore, the parallel running time is* $O(r)$, *where* r *is the number of iterations (see line 8 of Algorithm 2) of* TREECONTRACTION(G, par).

As mentioned in Section III-C, the number of iterations of TREECONTRACTION(G, par) is $O(\log \text{dep}(\text{par}))$ which implies that it needs $O(\log \text{dep}(\text{par}))$ rounds to implement in MPC model.

By following Lemma V.1, Lemma V.2 and Theorem III.8, we can get the following theorem which shows the number of rounds needed to implement Algorithm 3 in MPC model.

Theorem V.3. *Let graph* $G = (V, E), n = |V|, N = |V| + |E|$ *and* $m = \Theta(N^\gamma)$ *for some arbitrary* $\gamma \in [0, 2]$. *Let* $r > 0$ *be a round parameter.* CONNECTIVITY(G, m, r) *(Algorithm 3) can be implemented in* $(\gamma, \delta)-$MPC *model for any constant* $\delta \in (0, 1)$. *Furthermore, the parallel running time is* $O(R)$, *where* R *is the total number of iterations (see Definition III.7) of* CONNECTIVITY(G, m, r).

Here, we are able to conclude the following theorem for graph connectivity problem.

Theorem V.4. *For any* $\gamma \in [0, 2]$ *and any constant* $\delta \in (0, 1)$, *there is a randomized* $(\gamma, \delta) -$ MPC *algorithm (see Algorithm 3) which can output the connected components for any graph* $G = (V, E)$ *in* $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ *parallel time, where* D *is the diameter of* G, $n = |V|$, $N = |V| + |E|$ *and* $\gamma' = (1 + \gamma) \log_n \frac{2N}{n^{1/(1+\gamma)}}$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

B. Spanning Forest Algorithm

Our spanning forest algorithm was outlined a bit in Section I. In the following, we just state our results. Please see the full version [ASS+18] for details.

Theorem V.5. *For any* $\gamma \in [0, 2]$ *and any constant* $\delta \in (0, 1)$, *there is a randomized* $(\gamma, \delta) -$MPC *algorithm which can output the rooted spanning forest for any graph* $G = (V, E)$ *in* $O(\min(\log D \cdot \log \frac{1}{\gamma}, \log n))$ *parallel time, where* D *is the diameter of* G, $n = |V|$, $N = |V| + |E|$ *and* $\gamma' = (1 + \gamma) \log_n \frac{2N}{n^{1/(1+\gamma)}}$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

A byproduct is an estimator of the diameter of the graph.

Theorem V.6. *For any* $\gamma \in [0, 2]$ *and any constant* $\delta \in (0, 1)$, *there is a randomized* $(\gamma, \delta) -$MPC *algorithm which can output an diameter estimator* D' *for any graph* $G = (V, E)$ *in* $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ *parallel time such that* $D \le D' \le D^{O(\log(1/\gamma'))}$, *where* D *is the diameter of* G, $n = |V|$, $N = |V| + |E|$ *and* $\gamma' = (1 + \gamma) \log_n \frac{2N}{n^{1/(1+\gamma)}}$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

C. DFS Sequence

As stated in Section I, we can output a DFS sequence for a tree graph in MPC model. In the following, we just state our results. Please see the full version [ASS+18] for details.

Theorem V.7. *For any* $\gamma \in [\beta, 2]$ *and any constant* $\delta \in (0, 1)$, *there is a randomized* $(\gamma, \delta) -$MPC *algorithm which can output a Depth-First-Search sequence for any tree graph* $G = (V, E)$ *in* $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ *parallel time, where* $n = |V|$, $\beta = \Theta(\log \log n / \log n)$, D *is the diameter of* G, *and* $\gamma' = \gamma + \Theta(1/\log n)$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

VI. MINIMUM SPANNING FOREST

In this section, we discuss how to apply our connectivity/spanning forest algorithm to the Minimum Spanning Forest (MSF) and Bottleneck Spanning Forest (BSF) problem.

The input of MSF/BSF problem is an undirected graph $G = (V, E)$ together with a weight function $w : E \to \mathbb{Z}$, where E contains m edges $e_1, e_2, \cdots, e_m$ with $w(e_1) \le w(e_2) \le \cdots \le w(e_m)$. The goal of MSF is to output a spanning forest such that the sum of weights of the edges in the forest is minimized. The goal of BSF is to output a spanning forest such that the maximum weight of the edges in the forest is minimized. D is the diameter (with respect to the number of hops/edges) of the minimum spanning forest. If there are multiple choices of the MSF, then let D be the minimum diameter among all the MSFs.

Lemma VI.1. *Given a graph* $G = (V, E)$ *for* $E = \{e_1, e_2, \cdots, e_m\}$ *together with a weight function* w *which satisfies* $w(e_1) \le w(e_2) \le \cdots \le w(e_m)$, $\forall 1 \le i < j \le m$, *an edge* e *from* $\{e_i, e_{i+1}, \cdots, e_j\}$ *is in the minimum spanning forest of* G *if and only if* e' *from* $\{e'_i, e'_{i+1}, \cdots, e'_j\}$ *is in the minimum spanning forest of* G', *where the vertices of* G' *is obtained by contracting all the edges* $e_1, e_2, \cdots, e_{i-1}$ *of* G, *and* $e', e'_i, \cdots, e'_j$ *are the edges (or vertices) in* G' *which corresponds to edges* $e, e_i, \cdots, e_j$ *before contraction.*

A natural way to apply Lemma VI.1 to parallel minimum spanning forest algorithm is that we can divide the edges into several groups, and recursively solve the minimum spanning forest for each group of edges. More precisely, suppose we have total space $\Theta(km)$, we can divide E into k groups $E_1, E_2, \cdots, E_k$, where $E_i = \{e_{(i-1)\cdot m/k+1}, e_{(i-1)\cdot m/k+2}, \cdots, e_{i\cdot m/k}\}$. We can compute graph $G_1, G_2, \cdots, G_k$ where the vertices of G_i is obtained by contracting all the edges from e_1 to $e_{(i-1)\cdot m/k}$, the edges of G_i are corresponding to the edges in E_i. Then by Lemma VI.1, we can obtain the whole minimum spanning forest by solving these k size $O(m/k)$ minimum spanning forest problems. For each sub-problem, we can assign it $\Theta(m)$ working space, so each sub-problem has $\Theta(k)$ factor more space. We can recursively apply the above argument.

Theorem VI.2. *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$MPC algorithm which can output a minimum spanning forest for any weighted graph $G=(V,E)$ with weights $w: E \to \mathbb{Z}$ in $O(\min(\log D \cdot \log(1/\gamma'), \log n)\cdot 1/\gamma')$ parallel time, where $n=|V|$, $\forall e \in E, |w(e)| \leq \mathrm{poly}(n)$, D is the diameter of a minimum spanning forest of G, and $\gamma' = \gamma/2 + \Theta(1/\log n)$. The success probability is at least 0.98. In addition, if the algorithm fails, then it will return* FAIL.

In the following, we show that Lemma VI.1 can also be applied in approximate minimum spanning forest problem.

Theorem VI.3. *For any $\gamma \in [\beta,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$MPC algorithm which can output a $(1+\epsilon)$ approximate minimum spanning forest for any weighted graph $G=(V,E)$ with weights $w: E \to \mathbb{Z}_{\geq 0}$ in $O(\min(\log D\cdot\log(1/\gamma'), \log n))$ parallel time, where $n=|V|$, $N=|V|+|E|$, $\beta = \Theta(\log(\epsilon^{-1}\log n)/\log n)$, $\forall e \in E, |w(e)| \leq \mathrm{poly}(n)$, D is the diameter of a minimum spanning forest of G, and $\gamma' = (1+\gamma-\beta)\log_n \frac{2N}{n^{1/(1+\gamma-\beta)}}$. The success probability is at least 0.98. In addition, if the algorithm fails, then it will return* FAIL.

In the following, we show that if we only need to find the largest edge in the minimum spanning tree, then we are able to get a better parallel time. It is an another application of our *double exponential speed problem size reduction* technique.

Theorem VI.4. *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$MPC algorithm which can output a bottleneck spanning forest for any weighted graph $G=(V,E)$ with weights $w: E \to \mathbb{Z}$ in $O(\min(\log D \cdot \log(1/\gamma'), \log n)\cdot \log(1/\gamma'))$ parallel time, where $n=|V|$, $\forall e \in E, |w(e)| \leq \mathrm{poly}(n)$, D is the diameter of a minimum spanning forest of G, and $\gamma' = \gamma/2 + \Theta(1/\log n)$. The success probability is at least 0.98. In addition, if the algorithm fails, then it will return* FAIL.

VII. Directed Reachability vs. Boolean Matrix Multiplication

Consider the multi-query directed graph reachability problem. In this problem, we are given a directed graph $G=(V,E)$ together with $|V|+|E|$ queries where each query queries the reachability from vertex u to vertex v. The goal is to answer all these queries. A similar problem in the undirected graph is called multi-query undirected graph connectivity problem. According to Theorem V.4, there is a polynomial local running time fully scalable $\sim \log D$ parallel time $(0,\delta)-$MPC algorithm for multi-query undirected graph connectivity problem. Here polynomial local running time means that there is a constant $c>0$ (independent from δ) such that every machine in one round can only have $O((n^\delta)^c)$ local computation. For multi-query directed graph reachability problem, we show that if there is a polynomial local running time fully scalable $(\gamma,\delta)-$MPC algorithm which can solve multi-query reachability problem in $O(n^\alpha)$ parallel time, then we can solve all-pair directed graph reachability problem in $O(n^2\cdot n^{2\gamma+\alpha+\epsilon})$ sequential running time for any arbitrarily small constant $\epsilon>0$.

Theorem VII.1. *If there is a polynomial local running time fully scalable $(\gamma,\delta)-$MPC algorithm which can answer $|V|+|E|$ pairs of reachability queries simultaneously for any directed graph $G=(V,E)$ in $O(|V|^\alpha)$ parallel time, then there is a sequential algorithm which can compute the multiplication of two $n\times n$ boolean matrices in $O(n^2\cdot n^{2\gamma+\alpha+\epsilon})$ time, where constant $\epsilon \in \mathbb{R}_{>0}$ can be arbitrarily small.*

Acknowledgment

We thank Paul Beame, Lijie Chen, Xi Chen, Mika Göös, Bernhard Haeupler, Rasmus Kyng, Zhengyang Liu, Yin Tat Lee, Jelani Nelson, Eric Price, Aviad Rubinstein, Timothy Sun, Ola Svensson, Erik Waingarten, Omri Weinstein, David P. Woodruff, Mihalis Yannakakis, and Huacheng Yu for helpful comments.

Alexandr Andoni and Peilin Zhong are supported in part by Simons Foundation (#491119 to Alexandr Andoni), NSF (CCF-1617955, CCF-1740833), and Google Research Award. Peilin Zhong is also supported in part by NSF grant (CCF-1703925). Part of the work is done while Zhao Song is visiting IBM Almaden (hosted by David P. Woodruff) and Harvard University (hosted by Jelani Nelson). Clifford Stein is supported in part by NSF grants CCF-1421161 and CCF-1714818. Part of the work is done while Peilin Zhong is visiting IBM Almaden (hosted by David P. Woodruff).

References

[ABB+17] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *arXiv preprint*. http://arxiv.org/pdf/1711.03076, 2017.

[AG18] 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.

[AK17] Sepehr Assadi and Sanjeev Khanna. Randomized composable coresets for matching and vertex cover. In *SPAA*. https://arxiv.org/pdf/1705.08242, 2017.

[ANOY14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the Symposium on Theory of Computing (STOC)*. http://arxiv.org/pdf/1401.0042, 2014.

[ASS+18] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *FOCS*. https://arxiv.org/pdf/1805.03055, 2018.

[ASW18] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In *ArXiv preprint*. https://arxiv.org/pdf/1805.02974, 2018.

[BH89] Paul Beame and Johan Håstad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM*, 36(3):643–670, 1989.

[BKS13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 273–284. ACM, 2013.

[CLM+18] Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the Symposium on Theory of Computing (STOC)*. https://arxiv.org/pdf/1707.03478, 2018.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[EIM11] Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using MapReduce. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 681–689. ACM, 2011.

[FMS+10] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms*, 6(4), 2010. Previously in SODA'08.

[Goo99] Michael T Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.

[GSZ11] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *ISAAC*, volume 7074, pages 374–383. Springer, 2011.

[HHW18] Bernhard Haeupler, D. Ellis Hershkowitz, and David Wajc. Round- and message-optimal distributed graph algorithms. In *arXiv preprint*. http://arxiv.org/pdf/1801.05127, 2018.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[IMS17] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 798–811, 2017.

[KLM+14] Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

[ŁMW18] Jakub Łącki, Vahab Mirrokni, and Michał Włodarczyk. Connected components at scale via local contractions. *arXiv preprint arXiv:1807.10727*, 2018.

[McG09] Andrew McGregor. Graph mining on streams. *Encyclopedia of Database Systems*, pages 1271–1275, 2009.

[O'M08] Owen O'Malley. Terabyte sort on apache hadoop. *Yahoo Tech. Rep*, 2008.

[PRS16] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and mst in large graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 429–438. ACM, 2016.

[RMCS13] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 50–61. IEEE, 2013.

[RVW16] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits: (on lower bounds for modern parallel computation). In Christian Scheideler and Seth Gilbert, editors, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 1–12. ACM, 2016.

[SV82] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[Whi12] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.