New Distributed Algorithms in Almost Mixing Time via Transformations from Parallel Algorithms

Mohsen Ghaffari
ETH Zurich, Switzerland
ghaffari@inf.ethz.ch

Jason Li
Carnegie Mellon University, USA
`http://cs.cmu.edu/~jmli`

Abstract

We show that many classical optimization problems – such as $(1 \pm \epsilon)$-approximate maximum flow, shortest path, and transshipment – can be computed in $\tau_{\text{mix}}(G) \cdot n^{o(1)}$ rounds of distributed message passing, where $\tau_{\text{mix}}(G)$ is the mixing time of the network graph G. This extends the result of Ghaffari et al. [PODC'17], whose main result is a distributed MST algorithm in $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n \log \log n})}$ rounds in the CONGEST model, to a much wider class of optimization problems. For many practical networks of interest, e.g., peer-to-peer or overlay network structures, the mixing time $\tau_{\text{mix}}(G)$ is small, e.g., polylogarithmic. On these networks, our algorithms bypass the $\tilde{\Omega}(\sqrt{n} + D)$ lower bound of Das Sarma et al. [STOC'11], which applies for worst-case graphs and applies to all of the above optimization problems. For all of the problems except MST, this is the first distributed algorithm which takes $o(\sqrt{n})$ rounds on a (nontrivial) restricted class of network graphs.

Towards deriving these improved distributed algorithms, our main contribution is a general transformation that simulates any work-efficient PRAM algorithm running in T parallel rounds via a distributed algorithm running in $T \cdot \tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n})}$ rounds. Work- and time-efficient parallel algorithms for all of the aforementioned problems follow by combining the work of Sherman [FOCS'13, SODA'17] and Peng and Spielman [STOC'14]. Thus, simulating these parallel algorithms using our transformation framework produces the desired distributed algorithms.

The core technical component of our transformation is the algorithmic problem of solving *multi-commodity routing* – that is, roughly, routing n packets each from a given source to a given destination – in random graphs. For this problem, we obtain a new algorithm running in $2^{O(\sqrt{\log n})}$ rounds, improving on the $2^{O(\sqrt{\log n \log \log n})}$ round algorithm of Ghaffari, Kuhn, and Su [PODC'17]. As a consequence, for the MST problem in particular, we obtain an improved distributed algorithm running in $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n})}$ rounds.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Graph Algorithms, Mixing Time, Random Graphs, Multi-Commodity Routing

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.31

1 Introduction and Related Work

This paper presents a general method that allows us to transform work-efficient parallel algorithms – formally in the PRAM model – into efficient distributed message-passing algorithms – formally in the CONGEST model – for a wide range of network graphs of practical interest. We believe that this method can be of significance for the following reasons: (1) parallel algorithms have been studied extensively since the late 1970s [11, 14, 30] and there is a vast collection of known parallel algorithms for a variety of problems, and (2) there is a rather active community of research on developing new parallel algorithms. Our transformation opens the road for exporting these algorithms to the distributed setting and bridging the research in these two subareas in a concrete and formal manner. As immediate corollaries, by translating the recent work-efficient parallel algorithms for flow-type problems, we obtain new distributed algorithms for approximate maximum flow, shortest path, and transshipment.

Of course, such a transformation is bound to have some limitations. Due to the reasons that shall be explained soon, such a general transformation would be inefficient in worst-case network graphs. But we show that there are efficient transformations for many graph families of practical interest, and we also exhibit that these transformations entail interesting and non-trivial theoretical aspects.

To explain our transformations, we first recall (informal) descriptions of the two computational models that we discuss, the distributed model and the parallel model. The more detailed model definitions are presented later in Section 3.

The Distributed Computing Model – CONGEST [27]. The network is abstracted as an n-node undirected graph $G = (V, E)$. There is one processor on each node of the network. At the risk of a slight informality, we use the words processor and node interchangeably. Each node has a unique $\Theta(\log n)$-bit identifier. Communication happens in synchronous rounds where per round, each node can send one B-bit message to each of its neighboring nodes in the network graph G, where typically one assumes $B = O(\log n)$. During each round, each processor can perform unbounded computation with the information that it has at the time. The graph is known in a distributed fashion: each processor knows the edges incident on its own node. In case that the edges are weighted, the weight is known to both endpoints. At the end of the computation, each node should know its own part of the output: e.g., in computing a coloring, each node should know its own color. One can measure the efficiency of an algorithm in the CONGEST model in different ways, such as number of rounds taken, or total number of messages sent among all nodes. In this paper, we only focus on minimizing the number of rounds that an algorithm takes.

The Parallel Model – PRAM [15, 18]. The system is composed of p processors, each with a unique ID in $\{1, 2, \ldots, p\}$, and a shared memory block of M entries, including an output tape. In every round, each processor can read from or write to any memory entry (Concurrent Read and Concurrent Write, aka, CRCW); if multiple processors write to the same entry, an arbitrary one takes effect.[1] The input is provided in the shared memory cells in a manner that can be addressed easily, e.g., in the case of a graph, the input can be given as an adjacency list where there is one memory cell for the j^{th} neighbor of the i^{th} node.

Limitations to General Transformations? Notice that the two models are intrinsically focused on different issues. The PRAM model is about speeding up computations, via using more processors, and tries to understand how much parallelism can help in reducing time. On the other hand, the distributed model is relevant where the system is by nature made of

[1] We can also support parallel algorithms that work under a more powerful model: if multiple processors write to the same memory, then we can take any associative function (min, max, sum) on the words written, and write that result into memory. However, for simplicity, we will work under the arbitrary CRCW model.

autonomous entities, each of which knows a part of the problem. For instance, in computer networks, which were historically the primary motivation for distributed models such as CONGEST, the computers in the network each know a part of the network graph and they cooperate to compute something about it, e.g., variants of shortest paths or routing tables. Here, *locality of the data* and *limited communication bandwidth* are the main challenges. As such, it is arguably unreasonable to seek a general efficient transformation of *any* parallel algorithm to a distributed one in *any* arbitrary network graph. Let us elaborate on this. (1) The PRAM model is not limited by any *locality* – each processor can asses any single register – while this is an intrinsic limitation in distributed systems – it can take time proportional to the diameter of the network graph for a processor to be informed of some bit residing in a far away corner of the network graph [2]. (2) Similarly, the network graph may have a small cut, which means transferring information across this cut, i.e., from the processors on one side of the cut to the other size, may take a long time, while this can be done much faster in the PRAM model.

So What Can We Hope For? The above discussions and the two concrete points on *locality* and *congestion* (or in other words communication bandwidth) suggest that there may be some hope left: at least in network graphs that satisfy some mild conditions on diameter and cut sizes (or alternatively expansion, conductance, or other forms of formalizing lack of "communication bottlenecks"), we might be able to find some general transformation. Arguably, these actually capture a range of network graphs of practical interest. For instance, overlay and peer-to-peer networks are designed and dynamically maintained over time in a fashion that ensures these good properties.

One way of classifying some such nice graph families is by selecting all graphs whose mixing time for a random walk is relatively small. We define mixing time in Section 1.1.2, but informally, the mixing time of a graph is the number of steps a lazy random walk needs to take so that the distribution of the last vertex of the walk is roughly uniform over all n vertices. A wide range of the (overlay) networks used in practical distributed applications exhibit a good (e.g. polylogarithmic in n) mixing time. This holds for example for the networks in consideration in [2, 3, 23, 25, 20, 24, 26, 33].

A canonical reason for this good mixing time is because many of these overlay networks are formed in a way where each node is connected to $\Theta(\log n)$ randomly chosen nodes. Indeed, we present our general transformation primarily for such random graphs. We then also explain how to emulate the communication on random graphs atop arbitrary networks with a round-complexity overhead related to the mixing time of the graph, thus enabling us to extend the transformation to general graphs, with a round complexity overhead proportional to the mixing time.

1.1 Our Results

Our results build off of those in [13], whose main result is a distributed MST problem running in nearly *mixing time*. We improve upon their results in two dimensions, one technical and one primarily conceptual. The technical contribution is an improved algorithm for the *multicommodity routing* problem in random graphs, which is equivalent to the *permutation routing* problem in [13] up to $\tilde{O}(1)$ factors. We solve this problem in $2^{O(\sqrt{\log n})}$ rounds,

[2] And that bit may be relevant, as is in global problems such as minimum spanning tree, shortest path, etc.

improving upon the $2^{O(\sqrt{\log n \log\log n})}$ round algorithm in [13]. Together with the ideas in [13], this immediately improves the distributed MST algorithm from $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n \log\log n})}$ to $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n})}$.

Our second, more conceptual contribution is in applying the multicommodity routing problem in a more general way. In particular, we use it to develop a framework that transforms work-efficient algorithms in the PRAM model to distributed algorithms. This *transformation* allows us to port the recent work-efficient parallel algorithms [28, 31, 32, 4] for approximate maximum flow, shortest path, and transshipment to run in the CONGEST model, taking $\tau_{\text{mix}}(G) \cdot n^{o(1)}$ rounds for all three problems.

We first describe our multi-commodity routing result for random graphs, our main technical result and a key component in our transformations. We believe that this multicommodity routing scheme and the hierarchical graph partitioning underlying it may be of independent interest. We then state our transformation results and overview some of their applications in deriving efficient distributed algorithms for some central graph problems.

1.1.1 Multicommodity Routing on Random Graphs

Random Graph Model. We work with the following random (multi-)graph model $G(n, d)$ is as follows: each node $v \in V$ picks $d = \Omega(\log n)$ random nodes in V independently with replacement, called the **outgoing** neighbors of v. The network graph consists of all edges (u, v) where u is an outgoing neighbor of v or vice versa. For $d = \Omega(\log n)$, this model behaves very similarly to the Erdös-Rényi model $\mathcal{G}(n, d/n)$ [9]; we use our variant for convenience. [3]

Multicommodity Routing. Consider a random graph $G(n, p)$ for $p = O(\log n)$, and suppose that we have pairs of nodes $(s_i, t_i) \in V \times V$. Suppose each node s_i wants to communicate with its respective node t_i; we assume that node t_i does not know s_i beforehand. Our goal is to identify a path P_i in G between each pair s_i and t_i. We refer to this problem as **multicommodity routing**, to be formally defined in Section 2. In addition, if every node $v \in V$ appears at most W times as s_i or t_i, then we say that this multicommodity routing instance has **width** W.

Our main technical contribution is an improved multi-commodity routing algorithm on random graphs with round complexity $2^{O(\sqrt{\log n})}$. This improves on a solution of Ghaffari et al. [13] which has round complexity $2^{O(\sqrt{\log n \log\log n})}$. In its simplest form, the theorem can be stated as follows.

▶ **Theorem 1.** *Consider a multicommodity routing instance of width $\tilde{O}(1)$. There is a multicommodity routing algorithm on $G(n, \Omega(\log n))$ that runs in time $2^{O(\sqrt{\log n})}$.*

General Graphs and Mixing Time. In fact, our result generalizes to more than random graphs in the same way as [13]. As shown by [13], random graphs can be "embedded" into any network graph with an overhead proportional to the *mixing time* τ_{mix} of the network graph, which we define below. Thus, we can generalize the multicommodity routing algorithm to work on any graph.

[3] Moreover, for many other models of random graphs, we can embed one round of this model (i.e., connecting each node to $O(\log n)$ randomly selected nodes) with a small, typically poly$(\log n)$ round, overhead. This would be by using $O(n \log n)$ random walks, $O(\log n)$ starting from each node, and walking them until the mixing time, which is like selected a random connection endpoint. This is similar to [13]. In many random graph families, these walks would mix in poly$(\log n)$ rounds [6].

Identically to [13], we define (lazy) random walks as follows: in every step, the walk remains at the current node with probability $1/2$, and otherwise, it transitions to a uniformly random neighbor. We formally define the mixing time of a graph as follows:

▶ **Definition 2.** For a node $u \in V$, let $\{P_u^t(v)\}_{v \in V}$ be the probability distribution on the nodes $v \in V$ after t steps of a (lazy) random walk starting at u. The **mixing time** of the graph, denoted τ_{mix}, is the minimum integer t such that for all $u, v \in V$, $\left| P_u^t(v) - \frac{\deg(v)}{2m} \right| \le \frac{\deg(v)}{2mn}$.

Our multicommodity routing algorithm for general graphs is therefore as follows:

▶ **Theorem 3.** *There is a distributed algorithm solving multicommodity routing in $\tau_{mix} \cdot 2^{O(\sqrt{\log n})}$ rounds.*

Finally, by substituting our multicommodity routing algorithm into the one in [13], we get an improvement on distributed MST in mixing time.

▶ **Theorem 4.** *There is a distributed MST algorithm running in $\tau_{mix} \cdot 2^{O(\sqrt{\log n})}$ rounds.*

We remark that, by a standard doubling trick, we can assume that the algorithm does not even know the mixing time τ_{mix} beforehand.[4]

1.1.2 Transformation

Our second, more conceptual contribution is a transformation from parallel algorithms to distributed algorithms on random graphs. In particular, we show that any work-efficient parallel algorithm running in T rounds can be simulated on a distributed random graph network in $T \cdot \tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$ rounds. The actual theorem statement, Theorem 14, requires formalizing the parallel and distributed models, so we do not state it here.

Applications. For applications of this transformation, we look at a recent line of work on near-linear time algorithms for flow-type problems. In particular, we investigate the approximate versions of shortest path, maximum flow, and transshipment (also known as uncapacitated minimum cost flow). Parallel $(1 \pm \epsilon)$-approximation algorithms for these problems running in $O(m^{1+o(1)})$ work and $O(m^{o(1)})$ time result from gradient descent methods combined with a parallel solver for symmetric diagonally dominant systems [28, 31, 32, 4]. Therefore, by combining these parallel algorithms with our distributed transformation, we obtain the following corollaries:

▶ **Corollary 5.** *There are distributed algorithms running in time*

$$\tau_{mix} \cdot 2^{O(\sqrt{\log n})}$$

for $(1 + \epsilon)$-approximate single-source shortest path and transshipment, and running time

$$\tau_{mix} \cdot 2^{O(\sqrt{\log n \log \log n})}$$

for $(1 - \epsilon)$-approximate maximum flow.

[4] Indeed, begin with a guess $\tau = 1$ for the value of τ_{mix} and run the algorithm, assuming that $\tau_{\text{mix}} = \tau$. If the algorithm takes more than $\tau \cdot 2^{O(\sqrt{\log n})}$ rounds, then every node in the distributed network immediately terminates the algorithm early and restarts with τ multiplied by 2.

Finally, in the case of random graphs, another classical problem is the computation of a Hamiltonian cycle. Since an $\tilde{O}(n)$-work, $\tilde{O}(1)$-time parallel algorithm is known [7], we have an efficient distributed algorithm to compute Hamiltonian cycles.

▶ **Corollary 6.** *For large enough constant C, we can find a Hamilton cycle on $G(n,d)$ with $d = C\log n$ in $2^{O(\sqrt{\log n})}$ rounds, w.h.p.*

This problem has attracted recent attention in the distributed setting. The main result of [5] is a distributed Hamiltonian cycle algorithm that runs in $\Omega(n^\delta)$ rounds for graphs $G(n,d)$ with $d = \Omega(\log n/n^\delta)$ for any constant $0 < \delta \le 1$. Thus, our algorithm greatly improves upon their result, both in number of rounds and in the parameter d.

1.2 Some Other Related Work

There has been a long history [34, 8, 16, 29, 10] in translating the ideal PRAM model into more practical parallel models, such as the celebrated BSP model of Valiant [34]. These transformations typically track many more parameters, such as communication and computation, than our transformation from PRAM to CONGEST, which only concerns the round complexity of the CONGEST algorithm.

There has also been work in the intersection of distributed computing and algorithms on random graphs. The task of computing a Hamiltonian cycle on a random graph was initiated by Levy et al. [22] and improved recently in [5]. Computation of other graph-theoretic properties on random graphs, such as approximate minimum dominating set and maximum matching, has been studied in a distributed setting in [17].

2 Multicommodity Routing

We formally define the multicommodity routing problem below, along with the congestion and dilation of a solution to this problem.

▶ **Definition 7.** A multicommodity routing instance consists of pairs of nodes $(s_i, t_i) \in V \times V$, such that each t_i is known to node s_i. A solution consists of a (not necessarily simple) path P_i connecting nodes s_i and t_i for every i, such that every node on P_i knows its two neighbors on P_i.

The input has **width** W if every node $v \in V$ appears at most W times as s_i or t_i.

For a given solution of paths, the **dilation** is the maximum length of a path, and the **congestion** is the maximum number of times an edge appears in total over all paths. More precisely, if $c_i(e)$ is the number of occurrences of edge $e \in E(G)$ in path P_i, then the congestion is $\max_{e\in E(G)} \sum_i c_i(e)$.

The significance of the congestion and dilation parameters lies in the following lemma from [12], whose proof uses the standard trick of *random delays* from packet routing [21]. In particular, if a multicommodity routing algorithm runs efficiently and outputs a solution of low congestion and dilation, then each node s_i can efficiently route messages to node t_i.

▶ **Theorem 8** ([12]). *Suppose we solve a multicommodity routing instance $\{(s_i, t_i)\}_i$ and achieve congestion c and dilation d. Then, in $\tilde{O}(c + d)$ rounds, every node s_i can send one $O(\log n)$-bit message to every node t_i, and vice versa.*

We now provide our algorithm for multicommodity routing, improving the congestion and dilation factors from $2^{O(\sqrt{\log n \log\log n})}$ in [13] to $2^{O(\sqrt{\log n})}$. Like [13], our algorithm uses the concept of *embedding* a graph, defined below.

▶ **Definition 9.** Let H and G be two graphs on the same node set. We say that an algorithm **embeds** H into G with congestion c and dilation d if the algorithm solves the following multicommodity routing instance on G: the (s_i, t_i) pairs are precisely the edges of H, the congestion is c, and the dilation is d. For each $(s, t) \in E(H)$, the path $P_{s,t}$ (in G from s to t) is called the **embedded path** for edge (s, t).

Our multicommodity routing algorithm will *recursively* embed graphs. We use the following helper lemma.

▶ **Lemma 10.** *Suppose there is a distributed algorithm $\mathcal{A}_1$ embedding graph G_1 into network G_0 with congestion c_1 and dilation d_1 in T_1 rounds, and another distributed algorithm $\mathcal{A}_2$ embedding graph G_2 into network G_1 with congestion c_2 and dilation d_2 in T_2 rounds. Then, there is a distributed algorithm embedding G_2 into network G_0 with congestion c_1c_2 and dilation d_1d_2 in $T_1 + T_2 \cdot \tilde{O}(c_1 + d_1)$ rounds.*

Proof. First, we provide the embedding without the algorithm. For each pair $(s, t) \in E(G_1)$, let $P^1_{s,t}$ be the embedded path in G_0, and for each pair $(s, t) \in E(G_2)$, let $P^2_{s,t}$ be the embedded path in G_1. To embed edge $(s, t) \in E(G_2)$ into E_0, consider the path $P^2_{s,t} := (s = v_0, v_1, v_2, \ldots, v_\ell = t)$; the embedded path for (s, t) in G_0 is precisely the concatenation of the paths $P^1_{v_{i-1},v_i}$ for $i \in [\ell]$ in increasing order. Since $\ell \le d_2$ and each path $P^1_{v_{i-1},v_i}$ has length at most d_1, the total length of the embedded path for (s, t) in G_0 is at most d_1d_2, achieving the promised dilation.

For congestion, let $c^1_{s,t}(e)$ denote the number of occurrences of edge $e \in E(G_0)$ in $P^1_{s,t}$. Since each edge $(s, t) \in E(G_1)$ shows up at most c_2 times among all $P^2_{s',t'}$, the number of times the path $P^1_{s,t}$ is concatenated in the embedding is at most $c^1_{s,t}(e) \cdot c_2$. Therefore, edge $e \in E(G_0)$ occurs at most $\sum_{s,t} c^1_{s,t}(e) \cdot c_2 \le c_1c_2$ times among all the concatenated paths embedding G_2 into G_0.

Finally, we describe the embedding algorithm. First, the algorithm on G_0 runs $\mathcal{A}_1$, obtaining the embedding of G_1 into G_0 in T_1 rounds. We now show how to emulate a single round of $\mathcal{A}_2$ running on network G_1 using $\tilde{O}(c_1 + d_1)$ rounds on network G_0. Suppose that, on a particular round, $\mathcal{A}_2$ has each node s send a message x to node t for every $(s, t) \in E(G_1)$. Since the embedding of G_1 into G_0 is a multicommodity routing instance, we use Theorem 8, where each node s tries to route that same message x to node t. This runs in $\tilde{O}(c_1 + d_1)$ rounds for a given round of $\mathcal{A}_2$. Altogether, we spend $T_1 + T_2 \cdot \tilde{O}(c_1 + d_1)$ rounds to emulate the entire $\mathcal{A}_2$. ◀

We now prove our main result, Theorem 1. We actually prove a stronger version of it, stated below.

▶ **Theorem 11.** *Consider a multicommodity routing instance of width $\tilde{O}(1)$. There is a multicommodity routing algorithm on $G(n, \Omega(\log n))$ that achieves congestion and dilation $2^{O(\sqrt{\log n})}$, and runs in time $2^{O(\sqrt{\log n})}$.*

Proof. Following [13], our strategy is to construct graph embeddings recursively, forming a hierarchical decomposition. We start off by embedding a graph of sufficiently high degree in G, similar to the "Level Zero Random Graph" embedding of [13]. Essentially, the embedded paths are random walks in G of length τ_{mix}; we refer the reader to [13] for details. Note that, like in [13], we are embedding the graph $G(m, d)$, not the graph $G(n, d)$.

▶ **Lemma 12** ([13], Section 3.1.1)**.** *On any graph G with n nodes and m edges, we can embed a random graph $G(m, d)$ with $d \ge 200 \log n$ into G with congestion $\tilde{O}(\tau_{mix} \cdot d)$ and dilation τ_{mix} in time $\tilde{O}(\tau_{mix} \cdot d)$.*

For our instance, $\tau_{\text{mix}} = O(\log n)$ since $G \sim G(n, \Omega(\log n))$. Let $d := 2^{10\sqrt{\log n}}$. For this value of d in the lemma, we obtain an embedding $G_0 \sim G(m, d)$ into G in time $\tilde{O}(2^{10\sqrt{\log n}})$.

Similarly to [13], our first goal is to obtain graphs $G_1, G_2, \ldots, G_K$ which form some hierarchical structure, such that each graph G_i embeds into G_{i-1} with small congestion and dilation. Later on, we will exploit the hierarchical structure of the graphs $G_0, G_1, G_2, \ldots, G_K$ in order to route each (s_i, t_i) pair.

To begin, we first describe the embedding of G_1 into G_0. Like [13], we first randomly partition the nodes of G_0 into β sets $A_1, \ldots, A_\beta$ so that $|A_i| = \Theta(m/\beta)$. Our goal is to construct and embed G_1 into G_0 with congestion 1 and dilation 2, where G_1 has the following structure: it is a disjoint union, over all $i \in [\beta]$, of a random graph $G_1^{(i)} \sim G(|A_i|, d/4)$ on the set A_i. By definition, G_0 and G_1 share the same node set. Note that [13] does a similar graph embedding, except with congestion and dilation $O(\log n)$; improving the factors to $O(1)$ is what constitutes our improvement.

Fix a set A_i; we proceed to construct the random graph in A_i. For a fixed node $u \in V(G_0)$, consider the list of outgoing neighbors of u in A_i. Note that since G_0 can have multi-edges, a node in A_i may appear multiple times in the list. Now, inside the local computation of node u, randomly group the nodes in the list into ordered pairs, leaving one element out if the list size is odd. For each ordered pair $(v_1, v_2) \in A_i \times A_i$, add v_2 into v_1's list of outgoing edges in $G_1^{(i)}$, and embed this edge along the path (v_1, u, v_2) of length 2. In this case, since the paths are short, node u can inform each pair (v_1, v_2) the entire path (v_1, u, v_2) in $O(1)$ rounds.

Since node u has d outgoing neighbors, the expected number of outgoing neighbors of u in A_i is d/β. By Chernoff bound, the actual number is at least $0.9d/\beta$ w.h.p., so there are at least $0.4d/\beta$ ordered pairs w.h.p. Over all nodes u, there are at least $0.4md/\beta$ pairs total.

We now argue that, over the randomness of the construction of G_0, the pairs are uniformly and independently distributed in $A_i \times A_i$. We show this by revealing the randomness of G_0 in two steps. If, for each node u, we first reveal which set A_j each outgoing neighbor of u belongs to, and then group the outgoing neighbors in A_i into pairs, and finally reveal the actual outgoing neighbors, then each of the at least $0.4md/\beta$ pairs is uniformly and independently distributed in $A_i \times A_i$. Therefore, each node $v \in A_i$ is expected to receive at least $\dfrac{0.4md/\beta}{m/\beta} = 0.4d$ outgoing neighbors, or at least $0.25d$ outgoing neighbors w.h.p. by Chernoff bound. Finally, we have each node in A_i randomly discard outgoing neighbors until it has $d/4$ remaining. The edges remaining in A_i form the graph $G_1^{(i)}$, which has distribution $G(|A_i|, d/4)$. Thus, we have embedded a graph G_1 consisting of β disjoint random graphs $G(\Theta(m/\beta), d/4)$ into G_0, where every embedded path is edge-disjoint in G_0 and has length 2. In other words, the embedding has congestion 1 and dilation 2.

We apply recursion in the same manner as in [13]: recurse on each $G_1^{(i)}$ (in parallel) by partitioning its vertices into another β sets $A_1, \ldots, A_\beta$, building a random graph on each set, and taking their disjoint union. More precisely, suppose the algorithm begins with a graph $H_0 \sim G(|V(G')|, d/4^{t-1})$ on depth k of the recursion tree (where the initial embedding of G_1 into G_0 has depth 1). The algorithm randomly partitions the nodes of H into $A_1, \ldots, A_\beta$ and defines a graph H_1 similar to G_1 from before: it is a disjoint union, over all $i \in [\beta]$, of a random graph $H_1^{(i)} \sim G(|A_i|, d/4^t)$ on the set A_i. Finally, the algorithm recurses on each $H_1^{(i)}$. This recursion stops when the graphs have size at most $2^{5\sqrt{\log n}}$; in other words, if $|V(H_0)| \le 2^{5\sqrt{\log n}}$, then the recursive algorithm exits immediately instead of performing the above routine.

Once the recursive algorithm finishes, we let G_k be the disjoint union of all graphs $H_1^{(i)}$ constructed on a recursive call of depth k. Observe that G_k has the same node set as G_0. Moreover, since, on each recursive step the sizes of the A_i drop by a factor of $1/\beta$ in expectation, or at most $2/\beta$ w.h.p., the recursion goes for at most $\log_{\beta/2} n \leq 2\sqrt{\log n}$ levels. Therefore, for each disjoint random graph in each G_k, the number of outgoing neighbors is always at least $d/4^{2\sqrt{\log n}} \geq 2^{6\sqrt{\log n}}$. In addition, since every embedding of G_k into G_{k-1} has congestion 1 and dilation 2, by applying Lemma 10 repeatedly, G_K embeds into G_0 with congestion 1 and dilation $2^{2\sqrt{\log n}}$, and into G with congestion and dilation $2^{O(\sqrt{\log n})}$. Moreover, on each recursion level k, the embedding algorithm takes a constant number of rounds on the graph G_{k-1}, which can be simulated on G in $2^{O(\sqrt{\log n})}$ rounds by Lemma 10.

Now we discuss how to route each (s_i, t_i) pair. Fix a pair (s, t); at a high level, we will iterate over the graphs $G_0, G_1, G_2, \ldots$ while maintaining the invariant that s and t belong to the same connected component in G_k. Initially, this holds for G_0; if it becomes false when transitioning from G_{k-1} to G_k, then we replace s with a node s' in the connected component of t in G_k. We claim that in fact, w.h.p., there is such a node s' that is *adjacent* to s in G_{k-1}; hence, s can send its message to s' along the network G_{k-1}, and the algorithm proceeds to G_k pretending that s is now s'. This process is similar to that in [13], except we make do without their notion of "portals" because of the large degree of G_0 – $2^{\Theta(\sqrt{\log n})}$ compared to $\Theta(\log n)$ in [13].

We now make the routing procedure precise. For a given G_k with $k < K$, if s and t belong to the same connected component of G_k, then we do nothing. Otherwise, since s has at least $2^{6\sqrt{\log n}} = \omega(\beta \log n)$ neighbors, w.h.p., node s has an outgoing neighbor s' in the connected component of G_k containing t; if there are multiple neighbors, one is chosen at random. Node s *relays* the message along this edge to s', and the pair (s, t) is replaced with (s', t) upon applying recursion to the next level. [5] Therefore, we always maintain the invariant that in each current (s, t) pair, both s and t belong in the same random graph.

We now argue that w.h.p., each vertex s' has $\tilde{O}(1)$ messages after this routing step. By assumption, every node $v \in V$ appears $\tilde{O}(1)$ times as t_j, so there are $|A_i| \cdot \tilde{O}(1)$ many nodes t_j that are inside A_i. For each such t_j with $s_j \notin A_i$, over the randomness of G_{k-1}, the neighbor s'_j of s_j inside A_i chosen to relay the message from s_j is uniformly distributed in A_i. By Chernoff bound, each node in A_i is chosen to relay a message $\tilde{O}(1)$ times when transitioning from G_{k-1} to G_k. In total, each node $v \in V$ appears $\tilde{O}(1)$ times as s_i in the beginning, and receives $\tilde{O}(1)$ messages to relay for each of $O(\sqrt{\log n})$ iterations. It follows that every node always has $\tilde{O}(1)$ messages throughout the algorithm.

Finally, in the graph G_K, we know that each (s_j, t_j) pair is in the same connected component of G_K. Recall that each connected component in G_K has at most $2^{5\sqrt{\log n}}$ nodes, each with degree at least $2^{6\sqrt{\log n}}$ (possibly with self-loops and parallel edges). It follows that w.h.p., each connected component is a "complete" graph, in the sense that every two nodes in the component are connected by at least one edge. Therefore, we can route each (s_j, t_j) pair trivially along an edge connecting them.

As for running time, since each graph $G_0, G_1, \ldots, G_K$ embeds into G with congestion and dilation $2^{O(\sqrt{\log n})}$ by Lemma 10, iterating on each graph G_k takes $2^{O(\sqrt{\log n})}$ rounds. Therefore, the total running time is $2^{O(\sqrt{\log n})}$, concluding Theorem 11. ◀

[5] In reality, node s does not know which set A_i contains node t. Like [13], we resolve this issue using $\tilde{O}(1)$-wise independence, which does not affect the algorithm's performance. Since $\Theta(W \log n)$ bits of randomness suffice for W-wise independence [1], we can have one node draw $\Theta(W \log n) = \tilde{O}(1)$ random bits at the beginning of the iteration and broadcast them to all the nodes in $\tilde{O}(1)$ time. Then, every node can locally compute the set A_i that contains any given node t; see [13] for details.

For general graphs, we can repeat the same algorithm, except we embed G_0 with congestion and dilation $\tilde{O}(\tau_{\text{mix}} \cdot 2^{10\sqrt{\log n}})$ instead of $\tilde{O}(2^{10\sqrt{\log n}})$, obtaining the following:

▶ **Corollary 13.** *Consider a multicommodity routing algorithm where every node $v \in V$ appears $\tilde{O}(1)$ times as s_i or t_i. There is a multicommodity routing algorithm that achieves congestion and dilation $\tau_{mix} \cdot 2^{O(\sqrt{\log n})}$, and runs in time $\tau_{mix} \cdot 2^{O(\sqrt{\log n})}$.*

Combining Theorem 8 and Corollary 13 proves Theorem 3.

3 Parallel to Distributed

In this section, we present our procedure to simulate parallel algorithms on distributed graph networks.

Parallel Model Assumptions. To formalize our transformation, we make some standard input assumptions to work-efficient parallel algorithms:

1. The input graph is represented in adjacency list form. There is a pointer array of size n, whose i'th element points to an array of neighbors of vertex v_i. The i'th array of input begins with $\deg(v_i)$, followed by the $\deg(v_i)$ neighbors of vertex v_i.
2. There are exactly $2m$ processors.[6] Each processor knows its ID, a unique number in $[2m]$, and has unlimited local computation and memory.
3. There is a shared memory block of $\tilde{O}(mT)$ entries, including the output tape, where T is the running time of the parallel algorithm.[7] In every round, each processor can read or write from any entry in unit time (CRCW model). If multiple processors write to the same entry on the same round, then an arbitrary write is selected for that round.
4. If the output is a subgraph, then the output tape is an array of the subgraph edges.

Distributed Model Assumptions. Similarly, we make the following assumptions on the distributed model.

1. Each node knows its neighbors in the input graph, as well as its ID, a unique number of $\Theta(\log n)$ bits. Each node has unlimited local computation and memory.
2. If the output is a subgraph, each node should know its incident edges in the subgraph.

▶ **Theorem 14.** *Under the above parallel and distributed model assumptions, a parallel graph algorithm running in T rounds can be simulated by a distributed algorithm in $T \cdot \tau_{mix} \cdot 2^{O(\sqrt{\log n})}$ rounds.*

Proof. Our goal is to simulate one round of the parallel algorithm in $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$ rounds in the distributed model, from which the theorem follows. To do so, we need to simulate the processors, input data, shared memory, and output.

[6] If the algorithm uses $m^{1+o(1)}$ processors, then we can have each of the $2m$ processors simulate $m^{o(1)}$ of them.

[7] If the algorithm uses much more than mT memory addresses, then we can hash the memory addresses down to a hash table of $\tilde{O}(mT)$ entries.

Processors. Embed a random graph $G_0 = G(2m, \Theta(\log n)/m)$ into the network graph, as in [13]. Every node in G_0 simulates one processor so that all $2m$ processors are simulated; this means that every node $v \in V$ in the original network simulates $\deg(v)$ processors. Let the nodes of G_0 and the processors be named (v, j), where $v \in V$ and $j \in [\deg(v)]$. Node/processor (v, j) knows the j'th neighbor of v, and say, $(v, 1)$ also knows the value of $\deg(v)$. Therefore, all input data to the parallel algorithm is spread over the processors (v, j). From now on, we treat graph G_0 as the new network graph in the distributed setting.

Shared memory. Shared memory is spread over all $2m$ processors. Let the shared memory be split into $2m$ blocks of size B each, where $B := \tilde{O}(1)$. Processor (v_i, j) is in charge of block $\sum_{i'<i} \deg(v_{i'}) + j$, so that each block is maintained by one processor. To look up block k in the shared memory array, a processor needs to write k as $\sum_{i'<i} \deg(v_{i'}) + j$ for some (v_i, j). Suppose for now that each processor knows the map $\phi : [2m] \to V \times \mathbb{N}$ from index k to tuple (v_i, j); later on, we remove this assumption.

On a given parallel round, if a processor wants to read or write to block k of shared memory, it sends a request to node $\phi(k)$. One issue is the possibility that many nodes all want to communicate with processor $\phi(k)$, and in the multicommodity routing problem, we only allow each target node to appear $\tilde{O}(1)$ times in the (s_i, t_i) pairs. We solve this issue below, whose proof is deferred to Appendix A.

▶ **Lemma 15.** *Consider the following setting: there is a node v_0, called the root, in possession of a memory block, and nodes $v_1, \ldots, v_k$, called leaves, that request this memory block. The root node does not know the identities of the leaf nodes, but the leaf nodes know the identity v_0 of the root node. Then, in $\tilde{O}(1)$ multicommodity routing calls of width $\tilde{O}(1)$, the nodes $v_1, \ldots, v_k$ can receive the memory block of node v_0.*

Now consider multiple such settings in parallel, where every node in the graph is a root node in at most one setting, and a leaf node in at most one setting. Then, in $\tilde{O}(1)$ multicommodity routing calls of width $\tilde{O}(1)$, every leaf node can receive the memory block of its corresponding root node.

Combining Lemma 15 and the multicommodity algorithm of Corollary 13 gives the desired distributed running time of $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$ per parallel round, modulo the assumption that each processor knows the map ϕ.

To remove the above assumption, we do the following as a precomputation step. We allocate an auxiliary array of size n, and our goal is to fill entry i with $\sum_{i'<i} \deg(v_{i'}) + j$. Let processor $(v_i, 1)$ be in charge of entry i. Initially, processor $(v_i, 1)$ fills entry i with $\deg(v_i)$, which it knows. Then, getting the array we desire amounts to computing prefix sums, and we can make the parallel prefix sum algorithm work here [19], since any processor looking for entry i knows to query $(v_i, 1)$ for it. Finally, for a node to determine the entry $\phi(k)$, it can binary search on this auxiliary array to find the largest i with $\sum_{i'<i} \deg(v_{i'}) < k$, and set $j := k - \sum_{i'<i} \deg(v_{i'})$, which is the correct (v_i, j).

Input data. If a processor in the parallel algorithm requests the value of $\deg(v)$ or the i'th neighbor of vertex v, we have the corresponding processor send a request to processor (v, i) for this neighbor. The routing details are the same as above.

Output. If the output is a subgraph of the original network graph G, then the distributed model requires each original node to know its incident edges in the subgraph. One way to do this is as follows: at the end of simulating the parallel algorithm, we can first sort the edges lexicographically using the distributed translation of a parallel sorting algorithm. Then, each node (v_i, i) binary searches the output to determine if the edge of v to its i'th neighbor u is in the output (either as (u, v) or as (v, u)). Since each original node $v \in V$ simulates each node/processor (v_i, i), node v knows all edges incident to it in the output subgraph. ◀

3.1 Applications to Parallel Algorithms

The task of approximately solving symmetric diagonally dominant (SDD) systems $Mx = b$ appears in many fast algorithms for ℓ_p minimization problems, such as maximum flow and transshipment. Peng and Spielman [28] obtained the first polylogarithmic time parallel SDD solver, stated below. For precise definitions of SDD, ϵ-approximate solution, and condition number, we refer the reader to [28].

▶ **Theorem 16** (Peng and Spielman [28]). *The SDD system $Mx = b$, where M is an $n \times n$ matrix with m nonzero entries, can be ϵ-approximately solved in parallel in $\tilde{O}(m \log^3 \kappa)$ work and $\tilde{O}(\log \kappa)$ time, where κ is the condition number of matrix M.*

Using our framework, we can translate this algorithm to a distributed setting, assuming that the input and output are distributed proportionally among the nodes.

▶ **Corollary 17.** *Let G be a network matrix. Consider a SDD matrix M with condition number κ, whose rows and columns indexed by V, and with nonzero entries only at entries $M_{u,v}$ with $(u, v) \in E$. Moreover, assume that each nonzero entry $M_{u,v}$ is known to both nodes u and v, and that each entry b_v is known to node v. In $\tilde{O}(\tau_{mix} \cdot \log^4 \kappa)$ distributed rounds, we can compute an ϵ-approximate solution x, such that each node v knows entry x_v.*

By combining parallel SDD solvers with gradient descent, we can compute approximate solutions maximum flow and minimum transshipment in parallel based on the recent work of Sherman and Becker et al. [31, 32, 4]. An added corollary is approximate shortest path, which can be reduced from transshipment [4].

▶ **Theorem 18** (Sherman, Becker et al. [31, 32, 4]). *The $(1 + \epsilon)$-approximate single-source shortest path and minimum transshipment problems can be solved in parallel in $m \cdot 2^{O(\sqrt{\log n})}$ work and $2^{O(\sqrt{\log n})}$ time. The $(1 - \epsilon)$-approximate maximum flow problem can be solved in parallel in $m \cdot 2^{O(\sqrt{\log n \log \log n})}$ work and $2^{O(\sqrt{\log n \log \log n})}$ time.*

▶ **Corollary 5.** *There are distributed algorithms running in time*

$$\tau_{mix} \cdot 2^{O(\sqrt{\log n})}$$

for $(1 + \epsilon)$-approximate single-source shortest path and transshipment, and running time

$$\tau_{mix} \cdot 2^{O(\sqrt{\log n \log \log n})}$$

for $(1 - \epsilon)$-approximate maximum flow.

Lastly, we consider the task of computing a Hamiltonian cycle on random graphs. This problem can be solved efficiently in parallel on random graphs $G(n, d)$, with $d = C \log n$ for large enough constant C, by a result of Coppersmith et al. [7]. We remark that [7] only states that their algorithm runs in $O(\log^2 n)$ time *in expectation*, but their proof is easily modified so that it holds w.h.p., at the cost of a larger constant C.

▶ **Theorem 19** (Coppersmith et al. [7]). *For large enough constant C, there is a parallel algorithm that finds a Hamiltonian cycle in $G(n, C \log n)$ in $O(\log^2 n)$ time, w.h.p.*

This immediately implies our fast distributed algorithm for Hamiltonian cycle; the result is restated below.

▶ **Corollary 6.** *For large enough constant C, we can find a Hamilton cycle on $G(n, d)$ with $d = C \log n$ in $2^{O(\sqrt{\log n})}$ rounds, w.h.p.*

4 Conclusion and Open Problems

In this paper, we bridge the gap between work-efficient parallel algorithms and distributed algorithms in the CONGEST model. Our main technical contribution lies in a distributed algorithm for multicommodity routing on random graphs.

The most obvious open problem is to improve the $2^{O(\sqrt{\log n})}$ bound in Theorem 1. Interestingly, finding a multicommodity routing solution with congestion and dilation $O(\log n)$ is fairly easy if we are allowed poly(n) time. In other words, while there exist good multicommodity routing solutions, we do not know how to find them efficiently in a distributed fashion. Hence, finding an algorithm that both runs in $\tilde{O}(1)$ rounds and computes a solution of congestion and dilation $\tilde{O}(1)$ is an intriguing open problem, and would serve as evidence that distributed computation on well-mixing network graphs is as easy as work-efficient parallel computation, up to $\tilde{O}(1)$ factors.

References

1 Noga Alon and Joel H Spencer. *The probabilistic method*. John Wiley & Sons, 2004.

2 John Augustine, Gopal Pandurangan, Peter Robinson, Scott Roche, and Eli Upfal. Enabling robust and efficient distributed computation in dynamic peer-to-peer networks. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 350–369. IEEE, 2015.

3 Baruch Awerbuch and Christian Scheideler. The hyperring: a low-congestion deterministic data structure for distributed environments. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 318–327. Society for Industrial and Applied Mathematics, 2004.

4 Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. *arXiv preprint arXiv:1607.05127*, 2016.

5 Soumyottam Chatterjee, Reza Fathi, Gopal Pandurangan, and Nguyen Dinh Pham. Fast and efficient distributed computation of hamiltonian cycles in random graphs. *arXiv preprint arXiv:1804.08819*, 2018.

6 Colin Cooper and Alan Frieze. Random walks on random graphs. In *International Conference on Nano-Networks*, pages 95–106. Springer, 2008.

7 Don Coppersmith, Prabhakar Raghavan, and Martin Tompa. Parallel graph algorithms that are efficient on average. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 260–269. IEEE, 1987.

8 David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp: Towards a realistic model of parallel computation. *ACM Sigplan Notices*, 28(7):1–12, 1993.

9 Paul Erdös and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

10 Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. A general pram simulation scheme for clustered machines. *International Journal of Foundations of Computer Science*, 14(06):1147–1164, 2003.

11 Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 114–118. ACM, 1978.

12 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 202–219. Society for Industrial and Applied Mathematics, 2016.

13 Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed mst and routing in almost mixing time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 131–140. ACM, 2017.

14 Leslie M Goldschlager. A unified approach to models of synchronous parallel machines. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 89–94. ACM, 1978.

15 Richard M Karp. A survey of parallel algorithms for shared-memory machines. Technical report, University of California at Berkeley, 1988.

16 Richard M Karp, Michael Luby, and F Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. *Algorithmica*, 16(4-5):517–542, 1996.

17 K Krzywdziński and Katarzyna Rybarczyk. Distributed algorithms for random graphs. *Theoretical Computer Science*, 605:95–105, 2015.

18 Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.

19 Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

20 Ching Law and Kai-Yeung Siu. Distributed construction of random expander networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 2133–2143. IEEE, 2003.

21 Tom Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 256–269. IEEE, 1988.

22 Eythan Levy, Guy Louchard, and Jordi Petit. A distributed algorithm to find hamiltonian cycles in g(n,p) random graphs. In *Workshop on Combinatorial and Algorithmic aspects of networking*, pages 63–74. Springer, 2004.

23 Peter Mahlmann and Christian Schindelhauer. Peer-to-peer networks based on random transformations of connected regular undirected graphs. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 155–164. ACM, 2005.

24 Gopal Pandurangan, Prabhakar Raghavan, and Eli Upfal. Building low-diameter peer-to-peer networks. *Selected Areas in Communications, IEEE Journal on*, 21(6):995–1002, 2003.

25 Gopal Pandurangan, Peter Robinson, and Amitabh Trehan. Dex: self-healing expanders. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 702–711. IEEE, 2014.

26 Gopal Pandurangan and Amitabh Trehan. Xheal: localized self-healing using expanders. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 301–310. ACM, 2011.

27 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

28 Richard Peng and Daniel A Spielman. An efficient parallel solver for sdd linear systems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 333–342. ACM, 2014.

29 Andrea Pietracaprina and Geppino Pucci. The complexity of deterministic pram simulation on distributed memory machines. *Theory of Computing Systems*, 30(3):231–247, 1997.

30 Walter J Savitch and Michael J Stimson. Time bounded random access machines with parallel processing. *Journal of the ACM (JACM)*, 26(1):103–118, 1979.

31 Jonah Sherman. Nearly maximum flows in nearly linear time. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 263–269. IEEE, 2013.

32 Jonah Sherman. Generalized preconditioning and undirected minimum-cost flow. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 772–780. SIAM, 2017.

33 Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

34 Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

A High Degree Communication

▶ **Lemma 15.** *Consider the following setting: there is a node v_0, called the root, in possession of a memory block, and nodes $v_1, \dots, v_k$, called leaves, that request this memory block. The root node does not know the identities of the leaf nodes, but the leaf nodes know the identity v_0 of the root node. Then, in $\tilde{O}(1)$ multicommodity routing calls of width $\tilde{O}(1)$, the nodes $v_1, \dots, v_k$ can receive the memory block of node v_0.*

Now consider multiple such settings in parallel, where every node in the graph is a root node in at most one setting, and a leaf node in at most one setting. Then, in $\tilde{O}(1)$ multicommodity routing calls of width $\tilde{O}(1)$, every leaf node can receive the memory block of its corresponding root node.

Proof (Lemma 15). We assume that every node has a unique ID in the range $\{1, 2, \dots, n\}$. The reduction from $\Theta(\log n)$-bit identifiers is standard: construct a BFS tree of depth D, where D is the diameter of the network graph, root the tree arbitrarily, and run prefix/infix/postfix ordering on the tree in $O(D)$ time. Since $\tau_{\text{mix}} \geq D$, this takes $O(\tau_{\text{mix}})$ time, which is negligible.

For now, consider the first setting of the lemma, with only one root node. Our goal is to establish a low-degree and low-depth (rooted) tree of communication, which contains the leaf nodes and possibly other nodes. Then, through calls of multicommodity routing, the root node sends the memory block to one of the nodes in this tree, which then gets propagated to all other nodes on the tree, including the leaf nodes. The key idea is that a *random low-depth tree* of communication, chosen from a certain distribution, will turn out to be low-degree as well. We now state the precise construction of this random tree.

Let K be a parameter that starts at $n/2$ and decreases by a factor of 2 for $T := \lceil \log_2(n/2) \rceil$ rounds. The node with ID 1 picks a hash function $f : V \times [K] \to V$ for this iteration, and broadcasts it to all other nodes in D rounds. At the end, we will address the problem of encoding hash functions, but for now, assume that the hash function has mutual independence.

On iteration i, each leaf node computes a private random number $k \in [K]$ and computes $f(v_0, k) \in V$, called the *connection point* for leaf node v_i. We will later show that, w.h.p., each node in V is the connection point of $\tilde{O}(1)$ leaf nodes. Assuming this, we form the multicommodity routing instance where each leaf node requests a routing to its connection point, so that afterwards, each connection point v_j learns its set S_j of corresponding leaf nodes. Each connection point elects a random node $v_j^* \in S_j$ as the *leader*, and routes the entire set S_j to node v_j^* in another multicommodity routing instance. All nodes in $S_j \backslash v_j^*$, which did not receive the set S_j, drop out of the algorithm, leaving the leader v_j^* to route to other nodes in later iterations. At the end of the algorithm, there is only one leader left, and that leader routes directly to the root node v_0, receiving the memory block. Finally, the memory block gets propagated from the leaders v_j^* to the other nodes in S_j in reverse iteration order.

We now show that, w.h.p., each node in V is a connection point to $\tilde{O}(1)$ leaf nodes; this would bound the width of the multicommodity instances by $\tilde{O}(1)$. Initially, there are at most n leaf nodes and $n/2$ possible connection points, so each connection point has at most 2 leaf nodes in expectation, or $O(\log n)$ w.h.p. On iteration $t > 1$, there are at most $n/2^{t-1}$ leaf nodes left, since each of the $n/2^{t-1}$ connection point elected one leader in the previous iteration and those are the only leaf nodes remaining. So each of the $n/2^t$ connection points has at most 2 leaf nodes in expectation, or $O(\log n)$ w.h.p.

Now consider the general setting, where we do the same thing in parallel over all groups of leaf nodes. On iteration t, let the set of remaining leaf nodes in each setting be $L_1, \dots, L_r$. For each set of leaf nodes L_i, a given node v_j has probability $1/2^t$ of being selected as a connection point for L_i, and if so, it is expected to have at most $\frac{|L_i|}{n/2^t}$ many leaf nodes in L_i, or $O(\frac{|L_i|}{n/2^t} \log n) = O(\log n)$ w.h.p., using that $|L_i| \le n/2^{t-1}$. Therefore, if X_j^i is the random variable of the number of leaf nodes in L_i assigned to node v_j, then $\mathbb{E}[X_j^i] \le |L_i|/n$, and $X_j^i = O(\log n)$ w.h.p. Conditioned on the w.h.p. statement, we use the following variant of Chernoff bound:

▶ **Theorem 20** (Chernoff bound). *If $X_1, \dots, X_n$ are independent random variables in the range $[0, C]$ and $\mu := \mathbb{E}[X_1 + \dots + X_n]$, then*

$$\Pr[X_1 + \dots + X_n \ge (1+\delta)\mu] \le \exp\left(-\frac{2\delta^2\mu^2}{nC^2}\right).$$

Taking the independent variables $X_j^1, \dots, X_j^r$ and setting $\delta := \Theta(\frac{\log^2 n}{\mu})$ and $C := O(\log n)$, we get that $\mu = \sum_i |L_i|/n \le 1$ and

$$\Pr[X_j^1 + \dots + X_j^r \ge \Theta(\log^2 n)] \le \exp(-O(\log^2 n)).$$

Therefore, w.h.p., every node has $O(\log^2 n)$ neighbors at any given round.

Lastly, we address the issue of encoding hash functions, which we solve using W-wise independent hash families for a small value W. Since the algorithm runs in $\tilde{O}(1)$ rounds, $W = \tilde{O}(1)$ suffices. It turns out that deterministic families of $2^{O(W \log n)}$ hash functions exist [1], so the node with ID 1 can simply pick a random $O(W \log n) = \tilde{O}(1)$-bit string and broadcast it to all other nodes in $D + \tilde{O}(1) = \tilde{O}(\tau_{\text{mix}})$ rounds. ◀