Lessons from the Congested Clique Applied to MapReduce

James W. Hegeman and Sriram V. Pemmaraju

Department of Computer Science
The University of Iowa, Iowa City
Iowa 52242-1419, USA

Abstract

The main result of this paper is a simulation algorithm which, under quite general constraints, transforms algorithms running in the Congested Clique model into algorithms running in the MapReduce model. As a case study of applying this simulation algorithm, we first present a distributed $O(\Delta)$-coloring algorithm running on the Congested Clique in $O(1)$ rounds, if $\Delta \geq \log^2 n$, and $O(\log \log \log n)$ rounds otherwise. Applying the simulation theorem to this Congested Clique $O(\Delta)$-coloring algorithm yields an $O(1)$-round $O(\Delta)$-coloring algorithm in the MapReduce model. We apply our simulation algorithm to other Congested Clique algorithms including the 2-ruling set and metric facility location algorithms of Berns et al. (ICALP 2012).

Our simulation algorithm illustrates a natural correspondence between per-node bandwidth in the Congested Clique model and memory per machine in the MapReduce model. In the Congested Clique (and more generally, any network in the $CONGEST$ model), the major impediment to constructing fast algorithms is the $O(\log n)$ restriction on message sizes. Similarly, in the MapReduce model, the combined restrictions on memory per machine and total system memory have a dominant effect on algorithm design. In showing a fairly general simulation algorithm, we highlight the similarities and differences between these models.

1. Introduction

The $CONGEST$ model of distributed computation is a synchronous, message-passing model in which the amount of information that a node can transmit along an incident edge in one round is restricted to $O(\log n)$ bits [33]. As the name suggests, the $CONGEST$ model focuses on congestion as an obstacle to distributed computation. Recently, a fair amount of research activity has focused on the design of distributed algorithms in the $CONGEST$ model assuming that the underlying communication network is a clique. Problems considered in the so-called Congested Clique model include facility location [14, 3], minimum spanning tree (MST) [26, 17, 16], shortest paths and distances [5, 18, 29], triangle finding [9, 8], subgraph detection [9], ruling sets [8, 17], sorting [33, 24], and routing [24]. Working in the Congested Clique model completely removes from the picture obstacles that might be due to nodes having to acquire information from distant nodes (since any two nodes are neighbors), thus allowing us to focus on the issue of congestion alone. Making this setting intriguing is also the fact that no non-trivial lower bounds for computation on the Congested Clique have been proved. In fact, in a recent paper, Lenzen [24] showed how to do load balancing deterministically so as to route up to $n^2$ messages (each of size $O(\log n)$) in $O(1)$ rounds in the Congested Clique setting, provided each node is the source of at most $n$ messages and the destination for at most $n$ messages. Thus a large volume of information can be moved around the network very quickly and any lower-bound approach in the Congested Clique setting will have to work around Lenzen’s routing-protocol result. Even more recently, Drucker et al. [10] showed that almost any non-trivial lower bound

\footnote{This work is supported in part by National Science Foundation grant CCF 1318166.}

E-mail: [james-hegeman,sriram-pemmaraju]@uiowa.edu.

Preprint submitted to Elsevier September 19, 2015

© 2015. This manuscript version is made available under the Elsevier user license http://www.elsevier.com/open-access/userlicense/1.0/
in the Congested Clique setting would imply lower bounds in circuit complexity, resolving long-standing open questions. While Lotker et al. [27] mention overlay networks as a possible practical application of distributed computation on a Congested Clique, as of now, research on this model is largely driven by a theoretical interest in exploring the limits imposed by congestion.

MapReduce [7] is a tremendously popular parallel-programming framework that has become the tool of choice for large-scale data analytics at many companies such as Amazon, Facebook, Google, Yahoo!, etc., as well as at many universities. While the actual time-efficiency of a particular MapReduce-like implementation will depend on many low-level technical details, Karloff et al. [20] have attempted to formalize key constraints of this framework to propose a MapReduce model and an associated MapReduce complexity class (MRC). Informally speaking, a problem belongs to MRC if it can be solved in the MapReduce framework using: (i) a number of machines that is substantially sublinear in the input size, i.e., \(O(n^{1-\epsilon})\) for constant \(\epsilon > 0\), (ii) memory per machine that is substantially sublinear in the input size, (iii) \(O(\text{poly}(\log n))\) number of map-shuffle-reduce rounds, and (iv) polynomial-time local computation at each machine in each round. Specifically, a problem is said to be in MRC\(^0\) if it can be solved in \(O(\log^2 n)\) map-shuffle-reduce rounds, while maintaining the other constraints mentioned above. Karloff et al. [20] show that minimum spanning tree (MST) is in MRC\(^0\) (i.e., MST requires \(O(1)\) map-shuffle-reduce rounds) on non-sparse instances. Following up on this, Lattanzi et al. [23] show that other problems such as maximal matching (with which the distributed computing community is very familiar) are also in MRC\(^0\) (again, for non-sparse instances). We give a more-detailed description of the MapReduce model in Section 1.1.

The volume of communication that occurs in a Shuffle step can be quite substantial and provides a strong incentive to design algorithms in the MapReduce framework that use very few map-shuffle-reduce steps. As motivation for their approach (which they call filtering) to designing MapReduce algorithms, Lattanzi et al. [23] mention that past attempts to “shoehorn message-passing style algorithms into the framework” have led to inefficient algorithms. While this may be true for distributed message-passing algorithms in general, we show in this paper that algorithms designed in the Congested Clique model provide many lessons on how to design algorithms in the MapReduce model. We illustrate this by first designing an expected-\(O(1)\)-round algorithm for computing a \(O(\Delta)\)-coloring for a given \(n\)-node graph with maximum degree \(\Delta \geq \log^2 n\) in the Congested Clique model. We then simulate this coloring algorithm in the MapReduce model and obtain a corresponding algorithm that uses a constant number of map-shuffle-reduce rounds to compute an \(O(\Delta)\)-coloring of the given graph. While both of these results are new, what we wish to emphasize in this paper is the simulation of Congested Clique algorithms in the MapReduce model. Our simulation can also be used to obtain efficient MapReduce-model algorithms for other problems such as 2-ruling sets [3] for which an expected-\(O(\log \log n)\)-round algorithm on a Congested Clique was recently developed.

1.1. Models

The Congested Clique Model. The Congested Clique is a variation on the CONGEST model. The underlying communication network is a size-\(n\) clique, i.e., every pair of nodes can directly communicate with each other. Computation proceeds in synchronous rounds and in each round a node (i) receives all messages sent to it in the previous round; (ii) performs unlimited local computation; and then (iii) sends a, possibly distinct, message of size \(O(\log n)\) to each other node in the network. We assume that nodes have distinct IDs that can each be represented in \(O(\log n)\) bits. We call this the Congested Clique model.

Our focus in this paper is graph problems and we assume that the input is a graph \(G\) that is a spanning subgraph of the communication network. Initially, each node in the network knows who its neighbors are in \(G\). Thus knowledge of \(G\) is distributed among the nodes of the network, with each node having a particular local view of \(G\). Note that \(G\) can be quite dense (e.g., have \(\Omega(n^2)\) edges) and therefore any reasonably fast algorithm for the problem will have to be “truly” distributed in the sense that it cannot simply rely on shipping off the problem description to a single node for local computation.

The MapReduce Model. Our description of the MapReduce model borrows heavily from the work of Karloff et al. [20] and Lattanzi et al. [23]. Introduced by Karloff et al. [20], the MapReduce model is an abstraction
of the popular MapReduce framework [7] implemented at Google and also in the popular Hadoop open-source project by Apache.

The basic unit of information in the MapReduce model is a \((key, value)\)-pair. At a high level, computation in this model can be viewed as the application of a sequence of functions, each taking as input a collection of \((key, value)\)-pairs and producing as output a new collection of \((key, value)\)-pairs. MapReduce computation proceeds in rounds, with each round composed of a map phase, followed by a shuffle phase, followed by a reduce phase. In the map phase, \((key, value)\) pairs are processed individually and the output of this phase is a collection of \((key, value)\)-pairs. In the shuffle phase, these \((key, value)\)-pairs are “routed” so that all \((key, value)\)-pairs with the same key end up together. In the last phase, namely the reduce phase, each key and all associated values are processed together. We next describe each of the three phases in more detail.

- The computation in the Map phase of round \(i\) is performed by a collection of mappers, one per \((key, value)\) pair. In other words, each mapper takes a \((key, value)\) pair and outputs a collection of \((key, value)\) pairs. Since each mapper works on an individual \((key, value)\) pair and the computation is entirely “stateless” (i.e., not dependent on any stored information from previous computation), the mappers can be arbitrarily distributed among machines. In the MapReduce model, keys and values are restricted to the word size of the system, which we assume is \(\Theta(\log n)\). Because of this restriction, a mapper takes as input only a constant number of words.

- In the Shuffle phase of round \(i\), which runs concurrently with the Map phase (as possible), key-value pairs emitted by the mappers are moved from the machine that produced them to the machine which will run the reducer for which they are destined; i.e., a key-value pair \((k, v)\) emitted by a mapper is physically moved to the machine which will run the reducer responsible for key \(k\) in round \(i\). The Shuffle phase is implemented entirely by the underlying MapReduce framework and we generally ignore the Shuffle phase and treat data movement from one machine to another as a part of the Map phase.

- In the Reduce phase of round \(i\), reducers operate on the collected key-value pairs sent to them; a reducer is a function taking as input a pair \(\{(k, \{v_{k,j}\}_j)\}_i\), where the first element is a key \(k\) and the second is a multiset of values \(\{v_{k,j}\}_j\) which comprises all of the values contained in key-value pairs emitted by mappers during round \(i\) and having key \(k\). Reducers emit a multiset of key-value pairs \(\{(k, v_{k,l})\}_l\), where the key \(k\) in each pair is the same as the key \(k\) of the input.

For our purposes, the concepts of a machine and a reducer are interchangeable, because reducers are allowed to be “as large” as a single machine on which they compute.

The MapReduce model of Karloff et al. [20] tries to make explicit three key resource constraints on the MapReduce system. Suppose that the problem input has size \(n\) (note that this is not referring to the input size of a particular reducer or mapper). We assume, as do Karloff et al. [20] and Lattanzi et al. [23], that memory is measured in \(O(\log n)\)-bit-sized words.

1. Key-sizes and value-sizes are restricted to a \(\Theta(1)\) multiple of the word size of the system. Because of this restriction, a mapper takes as input only a constant number of words.

2. Both mappers and reducers are restricted to using space consisting of \(O(n^{1-\epsilon})\) words of memory, and time which is polynomial in \(n\).

3. The number of machines, or equivalently, the number of reducers, is restricted to \(O(n^{1-\epsilon})\).

Given these constraints, the goal is to design MapReduce algorithms that run in very few – preferably constant – number of rounds. For further details on the justifications for these constraints, see [20].
Since our focus is graph algorithms, we can restate the above constraints more specifically in terms of graph size. Suppose that an \( n \)-node graph \( G = (V, E) \) is the input. Following Lattanzi et al. \[23\], we assume that each machine in the MapReduce system has memory \( \eta = n^{1+\epsilon} \) for \( \epsilon \geq 0 \). Since \( n^{1+\epsilon} \) needs to be “substantially” sublinear in the input size, we assume that the number of edges \( m \) of \( G \) is \( \Omega(n^{1+c}) \) for \( c > \epsilon \). Thus the MapReduce results in this paper are for non-sparse graphs.

1.2. Contributions

The main contribution of this paper is to show that fast algorithms in the Congested Clique model can be translated via a simulation theorem into fast algorithms in the MapReduce model. As a case study, we design a fast graph-coloring algorithm running in the Congested Clique model and then apply the simulation theorem to this algorithm and obtain a fast MapReduce algorithm. Specifically, given an \( n \)-node graph \( G \) with maximum degree \( \Delta \geq \log^2 n \), we show how to compute an \( O(\Delta) \)-coloring of \( G \) in \( O(1) \) rounds (with high probability) in the Congested Clique model. We also present an algorithm for small \( \Delta \); for \( \Delta < \log^2 n \) we present an algorithm that computes a \( \Delta + 1 \) coloring in \( O(\log \log \log n) \) rounds with high probability on a Congested Clique. The implication of this result to the MapReduce model (via the simulation theorem) is that for any \( n \)-node graph with \( \Omega(n^{1+c}) \) edges, for constant \( c > 0 \), there is a MapReduce algorithm that runs in \( O(1) \) map-shuffle-reduce rounds using \( n^{1+\epsilon} \) words of memory per machine, for \( 0 \leq \epsilon < c \) and \( n^{\epsilon c^{-\epsilon}} \) machines. Note that even when using \( n \) words of memory per machine and \( n^c \) machines the algorithm still takes \( O(1) \) rounds. This is in contrast to examples in Lattanzi et al. \[23\] such as maximal matching which require \( O(\log n) \) rounds if the memory per machine is \( n \). We also apply our simulation theorem to the algorithms of Berns et al. \[3, 4\] that solve the 2-ruling set and the metric facility location problems in the Congested Clique model.

The coloring algorithms in both models are new and faster than any known in the respective models, as far as we know. However, the bigger point of this paper is the connection between models that are studied in somewhat different communities.

1.3. Related Work

The earliest interesting algorithm in the Congested Clique model is an \( O(\log \log n) \)-round deterministic algorithm to compute a minimum spanning tree (MST), due to Lotker et al. \[27\]. As mentioned at the beginning of the paper, since then a fair amount of research activity has focused on solving problems in the Congested Clique model. Here we briefly review work most relevant to this paper.

Gehweiler et al. \[14\] presented a randomized \( O(1) \)-round algorithm in the Congested Clique model that produced a constant-factor approximation algorithm for the uniform metric facility location problem. Berns et al. \[3, 4\] considered the more-general non-uniform metric facility location in the Congested Clique model and presented a constant-factor approximation running in expected \( O(\log \log n) \) rounds. Berns et al. reduce the metric facility location problem to the problem of computing a 2-ruling set of a spanning subgraph of the underlying communication network and show how to solve this in \( O(\log \log n) \) rounds in expectation.

In 2013, Lenzen presented a routing protocol to solve a problem called an Information Distribution Task \[24\]. The setup for this problem is that each node \( i \in V \) is given a set of \( n^c \) messages, each of size \( O(\log n) \), \( \{m^1_i, m^2_i, \ldots, m^n_i\} \), with destinations \( d(m^j_i) \in V, j \in \{1, 2, \ldots, n^c\} \). Messages are globally lexicographically ordered by their source \( i \), destination \( d(m^j_i) \), and \( j \). Each node is also the destination of at most \( n \) messages. Lenzen’s routing protocol solves the Information Distribution Task in \( O(1) \) rounds.

In PODC 2015, Hegeman et al. \[16\] showed that MST can be solved in \( O(\log \log \log n) \) rounds (with high probability) using a randomized algorithm. As can be expected, this algorithm also solves the Connectivity problem in \( O(\log \log \log n) \) rounds.

Our main sources of reference on the MapReduce model and for graph algorithms in this model are the work of Karloff et al. \[20\] and Lattanzi et al. \[23\] respectively. Besides these, the work of Ene et al. \[12\] on algorithms for clustering in MapReduce model and the work of Kumar et al. \[22\] on greedy algorithms in the MapReduce model are relevant.
2. Coloring on the Congested Clique

In this section we present an algorithm, running in the Congested Clique model, that takes an \( n \)-node graph \( G \) with maximum degree \( \Delta \) and computes an \( O(\Delta) \)-coloring in expected \( O(\log \log \log n) \) rounds. Indeed, for high-degree graphs, i.e., when \( \Delta \geq \log^2 n \), our algorithm computes an \( O(\Delta) \)-coloring in \( O(1) \) rounds with high probability (i.e., probability at least \( 1 - 1/n^c \) for constant \( c \geq 1 \)). This algorithm, which we call Algorithm \textsc{HighDegCol}, is the main contribution of this section. For graphs with maximum degree less than \( \log^2 n \) we appeal to an already-known coloring algorithm that computes a \((\Delta + 1)\) coloring in \( O(\log \Delta) \) rounds and then modify its implementation so that it runs in \( O(\log \log \log n) \) rounds on a Congested Clique.

We first give an overview of Algorithm \textsc{HighDegCol}. The reader is advised to follow the pseudocode given in Algorithm 1 as they read the following. The key step of Algorithm \textsc{HighDegCol} is that each node picks a color group \( k \) from the set \( \{1, 2, \ldots, \lceil \Delta / \log n \rceil \} \) independently and uniformly at random (Step 3). Then each node broadcasts its color group choice to all other nodes (in Step 4) so that any node can identify all neighbors that share its color group. Let \( G_k \) be the graph induced by nodes with color \( k \). Then (in Step 5) each graph \( G_k \) is distributed to a different node for local coloring using Lenzen’s routing scheme [24]. Let \( x_1, x_2, \ldots, x_{\lceil \Delta / \log n \rceil} \) be distinct nodes in \( G \). Then each node \( u \) in color group \( k \) identifies all incident edges to other nodes in color group \( k \) and sends these edges to node \( x_k \). In Lemma 3 we show that the total number of edges in \( G_k \) is \( O(n) \) for all \( k \), with high probability. This ensures that all the prerequisites of Lenzen’s routing scheme are met (with high probability) and it will terminate in \( O(1) \) rounds. Every node that receives a graph induced by a color group, locally computes a proper coloring of the graph using one more color than the maximum degree of the graph it receives (Step 6). Furthermore, every such coloring in an iteration employs a distinct palette of colors.

Algorithm 1 \textsc{HighDegCol}

\textbf{Input:} An \( n \)-node graph \( G = (V, E) \), of maximum degree \( \Delta \)

\textbf{Output:} A proper node-coloring of \( G \) using \( O(\Delta) \) colors

1. Each node \( u \) in \( G \) computes and broadcasts its degree to every other node \( v \) in \( G \).
2. If \( \Delta \leq \log^2 n \) then use Algorithm \textsc{LowDegCol} instead.
3. Each node \( u \) chooses a color group \( k \) from the set \( \{1, 2, \ldots, \lceil \Delta / \log n \rceil \} \) independently and uniformly at random. (Let \( G_k \) be the subgraph of \( G \) induced by nodes of color group \( k \).)
4. Each node \( u \) broadcasts its choice of color group to all other nodes in \( G \).
5. Using Lenzen’s routing protocol, distribute all information about each color-group graph \( G_k \) to distinct nodes of \( G \).
6. For each graph \( G_k \), the recipient of \( G_k \) computes a coloring of \( G_k \) using \( O(\log n) \) colors. The color palette used for each \( G_k \) is distinct.
7. Each node that locally colors a subgraph informs each node in the subgraph the color it has been assigned.

We now analyze Algorithm \textsc{HighDegCol} and show that (i) it terminates in \( O(1) \) rounds with high probability and (ii) it uses \( O(\Delta) \) colors. Subsequently, we discuss an \( O(\log \log \log n) \)-round algorithm to deal with the small \( \Delta \) case.

**Lemma 1.** With high probability, all color groups have at most \( \frac{5n \log n}{\Delta} \) nodes.

**Proof:** The number of color groups is \( \lceil \Delta / \log n \rceil \). Thus, for any \( k \), the expected number of nodes in \( G_k \), denoted \( |V(G_k)| \), is at most \( n \cdot \frac{\log n}{\Delta} \). An application of a Chernoff bound then gives, for each \( k \),

\[
\mathbb{P}\left(|V(G_k)| > 5n \cdot \frac{\log n}{\Delta}\right) \leq 2^{-5n \cdot \frac{\log n}{\Delta}} < 2^{-5 \log n} = \frac{1}{n^5},
\]

Applying the union bound over all \( k \) completes the proof. \( \square \)
**Lemma 2.** With high probability, no node $u$ in $G$ has more than $5\log n$ neighbors in any color group.

**Proof:** Node $u$ has maximum degree $\Delta$, so for any $k$, the expected number of neighbors of $u$ which choose color group $k$ is bounded above by $\log n$. Therefore, applying a Chernoff bound gives

$$P\left(|N(u) \cap G_k| > 5 \log n\right) \leq 2^{-5\log n} = \frac{1}{n^5}$$

Taking the union over all $k$ and $u$ shows that, with probability at least $1 - \frac{1}{n^5}$, the assertion of the lemma holds. \qed

**Lemma 3.** Every color group graph $G_k$ has $O(n)$ edges, with high probability.

**Proof:** Consider a color group graph $G_k$. By Lemma 1, $G_k$ has at most $\frac{5n \log n}{\Delta}$ nodes. By Lemma 2, with high probability, each node in $G_k$ has at most $5 \log n$ neighbors in $G_k$. Thus, with high probability, the total number of edges in $G_k$ is at most $\frac{25n \log^2 n}{2\Delta}$. Since $\Delta > \log^2 n$, this quantity is bounded above by $25n/2$, yielding the result. \qed

**Lemma 4.** Algorithm HIGHDEGCOL runs in a constant number of rounds and uses $O(\Delta)$ colors, with high probability.

**Proof:** Lenzen’s routing protocol can be used (in Line 4) to successfully distribute information about each $G_k$ to a distinct node in constant rounds because the size of every graph $G_k$ is $O(n)$ with high probability, by Lemma 3. All other steps in the algorithm are local computations or communication that involves nodes sending at most one $O(\log n)$-sized message to each neighbor. Thus each step in the algorithm runs in $O(1)$ rounds (Line 4 runs in $O(1)$ rounds with high probability) and therefore the algorithm runs in $O(1)$ rounds with high probability.

A palette of size $O(\log n)$ colors suffices for each color group graph because we showed in Lemma 2 that with high probability the maximum degree in any color group graph is $5 \log n$. Since there are a total of $\lceil \Delta / \log n \rceil$ color groups and we use a palette of size $O(\log n)$ for each color group, we use a total of $O(\Delta)$ colors for all the color groups. \qed

### 2.1. Coloring low-degree graphs

Now we describe an algorithm that we call LOWDEGCOL that, given an $n$-node graph $G$ with maximum degree $\Delta < \log^2 n$, computes a proper $(\Delta + 1)$-coloring with high probability in $O(\log \log \log n)$ rounds in the Congested Clique model. The algorithm has two stages. The first stage of the algorithm is based on the simple, natural randomized coloring algorithm first analyzed by Johansson [19] and more recently by Barenboim et al. [2]. Each node $u$ starts with a color palette $C_u = \{1, 2, \ldots, \Delta + 1\}$. In each iteration, each as-yet uncolored node $u$ makes a tentative color choice $c(u) \in C_u$ by picking a color from $C_u$ independently and uniformly at random. If no node in $u$’s neighborhood picks color $c(u)$, then $u$ colors itself $c(u)$ and $c(u)$ is deleted from the palettes of all neighbors of $u$. Otherwise, $u$ remains uncolored and participates in the next iteration of the algorithm. We call one such iteration RANDCOLSTEP. Barenboim et al. [2] show (as part of the proof of Theorem 5.1) that if we executed $O(\log \Delta)$ iterations of RANDCOLSTEP, then with high probability the nodes that remain uncolored induce connected components of size $O(\text{poly}(\log n))$. Since we are evaluating a situation in which $\Delta < \log^2 n$, this translates to using $O(\log \log n)$ iterations of RANDCOLSTEP to reach a state with small connected components. Now notice that this algorithm uses only the edges of $G$ – the graph being colored – for communication. By utilizing the entire bandwidth of the underlying clique communication network, it is possible to exponentially speed up this algorithm and get it to complete in $O(\log \log \log n)$ rounds (see Theorem 3.1 in [23]). The trick to doing this is to rapidly gather, at each node $u$, all information needed by node $u$ to execute the algorithm locally.
Once we complete executing all $O(\log \log n)$ iterations of RANDCOLSTEP (in $O(\log \log \log n)$ rounds in the Congested Clique model) and all connected components induced by as-yet uncolored nodes become polylogarithmic in size, then Stage 2 of the algorithm begins. To identify the connected components of the graph induced by uncolored nodes, we use the recent $O(\log \log \log \log n)$ connectivity verification algorithm of Hegeman et al. [16]. At the end of this algorithm, every node in a connected component $C$ knows that it belongs to $C$ (for example, by being labeled with the smallest ID of a node in $C$). Then each connected component is gathered at a distinct node via the use of Lenzen’s routing scheme. This is similar to how the graphs $G_k$ were distributed to distinct nodes in Step 5 of Algorithm HIGHDEGCOL. Note that since each connected component has polylogarithmic number of vertices, it also has polylogarithmic number of edges and therefore the prerequisites of Lenzen’s routing scheme are satisfied. Then each connected component is locally (and independently) colored using $\Delta + 1$ colors.

The above description of LOWDEGCOL and accompanying discussion is summarized in the following lemma.

**Lemma 5.** Given an $n$-node graph $G$ with maximum degree $\Delta < \log^2 n$, Algorithm LOWDEGCOL computes a proper $(\Delta + 1)$-coloring in $O(\log \log \log n)$ rounds in the Congested Clique model.

Combining Lemmas 4 and 5 gives the following theorem.

**Theorem 1.** Given an $n$-vertex input graph $G = (V, E)$ with maximum degree $\Delta \geq \log^2 n$, Algorithm HIGHDEGCOL computes an $O(\Delta)$-coloring in $O(1)$ rounds with high probability in the Congested Clique model. For arbitrary $\Delta$, an $O(\Delta)$-coloring can be computed in $O(\log \log \log n)$ rounds with high probability in the Congested Clique model.

3. MapReduce Algorithms from Congested Clique Algorithms

In this section, we prove a simulation theorem establishing that Congested Clique algorithms (with fairly weak restrictions) can be efficiently implemented in the MapReduce model. The simulation ensures that a Congested Clique algorithm running in $T$ rounds can be implemented in $O(T)$ rounds (more precisely, $3 \cdot T + O(1)$ rounds) in the MapReduce model, if certain communication and “memory” conditions are met. The technical details of this simulation are conceptually straightforward, but the details are a bit intricate.

We will now precisely define restrictions that we need to place on Congested Clique algorithms in order for the simulation theorem to hold. We assume that each node in the Congested Clique possesses a word-addressable memory whose words are indexed by the natural numbers. For an algorithm $A_{CC}$ running in the Congested Clique, let $I_u^{(i)} \subset \mathbb{N}$ be the set of memory addresses used by node $u$ during the local computation in round $i$ (not including the sending and receipt of messages).

After the local computation in each round, each node in the Congested Clique may send (or not send) a distinct message of size $O(\log n)$ to each other node in the network. In defining notation, we make a special distinction for the case where a node $u$ sends the same message to every other node $v$ in a particular round; i.e., node $u$ sends a broadcast message. The reason for this distinction is that broadcasts can be handled more efficiently on the receiving end in the MapReduce framework than can distinct messages sent by $u$. Let $m_u^{(i)}$ denote a message sent by node $u$ to node $v$ in round $i$ and let $D_u^{(i)} \subseteq V$ be the set of destinations of messages sent by node $u$ in round $i$. Let $M_u^{(i)} = \{m_{u,v}^{(i)}: v \in D_u^{(i)} \subseteq V\}$ be the set of messages sent by node $u$ in round $i$ of algorithm $A_{CC}$, except let $M_u^{(i)} = \emptyset$ if $u$ has chosen to broadcast a message $b_u^{(i)}$ in round $i$. Similarly, let $\overline{M}_u^{(i)} = \{m_{u,v}^{(i)}: u \in D_v^{(i)} \text{ and } v \text{ is not broadcasting in round } i\}$ be the set of messages received by node $u$ in round $i$, except that we exclude messages $b_v^{(i)}$ from nodes $v$ that have chosen to broadcast in round $i$. We say that $A_{CC}$, running on an $n$-node Congested Clique, is $(K, N)$-lightweight if

(i) for each round $i$ (in the Congested Clique), $\sum_{u \in V}(|\overline{M}_u^{(i)}| + |I_u^{(i)}|) = O(K)$;
(ii) there exists a constant $C$ such that for each round $i$ and for each node $u$, $I_u^{(i)} \subseteq \{1, 2, \ldots, [C \cdot N]\}$; and

(iii) each node $u$ performs only polynomial-time local computation in each round.

In plain language: no node uses more than $O(N)$ memory for local computation during a round; the total amount of memory that all nodes use and the total volume of messages nodes receive in any round is bounded by $O(K)$. Regarding condition (iii), traditional models of distributed computation such as the CONGEST and \texttt{LOCAL} models allow nodes to perform arbitrary local computation (e.g., taking exponential time), but since the MapReduce model requires mappers and reducers to run in polynomial time, we need this extra restriction.

As a point of notation, we note that, in the theorem below, we will “encapsulate” $M^{(i)}_u$ and $\overline{M}^{(i)}_u$ in tuple sets $M^{(i)}_u$ and $\overline{M}^{(i)}_u$, in which a typical tuple has the format $(r, u, v, m^{(i)}_{u,v})$, fully describing the sender, receiver, and message contents (here $r$ is a reducer identifier and $u$ and $v$ are vertices of $G$).

**Theorem 2.** Let $\epsilon, c$ satisfy $0 \leq \epsilon \leq c$, and let $G = (V, E)$ be a graph on $n$ vertices having $O(n^{1+c})$ edges. If $\mathcal{A}_{CC}$ is a $(n^{1+c}, n^{1+c})$-lightweight Congested Clique-model algorithm running on input $G$ in $T$ rounds, then $\mathcal{A}_{CC}$ can be implemented in the MapReduce model with $n_r = n^{c-\epsilon}$ machines and $m_r = \Theta(n^{1+\epsilon})$ (words of) memory per machine such that the implementation runs in $O(T)$ Map-Shuffle-Reduce rounds on $G$.

The simulation that will prove the above theorem contains two stages: the \textit{Initialization} stage and the \textit{Simulation} stage. In the Initialization stage, the input to the MapReduce system is transformed from the assumed format (an unordered list of edges and vertices of $G$) into a format in which each piece of information, be it an edge, node, or something else, that is associated with a node of $G$ is gathered at a single machine. After this gathering of associated information has been completed, the MapReduce system can emulate the execution of the Congested Clique algorithm via repeated application of the Simulation stage. In the Simulation stage, one round of a Congested Clique algorithm is simulated in several steps. First, machines preliminarily simulate (in a “dry run”) the next round of the Congested Clique computation and message-sending/receiving in order to construct a “plan” of how to allocate machine memory to nodes in the subsequent round. Second, the information necessary to construct this plan is aggregated and the next distribution of nodes to machines is determined. Third, the next round of Congested Clique computation and communication is (actually) simulated; because the “plan” has been constructed beforehand, messages can be sent and received between machines (in order to simulate the Congested Clique communication) in such a way that machine memory constraints are not violated.

In order to assist the reader, we split the proof of Theorem 2 into two lemmas – one describing the Initialization stage (Lemma 6), and one describing the Simulation stage (Lemma 7).

**Lemma 6.** Let $\epsilon, c$ satisfy $0 \leq \epsilon \leq c$, and let $G = (V, E)$ be a graph on $n$ vertices having $O(n^{1+c})$ edges. If $\mathcal{A}_{CC}$ is a $(n^{1+c}, n^{1+c})$-lightweight Congested Clique-model algorithm running on input $G$ in $T$ rounds, then there exists a MapReduce algorithm which, after three-and-a-half Map-Shuffle-Reduce rounds (that is, three Map-Shuffle-Reduce rounds plus a fourth Map phase) partitions the nodes of the Congested Clique model to be simulated across the machines of the MapReduce model, and, furthermore, aggregates the information local to each node on the machine to which it has been assigned.

**Proof:** Initialization stage. Input (in this case, the graph $G$) in the MapReduce model is assumed to be presented as an unordered sequence of tuples of the form $(\emptyset, u)$, where $u$ is a vertex of $G$, or $(\emptyset, (u, v))$, where $(u, v)$ is an edge of $G$. The goal of the Initialization stage is to partition the input $G$ among the $n_r$ reducers such that each reducer $r$ receives a subset $P_r \subseteq V$ and all edges $E_r$ incident on nodes in $P_r$ such that $|P_r| + |E_r|$ is bounded above by $O(n^{1+\epsilon})$. This stage can be seen as consisting of two tasks:
(i) every reducer \( r \) learns the degree \( \deg_G(u) \) of every node \( u \) in \( G \); and (ii) every reducer computes a partition (the same one) given by the following partition function \( F_0 : V \rightarrow \{1, 2, \ldots, n_r\} \). Define \( L(x) = \{j : j < x \text{ and } F_0(j) = F_0(x-1)\} \). Then,

\[
F_0(x) = \begin{cases} 
1, & \text{if } x = 1, \\
F_0(x-1), & \text{if } \sum_{v \in L(x)} \deg_G(v) \leq C' \cdot n^{1+\epsilon}, \\
F_0(x-1) + 1, & \text{otherwise}. 
\end{cases}
\]

Here \( C' \) is a constant which is the sum of the two constants of conditions (i) [a constant in front of \( K \) in \( O(K) \)] and (ii) \( [C] \) in the definition of \( (K, N) \)-lightweight. As well, \( L(x) = \{v \colon v < x \text{ and } F_0(j) = F_0(x-1)\} \). All nodes in the same group in the partition are mapped to the same value by \( F_0 \) and will be assigned to a single reducer. Since the degree of each node is bounded above by \( n \), it is easy to see that for any \( r \in \{1, 2, \ldots, n_r\} \), \( F_0^{-1}(r) \) is a subset of nodes of \( G \) such that \( |F_0^{-1}(r)| + \sum_{u \in F_0^{-1}(r)} \deg_G(u) \) is \( O(n^{1+\epsilon}) \).

In other words, the vertices of \( G \) are partitioned among the reducers such that the sum of the degrees of the vertices assigned to each reducer is at most \( 2 \cdot n^{1+\epsilon} \), which allows all of the information (incident edges) associated with nodes \( v \) assigned to reducer \( r \) to fit into the \( \Theta(n^{1+\epsilon}) \) memory of machine \( r \). Then, for each vertex \( u \) received, a reducer \( r \) outputs the tuple \( (r, F(u), u) \), and for each edge \( (u, v) \) received, a reducer \( r \) outputs the tuples \( (r, F_0(u), (u, v)) \) and \( (r, F_0(v), (u, v)) \). Each of the two tasks mentioned above can be implemented in a (small) constant number of MapReduce rounds as follows:

- **Map phase 1:** In Map phase 1, for each tuple \( (\varnothing, u) \), a mapper chooses a random reducer \( r \) and emits the tuple \( (r, u) \). For each tuple \( (\varnothing, (u, v)) \), a mapper again chooses a random reducer \( r \) and emits the tuple \( (r, (u, v)) \). Because the reduce keys are chosen at random, with high probability (actually, the failure probability is exponentially small) each reducer in Reduce phase 1 will receive \( O(n^{1+\epsilon}) \) tuples.

- **Reduce phase 1:** In Reduce phase 1, a reducer \( r \) receives tuples whose values consist of some collection \( P_r \subseteq V \) of vertices and some collection \( E_r \subseteq E \) of edges of \( G \). For each value consisting of a vertex \( u \), a reducer \( r \) re-emits the tuple \( (r, u) \), and for each value consisting of an edge \( (u, v) \), reducer \( r \) re-emits the tuple \( (r, (u, v)) \). In addition, a reducer \( r \) emits, for each vertex \( u \) such that reducer \( r \) received an edge \( (u, v) \) or \( (v, u) \), a tuple \( (r, u, d_{r,u}) \), where \( d_{r,u} \) is the total number of edges received by reducer \( r \) containing \( u \). (In other words, \( d_{r,u} \) is the partial degree of \( u \) seen by reducer \( r \).)

- **Map phase 2:** In Map phase 2, mappers again load-balance tuples containing vertices or edges as values across the reducers uniformly at random (an action which is successful with high probability), as in Map phase 1. In addition, when a mapper processes a tuple of the form \( (r, u, d_{r,u}) \), it emits the tuple \( (u \mod n_r, u, d_{r,u}) \). Here \( (u \mod n_r) \) refers to the reduction of the identifier of node \( u \) modulo the number of reducers \( n_r \). There are at most \( n \cdot n_r = O(n^{1+c-\epsilon}) \) such tuples, and thus (i) each reducer is the destination of \( O(n) \) such tuples [of the form \( (u \mod n_r, u, d_{r,u}) \)]; and (ii) all tuples containing a partial degree sum of node \( u \) among their values are given the same key and thus sent to the same reducer during the second MapReduce round.

- **Reduce phase 2:** In Reduce phase 2, a reducer \( r \) again re-emits tuples \( (r, u) \) and \( (r, (u, v)) \) for each vertex or edge received as a value. For tuples of the form \( (r, u, d_{r,u}) \), reducer \( r \) aggregates the partial degree sums of \( u \) to compute the full degree \( \deg_G(u) \) of \( u \) in \( G \), and emits the tuple \( (r, u, \deg_G(u)) \).

- **Map phase 3:** In Map phase 3, mappers once again load-balance tuples containing vertices or edges as values across the reducers as in Map phases 1 and 2. For each tuple of the form \( (r, u, \deg_G(u)) \), a mapper emits \( n_r \) tuples \( (r_1, u, \deg_G(u)), (r_2, u, \deg_G(u)), \ldots, (r_{n_r}, u, \deg_G(u)) \) — one for each
• **Reduce phase 3:** In Reduce phase 3, a reducer \( r \) now has access to the degrees of all vertices of \( G \) and can thus compute the partition function \( F_0 \) defined earlier. Then, for each node \( u \) received, reducer \( r \) outputs the tuple \( (r, F_0(u), u) \). For each edge \( (u, v) \) received, reducer \( r \) outputs the tuples \( (r, F_0(u), (u, v)) \) and \( (r, F_0(v), (u, v)) \).

In addition to “packaging” the vertex and edge information of \( G \) so that incident edges of a node \( u \) can be collected at the reducer \( F_0(u) \) assigned to simulate computation at \( u \), reducers must also emit tuples which allow both (i) the currently collected degrees of nodes of \( G \) and (ii) the partition function \( F_0 \) to be propagated forward through the rounds of the MapReduce computation. Fortunately this is straightforward: for each degree tuple \( (r, u, \deg_G(u)) \) received by reducer \( r \), reducer \( r \) re-emits the same tuple. As well, \( F_0 : V \to \{1, \ldots, n_r\} \) can be fully described by \( n \) pairs \( (v, F_0(v)) \), and so reducer \( r \) emits the \( n \) tuples \( (r, v, F_0(v)) \), which will allow reducer \( r \) to “remember” the partition function \( F_0(\cdot) \) in the next round. Observe that the totality of the memory required to support the persistence of knowledge of the partition function and all degrees of nodes of \( G \) is \( O(n) \), and thus fits into the memory of a reducer without any trouble.

• **Map phase 4:** Finally, in Map phase 4, a mapper receives and processes two different tuple formats: (i) tuples of the form \( (r, r', z) \), where \( r' \) is another reducer index and \( z \) is some information (of length \( O(1) \) words) representing either a node or an edge; and (ii) tuples of the form \( (r, v, w, z) \), where \( v \) is a node identifier and \( z \) is either a degree value or a reducer identifier. In case (i) [tuples of the form \( (r, r', z) \)] a mapper emits the tuple \( (r', z) \). In case (ii) [tuples of the form \( (r, v, w, z) \)] a mapper simply outputs the same tuple \( (r, v, z) \) unchanged.

After the Map phase of round 4 of the MapReduce computation has completed, the Initialization phase is complete, and the simulation of \( A_{CC} \) is ready to begin.

**Lemma 7.** Let \( \epsilon, c \) satisfy \( 0 < \epsilon \leq c \), and let \( G = (V, E) \) be a graph on \( n \) vertices having \( O(n^{1+c}) \) edges. If \( A_{CC} \) is an \( (n^{1+c}, n^{1+c}) \)-lightweight Congested Clique-model algorithm running on input \( G \) in \( \tau \) rounds, then, after the Initialization stage of Lemma 6, \( A_{CC} \) can be implemented in the MapReduce model with \( n_r = n^{c-\epsilon} \) machines and \( m_r = \Theta(n^{1+c}) \) (words of) memory per machine such that the implementation runs in \( O(T) \) Map-Shuffle-Reduce rounds on \( G \).

**Proof:** **Simulation stage.** At a high level, a Reduce phase serves as the “local computation” phase of the Congested Clique simulation, whereas a Map phase (together with the subsequent shuffle phase) serves as the “communication” phase of the simulation. However, there is, in general, a constant-factor slow-down because it may be that the sending and receiving of messages during \( A_{CC} \) could cause the subset of nodes assigned to a reducer to aggregate more than \( O(n^{1+c}) \) memory, necessitating a re-partitioning of the nodes among the reducers so as not to violate the memory-per-machine constraint.

Recall that \( I_u^{(i)} \) denotes the set of memory addresses used by a node \( u \) in round \( i \) of \( A_{CC} \). Let \( h_u^{(i)} \) be the value of word \( j \in I_u^{(i)} \) in the memory of node \( u \) after node \( u \) has completed local computation in round \( i \) of \( A_{CC} \), but before messages have been sent and received in this round. For \( i > 0 \), define a tuple set

\[
H_u^{(i)} = \left\{ (F_i(u), (u, i, h_u^{(i)})) : j \in I_u^{(i)} \right\},
\]

where \( F_i(\cdot) \) is the partition function used in the simulation of round \( i \) [of \( A_{CC} \)]. Like \( F_0 \), defined in the Initialization stage, \( F_i \) partitions \( G \) into \( n_r \) groups, one per reducer, so that reducer memory constraints are not violated during the simulation of round \( i \). The collection of tuples \( H_u^{(i-1)} \) is a representation, in the MapReduce key-value format, of the information necessary to simulate the computations of node \( u \) in
round $i$ of the Congested Clique algorithm $\mathcal{A}_{CC}$. The use of $F_i(u)$ as the key in each of the tuples in $\mathcal{H}_u^{(i)}$ ensures that all information needed to simulate a local computation at $u$ in $\mathcal{A}_{CC}$ goes to the same reducer. Additionally, note that the inclusion of the identifier of $u$ with the values allows the words from $u$’s memory to be reassembled and distinguished from information associated with other nodes $v \in F_{i-1}^{-1}(u)$. As an example, $\mathcal{H}_u^{(0)} = \{(F_0(u), u) \cup \{(F_0(u), (u, v)) : \text{vis a neighbor of } u\}$.

Once an initial partition function $F_0(\cdot)$ has been computed and the initial collections $\mathcal{H}_u^{(0)}$ have been assembled, the main goals of our simulation algorithm are to (i) provide a mechanism for transforming $\mathcal{H}_u^{(i-1)}$ into $\mathcal{H}_u^{(i)}$ during the reduce phase of a MapReduce round; and (ii) provide a means of transmitting messages to reducers of a subsequent round (corresponding to messages transmitted in the Congested Clique at the end of each round). Since we assume messages to be sent and received after local computation has occurred during a Congested Clique round, $\mathcal{M}_u^{(i)}$ can be determined from $\mathcal{H}_u^{(i)}$; in turn, $\mathcal{H}_u^{(i)}$ is a function of $\mathcal{H}_u^{(i-1)}$ and $\mathcal{M}_u^{(i-1)}$.

We describe the details of the simulation of a single round (round $i$) of a Congested Clique algorithm $\mathcal{A}_{CC}$ below. Let $j = 3i + 2$. Round $i$ of $\mathcal{A}_{CC}$ is simulated by three MapReduce rounds (a total of six Map or Reduce phases) – Reduce $j - 1$, Map $j$, Reduce $j$, Map $j + 1$, Reduce $j + 1$, and Map $j + 2$. We assume inductively that as input to Reduce phase $j - 1$ below, each reducer receives, in addition to data tuples, $O(n)$ metadata tuples containing a description of a partition function $F_{i-1}(\cdot)$ such that for each $r$,

$$\sum_{u \in P_r} \left( |\mathcal{H}_u^{(i-1)}| + |\mathcal{M}_u^{(i-1)}| \right) = O(n^{1+\epsilon})$$

where $P_r = F_{i-1}^{-1}(r)$.

- **Reduce phase $j - 1$**: The purpose of the “first” Reduce phase in the Simulation stage is to “profile” the memory usage of the next round of computation/communication for the Congested Clique algorithm being simulated. This profiling is necessary before the actual simulation steps can be performed because the information gathered is used to ensure (later) that machine memory constraints are not violated. In Reduce phase $j - 1$, a reducer $r$ receives input consisting of $\mathcal{H}_u^{(i-1)}$ together with $\mathcal{M}_u^{(i-1)}$ for each $u \in P_r$; for each such $u$, reducer $r$ performs the following steps:

  (i) Reducer $r$ simulates the local computation of round $i$ of $\mathcal{A}_{CC}$ at $u$.

  (ii) Reducer $r$ computes $\mathcal{H}_u^{(i)}$ from $\mathcal{H}_u^{(i-1)}$ and $\mathcal{M}_u^{(i-1)}$, but does not yet output any tuples of $\mathcal{H}_u^{(i)}$; rather, reducer $r$ outputs only a tuple $(r, u, s_u)$ containing the size $s_u$ of the information; $s_u = |\mathcal{H}_u^{(i)}|$.

  (iii) Reducer $r$ computes $\mathcal{M}_u^{(i)}$ from $\mathcal{H}_u^{(i)}$, but again, does not output any elements of $\mathcal{M}_u^{(i)}$. Rather, reducer $r$ computes $\mathcal{M}_u^{(i)}$ solely for the purpose of keeping track of, for each $v \in V$ and $u \in P_r$, the number of messages in $\mathcal{M}_u^{(i)}$ destined for $v$ (for each $u$, this number is either 1 or 0). Reducer $r$ computes, for each $v \in V$, the aggregate count $c_{r,v}$ of messages emanating from nodes in $P_r$ and destined for $v$, and outputs the tuple $(r, v, c_{r,v})$.

  (iv) Additionally, reducer $r$ also outputs the exact same tuples it received as input, $\mathcal{H}_u^{(i-1)}$ and $\mathcal{M}_u^{(i-1)}$.

By assumption, the union over $u \in P_r$ of the sets output in (iv) requires only $O(n^{1+\epsilon})$ memory on the machine running reducer $r$; as well, reducer $r$ emits $n$ tuples of the form $(r, v, c_{r,v})$ and $O(n)$ tuples of the form $(r, u, s_u)$.

- **Map phase $j$**: Before message tuples can be generated and aggregated (as a collection $\mathcal{M}_u^{(i)}$ at a reducer $F_i(u)$) a rebalancing of the nodes to reducers must be performed to ensure that the reducer-
memory constraint is not violated. In Map phase \( j \), a mapper forwards tuples from either a \( \mathcal{H}_u^{(i-1)} \) or a \( \mathcal{M}_u^{(i-1)} \) through unchanged. However, for each tuple of the form \((r, u, c_{r,u})\), a mapper outputs the tuple \((u \mod n_r, u, c_{r,u})\). In addition, for each tuple of the form \((r, u, s_u)\), a mapper outputs \( n_r \) tuples \((r', u, s_u)\) – one for each reducer \( r' \) – so that every reducer can know the future size of \( \mathcal{H}_u^{(i)} \).

- **Reduce phase \( j \):** In Reduce phase \( j \), a reducer \( r \) receives as input nearly the exact same input (and output) of reducer \( r \) in the previous MapReduce round – the union of \( \mathcal{H}_u^{(i-1)} \) and \( \mathcal{M}_u^{(i-1)} \) for each \( u \in P_r \) – except that instead of receiving tuples of the form \((r, u, c_{r,u})\) for each \( u \in V \), reducer \( r \) receives all partial message counts for the subset of vertices \( u \) for which \( u \mod n_r = r \); as well, each reducer receives \( n \) tuples of the form \((r, u, s_u)\) describing the amount of memory required by node \( u \) in round \( i \) of \( A_{CC} \). Reducer \( r \) aggregates tuples of the form \((u \mod n_r, u, c_{r,u})\) and outputs \((r, u, |\mathcal{M}_u^{(i)}|)\), since \(|\mathcal{M}_u^{(i)}|\) is precisely the sum of the partial message counts \( c_{r,u} \). (Notice that a reducer \( r \) receives \( O(n) \) such tuples.) Reducer \( r \) forwards all other tuples through unchanged to the next MapReduce round.

- **Map phase \( j+1 \):** In Map phase \( j+1 \), a mapper continues to forward all tuples through unchanged to Reduce phase \( j+1 \), except that for each tuple of the form \((r, u, |\mathcal{M}_u^{(i)}|)\), a mapper outputs \( n_r \) tuples \((r', u, |\mathcal{M}_u^{(i)}|)\) – one for each reducer \( r' \). In this way, each reducer in Reduce phase \( j+1 \) can come to know all \( n \) message counts for each node \( u \in V \).

- **Reduce phase \( j+1 \):** In Reduce phase \( j+1 \), each reducer receives all \( n \) message counts (for each node \( u \in V \)) in addition to the sizes \( s_u \) of the state needed by each node \( u \) in round \( i \) of \( A_{CC} \). Each reducer thus has enough information to determine the next partition function \( F_i : V \rightarrow \{1, \ldots, n_r\} \), defined by

\[
F_i(x) = \begin{cases} 
1, & \text{if } x = 1, \\
F_i(x-1), & \text{if } \sum_{v \in L(x)} (s_v + |\mathcal{M}_v^{(i)}|) \leq C' \cdot n^{1+\epsilon}, \\
F_i(x-1) + 1, & \text{otherwise.}
\end{cases}
\]

As before, \( C' \) is a constant (the sum of the constants in the definition of \((K, N)\)-lightweight) and \( L(x) = \{ v : v < x \text{ and } F_i(v) = F_i(x-1) \} \). After determination of the new partition function \( F_i \), reducers are now able to successfully output the “packaged memory” \( \mathcal{H}_u^{(i)} \) of round \( i \) of \( A_{CC} \), as well as the new messages \( m_{u,v}^{(i)} \) sent in round \( i \), because the new partition function \( F_i \) is specifically designed to correctly load-balance these tuple sets across the reducers while satisfying the memory constraint. Therefore:

(i) Reducer \( r \) now simulates the local computation at each \( u \in P_r \) and thus outputs the set \( \mathcal{H}_u^{(i)} \) (which can be computed from \( \mathcal{H}_u^{(i-1)} \) and \( \mathcal{M}_u^{(i-1)} \)). It is important to recall here that because mappers operate on key-value pairs one at a time in the MapReduce model, there is no restriction on the size of the output from any reducer \( r \) in any MapReduce round, other than that it be polynomial \([20]\). Therefore, a reducer \( r \) may output (and thus free-up its memory) each tuple set \( \mathcal{H}_u^{(i)} \) as it is created (as reducer \( r \) processes the nodes in \( P_r \) one at a time), and so there is no concern about reducer \( r \) attempting to maintain in memory all sets \( \mathcal{H}_u^{(i)} \) for \( u \in P_r \) at once. Note that \( \mathcal{H}_u^{(i)} \), as generated by a reducer \( r \), should contain tuples of the form \((r, F_i(u), u, h_{u,l})\) so that mappers in MapReduce round \( j+2 \) can correctly deliver \( \mathcal{H}_u^{(i)} \) to reducer \( F_i(u) \). Recall that \( h_{u,l}^{(i)} \) denotes the content of the word with address \( l \) in node \( u \)'s memory at the end of local computation in round \( i \).

(ii) As a reducer \( r \) processes, and simulates the computation at, each node \( u \in P_r \) one at a time, generating \( \mathcal{H}_u^{(i)} \), reducer \( r \) also uses \( \mathcal{H}_u^{(i)} \) to generate the messages \( M_{u,v}^{(i)} \) to be sent by node \( u \)
in round $i$ of $A_{CC}$. Reducer $r$ encapsulates $M_u^{(i)}$ in the tuple set $M_u^{(i)}$ and outputs it alongside $H_u^{(i)}$ before moving on to the next node in $P_r$. As with $H_u^{(i)}$, tuples in $M_u^{(i)}$ should initially be generated by a reducer $r$ in the form $(r, F_i(v), u, v, m_u^{(i)}, v)$ so that mappers in MapReduce round $j + 2$ can correctly deliver the set $M_v^{(i)}$ to reducer $F_i(v)$.

(iii) Lastly, regarding the simulation procedure, whenever a node $u \in P_r$ is simulated broadcasts a message $b_u^{(i)}$, reducer $r$ outputs the tuple $(r, u, b_u^{(i)})$.

(iv) After simulation of each node $u \in P_r$ is complete, reducer $r$ also outputs a description of the new partition function $F_i$.

- **Map phase** $j + 2$: In Map phase $j + 2$, a mapper simply transforms the key in a data tuple as appropriate: for each tuple $(r, F_i(u), u, h_u^{(i)})$, a mapper simply emits the tuple $(F_i(u), u, h_u^{(i)})$; for each tuple $(r, F_i(v), u, v, m_u^{(i)}, v)$, a mapper simply emits the tuple $(F_i(v), u, v, m_u^{(i)}, v)$. The exception to this is that tuples $(r, u, b_u^{(i)})$ containing broadcast messages are expanded: for each, a mapper emits $n_r$ tuples $(r', u, b_u^{(i)})$—one for each reducer $r'$—so that every reducer in Reduce phase $j + 2$ receives a single copy of each message broadcast during round $i$ of $A_{CC}$.

- Tuples carrying metadata describing the (new) partition function $F_i$ are forwarded unchanged, because there already exists one copy of each such metadata tuple for each reducer, and there need be only one such copy per reducer as well. After Map phase $j + 2$, tuples from the sets $H_u^{(i)}$ and $M_u^{(i)}$ have been emitted with keys $F_i(u)$, and for each broadcast message $b_u^{(i)}$, one tuple containing a copy of $b_u^{(i)}$ has been emitted for each reducer as well; thus, in Reduce phase $j + 2$, simulation of round $i + 1$ of algorithm $A_{CC}$ can begin.

It remains to comment on the memory-per-machine constraint which must be satisfied during each MapReduce round. Observe that, inductively, for each $r$, the sum $\sum_{u \in P_r} (|H_u^{(i-1)}| + |M_u^{(i-1)}|) = O(n^{1+\epsilon})$. These data tuples are forwarded unchanged until Reduce phase $j + 1$, in which the new partition function $F_i(\cdot)$ for the next round of simulation is computed, and then collectively $H_u^{(i-1)}$ and $M_u^{(i-1)}$ are transformed into $H_u^{(i)}$ and $M_u^{(i)}$. By construction of the partition functions $F_{i-1}$ and $F_i$, and by the assumption that $A_{CC}$ is a $(n^{1+\epsilon}, n^{1+\epsilon})$-lightweight algorithm, it follows that these data tuples are never present on any reducer in a number that exceeds $\Theta(n^{1+\epsilon})$. Secondly, it should be mentioned that because broadcast messages are not duplicated at any reducer $r$, no reducer will ever receive more than $n = O(n^{1+\epsilon})$ such broadcast messages. Third, tuples containing state or message counts are never present in a number exceeding $n$ at any reducer, and partial message counts are explicitly load-balanced so that only $O(n)$ such information is passed to a single reducer as well. Finally, metadata tuples describing a partition function never exceed $\Theta(n)$ on any reducer because the domain of each partition function has size $n$.

4. Algorithms in the MapReduce Framework

Using the simulation theorem of Section 3, we can simulate several recent Congested Clique algorithms, including Algorithm HIGHDEGCOL of Section 2 in the MapReduce model and thereby achieve corresponding fast MapReduce algorithms. In addition to our $O(\Delta)$-coloring algorithm of Section 2, we discuss the 2-ruling set algorithm and metric facility location approximation algorithm of Berns et al. [3,4]. As in Lattanzi et al. [23], we consider graphs with $\Omega(n^{1+\epsilon})$ edges, $\epsilon > 0$.

4.1. Coloring

From Algorithm HIGHDEGCOL of Section 2, we get an $O(\Delta)$-coloring MapReduce algorithm running in $O(1)$ rounds with high probability. Before proving this assertion, however, we first make a preliminary observation related to Lenzen’s routing protocol and the Congested Clique routing problem [24].

13
Theorem 3. The routing problem solved by Lenzen’s routing protocol [24] can be solved in the MapReduce model with \( n_r = n^{c-\epsilon} \) machines and \( m_r = \Theta(n^{1+\epsilon}) \) (words of) memory per machine in three Reduce phases and three Map phases if the total number of messages to be sent/received is \( O(n^{1+c}) \).

Proof: If the total number of messages to be sent/received is \( O(n^{1+c}) \), then the conditions for \((n^{1+c}, n)\)-lightweightness are satisfied in the sense that

(i) \( \sum_{u \in V}(|M_u| + |I_u|) = O(n^{1+c}) \);

(ii) there exists a constant \( C \) such that for each node \( u \), \( I_u \subseteq \{1, 2, \ldots, [C \cdot n]\} \); and

(iii) each node \( u \) performs only polynomial-time local computation.

Thus, to solve the routing problem (as does Lenzen’s routing protocol), we consider that, at the end of either the Initialization stage or a Simulation stage of our Congested Clique emulation in the MapReduce model, Mappers emit tuples so that each Reducer will possess knowledge corresponding to some subset of nodes of \( G \). Therefore, each Reducer \( r \) can locally determine the number of messages \( q_{r,v} \) it possesses destined for the Reducer managing node \( v \). Reducers forward this size information \( q_{r,v} \) to the next phase of Mappers, and the Mappers load-balance it back across the Reducers (so that the Reducer managing node \( v \) receives all numbers \( q_{r,v} \), for each \( r \) – in the same manner as information about \( |M_u| \) in a Simulation stage). In the next (second) Reduce phase, Reducers can thus compute the total number of messages destined for each node \( v \). Mappers then distribute this information globally to all Reducers, allowing the Reducers to compute a partition function, similar to \( F_i \), partitioning the nodes across the Reducers so as not to violate the memory-per-machine constraint upon receipt of messages. In the same (third) Reduce phase, then, each Reducer is now able to output the messages sent by nodes it is currently managing, with their correct intended destinations (i.e., which Reducers will be managing the destination nodes in the next stage). Mappers in the final round are able to forward these message tuples appropriately to Reducers (for the next Simulation stage) such that the memory-per-machine constraint is respected, and then an ongoing simulation (run via the simulation theorem) may be restarted, or another round of message routing may be performed. \( \square \)

Theorem 4. When the input graph \( G \) has \( \Omega(n^{1+c}) \) edges, and \( 0 \leq \epsilon < c \), there exists an \( O(\Delta) \)-coloring algorithm running in the MapReduce model with \( \Theta(n^{c-\epsilon}) \) machines and \( \Theta(n^{1+c}) \) memory per machine, and having a running time of \( O(1) \) rounds with high probability.

Proof: It is easy to examine the lines of code in Algorithm HIGHDEGCOL to ascertain that the total amount of non-broadcast communication in any round is bounded above by \( O(n^{1+c}) \). The one exception to this is in the use of Lenzen’s routing protocol in Line 11; for all other lines of code, the volume of total non-broadcast communication is bounded by \( O(n) \). To handle the simulation of the line using Lenzen’s routing protocol, we actually depart slightly from our simulation theorem.

Instead of attempting to apply the simulation theorem to Line 11, we do the following: Since there are \( O(n^{1+c}) \) edges in \( G \), the union of all color groups therefore requires only \( O(n^{1+c}) \) information to fully describe. We therefore perform the following direct simulation of Line 11 in MapReduce: First, “pause” the simulation being performed via the simulation theorem – while “paused”, each Reducer \( r \) outputs its current knowledge as tuples with Reduce key \( r \) (thus identifying this information as belonging to Reducer \( r \)), and each Mapper simply forwards each tuple received to the Reducer having the same index as the key. This procedure accomplishes the goal of maintaining the current state of the simulation being run via the simulation theorem.

Next, by Theorem\[3\] we can simulate Line 11 in a constant number of six-phase, Reduce-Map-Reduce-Map-Reduce-Map stages which are similar to a Simulation stage as described previously, but involve no algorithmic emulation (in a computational sense) – rather only message routing. At the end of such a constant number of MapReduce phases, the simulation can be restarted, and Reducers can compute colorings.
of the color-group graphs for which they are responsible – for each \( u \) in a color-group graph assigned to Reducer \( r \), Reducer \( r \) can then emit a tuple \((r, u, c(u))\) informing \( u \) of its color which can be routed to the Reducer responsible for \( u \).

Finally, it is easy to examine the lines of code in Algorithm \textsc{HighDegCol} to verify that the total memory (in words) used by all nodes for their local computations in any one round is bounded above by \( O(n^{1+\epsilon}) \). Finally, it is also easy to verify that the maximum amount of memory used by a node in any round of computation is \( O(n) \).

Thus, Algorithm \textsc{HighDegCol} is an \((n^{1+\epsilon}, n)\)-lightweight algorithm on the Congested Clique (with the exception of the Lenzen’s routing protocol step simulated in the preceding paragraph), and applying the Simulation Theorem (Theorem 2) to this algorithm yields the claimed result. \( \square \)

It is worth emphasizing that the result holds even when \( \epsilon = 0 \); in other words, even when the per-machine memory is \( O(n) \), the algorithm can compute an \( O(\Delta) \)-coloring in \( O(1) \) rounds. This is in contrast with the results of Lattanzi et al. \cite{Lattanzi2018}, where \( O(1) \)-round algorithms were obtained (e.g., for maximal matching) with \( n^{1+\epsilon} \) per-machine memory only when \( \epsilon > 0 \). In their work, setting \( \epsilon = 0 \) (i.e., using \( \Theta(n) \) memory per machine) resulted in \( O(\log n) \)-round algorithms.

One can also note that the simulation theorem of Section 3 generalizes naturally to settings when \( \Delta = o(n^c) \) for all \( \epsilon > 0 \); say, \( \Delta = \log n \). The critical simulation and load-balancing steps are possible as long as the total system memory is \( \Omega(|E[G]|) \) and each machine has a memory of size \( \Omega(n) \) words. In order to maintain consistency with previously used notation, we did not state Theorem 2 in this full generality.

The following corollary is an immediate consequence of Theorem 4.

**Corollary 5.** The problem of computing an \( O(\Delta) \)-coloring for an \( n \)-node graph with maximum degree \( \Delta \) and at least \( \Omega(n^{1+c}) \) edges, for \( c > 0 \), is in \( \mathcal{MRC}^0 \).

### 4.2. Metric Facility Location and Ruling Sets

As a second example of an application of the Simulation Theorem to derive a MapReduce algorithm from a Congested Clique algorithm, we consider the 2-ruling set algorithm and the Metric Facility Location algorithm described by Berns et al. \cite{Berns2018}. The metric facility location problem on a Congested Clique is called \textsc{CliqueFacLoc} in \cite{Berns2018} and is defined as follows (as in \cite{Berns2018}). The input to the problem consists of opening costs \( f_i \) associated with each node \( i \), \( 1 \leq i \leq n \) and connection costs \( D(i,j) \) between every pair of nodes \( i \) and \( j \). The connection costs form a metric. The goal is to find a subset \( F \) of nodes (facilities) to open so as to minimize the facility opening costs plus connection costs, i.e.,

\[
\sum_{i \in F} f_i + \sum_{j \not\in F} D(F,j)
\]

where \( D(F,j) := \min_{i \in F} D(i,j) \). Initially, each node \( i \) knows \( f_i \) and connection costs \( D(i,j) \) for all \( j \). Thus the facility has size \( \Theta(n^2) \) that is initially distributed equally among \( n \) machines.

Facility location is a well-studied, classical problem \cite{Aggarwal1987, Berman1994, Chekuri2000, Xiong2006} which recently has been used as an abstraction for various distributed problems as well \cite{Majumdar2015, Bounni2018, Bounni2019, Bounni2020, Bounni2020a}. As we describe in our previous work \cite{Berns2018}, Gehweiler et al. \cite{Gehweiler2020} studied a special case of \textsc{CliqueFacLoc} in which all facilities have the same opening cost, and they present an algorithm that achieves a constant-factor approximation for this setting in a constant number of rounds. Berns et al. \cite{Berns2018} consider the more general problem with arbitrary opening costs and present a constant-factor approximation in the Congested Clique model that runs in expected-\( O(\log \log n) \) rounds. A key step in the Berns et al. \textsc{CliqueFacLoc} algorithm is a Congested Clique algorithm that solves the 2-ruling set problem in expected-\( O(\log \log n) \) rounds. A 2-ruling set of a graph \( G \) is an independent set \( I \) such that every node is within 2 hops of \( I \). A 2-ruling set is a natural generalization of a maximal independent set (which can be viewed as a 1-ruling set) and solving the 2-ruling set problem is of independent interest to the distributed computing community \cite{Bein2014, Xiong2015}. Berns et al. present a
2-ruling set algorithm on an input graph $G$ (a spanning subgraph of the underlying clique network) that runs in expected $O(\log \log n)$ rounds and use this to solve CLIQUEFA-CLOC in expected-$O(\log \log n)$ rounds.

In the following we apply the Simulation Theorem to the 2-ruling set algorithm and to the CLIQUEFA-CLOC algorithm of Berns et al. and derive corresponding MapReduce algorithms. We start with an informal description of the 2-ruling set algorithm; for more details the reader is invited to see Algorithm 3 in Berns et al. [3, 4]. The algorithm proceeds in iterations and in each iteration some number of nodes leave $G$. Progress is measured by the number of edges remaining in $G$, as nodes leave $G$. In an iteration, each node remaining in $G$ joins a “Test” set $T$ independently with probability $q = \sqrt{\frac{n}{m}}$, where $m$ is the number of edges remaining in $G$. The probability $q$ is set such that the expected number of edges in the induced subgraph $G[T]$ is equal to $n$. Nodes then collaborate to determine if indeed the number of edges in $G[T]$ is at most $4n$. This is easy to do in constant rounds. If the number of edges in $G[T]$ is at most $4n$, then $G[T]$ is sent to a single node which computes a maximal independent set $L$ of $G[T]$. $L$ is added to the solution and $T$ and its neighborhood are deleted from $G$. Then the next iteration starts.

**Theorem 6.** When the input graph $G$ has $\Omega(n^{1+c})$ edges, for any $c$ such that $0 \leq c \leq 1$, there exists a 2-ruling set algorithm running in the MapReduce model with $\Theta(n^{c-\epsilon})$ machines and $\Theta(n^{1+c})$ memory per machine, and having an expected running time of $O(\log \log n)$ rounds.

**Proof:** With the exception of computing a maximal independent set of a subgraph having a linear ($O(n)$) number of edges (Line 8 of Algorithm 3 of [4]), all communication in the 2-ruling set algorithm in question is via broadcast. We therefore restrict our attention to the auxiliary subgraph dissemination algorithm (Algorithm 2 of [4]) of Line 8.

In this (fairly trivial) subgraph-dissemination algorithm, most of the communication occurs again via broadcast, except for in Line 4 of Algorithm 2 of [4], in which each node distributes its incident edges “modulo $n$”. To quote, “$x_i$ sends each outgoing edge $e$ to the node $x_j$ of rank $\rho_j = (\ell(e) \mod n)$.” However, since Algorithm 2 is used only for $O(n)$-sized subgraphs, we see that in any round, the total number of non-broadcast messages is $O(n)$.

Finally, it is easy to see that no node needs more than $O(n)$ words of memory to run Algorithms 2 & 3 of [4]. Thus, the 2-ruling set algorithm of Berns et al. is $(n^{1+c}, n^{1+\epsilon})$-lightweight for any $\epsilon$ between 0 and $c$, inclusive. Applying the Simulation Theorem (Theorem [2]) yields the result. □

When $\epsilon$ is a constant strictly greater than 0, and the memory per machine is thus “strictly” super-linear (polynomially), we can obtain the following stronger result.

**Theorem 7.** When the input graph $G$ has $\Omega(n^{1+c})$ edges and $\epsilon$ is a constant such that $0 < \epsilon \leq c$, there exists a 2-ruling set algorithm running in the MapReduce model with $\Theta(n^{c-\epsilon})$ machines and $\Theta(n^{1+c})$ memory per machine, and having an expected running time of $O(1)$ rounds.

**Proof:** For any constant $\epsilon > 0$, only a constant number of iterations of the main loop of Algorithm 3 in [4] are required (in expectation) before the number of edges in the remaining graph falls below $O(n^{1+\epsilon})$. Thus, by the Simulation Theorem, only a constant number of corresponding MapReduce rounds are required (in expectation) to achieve the same intermediate result.

Once the number of edges remaining in the graph is at most $O(n^{1+\epsilon})$, only a constant number of MapReduce rounds are necessary before the computation can complete. (Note that we are departing from the use of the Simulation Theorem at this point.) Specifically, in the next Map phase, the entirety of the information remaining in the tuples can be mapped to each reducer, since this information is of size $O(n^{1+\epsilon})$ and each reducer has that much local storage (in words) available. Then, each reducer (or just a single reducer, depending on the desired format of the output) can complete the computation locally; it is clear, using the naïve algorithm, that a 2-ruling set, or more generally a maximal independent set, can be computed in $O(n^{1+c})$ space when the input graph has size $O(n^{1+c})$. □
Corollary 8. The problem of computing a 2-ruling set for an \( n \)-node graph with at least \( \Omega(n^{1+c}) \) edges, with \( c > 0 \), is in \( \mathcal{MRC}^0 \) (in an expected sense).

The 2-ruling set algorithm of Berns et al. is used as a sub-procedure within their CLIQUEFACLOC algorithm. Recall that the input to this problem is a clique graph with weights (distances) associated with the edges and an opening cost associated with each node (i.e., facility). Thus, the size of the input is \( O(n^2) \) words, and so \( c = 1 \). An examination of Algorithm 1 of [4] reveals that all communication outside of the 2-ruling-set computation is via broadcast, and each node maintains at most \( O(n) \) information (as words) in memory. We thus have the following additional results.

Theorem 9. (i) There exists an \( O(1) \)-approximation algorithm to the CLIQUEFACLOC problem which runs in the MapReduce model with \( \Theta(n) \) machines and \( \Theta(n) \) memory per machine and has an expected running time of \( O(\log \log n) \) rounds. (ii) If \( \epsilon \) is a constant strictly greater than 0 (and at most 1), there exists an \( O(1) \)-approximation algorithm to CLIQUEFACLOC which runs in the MapReduce model with \( \Theta(n^{1-\epsilon}) \) machines and \( \Theta(n^{1+\epsilon}) \) memory per machine and has an expected running time of \( O(1) \) rounds.

Corollary 10. The problem of computing an \( O(1) \)-approximation to the clique metric facility location problem is in \( \mathcal{MRC}^0 \) (in an expected sense).

5. Conclusions

The results in this paper connect two models that are usually studied by different research communities. In general, it would be interesting to see if this connection has benefits beyond those discussed in this paper. Also, it would be interesting to study differences between these two models. For example, the Congested Clique model allows nodes to remember an arbitrary amount of information from one round to the next. Does this give the Congested Clique model a provable advantage over the “stateless” MapReduce model?

For the “small \( \Delta \)” case, i.e., when \( \Delta = O(\text{poly}(\log n)) \), our paper presents an \( O(\log \log \log n) \)-round \( (\Delta + 1) \)-coloring algorithm on the Congested Clique. One question that interests us is whether \( O(1) \) rounds will suffice to compute an \( O(\Delta) \)-coloring even when \( \Delta \) is small?

Following the lead of Lattanzi et al. [23], we have assumed that each machine in the MapReduce model contains at least \( \Omega(n) \) memory for processing an \( n \)-node graph. Relaxing this assumption (for example, when \( \Delta = o(n) \)) is interesting and leads to the question of whether for some \( \epsilon > 0 \), \( O(1) \) MapReduce rounds would suffice to compute an \( O(\Delta) \)-coloring, even when the per-machine memory is \( O(n^{1-\epsilon}) \).

6. References

[1] M. L. Balinski. On finding integer solutions to linear programs. In *Proceedings of IBM Scientific Computing Symposium on Combinatorial Problems*, pages 225—248, 1966.

[2] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 321–330, 2012.

[3] Andrew Berns, James Hegeman, and Sriram V. Pemmaraju. Super-fast distributed algorithms for metric facility location. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *ICALP (2)*, volume 7392 of *Lecture Notes in Computer Science*, pages 428–439. Springer, 2012.

[4] Andrew Berns, James Hegeman, and Sriram V. Pemmaraju. Super-fast distributed algorithms for metric facility location. *CoRR*, abs/1308.2473, August 2013.
[5] Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015, pages 143–152, 2015.

[6] G. Cornuejols, G. Nemhauser, and L. Wolsey. Discrete Location Theory. Wiley, 1990.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. Commun. ACM, 51(1):107–113, January 2008.

[8] Danny Dolev, Christoph Lenzen, and Shir Peled. “Tri, Tri again”: Finding triangles and small subgraphs in a distributed setting. In Proceedings of the 26th International Conference on Distributed Computing, DISC’12, pages 195–209, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. The communication complexity of distributed task allocation. In Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing, PODC ’12, pages 67–76, New York, NY, USA, 2012. ACM.

[10] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC ’14, pages 367–376, New York, NY, USA, 2014. ACM.

[11] M. V. Eede, P. Hansen, and L. Kaufman. A plant and warehouse location problem. Operational Research Quarterly, 28(3):547—554, 1977.

[12] Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using MapReduce. In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’11, pages 681–689, New York, NY, USA, 2011. ACM.

[13] C. Frank. Algorithms for Sensor and Ad Hoc Networks. Springer, 2007.

[14] Joachim Gehweiler, Christiane Lammersen, and Christian Sohler. A distributed O(1)-approximation algorithm for the uniform facility location problem. In Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA ’06, pages 237–243, New York, NY, USA, 2006. ACM.

[15] M. J. Hamburger and A. A. Kuehn. A heuristic program for locating warehouses. Management science, 9(4):643—666, 1963.

[16] James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015, pages 91–100, 2015.

[17] James W. Hegeman, Sriram V. Pemmaraju, and Vivek B. Sardeshmukh. Near-constant-time distributed algorithms on a congested clique. In Proceedings of the 28th International Symposium on Distributed Computing (DISC), pages 514–530, 2014.

[18] Stephan Holzer and Nathan Pinsker. Approximation of distances and shortest paths in the broadcast congest clique. CoRR, abs/1412.3445, 2014.

[19] Öjvind Johansson. Simple distributed (δ + 1)-coloring of graphs. Inf. Process. Lett., 70(5):229–232, June 1999.
[20] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[21] Kishore Kothapalli and Sriram V. Pemmaraju. Super-fast 3-ruling sets. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, pages 136–147, 2012.

[22] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in MapReduce and streaming. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’13, pages 1–10, New York, NY, USA, 2013. ACM.

[23] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: A method for solving graph problems in MapReduce. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’11, pages 85–94, New York, NY, USA, 2011. ACM.

[24] Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC ’13, pages 42–50, 2013.

[25] Christoph Lenzen and Roger Wattenhofer. Brief announcement: Exponential speed-up of local algorithms using non-local communication. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC ’10, pages 295–296, New York, NY, USA, 2010. ACM.

[26] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM Journal on Computing*, 35(1):120–131, 2005.

[27] Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed MST for Constant Diameter Graphs. *Distributed Computing*, 18(6):453–460, 2006.

[28] T. Moscibroda and R. Wattenhofer. Facility location: distributed approximation. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 108—117, New York, NY, USA, 2005. ACM, ACM Press.

[29] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the 46th ACM Symposium on Theory of Computing (STOC)*, pages 565–573, 2014.

[30] S. Pandit and S. V. Pemmaraju. Return of the primal-dual: distributed metric facility location. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC ’09, pages 180—189, New York, NY, USA, 2009. ACM, ACM Press.

[31] S. Pandit and S. V. Pemmaraju. Rapid randomized pruning for fast greedy distributed algorithms. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 325—334. ACM, 2010.

[32] Saurav Pandit and Sriram V. Pemmaraju. Finding facilities fast. In *Distributed Computing and Networking, 10th International Conference, ICDCN 2009, Hyderabad, India, January 3-6, 2009. Proceedings*, pages 11–24, 2009.

[33] Boaz Patt-Shamir and Marat Teplitsky. The round complexity of distributed sorting: Extended abstract. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC ’11, pages 249–256, New York, NY, USA, 2011. ACM.
[34] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*, volume 5. Society for Industrial Mathematics, 2000.

[35] Johannes Schneider, Michael Elkin, and Roger Wattenhofer. Symmetry breaking depending on the chromatic number or the neighborhood growth. *Theoretical Computer Science*, 509:40–50, 2013.

[36] John F. Stollsteimer. A working model for plant numbers and locations. *Management Science*, 45(3):631–645, 1963.