Efficient Distributed Algorithms in the k-machine model via PRAM Simulations

John Augustine[1]
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Chennai, India 600 004
Email: augustine@iitm.ac.in

Kishore Kothapalli[2]
Center for Security, Theory, and Algorithmic Research
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India 500 032.
Email: kkishore@iiit.ac.in

Gopal Pandurangan[3]
Department of Computer Science
University of Houston
Houston, Texas, USA 77204
Email: gopal@cs.uh.edu

***Abstract*—We study several fundamental problems in the k-machine model, a message-passing model for large-scale distributed computations where $k \geqslant 2$ machines jointly perform computations on a large input of size N, (typically, $N \gg k$). The input is initially partitioned (randomly or in a balanced fashion) among the k machines, a common implementation in many real-world systems. Communication is point-to-point, and the goal is to minimize the number of communication rounds of the computation.**

Our main result is a general technique for designing efficient deterministic distributed algorithms in the k-machine model using PRAM algorithms. Our technique works by efficiently simulating PRAM algorithms in the k-machine model in a *deterministic* way. This simulation allows us to arrive at new algorithms in the k-machine model for some problems for which no efficient k-machine algorithms are known before and also improve on existing results in the k-machine model for some problems.

While our simulation allows us to obtain k-machine algorithms for any problem with a known PRAM algorithm, we mainly focus on graph problems. For an input graph on n vertices and m edges, we obtain $\tilde{O}(m/k^2)$ round[4] algorithms for various graph problems such as r–connectivity for $r = 1, 2, 3, 4$, minimum spanning tree (MST), maximal independent set (MIS), $(\Delta + 1)$-coloring, maximal matching, ear decomposition, and spanners under the assumption that the edges of the input graph are partitioned (randomly, or in an arbitrary, but balanced, fashion) among the k machines. For problems such as connectivity and MST, the above bound is (essentially) the best possible (up to logarithmic factors). Our simulation technique allows us to obtain the first known efficient *deterministic* algorithms in the k-machine model for other problems with known deterministic PRAM algorithms.

[1]Research supported in part by DST/SERB Extra Mural Grant EMR/2016/00301 and DST/SERB MATRICS Grant MTR/2018/001198.

[2]Research supported in part by DST/SERB Extra Mural Grant EMR/2016/007668 and DST/SERB MATRICS Grant MTR/2017/000849.

[3]Research supported, in part, by NSF grants CCF-1540512, IIS-1633720, CCF-BSF-1717075, BSF award 2016419, and by the VAJRA faculty program of the Government of India.

[4]$\tilde{O}$ notation hides a $\mathrm{polylog}(\cdot)$ factor and an additive $\mathrm{polylog}(\cdot)$ term.

I. Introduction

The focus of this paper is on the distributed processing of large-scale inputs, which has become increasingly important with the rise of massive datasets in general and massive graphs such as the Web graph, social networks, and biological networks in particular. A key goal in distributed large-scale computation is to *minimize* the amount of *communication* across machines, as this typically dominates the overall cost of the computation (see, e.g. [52]). Frameworks such as the Map-Reduce [37] and Spark [2] are popular for distributed computation on massive datasets. In the context of graphs, several large-scale graph processing systems such as Pregel [40], Giraph [1], and Spark's GraphX [3] have been designed based on the *message-passing* distributed computing model [39], [46]. In these systems, the input which is simply too large to fit into a single machine, is distributed across a group of machines connected via a communication network. The machines jointly perform computation in a distributed fashion by sending/receiving messages.

Motivated by the above considerations, we study a number of fundamental problems in a message-passing distributed computing model for large-scale computations called the k-*machine model* [30] (explained in detail in Section II). In this model, the input is distributed across a group of k machines that are pairwise interconnected via a communication network. The k machines jointly perform computations to produce the output. The communication is point-to-point via message passing. The computation advances in synchronous rounds, and there is a constraint on the amount of data that can cross each link of the network in each round. The complexity of algorithms is expressed as the *round complexity* and is measured as the number of *communication rounds* required by the computation. The aim of this model is to investigate the amount of "speed-up" possible vis-a-vis the number of available machines in the following sense: when k machines are used, how does the *round complexity scale in* k?

There are several recent results concerning graph algorithms in the k-machine model [30], [44], [6], [45], [23], [32], [18]. All the above papers crucially assume an input distribution known as the *Random Vertex Partition (RVP)* model where each vertex of the input graph and each of its incident edges is assigned to a machine chosen independently and uniformly at random. It can be shown ([30]) that, on a graph of n vertices and m edges, the RVP model gives a balanced partition such that each machine gets $\tilde{O}(n/k)$ vertices and $\tilde{O}(m/k + \Delta)$ edges, where Δ is the maximum degree of the graph (which can be $\Theta(n)$). Under the RVP model, Klauck et al. [30] presented lower and upper bounds for several graph problems including connectivity, minimum spannning tree (MST), maximal independent set (MIS), shortest paths, spanners, PageRank, triangle counting and enumeration, etc. In particular, assuming that each link has a bandwidth of $O(\text{polylog}\, n)$ bits[5] per round, they show a lower bound of $\tilde{\Omega}(n/k^2)$ rounds for the graph connectivity problem. They also present an $\tilde{O}(n/k)$-round algorithm for graph connectivity and spanning tree (ST) verification.

The work of [45], [44] presents an $\tilde{O}(n/k^2)$-round algorithm in the RVP model for graph connectivity, thus achieving a speedup quadratic in k and also matches the lower bound up to polylogarithmic (in n) factors. This result is significant since it demonstrated that there are non-trivial graph problems for which we can obtain *superlinear* (in k) speed-up.

Despite these spurts of progress in designing algorithms in the k-machine model, we highlight two prominent questions that arise from the current progress on distributed computation in the k-machine model. Firstly, as mentioned earlier, most of the current algorithms for graph problems are designed under the RVP model. While the RVP model is used in many graph processing systems, for many graph problems, the input graph is typically available as a list of edges (e.g., graphs in the popular SNAP dataset [35]). For this setting, the more *natural* input model is the *Random Edge Partition* (REP) model where the m edges of the input graph are distributed among the k machines (randomly or in a balanced fashion) so that each machine gets $\tilde{O}(m/k)$ edges. It is worth noting that algorithms that work in the RVP model ([30], [44], [6], [45], [23]) do not necessarily yield optimal algorithms in the REP model (cf. Section III-A). Secondly, many of the efficient algorithms in the k-machine model use randomization (cf. Section III-A). Our work presents the first-known efficient *deterministic* algorithms for many problems (including graph connectivity and MST) in the REP model.

The main contribution of this paper is a general technique to derive *efficient* and *deterministic* distributed algorithms in the k-machine model for various fundamental graph problems. Our technique allows one to convert algorithms designed in the standard *PRAM model* to the k-machine model. Our technique has wide applicability as the PRAM model is a well-studied model for algorithm design in the parallel setting with a wealth of literature on PRAM algorithms for many problems from various domains.

While this paper focuses on the theory of the design and analysis of k-machine model algorithms, as part of further work (cf. Section VI), we plan to implement these algorithms and study their performance. It is fairly straightforward to implement k-machine algorithms using message-passing platforms such as MPI. We have implemented such algorithms for PageRank and MST from [30] in [4].

A. Other Related Work

Theoretical study of large-scale graph computation in distributed systems is relatively new. Several works have been devoted to developing graph algorithms in the MapReduce or the Massively Parallel Computing (MPC) model (e.g., see [7], [37], [33], [36], [28], [23], [5] and references therein).

Simulating algorithms originally designed in one computational model to other models of computation is the central theme in a number of prior works [29], [55], [51], [27], [21], [22], [28], [30], [17]. In particular, see [21] for a survey of PRAM simulation results and techniques. Several papers study simulation of PRAM in a distributed memory machine (DMM), see e.g., [15], [55], [27], [29] and the references therein. Upfal and Wigderson [55] give a deterministic scheme to simulate a PRAM step in $O(\log n(\log\log n)^2)$ time when there are n processors in both the PRAM and the DMM has n memory modules. The subsequent work of Karp et al. [29] give a more efficient *randomized* simulation, i.e., each PRAM step can be simulated in $O(\log\log n \log^* n)$ time. Note that in the DMM simulations every processor/memory module pair can only answer one request per round and typically the bounds are for the case when the number of processors in both the PRAM and DMM are the same. In contrast, in the k-machine model (each machine has its own memory), k is much smaller than n (sublinear in n) and there is congestion at the machine level (only) due to bandwidth limitations.

Karloff et al. [28] show that PRAM algorithms can be translated to work in the Map-Reduce model. The simulation from [28] converts a $t(n)$-time and $O(n^{2-2\epsilon})$-processor CREW PRAM algorithm to $O(t(n))$ round Map-Reduce algorithm. The number of mappers and reducers used in the simulation is also in $O(n^{2-2\epsilon})$ unlike our results which delink the number of processors used in the PRAM algorithm and the number of machines in the k-machine model. Hegeman and Pemmaraju [22] show that algorithms designed in the congested clique model can be translated to run in the MapReduce model. Klauck et al. [30] show several graph algorithms in the k-machine model by converting algorithms designed in the CONGEST model.

II. The Model

The k-machine model. We now describe the adopted model of distributed computation, the *k-machine model* (a.k.a. the *Big Data model*), introduced in [30] and further investigated in [43], [48], [13], [6], [45], [23], [32], [16]. The model consists of a set of $k \geqslant 2$ machines $\{M_1, M_2, \ldots, M_k\}$ that are pairwise interconnected by bidirectional point-to-point

[5]If the bandwidth is B bits per round, the bounds are to be divided by B.

communication links. Each machine executes an instance of a distributed algorithm. The computation advances in synchronous rounds where, in each round, machines can exchange messages over their communication links and perform some local computation. Each link is assumed to have a bandwidth of B bits per round, i.e., B bits can be transmitted over each link in each round; unless otherwise stated, we assume $B = \Theta(\text{polylog}\, N)$ (where N is the input size), although our bounds can be easily rewritten in terms of B [45].[6] Each machine has its own (internal) memory (assumed to be unlimited[7]) which it can access for free (any number of its own memory locations can be accessed simultaneously in one round). Machines do not. share any memory and have no other means of communication.

Local computation within a machine is ignored since our focus is on the exchange of messages between machines, which is the costly operation. However, we note that in all the algorithms of this paper, every machine in every round performs lightweight computations; in particular, these computations are bounded by a polynomial (typically, even linear) in the size of the input assigned to that machine. The *round complexity* of an algorithm is the maximum number of rounds until termination.

Input and Output Requirement Initially, the entire input is not known to any single machine, but rather partitioned among the k machines in a "balanced" fashion. Eventually, each machine M_i, for $1 \leqslant i \leqslant k$, must set a designated local output variable o_i (which need not depend on the set of vertices assigned to M_i), and the *output configuration* $o = \langle o_1, \ldots, o_k \rangle$ must satisfy certain feasibility conditions for the problem at hand.

PRAM model. The PRAM model (cf. [24]) is a popular model that is used in designing parallel algorithms for a wide variety of problems over decades. Key elements of this model are the availability of a large number of processors that have shared access to a pool of memory that can be accessed at uniform cost. The processors execute in a synchronous manner. Given the shared nature of memory, the model can be classified further as the Exclusive-Read-Exclusive-Write (EREW), the Concurrent-Read-Exclusive-Write (CREW), and the Concurrent-Read-Concurrent-Write (CRCW) models. As these names indicate, algorithms in the the EREW model do not have more than one processor reading from or writing to the same cell of memory at the same time. Algorithms in the CREW model allow concurrent reads but forbid concurrent writes. In the CRCW model, also concurrent writes are supported by further specifying the semantics of the concurrent write operation. The parameters of interest are: the time taken by the parallel algorithm, the number of processors used, and the overall work done by the parallel algorithm.

[6]There is an alternative (but equivalent) way to view this communication restriction: instead of putting a bandwidth restriction on the links (which increases with the number of machines), we can put a restriction on the amount of information that each *machine* can communicate (i.e., send/receive) in each round [45]. This is similar to the restriction imposed in the popular Map-Reduce/MPC models as well [28], [7]. In such model, the memory and bandwidth of each machine are restricted to sublinear in the size of the input; this is equivalent to choosing k sublinear in the input size in the k-machine model.

[7]In many k-machine algorithms, the memory usage is limited, essentially proportional to the size of the input allocated to that machine.

PROBLEM	UPPER BOUND **This Paper**	LOWER BOUND
Graph Algorithms		
Connectivity Strong Connectivity* (Directed Graphs) Minimum Spanning Tree (MST)	$\tilde{O}(\min\{m/k^2, n/k\})$	$\Omega(\min\{m/k^2, n/k\})$ [56]
r-Connectivity $r = 2, 3, 4$	$\tilde{O}(m/k^2)$	
Maximal Independent Set, Coloring, Maximal matching	$\tilde{O}(m/k^2)$	$\tilde{\Omega}(n/k^2)$ for MIS [32]
Breadth First Search	$\tilde{O}(\frac{m+n}{k^2} + D)$	
Approximate SSSP*, (weighted, undirected)	$\tilde{O}(\frac{mn^\delta}{k^2})$, $\delta > 0$	
Other Problems		
Matrix multiplication[8]	$\tilde{O}(M(n)/k^2)$	
Convex hull, Voronoi diagram	$\tilde{O}(N/k^2)$	
Suffix tree[9]	$\tilde{O}(N \log \Sigma/k^2)$	

TABLE I

SUMMARY OF OUR RESULTS. THE ALGORITHMS FOR SSSP ARE $(1 + O(1/\text{POLYLOG}(n)))$-APPROXIMATE. PROBLEMS MARKED WITH A * IN THE FIRST COLUMN INDICATE THAT THE CORRESPONDING ALGORITHMS ARE RANDOMIZED IN NATURE.

III. OUR CONTRIBUTIONS AND TECHNIQUES

A. Our Results

Our main result is a *general* technique for designing *efficient deterministic* distributed algorithms in the k-machine model. Our technique is based on efficiently simulating PRAM algorithms in a deterministic way in the k-machine model (cf. Section IV). This is stated in Theorem 1 below as the *Deterministic Simulation Theorem*. Theorem 1 allows us to translate PRAM algorithms to the k-machine model as shown in Table I. Our simulation extends to also randomized PRAM algorithms. Although our Deterministic Simulation Theorem can be applied for any PRAM algorithm, we mainly focus on graph problems as graph problems are one of the natural problems to solve in the k-machine model.

Theorem 1 (Deterministic Simulation Theorem)**.** *A PRAM algorithm $\mathcal{P}$ that runs in $T(N)$ time using $P(N) \geqslant k$ processors, $M(N)$ memory words, and does $W(N)$ total work to solve an instance of problem Π of size N can be translated into an algorithm $\mathcal{K}$ for the k-machine model that runs in $\tilde{O}\Big(\frac{W(N)}{k^2} + T(N)\Big)$ rounds.*

Table I lists some of the **new** and **efficient** deterministic algorithms in the k-machine model for various fundamental problems under an appropriate input model that we obtain using the Deterministic Simulation Theorem. (See also Section V). Importantly, the algorithms we show are *deterministic* algorithms; if the PRAM algorithm is deterministic, then since

[6]$M(n)$ is the worst case work complexity of the best known PRAM algorithm for multiplying two $n \times n$ matrices.

[7]The input is a string of N characters from an alphabet Σ.

the simulation is deterministic we obtain deterministic k-machine algorithms.

B. An Overview of Techniques

As a warmup, we first show that a single step of any (deterministic) PRAM algorithm can be simulated on the k-machine model so that each of the k machines simulates an equal number of processors as used in a PRAM algorithm (cf. Theorem 3). We use processors to denote PRAM processors and machines to denote the machines in the k-machine model. Similarly, time is used to denote the timesteps needed by a PRAM algorithm and rounds to denote the number of communication rounds of a k-machine algorithm. This simulation is randomized and makes certain assumptions such as a uniform mapping of shared memory cells to the machines. These assumptions are removed by using techniques that are shown in a related context by Upfal and Widgerson [55]. Upfal and Widgerson show how to deterministically simulate one step of an n-processor PRAM algorithm in a *complete network* of n nodes where nodes communicate via links by message passing with no shared memory. Furthermore, each link has a limited bandwidth (each link can send at most one data item per round). The challenge is to efficiently simulate (up to) n shared memory accesses of the n processors in a distributed network with no shared memory. The main technical idea behind the simulation is a scheme to distribute the shared memory among the processors. This distribution is done by using properties of a bipartite expander graph and using multiple (logarithmic) copies for each word of the shared memory. We adapt this technique for the k-machine model by solving additional technical challenges.

Firstly, [55] is only concerned with efficiently accessing shared memory. In our case, we have to handle both shared memory as well as processors, so we replicate both shared memory words and processors and place them in the k machines. The placement is in a well-distributed manner that ensures that for any (adversarial) choice of $\tilde{\Omega}(k)$ processors (resp., shared memory words), the replicated *executions* (resp., *copies* of the shared memory words) are forced to spread across the $\tilde{\Omega}(k)$ machines so that no small subset of machines is over-burdened. In addition to balancing the load across machines, this also ensures that communication is also load balanced because all the pair-wise links between the $\tilde{\Omega}(k)$ machines are exercised. This results in a speedup of k^2 in the k-machine round complexity with respect to the work complexity of the PRAM algorithm.

Secondly, since each processor is replicated into multiple executions, we should ensure that the multiple executions of a processor coordinate appropriately. We achieve this by ensuring that at most one execution takes the lead and executes the step, and then shares the updated state to other executions so that we can consistently ensure that a majority of the executions (resp., copies) are up-to-date for every processor (resp., shared memory word).

Thirdly, we have to be careful to stop short of keeping all executions and copies up-to-date because this can make the simulation sub-optimal in terms of work. For this, we introduce new techniques to enable machines to infer when to give up on some executions/copies. Our techniques result in Theorem 1 where the simulation runs in a deterministic manner while adding a logarithmic overhead in the process.

IV. Deterministic Simulation of PRAM Algorithms in the k-machine Model

In this section, we present simulation techniques to execute PRAM algorithms in the k-machine model. We begin with a warm up where we show a randomized simulation which succeeds with high probability. Since practitioners typically expect much stronger (in particular deterministic) guarantees, we follow up with a deterministic simulation which is also more efficient. Our deterministic simulation will require a single initial setup for all subsequent simulations.

A. The General Setting

Consider a PRAM algorithm $\mathcal{P}$ for a problem Π. Suppose, to solve an instance π of input size N of Π, $\mathcal{P}$ requires $P(N)$ processors, takes $T(N)$ time, and performs work $W(N)$. The $P(N)$ processors access a shared memory of at most $M(N)$ words, each of size at most $O(\log N)$. When clear from the context, we drop the problem size parameter N. We assume further that each processor has $O(1)$ words of local/private memory (typically in the form of its registers). The contents of this private memory of a processor is called its *state*. In each time step t, $1 \leqslant t \leqslant T$, we assume that some W_t processors are active, i.e., processors that read/write into the shared memory. Thus, the total work performed by $\mathcal{P}$ is given by $W = \sum_{1 \leqslant t \leqslant T} W_t \leqslant P \times T$. In each time step, each active processor reads $O(1)$ items from the shared memory, performs computation on these items, and writes back $O(1)$ items (without loss of generality, a subset of the read items as not every item read need not be written back) into the shared memory.

B. Warmup: A Randomized Simulation

Our goal is to efficiently convert a PRAM algorithm $\mathcal{P}$ into an algorithm $\mathcal{K}$ that runs on the k-machine model. We assume that $P(N) \geqslant k$ and each machine simulates at most $\lceil P(N)/k \rceil$ number of PRAM processors[8] chosen in some *arbitrary* way. Recall that we use processors to denote PRAM processors and machines to denote the machines in the k-machine model. Similarly, *time* is used to denote the timesteps needed by a PRAM algorithm and *rounds* to denote the number of communication rounds of a k-machine algorithm. We assume that the entire shared memory used by the PRAM algorithm is partitioned across the k machines so that *each word in the memory* is placed *independently and uniformly at random* in one of the k machines. Furthermore, every machine knows this memory mapping (typically implemented in the form of a hash function) so that any machine that requires a

[8]Notice that the simulation we seek is different in spirit to the Brent's slowdown simulation [9] in the context of PRAM algorithms as Brent's slowdown simulation still assumes the availability of shared memory.

particular shared memory word will know which machine to request for it.

To convert $\mathcal{P}$ to the corresponding $\mathcal{K}$, we first focus on the $\tilde{O}(P(N)/k)$ processors in some machine i and analyze the time it takes to execute one PRAM time step. The $\tilde{O}(P(N)/k)$ memory accesses issued by these processors can be viewed as $O(P(N)/k)$ balls being thrown randomly (uniformly and independently) into k bins. Thus, with high probability each link incident to machine i will be used for at most $\tilde{O}(P(N)/k^2+1)$ updates. Since all the k machines perform this in parallel and this process is repeated for T time steps, $\mathcal{P}$ can be simulated in $\tilde{O}\left(\frac{P(N)T(N)}{k^2} + T(N)\right)$ rounds of the k-machine with high probability.

Notice that concurrent reads are a trivial extension. For concurrent writes, suppose multiple PRAM processors need to concurrently write to a single word w in machine j, and let machine i contain one or more of such processors. Then, the processors in machine i first resolve the appropriate write policy and send exactly one write request for w to machine j. Machine j might receive multiple such write requests from several machines, so it must write into w in accordance with the appropriate concurrent write policy.

Theorem 2 (Randomized Simulation Theorem)**.** *Let $\mathcal{P}$ be a PRAM algorithm for problem Π that uses $P(N) \geqslant k$ processors and $M(N)$ memory words to solve an instance π of input size N. Then, we can translate $\mathcal{P}$ into an algorithm $\mathcal{K}$ for the k-machine model such that $\mathcal{K}$ requires at most $\tilde{O}((P(N)T(N)/k^2 + T(N))$ rounds and each machine requires at most $\tilde{O}((P(N) + M(N))/k)$ memory with probability at least $1 - N^{-c}$ for any fixed constant $c > 0$.*

The probabilistic guarantee given by Theorem 2 is a shortcoming that we must address. In addition, despite achieving a speedup of $O(k^2)$, the guarantees are loose because of the assumption that every processor is active in every PRAM time step. In reality, the work performed by a PRAM algorithm may be far less than $P(N) \times T(N)$. A good example is the parallel BFS algorithm [8] where the number of processors that are active in any given timestep depend on the number of vertices in the BFS frontier. The simulation itself can be easily adapted to situations where fewer processors execute at any given time step, but we need to recognize an important subtlety that will limit our speedup to k. When we seek deterministic guarantees, the simulation could be stymied by a PRAM algorithm $\mathcal{P}$ in which the processors in some machine a execute every time step, while others are only active for, say, one round. Suppose further that all the required data by processors in machine a is concentrated in another machine b. Then, all communication is restricted to just the single link between machines a and b that in effect results in no scaling.

One possible way to guard against such PRAM algorithms is to use deterministic routing techniques from the work of Lenzen [34] that can distribute the load to all the possible links thereby improving the scaling to a factor of $1/k$.

To go beyond a speedup of k, we will bring another technique into play – redundancy. In particular, we consider creating $O(\log N)$ copies of each processor and each memory word in the PRAM model. By placing these copies uniformly at random across the k machines and insisting that only a majority of these copies need to be executed for executing any given time step in the PRAM algorithm, we will show that the executions and communication will be more balanced across the k machines and the $k(k-1)/2$ links respectively. This will result in achieving a scaling down by an $O(1/k^2)$ factor as desired.

C. The Deterministic Simulation

In the rest of the section, we will present a deterministic simulation that is work-efficient in the sense that it ensures that the round complexity in the k-machine model is related to the work performed by $\mathcal{P}$. We will only present the simulation for one PRAM time step $t \in [T]$, which of course can be repeated for T time steps. The key idea is to maintain $2c - 1 = O(\log N)$ copies of each word (a la Upfal and Wigderson [55]) and additionally $2c-1$ redundant executions (or just executions) for each processor. Executions are represented by the state of the processor, i.e., the contents of their registers. The $2c-1$ executions for each processor are therefore stored as states in $2c - 1$ different machines. The simulation must ensure the invariant that a majority consisting of at least c copies of each word and c executions (i.e., the copies of the states) of each processor are up-to-date at all times. We therefore require a preprocessing step (that must be performed just once and used for any number of simulations) wherein the copies of memory words and the copies of the executions are assigned to appropriate machines. The exact placement of the memory copies and executions will be addressed in the analysis, but for now we assume that each machine is aware of the machine in which each copy and execution resides.

We refer the reader to Algorithm 1 for an overview of the steps we use to simulate one PRAM time step in the k machine model. Each machine $i \in [k]$ is responsible for the execution of processors $A_i \subseteq P$ during the current time step t. Note that multiple machines will be responsible for each processor j that has to be executed in the current time step, so those machines responsible for each processor j must coordinate with each other to execute the processor.

In Algorithm 1, we call machines that hold executions that need to be executed in the current time step as *active machines* and the rest as *inactive machines*. Inactive machines inform other machines about their inactive status and wait till the next time step. Algorithm 1 is to be executed by all active machines $i \in [k]$.

The `while` loop of Algorithm 1 consists of three major steps for each machine i: *acquiring* a batch of processors to execute a PRAM step, *executing* that batch of processors, and a *cleanup* step. We describe each of these steps, detailed in Algorithms 2–4 respectively, in the following.

In Algorithm 2, each machine i sends out a "request to execute j", as an REQ_EXEC message to a *majority* of the machines (including itself) that are responsible for processor j. We prove shortly (cf. Lemma 3) that the $2c - 1$

different executions of each processor can be placed in the k machines so that for any choice of processors, a majority of the executions for each of the processors can be chosen from the k machines in a balanced fashion. This allows machine i to therefore send the requests to all the $k-1$ other machines in a balanced fashion. As explained earlier, without this load balancing, we risk the situation where all requests are routed through a single incident link, thereby incurring an extra factor k in the round complexity.

If all machines in the majority respond positively, it is clear that no other machine can attain this sole responsibility. However, we should ensure that some machine will get that majority and therefore execute processor j. Consider the first time that at least one machine raises the request to execute j. If exactly one machine raises the request, then that machine will get the sole responsibility by receiving OK messages from the machines it sent REQ_EXEC messages to. However, if multiple machines raise it, then machine i acts as follows with respect to processor j. From all the machines i' that sent an REQ_EXEC message to i for processor j, machine i sends an OK message to only the machine i^* with the highest ID among machines that have the largest time stamp when that machine executed processor j. As shown in Line 8 of Algorithm 2, $i^* = \max\{i' \mid i \text{ received an REQ\_EXEC message from } i'\}$.

Since each machine contacts a strict majority, all other machines will be (implicitly) rejected by at least one machine thereby not having at least c OK messages as mentioned in Line 11 of Algorithm 2. Suppose machine i gains the sole responsibility to execute processor j. In order to execute j, machine i first obtains the state of j from the machine that has the latest timestamp from all the OK messages received for processor j.

Algorithm 3 shows the steps that machine i uses to execute one step of processor j. Machine i must read from and/or write to a majority of the copies of words accessed in the execution of processor j. In fact, i may be responsible for several (up to k) such executions and if all of them access words from the same machine i', we will again face congestion along the link between i and i'. However, as we place the copies of the shared memory words in the k machines, these memory accesses can again be performed in a balanced fashion as shown in Lemma 3. Finally, the winner, after executing processor j, informs a majority of machines responsible for j, so no subsequent machine will be allowed to execute j in the current time step.

Notice from Algorithm 1 that in each iteration of the `while` loop, each machine i removes k processors from its A_i (except the last iteration which may be fewer removals). Consider the set of all processors $A = \cup_i A_i$ that are to be executed in the current time step t and let $W_t = |A|$ denote the amount of work that $\mathcal{P}$ performs in the current time step. If the executions of processors in A are evenly distributed across the machines, then, this will imply immediately that time step t completes in $O(cW_t/k^2 + 1) = \tilde{O}(W_t/k^2 + 1)$ rounds of the k machine. However, A can be quite arbitrary.

Algorithm 1 Deterministic simulation of one PRAM time step from the perspective of a machine i.

1: Let A_i denote the processors that must be executed during the current time step and have an execution hosted by machine i;
2: **while** A_i is non-empty **do**
3: Use Algorithm Acquire (Algorithm 2) to get ownership of k processors B^* whose instruction(s) is to be executed locally;
4: **for all** processors P in B^* **do**
5: Execute P using Algorithm Execute (Algorithm 3)
6: Invoke Cleanup (Algorithm 4)

Algorithm 2 Acquire: Subroutine to acquire ownership of processors to execute.

1: Create a batch $B \subseteq A_i$ of cardinality k (or as many that are left). Set $A_i \leftarrow A_i \setminus B$.
2: Find the set of machines O_j that have an execution for processor j in B.
3: **for all** $j \in B$ **do**
4: Send message $\langle REQ\_EXEC, i', j\rangle$ to c machines i' in $O_j \cup \{i\}$
5: /* Let T_{ij} be the time step at which processor j was last executed by machine i */
6: $J = \{$Processors j for which there is at least one REQ_EXEC message received$\}$
7: **for all** $j \in J$ **do**
8: Send $\langle OK, i^*, j, T_{ij}\rangle$ if $i^* = \max\{i' \mid i$ received an REQ_EXEC message from $i'\}$
9: $B^* \leftarrow \emptyset$.
10: **for all** $j \in B$ **do**
11: if there are at least c OK messages then
12: add j to B^*.
13: obtain the state of j from the machine i^* that has the latest timestamp from all the OK messages for j
14: endif
15: Return B^*

For example, all of the W_t processors could be in one machine. So if we allow all the machines i to execute the while loop until their respective A_i's are all processed, this could lead to a situation where such heavily loaded machines require $\tilde{O}(W_t/k + 1)$ communication rounds; notice that the speedup is by factor k^{-1} as opposed to k^{-2}. To avoid this situation, we keep track of the machines that are no longer active, i.e., exited the `while` loop. Furthermore, active machines prune out processors whose copies are in machines that are no longer active. This is illustrated in Algorithm 4.

Thus, we have the following lemma that shows that the simulation correctly indeed executes all the processors that have an action to execute for a given PRAM time step.

Lemma 1. *A machine i will only exit the* `while` *loop if all processors in A_i have been executed by some machine.*

From Lemma 1, it is clear that some machine has executed every processor in A. We still need to show that the PRAM time step t can be simulated in $\tilde{O}(W_t/k^2 + 1)$ rounds regardless of the choice of A. For this and other purposes, we first prove a few balls-into-bins lemmas. We use balls as analogues of both processors and memory words, and bins as analogues of machines. In our balls-into-bins analogy, we don't throw balls, but rather copies of the balls (analogous to how we store copies of memory words or executions of processors in machines).

Algorithm 3 Subroutine to execute one step of processors in B^*.

INPUT: a set of processor B^*
/*We now execute processors in B^* such that for each memory word accessed by those processors a majority of the copies are updated.*/
$M = \emptyset$
for all processors $j \in B^*$ **do**
  $M := M \cup$ memory words accessed by j
for all words $w \in M$ **do**
  Find a set Q_w of c machines such that each machine $q \in Q_w$ has a copy of w ;
  /*Note that for every machine q, $|\{w \mid w \in M \wedge q \in Q_w\}| = \tilde{O}(1)$ due to Lemma 3.*/
  for all $q \in Q_w$ **do**
    /* Read */
    Read w from all the machines in Q_w along with the time stamps T_w corresponding to when the word w was last updated in Q_w
    Use the value of w
    /* Execute */
    Execute the time step for processor j
    /*Write, if needed */
    If w has to be written, send $\langle UPDATE, w, Q_w, T_w \rangle$
    /* T_w is the current time stamp that will be used to mark the update with. */
    for all messages received of the form $\langle UPDATE, r, Q_r, T_r \rangle$ **do**
      Update the local copy of r and set the timestamp of the update
    /*Also, update local variable copy updates sent by other machines.*/
    Send the updated state of processor j to all machines in B^* that have an execution for j.
    Receive updated processor states and update corresponding local executions to the updated states.
    /* This ensures that a majority of the executions have, in effect, executed processor j. */

Algorithm 4 Cleanup

/*This step crucially ensures that all machines will complete around the same time even under worst-case selection of processors.*/
Let $C = \{j \in A_i \mid$ a copy of j is in a machine that is no longer active in the current time step$\}$.
$A_i \leftarrow A_i \setminus C$.
/*Pruning out executions of processors that other machines have already executed.*/
if $A_i = \emptyset$ then inform all machines that i is no longer active for the current time step.

We now provide a lemma inspired by Upfal and Wigderson [55]. Consider the random experiment of throwing $2c-1$ copies of N^δ distinguishable balls into k bins where $\delta > 0$ is a fixed constant. We ensure that no two copies of a ball are in the same bin. For a given ball, the set of bins that have a copy of this ball is called as the *span* of the ball. For a set X of balls, the span of X is naturally the set of bins that have at least one copy of some ball in X. We say that a mapping of these copies of balls into bins is *smooth* if for any subset X of balls with c copies of each ball, the span of X has a size of at least $\alpha \cdot \min(\beta \log N, k/\log N)$ where $\beta = |X|$ and $\alpha > 0$ is some fixed constant.

Lemma 2. *With high probability, an independent and uniformly-at-random distribution of $2c-1$ copies of N^δ balls into k bins is smooth.*

Proof. The proof is based on a similar lemma proved by Upfal and Wigderson (cf. Lemma 3.2 in [55]). □

Remark 1. *For our deterministic simulation, we assume that a smooth mapping exists and is accessible by the machines. The above lemma shows the existence of such a mapping; in fact it shows that most "random" mappings are smooth. We note that such a mapping has to be constructed only once for all simulations (as a preprocessing step). In addition, we need only $O(c)$-wise independence for the assignment of balls (and their copies) into bins. This means that such a mapping can be done by choosing a c-wise universal hash function [10]. Such a hash function can be constructed by using c random bits.*

Lemma 3. *Consider again the assignment of the $2c-1$ copies of the N^δ balls into k bins as stated under Lemma 2 and any choice of balls X of cardinality $\beta \leqslant N^\delta$ balls. Then, for each ball in X, there is a choice of c copies such that no bin has more than $\tilde{O}(\beta/k + 1)$ chosen copies. (In particular, when $\beta = k$, we get $\tilde{O}(1)$ copies per bin.)*

Proof. As an aid for our proof, we present a procedure that operates in rounds. At the beginning, all copies of all balls are unmarked. A ball is said to be complete if at least c of its copies are marked, and incomplete otherwise. In each round r, for each bin that has at least one copy of an incomplete ball, we mark a copy of one of its incomplete balls chosen arbitrarily. We terminate when all balls are complete, thereby guaranteeing that each ball has at least c of its copies marked. The total number of rounds to termination is therefore an upper bound on the number of marked balls in each bin. So the rest of the proof is aimed at showing that the total number of rounds is $\tilde{O}(\beta/k + 1)$.

Let β_r be the number of incomplete balls at the start of round r. Consider as phase 1, the rounds r when $\beta_r \geqslant k/\log^2 N$. From Lemma 2, we know that the span of the β_r incomplete balls considering only the unmarked copies will be at least $k/\log N$. Since each bin in the span will be able to mark one of its balls, each round r will mark at least $\Omega(k/\log N)$ copies. At least $\Omega(k/\log^2 N)$ of those marked copies will be useful in the sense that they will count towards marking a copy of an incomplete ball. To see why, notice that a ball that was incomplete at the start of the round with $a < c$ marked copies may become complete with $a' \geqslant c$ marked copies by the end of the round. Out of the $a' - a < 2c - 1$ marked copies, at least one will count towards reaching the complete state. Thus, in $O(\frac{\beta \log^3 N}{k})$ rounds, we will be able to reduce the number of incomplete balls to fewer than $k/\log N$, thus ending phase 1.

Phase 2 are the rounds r when $0 < \beta_r < k/\log^2 N$, again using Lemma 2, we know that the span will be at least $\Omega(\beta \log N)$. Thus, $\Omega(\beta \log N)$ copies will be marked out of which at least $\Omega(\beta)$ will be useful (as argued earlier) in reducing the number of unmarked copies in incomplete bins. Since the number of copies that need to be usefully marked in phase 2 is at most $c \times k/\log^2 N = O(k/\log N)$, the number of rounds in phase 2 is the smallest integer x such that $\frac{k}{\log N}\left(1 - \frac{1}{\log N}\right)^x < 1$, which is at most $O(\log^2 N)$. □

Lemma 4. *Each machine i performs at most $\tilde{O}(W_t/k^2+1)$ iterations of the* `while` *loop in Algorithm 1.*

Lemma 5. *Algorithm 1 simulates a time step t in the PRAM algorithm $\mathcal{P}$ in $\tilde{O}(W_t/k^2+1)$ rounds of the k machine model. Here, W_t is the work performed by $\mathcal{P}$ during time step t.*

Thus we obtain the following theorem (akin to Theorem 1).

Theorem 3 (Deterministic Simulation Theorem)**.** *Let $\mathcal{P}$ be a PRAM algorithm that runs in $T(N)$ time using $P(N) \geqslant k$ processors, $M(N)$ memory words, and does $W(N)$ total work to solve an instance of a problem Π of size N. Then, we can translate $\mathcal{P}$ into an algorithm $\mathcal{K}$ for the k-machine model so that $\mathcal{K}$ runs in $\tilde{O}\Big(\frac{W(N)}{k^2}+T(N)\Big)$ rounds and each machine will require at most $\tilde{O}((P(N)+M(N))/k)$ memory.*

Theorem 3 is significantly stronger than Theorem 2. Firstly, $\mathcal{K}$ will be a deterministic algorithm iff $\mathcal{P}$ is deterministic. If $\mathcal{P}$ is randomized, then $T(N)$ and the round complexity of $\mathcal{K}$ will be random variables. Secondly, the round complexity of $\mathcal{K}$ is significantly more nuanced in that it depends on the work performed by $\mathcal{P}$ rather than the processor-time product. The latter could be significantly higher as the case of parallel BFS [8] shows.

V. Applications

We now show the following k-machine algorithms for various problems. Each of the theorems we state in this section is the result of a direct application of Theorem 3. Although our Simulation Theorem can be applied for any PRAM algorithm, we mainly focus on graph problems and discuss other problems briefly. In particular, when we apply our Simulation Theorem for graph problems, we obtain the *first* ever algorithms in the k-machine model for problems such as higher order connectivity, and also *deterministic* algorithms for fundamental problems such as symmetry breaking.

A. Graph Algorithms

Graph Input. Consider an input graph G with n vertices, each associated with a unique integer ID from $[n]$, and m edges. To avoid trivialities, we will assume that $n, m \geqslant k$ (typically, $m, n \gg k$, and also we generally assume that $m = \Omega(n)$). Although k can be any value (between 2 and n), the interesting regime for our bounds is when k is sublinear in n, i.e., $k = n^{\epsilon}$ for any constant $\epsilon > 0$. For the algorithms in this paper, we assume the REP model ([56], [45]) whereby edges are assigned independently and uniformly at random to one of the k machines in a *balanced* fashion. Each machine gets $\tilde{O}(m/k)$ edges. However, we note that our algorithms will work for any arbitrary, but balanced, partitioning of the edges. It is easy to show that one can transform the input partition from the REP model to the RVP model in $\tilde{O}(m/k^2+n/k)$ rounds [4] using techniques from [30], [45] or [34]. Therefore, one is interested in algorithms that improve this *trivial* $\tilde{O}(m/k)$ upper bound.

1) Connectivity, Connected Components, and MST: Chong et al. [12] show a deterministic PRAM algorithm to obtain the MST of a given graph, which also implies an algorithm for checking connectivity and connected components, using $O(m+n)$ processors and $O(\log n)$ time. Using this algorithm and Theorem 3, we obtain the following theorem.

Theorem 4. *Graph connectivity, connected components, and MST can be solved in $\tilde{O}(\min\{m/k^2, n/k\})$ rounds in the k-machine model where m is the number of edges in the graph.*

Theorem 3 gives the $\tilde{O}(m/k^2)$ upper bound. We obtain a deterministic $\tilde{O}(n/k)$ algorithm for MST in the k-machine model in the REP model as follows. The high-level idea is based on repeated filtering (see e.g., [33]). Initially all k machines are active. One phase of the algorithm consists of: (i) "Filter" the edges in each active machine using the cut and cycle properties of the MST [26]; this leaves each active machine with $O(n)$ edges. (ii) Pair up the active machines (this pairing can be hardcoded apriori). Each machine sends its list of edges to its corresponding paired machine by using the deterministic routing of Lenzen [34] which takes $O(n/k)$ rounds. Each phase halves the number of active machines. This is repeated over $O(\log k)$ phases for a total of $\tilde{O}(n/k)$ rounds. At the end, one machine (say machine 1) will have all the MST edges.

Thus, MST (and connectivity and connected components) can be solved in $\tilde{O}(\min\{m/k^2, n/k\})$ rounds *deterministically* with a (an almost) matching lower bound (cf. Section III-A).

2) r-connectivity for $r = 2, 3, 4$: Using Theorem 3, and the PRAM algorithms for testing r-connectivity via the works of Tarjan and Vishkin [54] for $r = 2$, Miller and Ramachandran [42] for $r = 3$, and Kanevsky and Ramachandran [25] for $r = 4$, we obtain Theorem 5. These are the *first* known and *deterministic* k-machine algorithms for 2-,3-, and 4-connectivity.

Theorem 5. *Graph r-connectivity, for $r = 2, 3, 4$, can be solved in $\tilde{O}(m/k^2)$ deterministic rounds in the k-machine model where m is the number of edges in the graph.*

3) Symmetry breaking: For symmetry breaking problems such as maximal matching (MM), maximal independent set (MIS), and coloring (COL), we obtain the following theorem via the respective derandomized PRAM algorithms of Luby [38] and Han [19]. These are the *first* known determinsitic algorithms for symmetry breaking in the k–machine model (under REP). For randomized algorithms, Konrad et al. [32] show an upper bound of $\tilde{O}(\min\{n/k, m/k^2\})$ and a lower bound of $\tilde{\Omega}(n/k^2)$ for MIS in the RVP model.

Theorem 6. *An MIS, $\Delta+1$ coloring, and a maximal matching of a graph can be solved deterministically in $\tilde{O}(m/k^2)$ rounds in the k-machine model.*

4) Ear Decomposition: An ear decomposition of a graph is a partitioning of the edges of the graph into edge disjoint paths (*ears*) P_i, for $i \geqslant 1$, such that P_1 is a simple cycle, and the end points of the i^{th} ear, P_i for $i > 1$, are the only vertices common to P_i and the vertices of $\cup_{j=1}^{i-1} P_j$. Using the PRAM algorithm of Ramachandran [49], the following theorem holds.

Theorem 7. *An ear decomposition of a graph can be obtained in $\tilde{O}(m/k^2)$ rounds in the k-machine model.*

As the ear decomposition of a graph has applications to several important problems such as planarity testing and finding planar embedding [50], the above theorem leads to *first* known $\tilde{O}(n/k^2)$-round algorithms for planar graphs in the k-machine model.

5) BFS: Given an unweighted graph G of diameter D, and a source vertex s, the PRAM algorithm for BFS as designed by Blelloch and Maggs [8, Section 4.2] runs in $O(D)$ parallel time using $O(m+n)$ work. This translates to the following theorem.

Theorem 8. *Given a directed weighted graph $G = (V, E)$ and a source vertex s, a breadth first traversal from s can be obtained in $\tilde{O}((\frac{m+n}{k^2} + D))$-rounds in the k-machine model.*

Noticeably, the product of the number of processors and the parallel run time of the BFS algorithm is much higher at $O((m+n) \cdot D)$ whereas the deterministic simulation theorem is able to arrive at a k-machine algorithm that has a round complexity of $\tilde{O}(\frac{m+n}{k^2} + D))$.

6) Shortest Paths: Obtaining shortest paths from a given source in a weighted directed graph is a problem with several applications. Klein and Subramanian [31] show that the shortest paths from a source s can be computed in $\tilde{O}(\sqrt{n} \log L)$ PRAM time with high probability using $\tilde{O}(m\sqrt{n} \log L)$ work where L is the sum of the weights of the edges of G. We obtain the following theorem from Theorem 3.

Theorem 9. *Given a directed weighted graph $G = (V, E)$ and a source vertex s, the shortest paths from s to all other vertices can be found in $\tilde{O}(\frac{m\sqrt{n} \log L}{k^2})$-rounds in the k-machine model with high probability.*

For undirected graphs and approximation algorithms, a few improvements to Lemma 9 can be obtained through the following theorem using the work of Cohen [14].

Theorem 10. *Given a directed weighted graph $G = (V, E)$ and a source vertex s, $(1+\epsilon)$–approximate shortest paths from s to all other vertices in G can be found in $\tilde{O}(\frac{mn^{\delta}}{k^2})$-rounds in the k-machine model. In the above, $\epsilon = O(1/polylog(n))$ and $\delta > 0$ is any constant.*

For sparse graphs with $m < n^{1-\delta}k^{3/2}$, Theorem 10 improves the current knowledge on $(1+\epsilon)$–approximate single source shortest paths from [30].

All Pairs Shortest Paths (APSP). Using Theorem 10 over n times, one can see that the $(1+\epsilon)$–approximate APSP problem can be solved in the k-machine model using $\tilde{O}(mn^{1+\delta}/k^2)$ rounds. This result improves the corresponding result from [30] designed in the RVP model, provided $m = O(n^{\frac{1}{2}-\delta} \cdot k)$.

7) Directed Graphs: Following the results of Schudy [53], we get the *first* known (randomized) algorithms in the k-machine model for strong connectivity of directed graphs.

Theorem 11. *The strongly-connected components and a topological sort of a directed graph G can be obtained in $\tilde{O}(m/k^2)$ rounds in the k-machine model, with high probability.*

8) Spanners: Graph spanners are a useful tool in the context of approximate shortest paths. A t-spanner of a graph G is a subgraph H such that distances in H are no more than distances in G by a multiplicative factor of t. The recent work of Miller et al. [41] results in the following theorem.

Theorem 12. *Given a weighted graph G with n vertices and m edges with each edge weight at most U and a parameter $t \geqslant 1$, an $O(t)$-spanner for G containing $O(n^{1+1/t} \log t)$ edges in expectation can be obtained in $\tilde{O}(\frac{m \log U}{k^2})$ rounds in the k-machine model with high probability.*

B. Other Applications

1) Algebraic Computations: For matrix algorithms, we use the Random Matrix Element Partition (RMEP) model where the elements of the input matrix (matrices) are partitioned across the k machines independently and uniformly at random. This is similar to the REP model we use for graph algorithms. Given a square matrix of size $n \times n$, each machine in the k-machine model gets $\tilde{O}(n^2/k)$ elements of (each of) the matrix. Existing algorithms, for instance matrix multiplication [11], assume that the each row of the matrix is stored at some machine corresponding to Random Matrix Row Partition (RMRP) model.

Let us denote by $M(n)$ the worst-case work complexity of the best known PRAM algorithm for multiplying two $n \times n$ matrices. For example, the algorithm of Strassen (cf. [24]) has $M(n) = O(n^{\log_2 7})$. In the RMEP model, Theorem 3 gives us the following theorem.

Theorem 13. *Multiplying two $n \times n$ matrices can be achieved in $O(M(n)/k^2)$ rounds in the k-machine model.*

2) Algorithms from Strings: We consider the problem of constructing the suffix tree of a given string. Suffix tree construction is fundamental to the field of bioinformatics. In this case, we assume that for a string of length N, each machine of the k-machine model is given N/k characters of the string chosen independently and uniformly at random. Using the PRAM algorithm of Hariharan [20], we obtain:

Theorem 14. *The suffix tree of a string of N characters from an alphabet Σ can be constructed in $\tilde{O}(N \log |\Sigma|/k^2)$ rounds in the k-machine model.*

3) Geometric Algorithms: In this case, we assume that the input is N points on the plane and each machine in the k-machine model is given $O(N/k)$ points of the input chosen independently and uniformly at random. Owing to NC algorithms for geometric computations such as the convex hull and Voronoi diagrams, [47], we obtain $\tilde{O}(N/k^2)$ round algorithms for these problems in the k-machine model.

VI. Conclusion

We presented a general technique for designing efficient deterministic distributed algorithms in the k-machine model and

showed its application for many fundamental graph problems. While some algorithms are optimal with respect to the REP input model, for some problems such as shortest paths, it is not clear whether these are the best possible. These are interesting questions for further research. We also plan to implement our algorithms using a message-passing distributed computing platform such as MPI or GraphX and study their performance.

REFERENCES

[1] Giraph, 2016. http://giraph.apache.org/.

[2] spark, 2016. http://spark.apache.org/.

[3] Apache Spark, Graphx, 2018. https://spark.apache.org/graphx/.

[4] E. Ajieren, K. Hourani, W. Moses, and G. Pandurangan. Communication and computation efficient distributed algorithms for mst and connectivity in large graphs. Under submission, 2020.

[5] A. Andoni, Z. Song, C. Stein, Z. Wang, and P. Zhong. Parallel graph connectivity in log diameter rounds. In *Proc. IEEE FOCS*, pages 674–685, 2018.

[6] S. Bandyapadhyay, T. Inamdar, S. Pai, and S. V. Pemmaraju. Near-optimal clustering in the k-machine model. In *Proc. ICDCN*, 2018.

[7] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017.

[8] G. E. Blelloch and B. M. Maggs. Parallel algorithms. In *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*. Chapman & Hall/CRC, 2010.

[9] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[10] L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[11] Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Proc. ACM PODC*, pages 143–152. ACM, 2015.

[12] K. Wong. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM*, 48(2):297–323, 2001.

[13] F. Chung and O. Simpson. Distributed algorithms for finding local clusters using heat kernel pagerank. In *Proc. WAW*, pages 77–189, 2015.

[14] E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *Proc. ACM STOC*, pages 16–26, 1994.

[15] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Contention resolution in hashing based shared memory simulations. *SIAM J. Comput.*, 29(5):1703–1739, 2000.

[16] R. Fathi, A. R. Molla, and G. Pandurangan. Efficient distributed algorithms for the k-nearest neighbors problem. In *SPAA*, pages 527–529. ACM, 2020.

[17] M. Ghaffari and J. Li. New distributed algorithms in almost mixing time via transformations from parallel algorithms. In *Proc DISC.*, volume 121, pages 31:1–31:16, 2018.

[18] S. Gilbert and L. Li. How fast can you update your MST. In *Proc. ACM SPAA*, pages 531–533, 2020.

[19] Y. Han. An improvement on parallel computation of a maximal matching. *Inf. Process. Lett.*, 56(6):343–348, 1995.

[20] Ramesh Hariharan. Optimal parallel suffix tree construction. *J. Comput. Syst. Sci.*, 55(1):44–69, 1997.

[21] T. J. Harris. A survey of PRAM simulation techniques. *ACM Comput. Surv.*, 26(2):187–206, 1994.

[22] J. W. Hegeman and S. V. Pemmaraju. Lessons from the congested clique applied to mapreduce. *Theor. Comput. Sci.*, 608:268–281, 2015.

[23] T. Inamdar, S. Pai, and S. V. Pemmaraju. Large-scale distributed algorithms for facility location with outliers. In *Proc. OPODIS*, 2018.

[24] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[25] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. *J. Comput. Syst. Sci.*, 42(3):288–306, 1991.

[26] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[27] A. R. Karlin and E. Upfal. Parallel hashing: an efficient implementation of shared memory. *J. ACM*, 35(4):876–892, 1988.

[28] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proc. ACM SPAA*, pages 938–948, 2010.

[29] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, 16(4/5):517–542, 1996.

[30] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson. Distributed computation of large-scale graph problems. In *Proc. ACM SODA*, pages 391–410, 2015.

[31] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, 1997.

[32] C. Konrad, S. V. Pemmaraju, T. Riaz, and P. Robinson. The complexity of symmetry breaking in massive graphs. In *Proc. DISC*, volume 146, pages 26:1–26:18, 2019.

[33] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *Proc. ACM SPAA*, pages 85–94, 2011.

[34] C. Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proc. ACM PODC*, pages 42–50, 2013.

[35] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, jun 2014.

[36] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[37] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[38] M. Luby. Removing randomness in parallel computation without a processor penalty. *J. Comp. and System Sciences*, 47(2):250–286, 1993.

[39] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[40] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD*, pages 135–146, 2010.

[41] G. L. Miller, R. Peng, A. Vladu, and S. C. Xu. Improved parallel algorithms for spanners and hopsets. In *Proc. ACM SPAA*, pages 192–201. ACM, 2015.

[42] G. L. Miller and V. Ramachandran. A new graph triconnectivity algorithm and its parallelization. *Combinatorica*, 12(1):53–76, 1992.

[43] G. Pandurangan, P. Robinson, and M. Scquizzato. Fast distributed algorithms for connectivity and MST in large graphs. In *Proc. ACM SPAA*, pages 429–438, 2016.

[44] G. Pandurangan, P. Robinson, and M. Scquizzato. Fast distributed algorithms for connectivity and MST in large graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(1):4:1–4:22, 2018.

[45] G. Pandurangan, P. Robinson, and M. Scquizzato. On the distributed complexity of large-scale graph computations. In *Proc. ACM SPAA*, pages 405–414, 2018.

[46] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

[47] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction*. Texts and Monographs in Computer Science. Springer, 1985.

[48] J. Qiu, S. Jha, A. Luckow, and G. C. Fox. Towards HPC-ABDS: an initial high-performance big data stack. 2014. Available: http://grids.ucs.indiana.edu/ptliupages/publications/nist-hpc-abds.pdf.

[49] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In J. H. Reif, editor, *Algorithms for Parallel Processing*, volume 105, pages 1 – 18. Morgan Kaufmann Publishers Inc., 1993.

[50] V. Ramachandran and J. H. Reif. Planarity testing in parallel. *J. Comput. Syst. Sci.*, 49(3):517–561, 1994.

[51] A. G. Ranade. How to emulate shared memory. *J. Comput. Syst. Sci.*, 42(3):307–326, 1991.

[52] D. A. Reed and J. Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, 2015.

[53] W. Schudy. Finding strongly connected components in parallel using $O(\log^2 n)$ reachability queries. In *Proc. ACM SPAA*, pages 146–151, 2008.

[54] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.

[55] E. Upfal and A. Wigderson. How to share memory in a distributed system. *J. ACM*, 34(1), January 1987.

[56] D. P. Woodruff and Q. Zhang. When distributed computation is communication expensive. *Distrib. Comput.*, 30(5):309–323, 2017.