



Distributed Computing with the Cloud

Yehuda Afek¹, Gal Giladi², and Boaz Patt-Shamir³(✉)

¹ School of CS, Tel Aviv University, 6997801 Tel Aviv, Israel
afek@tauex.tau.ac.il

² School of CS, Tel Aviv University, 6997801 Tel Aviv, Israel

³ School of EE, Tel Aviv University, 6997801 Tel Aviv, Israel
boaz@tau.ac.il

Abstract. Motivated by cloud storage (à la Dropbox, Google Drive, etc.), we investigate distributed computing in message passing networks that contain a passive node that can only store and share data, and does not carry out any computations. Using basic primitives of collaborative transmission of a file from and to the cloud, we implement more complex tasks where the goal is to combine input values: e.g., each node holds a vector (or a matrix) as input and the sum (or product) of all the inputs should be stored in the cloud. We present near-optimal algorithms for these tasks. Finally we consider applications such as federated learning and file deduplication in this new model. Our results show that utilizing both node-cloud and node-node communication links can substantially speed up computation with respect to systems where processors communicate either only through the cloud or only through the network links.

1 Introduction

In 2018 Google announced that the number of users of Google Drive is surpassing one billion [25]. Earlier that year, Dropbox stated that in total, more than an exabyte (10^{18} bytes) of data has been uploaded by its users [14]. Other cloud-storage services, such as Microsoft's OneDrive, Amazon's S3, or Box, are thriving too. The driving force of this paper is our wish to let *other* distributed systems to take advantage of the enormous infrastructure that makes up the complexes called “clouds.” Let us explain how.

The computational and storage capacities of servers in cloud services are relatively well advertised. A lesser known fact is that a cloud system also entails a massive component of *communication*, that makes it appear close almost everywhere on the Internet. (This feature is particularly essential for cloud-based video conferencing applications, such as Zoom, Cisco's Webex and others.) In view of the existing cloud services, our fundamental idea is to *abstract a complete cloud system as a single, passive storage node*.

To see the benefit of this approach, consider a network of the “wheel” topology: a single cloud node is connected to n processing nodes arranged in a cycle (see Fig. 1). Suppose each processing node has a wide link of bandwidth n to its

The original version of this chapter was revised: An error in the presentation of Gal Giladi's affiliation was corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-91081-5_41

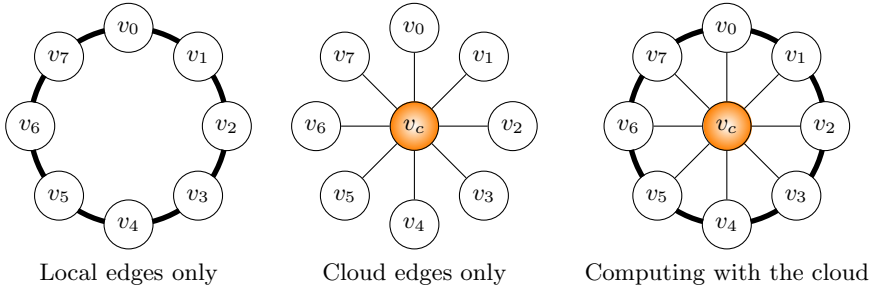


Fig. 1. Wheel topology with $n = 8$. The v_i nodes are processing nodes connected by a ring of high-bandwidth links. The cloud node v_c is connected to the processing nodes by lower-bandwidth links. All links are bidirectional and symmetric.

cycle neighbors, and a narrower link of bandwidth \sqrt{n} to the cloud node. Further suppose that each processing node has an n -bit vector, and that the goal is to calculate the sum of all vectors. Without the cloud (Fig. 1, left), such a task requires at least $\Omega(n)$ rounds – to cover the distance; on the other hand, without using the cycle links (Fig. 1, middle), transmitting a single vector from any processing node (and hence computing the sum) requires $\Omega(n/\sqrt{n}) = \Omega(\sqrt{n})$ rounds – due to the limited bandwidth to the cloud. But using both cloud links and local links (Fig. 1, right), the sum can be computed in $\tilde{O}(\sqrt[4]{n})$ rounds, as we show in this paper.

More generally, in this paper we initiate the study of the question of how to use an omnipresent cloud storage to speed up computations, if possible. We stress that the idea here is to develop a framework and tools that facilitate computing *with* the cloud, as opposed to computing *in* the cloud.

Specifically, in this paper we introduce the *computing with the cloud* model (CWC), and present algorithms that efficiently combine distributed inputs to compute various functions, such as vector addition and matrix multiplication. To this end, we first implement (using dynamic flow techniques) primitive operations that allow for the exchange of large messages between processing nodes and cloud nodes. Given the combining algorithms, we show how to implement some applications such as federated learning and file de-duplication (dedup).

1.1 Model Specification

The “Computing with the Cloud” (CWC) model is a synchronous network whose underlying topology is described by a weighted directed graph $G = (V, E, w)$. The node set consists of two disjoint subsets: $V = V_p \cup V_c$, where V_p is the set of *processing nodes*, and V_c is the set of *cloud nodes*. Cloud nodes are passive nodes that function as shared storage: they support read and write requests, and do not perform any other computation. We use n to denote the number of processing nodes (the number of cloud nodes is typically constant).

We denote the set of links that connect two processing nodes by E_L (“local links”), and by E_C (“cloud links”) the set of links that connect processing nodes to cloud nodes. Each link $e \in E = E_L \cup E_C$ has a prescribed bandwidth $w(e)$

(there are no links between different cloud nodes). We denote by $G_p \stackrel{\text{def}}{=} (V_p, E_L)$ the graph $G - V_c$, i.e., the graph spanned by the processing nodes.

Our execution model is the standard synchronous network model, where each round consists of processing nodes receiving messages sent in the previous round, doing an arbitrary local computation, and then sending messages. The size of a message sent over a link e in a round is at most $w(e)$ bits.

Cloud nodes do not perform any computations: they can only receive requests we denote by **FR** and **FW** (file read and write, respectively), to which they respond in the following round. More precisely, each cloud node has unbounded storage; to write, a processing node v_i invokes **FW** with arguments that describe the target cloud node, a filename f , a bit string S , and the location (index) within f that S needs to be written in. It is assumed that $|S| \leq w(v_i, v_c)$ bits (longer writes can be broken to a series of **FW** operations). To read, a processing node v_i invokes **FR** with arguments that describe the cloud node, a filename f and the range of indices to fetch from f . Again, we assume that the size of the range in any single **FR** invocation by node v_i is at most $w(v_i, v_c)$.¹

FW operations are exclusive, i.e., no other operation (read or write) to the same file location is allowed to take place simultaneously. Concurrent **FR** operations from the same location are allowed.

Discussion. We believe that our model is fairly widely applicable. A processing node in our model may represent anything from a computer cluster with a single gateway to the Internet, to cellphones or even smaller devices—anything with a non-shared Internet connection. The local links can range from high-speed fiber to Bluetooth or infrared links. Typically in this setting the local links have bandwidth much larger than the cloud links (and cloud downlinks in many cases have larger bandwidth than cloud uplinks). Another possible interpretation of the model is a private network (say, in a corporation), where a cloud node represents a storage or a file server. In this case the cloud link bandwidth may be as large as the local link bandwidth.

1.2 Problems Considered and Main Results

Our main results in this paper are efficient algorithms in the CWC model to combine values stored at nodes. These algorithms use building blocks that facilitate efficient transmission of large messages between processing nodes and cloud nodes. These building blocks, in turn, are implemented in a straightforward way using dynamic flow techniques. Finally, we show how to use the combining algorithms to derive new algorithms for federated learning and file de-duplication (dedup) in the CWC model.

More specifically, we provide implementations of the following tasks.

¹ For both the **FW** and **FR** operations we ignore the metadata (i.e., v_c 's descriptor, the filename f and the indices) and assume that the total size of metadata in a single round is negligible and can fit within $w(v_i, v_c)$. Otherwise, processing nodes may use the metadata parameters to exchange information that exceeds the bandwidth limitations (for example, naming a file with the string representation of a message whose length is larger than the bandwidth).

Basic Cloud Operations: Let v_c denote a cloud node below.

cW_i (*cloud write*): write an s -bits file f stored at node $i \in V_p$ to node v_c .

cR_i (*cloud read*): fetch an s -bits file f from node v_c to node $i \in V_p$.

cAW (*cloud all write*): for each $i \in V_p$, write an s -bits file f_i stored at node i to node v_c .

cAR (*cloud all read*): for each $i \in V_p$, fetch an s -bits file f_i from node v_c to node i .

Combining and Dissemination Operations

$cComb$: (*cloud combine*): Each node $i \in V_p$ has an s -bits input string S_i , and there is a binary associative operator $\otimes : \{0, 1\}^s \times \{0, 1\}^s \rightarrow \{0, 1\}^s$ (the result is as long as each operand). The requirement is to write to a cloud node v_c the s -bits string $S_1 \otimes S_2 \otimes \dots \otimes S_n$. Borrowing from Group Theory, we call the operation \otimes *multiplication*, and $S_1 \otimes S_2$ is the *product* of S_1 by S_2 . In general, \otimes is not necessarily commutative. We assume the existence of a unit element for \otimes , denoted $\tilde{1}$, such that $\tilde{1} \otimes S = S \otimes \tilde{1} = S$ for any s -bits strings S . The unit element is represented by a string of $O(1)$ bits. Examples for commutative operators include vector (or matrix) addition over a finite field, logical bitwise operations, leader election, and the top- k problem. Examples for non-commutative operators may be matrix multiplication (over a finite field) and function composition.

$cCast$ (*cloudcast*): All the nodes $i \in V_p$ simultaneously fetch a copy of an s -bits file f from node v_c (Similar to network broadcast).

Applications. $cComb$ and $cCast$ can be used directly to provide matrix multiplication, matrix addition, and vector addition. We also outline the implementation of the following.

Federated Learning (FL) [31]: In FL, a collection of agents collaborate in training a neural network to construct a model of some concept, but the agents want to keep their data private. Unlike [31], in our model the central server is a passive storage device that does not carry out computations. We show how elementary secure computation techniques, along with our combining algorithm, can efficiently help training an ML model in the federated scheme implemented in CWC, while maintaining privacy.

File Deduplication: Deduplication (or dedup) is a task in file stores, where redundant identical copies of data are identified (and possibly unified)—see, e.g., [32]. Using $cComb$ and $cCast$, we implement file dedup in the CWC model on collections of files stored at the different processing nodes. The algorithm keeps a single copy of each file and pointers instead of the other replicas.

Special Topologies. The complexity of the general algorithms we present depends on the given network topology. We study a few cases of interest.

First, we consider *s-fat-links* network, defined to be, for a given parameter $s \in \mathbb{N}$, as the CWC model with the following additional assumptions:

- All links are symmetric, i.e., $w(u, v) = w(v, u)$ for every link $(u, v) \in E$.

- Local links have bandwidth at least s .
- There is only one cloud node v_c .

The fat links model seems suitable in many real-life cases where local links are much wider than cloud links (uplinks to the Internet), as is the intuition behind the HYBRID model [4].

Another topology we consider is the *wheel network*, depicted schematically in Fig. 1 (right). In a wheel system there are n processing nodes arranged in a ring, and a cloud node connected to all processing nodes. In the *uniform* wheel, all cloud links have the same bandwidth b_c and all local links have the same bandwidth b_l . In the uniform wheel model, we typically assume that $b_c \ll b_l$.

The wheel network is motivated by non-commutative combining operations, where the order of the operands induces a linear order on the processing nodes, i.e., we view the nodes as a line, where the first node holds the first input, the second node holds the second input etc. For symmetry, we connect the first and the last node, and with a cloud node connected to all—we’ve obtained the wheel.

Overview of Techniques. As mentioned above, the basic file operations (cW, cR, cAW and cAR) are solved optimally using dynamic flow techniques, or more specifically, *quickest flow* (Sect. 2). In the full version, we present closed-form bounds on cW and cR for the wheel topology.

We present tight bounds for cW and cR in the s -fat-links network, where s is the input size at all nodes. We then continue to consider the tasks cComb with *commutative operators* and cCast, and prove nearly-tight bounds on their time complexity in the s -fat-links network (Theorem 11, Theorem 13, Theorem 15). The idea is to first find, for every processing node i , a cluster of processing nodes that allows it to perform cW in an optimal number of rounds. We then perform cComb by combining the values within every cluster using convergecast [33], and then combining the results in a computation-tree fashion. Using sparse covers [5], we perform the described procedure in near-optimal time.

Non-commutative operators are explored in the natural wheel topology. We present algorithms for wheel networks with *arbitrary* bandwidth (both cloud and local links). We prove an upper bound for cComb (Theorem 18).

Finally, in Sect. 5, we demonstrate how the considered tasks can be applied for the purposes of Federated Learning and File Deduplication.

Paper Organization. Due to space constraints, many details and proofs are omitted from this version. They can be found in the full version [2].

1.3 Related Work

Our model is based on, and inspired by, a long history of theoretical models in distributed computing. To gain some perspective, we offer here a brief review.

Historically, distributed computing is split along the dichotomy of message passing vs shared memory [16]. While message passing is deemed the “right” model for network algorithms, the shared memory model is the abstraction of choice for programming multi-core machines.

The prominent message-passing models are LOCAL [28], and its derived CONGEST [33]. In these models, a system is represented by a connected (typically undirected) graph, in which nodes represent processors and edges represent communication links. In LOCAL, message size is unbounded, while in CONGEST, message size is restricted, typically to $O(\log n)$ bits. Thus, CONGEST accounts not only for the distance information has to traverse, but also for information volume and the bandwidth available for its transportation.

While most algorithms in the LOCAL and CONGEST models assume fault-free (and hence synchronous) executions, in the distributed shared memory model, asynchrony and faults are the primary source of difficulty. Usually, in the shared memory model one assumes that there is a collection of “registers,” accessible by multiple threads of computation that run at different speeds and may suffer crash or even Byzantine faults (see, e.g., [3]). The main issues in this model are coordination and fault-tolerance. Typically, the only quantitative hint to communication cost is the number and size of the shared registers.

Quite a few papers consider the combination of message passing and shared memory, e.g., [1, 12, 18, 19, 30, 35]. The uniqueness of the CWC model with respect to past work is that it combines passive storage nodes with a message passing network with restrictions on the links bandwidth.

The CONGESTED CLIQUE (CC) model [29] is a special case of CONGEST, where the underlying communication graph is assumed to be fully connected. The CC model is appropriate for computing *in* the cloud, as it has been shown that under some relatively mild conditions, algorithms designed for the CC model can be implemented in the MapReduce model, i.e., run in datacenters [20]. Another model for computing in the cloud is the MPC model [22]. Very recently, the HYBRID model [4] was proposed as a combination of CC with classical graph-based communication. More specifically, the HYBRID model assumes the existence of two communication networks: one for local communication between neighbors, where links are typically of infinite bandwidth (exactly like LOCAL); the other network is a *node-congested* clique, i.e., a node can communicate with every other node directly via “global links,” but there is a small upper bound (typically $O(\log n)$) on the total number of messages a node can send or receive via these global links in a round. Even though the model was presented only recently, there is already a line of algorithmic work in it, in particular for computing shortest paths [4, 10, 23].

Discussion. Intuitively, our CWC model can be viewed as the classical CONGEST model over the processors, augmented by special cloud nodes (object stores) connected to some (typically, many) compute nodes. To reflect modern demands and availability of resources, we relax the very stringent bandwidth allowance of CONGEST, and usually envision networks with much larger link bandwidth (e.g., n^ϵ for some $\epsilon > 0$).

Considering previous network models, it appears that HYBRID is the closest to CWC, even though HYBRID was not expressly designed to model the cloud. In our view, CWC is indeed more appropriate for computation with the cloud. First, in most cases, global communication (modeled by clique edges

in HYBRID) is limited by link bandwidth, unlike HYBRID’s node capacity constraint, which seems somewhat artificial. Second, HYBRID is not readily amenable to model multiple clouds, while this is a natural property of CWC.

Regarding shared memory models, we are unaware of topology-based bandwidth restriction on shared memory access in distributed models. In some general-purpose parallel computation models (based on BSP [35]), communication capabilities are specified using a few global parameters such as latency and throughput, but these models deliberately abstract topology away. In distributed (asynchronous) shared memory, the number of bits that need to be transferred to and from the shared memory is seldom explicitly analyzed.

2 Communication Primitives in CWC

In this short section we state the complexity results for the basic operations, derived by straightforward application of *dynamic flow* techniques [34].

Intuitively, the concept of dynamic flow is a variant of maximum flow, where time is finite, links introduce delay, and the goal is to maximize the amount of flow shipped in the given time limit (the dual problem, where the amount of flow to ship is given and the goal is to minimize the time required to ship it, is called *quickest flow* [6, 9, 15, 21]). By reduction to min-cost max-flow, strongly polynomial algorithms to these problems are known. Using these results, we can prove the following statements. Details can be found in [2].

Theorem 1. *Given any instance of the CWC model, an optimal schedule realizing cW_i or cR_i can be computed in polynomial time.*

Theorem 2. *Given any instance of the CWC model, an optimal schedule realizing cAW or cAR for one cloud node can be computed in polynomial time.*

Theorem 3. *Given any instance of the CWC model and $\epsilon > 0$, a schedule realizing cAW or cAR of length at most $(1 + \epsilon)$ times the optimal can be computed in time polynomial in the instance size and ϵ^{-1} .*

3 Computing and Writing Combined Values

Flow-based techniques are not applicable in the case of writing a combined value, because the very essence of combining violates conservation constraints (i.e., the number of bits entering a node may be different than the number of bits leaving it). However, in Sect. 3.1 we explain how to implement $cComb$ in the general case using cAW and cAR . While simple and generic, these implementations can have time complexity much larger than optimal. We offer partial remedy in Sect. 3.2, where we present our main result: an algorithm for $cComb$ when \otimes is commutative and the local network has “fat links,” i.e., all local links have capacity at least s . For this important case, we show how to complete the task in time larger than the optimum by an $O(\log^2 n)$ factor.

Algorithm 1. High-level algorithm for cComb using cAW and cAR

```

1:  $m := n, j := 0$ 
2: for all  $i < n$  set  $X_i^0 = S_i$ , and for all  $i > n$ ,  $X_i^0 = \tilde{\mathbf{1}}$ 
3: while  $m > 1$  do
4:   run cAW with inputs  $S_i = X_i^j$ 
5:   run cAR with inputs  $S_i = X_{2i}^j$ 
6:   run cAR with inputs  $S_i = X_{2i+1}^j$ 
7:    $m := \lceil m/2 \rceil$ 
8:   for all  $i < m$  set  $X_i^{j+1} = X_{2i}^j \otimes X_{2i+1}^j$ , and for all  $i > m$ ,  $X_i^{j+1} = \tilde{\mathbf{1}}$ 
9:   for all  $i < m$ , in parallel, node  $i$  calculates  $X_i^{j+1}$  locally
10:   $j := j + 1$ 
11: end while
12: run cW from node 0 to write  $X_0^j$  to the cloud

```

3.1 Combining in General Graphs

We now present algorithms for cComb and for cCast on general graphs, using the primitives treated in Sect. 2. Note that with a non-commutative operator, the operands must be ordered; using renaming if necessary, we assume w.l.o.g. that in such cases the nodes are indexed by the same order of their operands.

Theorem 4. *Let T_s be the running time of cAW (and cAR) when all files have size s . Then Algorithm 1 solves cComb in $O(T_s \log n)$ rounds.*

In a way, cCast is the “reverse” problem of cComb, since it starts with s bits in the cloud and ends with s bits of output in every node. However, cCast is easier than cComb because our model allows for concurrent reads and disallows concurrent writes. We have the following result.

Theorem 5. *Let T_s be the time required to solve cAR when all files have size s . Then cCast can be solved in T_s rounds as well.*

3.2 Combining Commutative Operators in Fat Links Network

In the case of s -fat-links network (i.e., all local links are have bandwidth at least s , and all links are symmetric) we can construct a near-optimal algorithm for cComb. The idea is to use multiple cW and cR operations instead of cAW and cAR. The challenge is to minimize the number of concurrent operations per node; to this end we use sparse covers [5].

We note that if the network is s -fat-links but the operand size is $s' > s$, the algorithms still apply, with an additional factor of $\lceil s'/s \rceil$ to the running time. The lower bounds in this section, however, may change by more than that factor.

We start with a tight analysis of cW and cR in this setting and then generalize to cComb and cCast.

Implementation of cW and cR. Consider cW_i , where i wishes to write s bits to a given cloud node. The basic tension in finding an optimal schedule for cW_i

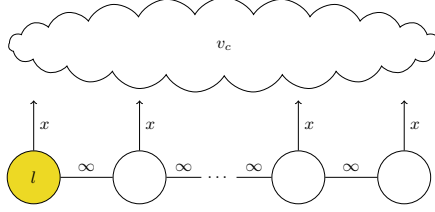


Fig. 2. A simple path example. The optimal distance to travel in order to write an s -bits file to the cloud would be $\sqrt{s/x}$.

is that in order to use more cloud bandwidth, more nodes need to be enlisted. But while more bandwidth reduces the transmission time, reaching remote nodes (that provide the extra bandwidth) increases the traversal time. Our algorithm looks for the sweet spot where the conflicting effects are more-or-less balanced.

For example, consider a simple path of n nodes with infinite local bandwidth, where each node is connected to the cloud with bandwidth x (Fig. 2). Suppose that the leftmost node l needs to write a message of s bits to the cloud. By itself, writing requires s/x rounds. Using all n nodes, uploading would take $O(s/nx)$ rounds, but $n - 1$ rounds are needed to ship the messages to the fellow-nodes. The optimal solution in this case is to use only $\sqrt{s/x}$ nodes: the time to ship the file to all these nodes is $\sqrt{s/x}$, and the upload time is $\frac{s/\sqrt{s/x}}{x} = \sqrt{s/x}$, because each node needs to upload only $s/\sqrt{s/x}$ bits.

In general, we define “cloud clusters” to be node sets that optimize the ratio between their diameter and their total bandwidth to the cloud. Our algorithms for cW and cR use nodes of cloud clusters. We prove that the running-time of our implementation is asymptotically optimal. Formally, we have the following.

Definition 1. Let $G = (V, E, w)$ be a CWC system with processor nodes V_p and cloud nodes V_c . The **cloud bandwidth** of a processing node $i \in V_p$ w.r.t. a given cloud node $v_c \in V_c$ is $b_c(i) \stackrel{\text{def}}{=} w(i, v_c)$. A **cluster** $B \subseteq V_p$ in G is a connected set of processing nodes. The **cloud (up or down) bandwidth** of cluster B w.r.t a given cloud node, denoted $b_c(B)$, is the sum of the cloud bandwidth to v_c over all nodes in B : $b_c(B) \stackrel{\text{def}}{=} \sum_{i \in B} b_c(i)$. The (strong) **diameter** of cluster B , denoted $\text{diam}(B)$, is the maximum distance between any two nodes of B in the induced graph $G[B]$: $\text{diam}(B) = \max_{u, v \in B} \text{dist}_{G[B]}(u, v)$.

We use the following definition for the network without the cloud.

Definition 2. Let $G = (V, E, w)$ be a CWC system with processing nodes V_p and cloud nodes V_c . The **ball of radius r around node $i \in V_p$** , denoted $B_r(i)$ is the set of nodes at most r hops away from i in G_p .

(Note that the metric here is hop-based— w indicates link bandwidths.) Finally, we define the concept of cloud cluster of a node.

Algorithm 2. cW_i

-
- 1: Construct a BFS spanning tree of B_i rooted at node i and assign for each index $1 \leq x \leq |B_i|$ a unique node $v(x) \in B_i$ according to their BFS order ($v(1) = i$)
 - 2: Broadcast S from node i to all nodes in B_i using the tree
 - 3: **for** all $x := 1$ **to** $|B_i|$, **in parallel do**
 - 4: Node $v(x)$ writes to the cloud the part of S starting at $s \cdot \frac{\sum_{y=1}^{x-1} b_c(v(y))}{b_c(B_i)}$ and extending for $s \cdot \frac{b_c(v(x))}{b_c(B_i)}$ bits, writing $b_c(v(x))$ bits in every round.
 - 5: Node $v(x) \neq i$ sends an acknowledgment to i when done, and halts
 - 6: **end for**
 - 7: Node i halts when all acknowledgments are received. // for cR reversal
-

Definition 3. Let $G = (V, E, w)$ be a CWC system with processing nodes V_p and cloud node v_c , and let $i \in V_p$. Given $s \in \mathbb{N}$, the s -**cloud radius** of node i , denoted $k_s(i)$, is defined to be $k_s(i) \stackrel{\text{def}}{=} \min(\{\text{diam}(G_p)\} \cup \{k \mid (k+1) \cdot b_c(B_k(i)) \geq s\})$. The ball $B_i \stackrel{\text{def}}{=} B_{k_s(i)}(i)$ is the s -**cloud cluster** of node i . The **timespan** of the s -cloud cluster of i is denoted $Z_i \stackrel{\text{def}}{=} k_s(i) + \frac{s}{b_c(B_i)}$. We sometimes omit the s qualifier when it is clear from the context.

In words, B_i is a cluster of radius $k(i)$ around node i , where $k(i)$ is the smallest radius that allows writing s bits to v_c by using all cloud bandwidth emanating from B_i for $k(i) + 1$ rounds. Z_i is the time required (1) to send s bits from node i to all nodes in B_i , and (2) to upload s bits to v_c collectively by all nodes of B_i . Note that B_i is easy to compute. We can now state our upper bound.

Theorem 6. Given a fat-links CWC system, Algorithm 2 solves the s -bits cW_i problem in $O(Z_i)$ rounds on B_i .

Next, we show that our solution for cW_i is optimal, up to a constant factor. We consider the case of an incompressible input string: such a string exists for any size $s \in \mathbb{N}$ (see, e.g., [27]). As a consequence, in any execution of a correct algorithm, s bits must cross any cut that separates i from the cloud node, giving rise to the following lower bound.

Theorem 7. cW_i in a fat-links CWC requires $\Omega(Z_i)$ rounds.

By reversing time (and hence information flow) in a schedule of cW , one gets a schedule for cR . Hence we have the following immediate corollaries.

Theorem 8. cR_i can be executed in $O(Z_i)$ rounds in a fat-links CWC.

Theorem 9. cR_i in a fat-links CWC requires $\Omega(Z_i)$ rounds.

► *Remark:* The lower bound (Theorem 7) and the definition of cloud clusters (Definition 3) show an interplay between the message size s , cloud bandwidth, and the network diameter; For large enough s , the cloud cluster of a node includes all processing nodes (because the time spent crossing the local network is negligible relative to the upload time), and for small enough s , the cloud cluster includes only the invoking node, rendering the local network redundant.

Implementation of cComb. Below, we first show how to implement cComb using any given cover. In fact, we shall use *sparse covers* [5], which allow us to get near-optimal performance.

Intuitively, every node i has a cloud cluster B_i which allows it to perform cW_i , and calculating the combined value within every cloud cluster B_i is straightforward (cf. Algorithm 4 and Lemma 1). Therefore, given a partition of the graph that consists of pairwise disjoint cloud clusters, cComb can be solved by combining the inputs in every cloud cluster, followed by combining the partial results in a computation-tree fashion using cW and cR. However, such a partition may not always exist, and we resort to a *cover* of the nodes. Given a cover \mathcal{C} in which every node is a member of at most $\text{load}(\mathcal{C})$ clusters, we can use the same technique, while increasing the running-time by a factor of $\text{load}(\mathcal{C})$ by time multiplexing. Using Awerbuch and Peleg’s sparse covers (see Theorem 12), we can use an initial cover \mathcal{C} that consists of all cloud clusters in the graph to construct another cover, \mathcal{C}' , in which $\text{load}(\mathcal{C}')$ is $O(\log n)$, paying an $O(\log n)$ factor in cluster diameters, and use \mathcal{C}' to get near-optimal results.

Definition 4. Let G be a CWC system, and let B be a cluster in G (see Definition 1). The **timespan of node i in B** , denoted $Z_B(i)$, is the minimum number of rounds required to perform cW_i (or cR_i), using only nodes in B . The **timespan of cluster B** , denoted $Z(B)$, is given by $Z(B) = \min_{i \in B} Z_B(i)$. The **leader of cluster B** , denoted $r(B)$, is a node with minimal timespan in B , i.e., $r(B) = \arg\max_{i \in B} Z_B(i)$.

In words, the timespan of cluster B is the minimum time required for any node in B to write an s -bit string to the cloud using only nodes of B .

Definition 5. Let G be a CWC system with processing node set V_p . A **cover of G** is a set of clusters $\mathcal{C} = \{B_1, \dots, B_m\}$ such that $\cup_{B \in \mathcal{C}} B = V_p$. The **load of node i in a cover \mathcal{C}** is the number of clusters in \mathcal{C} that contain i , i.e., $\text{load}_{\mathcal{C}}(i) = |\{B \in \mathcal{C} \mid i \in B\}|$. The **load of cover \mathcal{C}** is the maximum load of any node in the cover, i.e., $\text{load}(\mathcal{C}) = \max_{i \in V_p} \text{load}_{\mathcal{C}}(i)$. The **timespan of cover \mathcal{C}** , denoted $Z(\mathcal{C})$, is the maximum timespan of any cluster in \mathcal{C} , $Z(\mathcal{C}) = \max_{B \in \mathcal{C}} Z(B)$. The **diameter of cover \mathcal{C}** , denoted $\text{diam}_{\max}(\mathcal{C})$, is the maximum diameter of any cluster in \mathcal{C} , $\text{diam}_{\max}(\mathcal{C}) = \max_{B \in \mathcal{C}} \text{diam}(B)$.

We now give an upper bound in terms of any given cover.

Theorem 10. Given a cover \mathcal{C} , Algorithm 3 solves cComb in a fat-links CWC in $O(\text{diam}_{\max}(\mathcal{C}) \cdot \text{load}(\mathcal{C}) + Z(\mathcal{C}) \cdot \text{load}(\mathcal{C}) \cdot \log |\mathcal{C}|)$ rounds.

The basic strategy is to first compute the combined value in each cluster using only the local links, and then combine the cluster values using a computation tree. However, unlike Algorithm 1, we use cW and cR instead of cAW and cAR.

A high-level description is given in Algorithm 3. The algorithm consists of a preprocessing part (lines 1–2), and the execution part, which consists of the “low-level” computation using only local links (lines 3–5), and the “high-level” computation among clusters (line 6). We elaborate on each below.

Algorithm 3. High-level algorithm for cComb given a cover \mathcal{C}

-
- 1: For a node $i \in V_p$, let $C_i[1], C_i[2], \dots$ be the clusters containing i .
 - 2: For the rest of the algorithm, multiplex each round as $\text{load}(\mathcal{C})$ rounds, such that each node i operates in the context of cluster $C_i[j]$ in the j -th round .
 - 3: **for all** $B \in \mathcal{C}$, in parallel **do**
 - 4: Compute $P_B = \bigotimes_{j \in B} S_j$ using Alg. 4 // convergecast using local links only
 - 5: **end for**
 - 6: Apply Alg. 5 // the result is stored in the cloud
-

► *Preprocessing.* A major component of the preprocessing stage is computing the cover \mathcal{C} , which we specify later (see Theorem 11). In Algorithm 3 we describe the algorithm as if it operates in each cluster independently of other clusters, but clusters may overlap. To facilitate this mode of operation, we use time multiplexing: nodes execute work on behalf of the clusters they are member of in a round-robin fashion, as specified in lines 1–2 of Algorithm 3. This allows us to invoke operations limited to clusters in all clusters “simultaneously” by increasing the time complexity by a $\text{load}(\mathcal{C})$ factor.

► *Low levels: Combining within a single cluster.* To implement line 4 of Algorithm 3, we build, in each cluster $B \in \mathcal{C}$, a spanning tree rooted at $r(B)$, and apply Convergecast [33] using \otimes . Ignoring the multiplexing of Algorithm 3, we have:

Lemma 1. *Algorithm 4 computes $P_B = \bigotimes_{i \in B} S_i$ at node $r(B)$ in $O(\text{diam}(B))$ rounds.*

To get the right overall result, each input S_i is associated with a single cluster in \mathcal{C} . To this end, we require each node to select a single cluster in which it is a member as its *home cluster*. When applying Algorithm 4, we use the rule that the input of node i in a cluster $B \ni i$ is S_i if B is i 's home cluster, and $\bar{\mathbf{1}}$ otherwise.

Considering the scheduling obtained by Step 2, we get the following lemma.

Lemma 2. *Steps 3–5 of Algorithm 3 terminate in $O(\text{diam}_{\max}(\mathcal{C}) \cdot \text{load}(\mathcal{C}))$ rounds, with P_B stored at the leader node of B for each cluster $B \in \mathcal{C}$.*

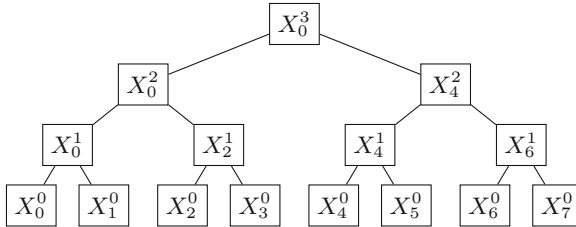


Fig. 3. Computation tree example. X_i^j denotes the result stored in i after iteration j .

Algorithm 4. Computing the combined result of cluster B at leader $r(B)$

```

1: Construct a BFS tree of  $B$  rooted at node  $r(B)$ . Let  $h$  be the height of the tree.
2: for  $d := h$  to 2 do
3:   for all  $i \in B$  at layer  $d$  of the tree, in parallel do
4:     if  $i$  is not a leaf then
5:        $i$  computes  $S'_i := S_i \otimes \bigotimes_{j \in \text{child}(i)} S'_j$ 
6:     else
7:        $S'_i := S_i$ 
8:     end if
9:      $i$  sends  $S'_i$  to its parent node in the tree
10:  end for
11: end for
12: Node  $r(B)$  computes  $P_B := S_{r(B)} \otimes \bigotimes_{j \in \text{child}(r(B))} S'_j$ 

```

Algorithm 5. Computing the high level tree-nodes values

```

1: for  $l := \lceil \log |C_1| \rceil$  to 1 do
2:   for all tree-nodes  $y$  in layer  $l$  of the computation tree, in parallel do
3:     Let  $B := \text{cl}(y)$ 
4:     if  $y$  is not a leaf then
5:       Let  $y_\ell$  and  $y_r$  be the left and the right children of  $y$ , respectively.
6:        $r(B)$  invokes cR for  $\text{vl}(y_\ell)$ 
7:        $r(B)$  invokes cR for  $\text{vl}(y_r)$ 
8:        $r(B)$  computes  $\text{vl}(y) := \text{vl}(y_\ell) \otimes \text{vl}(y_r)$ 
9:     else
10:       $\text{vl}(y) := P_B$  // if  $y$  is a leaf its value is already stored at  $r(B)$ 
11:    end if
12:     $r(B)$  invokes cW for  $\text{vl}(y)$ 
13:  end for
14: end for

```

► *High levels: Combining using the cloud.* When Algorithm 3 reaches Step 6, the combined result of every cluster is stored in the leader of the cluster. The idea is now to fill in a computation tree whose leaves are these values (see Fig. 3).

We combine the partial results by filling in the values of a computation tree defined over the clusters. The leaves of the tree are the combined values of the clusters of \mathcal{C} , as computed by Algorithm 4. To fill in the values of other nodes in the computation tree, we use the clusters of \mathcal{C} : Each node in the tree is assigned a cluster which computes its value using the cR and cW primitives.

Specifically, in Algorithm 5 we consider a binary tree with $|\mathcal{C}|$ leaves, where each non-leaf node has exactly two children. The tree is constructed from a complete binary tree with $2^{\lceil \log |\mathcal{C}| \rceil}$ leaves, after deleting the rightmost $2^{\lceil \log |\mathcal{C}| \rceil} - |\mathcal{C}|$ leaves. (If by the end the rightmost leaf is the only child of its parent, we delete the rightmost leaf repeatedly until this is not the case).

We associate each node y in the computation tree with a cluster $\text{cl}(y) \in \mathcal{C}$ and a value $\text{vl}(y)$, computed by the processors in $\text{cl}(y)$ are responsible to compute $\text{vl}(y)$. Clusters are assigned to leaves by index: The i -th leaf from the left is

associated with the i -th cluster of \mathcal{C} . For internal nodes, we assign the clusters arbitrarily except that we ensure that no cluster is assigned to more than one internal node. (This is possible because in a tree where every node has two or no children, the number of internal nodes is smaller than the number of leaves).

The clusters assigned to tree nodes compute the values as follows (see Algorithm 5). The value associated with a leaf y_B corresponding to cluster B is $\text{vl}(y_B) = P_B$. This way, every leaf x has $\text{vl}(x)$, stored in the leader of $\text{cl}(x)$, which can write it to the cloud using cW . For an internal node y with children y_l and y_r , the leader of $\text{cl}(y)$ obtains $\text{vl}(y_l)$ and $\text{vl}(y_r)$ using cR , computes their product $\text{vl}(y) = \text{vl}(y_l) \otimes \text{vl}(y_r)$ and invokes cW to write it to the cloud. The executions of cW and cR in a cluster B are done by the processing nodes of B .

Computation tree values are filled layer by layer, bottom up.

► *Remark.* We note that in Algorithm 5, Lines 6, 7 and 12 essentially compute cAR and cAW in which only the relevant cluster leaders have inputs. Therefore, these calls can be replaced with a collective call for appropriate cAR and cAW , making the multiplexing of Line 2 of Algorithm 3 unnecessary (similarly to Algorithm 1). By using optimal schedules for cAW and cAR , the running-time can only improve beyond the upper bound of Theorem 10.

Sparse Covers. We now arrive at our main result, derived from Theorem 10 using a particular flavor of covers. The result is stated in terms of the maximal timespan of a graph, according to the following definition.

Definition 6. Let $G = (V, E, w)$ be a CWC system with fat links. $Z_{\max} \stackrel{\text{def}}{=} \max_{i \in V_p} Z_i$ is the *maximal timespan* in G .

In words, Z_{\max} is the maximal amount of rounds that is required for any node in G to write an s -bit message to the cloud, up to a constant factor (cf. Theorem 7).

Theorem 11. Let $G = (V, E, w)$ be a CWC system with fat links. Then cComb with a commutative combining operator can be solved in $O(Z_{\max} \log^2 n)$ rounds.

To prove Theorem 11 we use sparse covers. We state the result from [5].

Theorem 12 ([5]). Given any cover \mathcal{C} and an integer $\kappa \geq 1$, a cover \mathcal{C}' that satisfies the following properties can be constructed in polynomial time.

- (i) For every cluster $B \in \mathcal{C}$ there exists a cluster $B' \in \mathcal{C}'$ such that $B \subseteq B'$.
- (ii) $\max_{B' \in \mathcal{C}'} \text{diam}(B') \leq 4\kappa \max_{B \in \mathcal{C}} \text{diam}(B)$
- (iii) $\text{load}(\mathcal{C}') \leq 2\kappa |\mathcal{C}|^{1/\kappa}$.

Proof of Theorem 11: Let \mathcal{C} be the cover defined as the set of all cloud clusters in the system. By applying Theorem 12 to \mathcal{C} with $\kappa = \lceil \log n \rceil$, we obtain a cover \mathcal{C}' with $\text{load}(\mathcal{C}') \leq 4 \lceil \log n \rceil$ because $|\mathcal{C}| \leq n$. By ii, $\text{diam}_{\max}(\mathcal{C}') \leq 4 \lceil \log n \rceil \cdot \text{diam}_{\max}(\mathcal{C})$. Now, let $B' \in \mathcal{C}'$. We can assume w.l.o.g. that there is a cluster $B \in \mathcal{C}$ such that $B \subseteq B'$ (otherwise B' can be removed from \mathcal{C}'). B is a cloud cluster of some node $i \in B'$, and therefore by Theorem 6 and by Definition 4,

we get that $Z(B') \leq Z(B) = O(Z_i) = O(Z_{\max})$. Since this bound holds for all clusters of \mathcal{C}' , $Z(\mathcal{C}') = O(Z_{\max})$.

An $O(\text{diam}_{\max}(\mathcal{C}) \cdot \log^2 n + Z_{\max} \cdot \log^2 n)$ time bound for **cComb** is derived by applying Theorem 10 to cover \mathcal{C}' . Finally, let $B_j \in \mathcal{C}$ be a cloud cluster of diameter $\text{diam}_{\max}(\mathcal{C})$. Recall that by Definition 3, $\text{diam}(B_j) \leq 2k(j) \leq 2Z_j \leq 2Z_{\max}$. We therefore obtain an upper bound of $O(Z_{\max} \log^2 n)$ rounds. ■

We close with a lower bound.

Theorem 13. *Let $G = (V, E, w)$ be a CWC system with fat links. Then **cComb** requires $\Omega(Z_{\max})$ rounds.*

cCast. To implement **cCast**, one can reverse the schedule of **cComb**. However, a slightly better implementation is possible, because there is no need to ever write to the cloud node. More specifically, let \mathcal{C} be a cover of V_p . In the algorithm for **cCast**, each cluster leader invokes **cR**, and then the leader disseminates the result to all cluster members. The time complexity for a single cluster B is $O(Z(B))$ for the **cR** operation, and $O(\text{diam}(B))$ rounds for the dissemination of S throughout B (similarly to Lemma 1). Using the multiplexing to load(\mathcal{C}) as in Step 2 of Algorithm 3, we obtain the following result.

Theorem 14. *Let $G = (V, E, w)$ be a CWC system with fat links. Then **cCast** can be performed in $O(Z_{\max} \cdot \log^2 n)$ rounds.*

Finally, we note that since any algorithm for **cCast** also solves cR_i problem for every node i , we get from Theorem 9 the following result.

Theorem 15. *Let $G = (V, E, w)$ be a CWC system with fat links. Any algorithm solving **cCast** requires $\Omega(Z_{\max})$ rounds.*

4 Non-commutative Operators and the Wheel Settings

In this section we consider **cComb** for non-commutative operators in the wheel topology (Fig. 1). Our description here omits many details that can be found in the full version [2].

Consider an instance with a non-commutative operator. Trivially, Algorithm 3 can be used (and Theorem 10 can be applied) if the ordering of the inputs happens to match ordering induced by the algorithm. While such a coincidence is unlikely in general, it seems reasonable to assume that processing nodes are physically connected according to their combining order. Neglecting other possible connections, assuming that the last node is also connected to the first node for symmetry, and connecting a cloud node to all processors, we arrive at the *wheel* topology, which we study in this section. We assume that all links are bidirectional and bandwidth-symmetric.

We start with the concept of intervals that refines the concept of clusters (Definition 1) to the case of the wheel topology.

Definition 7. The **cloud bandwidth** of a processing node $i \in V_p$ in a given wheel graph is $b_c(i) \stackrel{\text{def}}{=} w(i, v_c)$. An **interval** $[i, i+k] \stackrel{\text{def}}{=} \{i, i+1, \dots, i+k\} \subseteq V$ is a path of processing nodes in the ring. Given an interval $I = [i, i+k]$, $|I| = k+1$ is its **size**, and k is its **length**. The **cloud bandwidth** of I , denoted $b_c(I)$, is the sum of the cloud bandwidth of all nodes in I : $b_c(I) = \sum_{i \in I} b_c(i)$. The **bottleneck bandwidth** of I , denoted $\phi(I)$, is the smallest bandwidth of a link in the interval: $\phi(I) = \min \{w(i, i+1) \mid i, i+1 \in I\}$. If $|I| = 1$, define $\phi(I) = \infty$.

(Note that bottleneck bandwidth was not defined for general clusters).

Given these interval-related definitions, we adapt the notion of “cloud cluster” (Definition 3), for problems with inputs s , this time also accounting for the bottleneck of the interval. We define I_i to be the **cloud interval** of node i , and $Z_i = |I_i| + \frac{s}{\phi(I_i)} + \frac{s}{b_c(I_i)}$ to be the **timespan** of I_i .

Similarly to fat-links, we obtain the following results for cW_i and cR_i .

Theorem 16. *In the wheel settings, cW_i can be solved in $O(Z_i)$ rounds for every node i .*

Theorem 17. *In the wheel settings, Any algorithm for cW_i requires at least $\Omega(Z_i)$ rounds for every node i .*

Our main result in this section is an algorithm for $c\text{Comb}$ for the wheel topology with arbitrary bandwidths, that works in time bounded by $O(\log n)$ times the optimal. We note that by using standard methods [24], the presented algorithm can be extended to compute, with the same asymptotic time complexity, all prefix sums, i.e., compute $\bigotimes_{i=0}^j S_i$ for each $0 \leq j < n$.

Extending the notion of Z_{\max} to the wheel case, and adapting Algorithm 3, we obtain the following theorem.

Theorem 18. *In the wheel settings, $c\text{Comb}$ can be solved in $O(Z_{\max} \log n)$ rounds by Algorithm 3.*

This is a log factor improvement over the fat-links case. The main ideas are as follows:

- In the wheel case, for any minimal cover C' of the graph, $\text{load}_{C'}(i) \leq 2$ for every node i . Furthermore, a minimal cover is easy to find without resorting to Theorem 12 (see, e.g., [26]).
- Due to the limited local bandwidth, Algorithm 4 can't be used with the same time analysis as in the fat-links case in Steps 3–5 of Algorithm 3. Instead, we use pipelining to compute the inner product of every interval in the cover.

Pipelining. We distinguish between *holistic* and *modular* combining operators, defined as follows (see [2] for details). In modular combining, one can apply the combining operator to aligned, equal-length parts of operands to get the output corresponding to that part. For example, this is the case with vector (or matrix) addition: to compute any entry in the sum, all that is needed is

the corresponding entries in the summands. If the operand is not modular, it is called holistic (e.g., matrix multiplication). We call the aligned parts of the operands *grains*, and their maximal size g is the *grain size*. We show that in the modular case, using pipelining, a logarithmic factor can be shaved off the running time (more precisely, converted into an additive term), as can be seen in the following theorem:

Theorem 19. *Suppose \otimes is modular with grain size g , and that $w(e) \geq g$ for every link $e \in E$. Then $cComb$ can be solved in $O(Z_{\max} + \log n)$ rounds, where $Z_{\max} = \max\{Z_i \mid i \in V_p\}$.*

5 CWC Applications

In this section we briefly explore some of the possible applications of the results shown in this paper to two slightly more involved applications, namely Federated Learning (Sect. 5.1) and File Deduplication (Sect. 5.2).

5.1 Federated Learning in CWC

Federated Learning (FL) [11,31] is a distributed Machine Learning training algorithm, by which an ML model for some concept is acquired. The idea is to train over a huge data set that is distributed across many devices such as mobile phones and user PCs, without requiring the edge devices to explicitly exchange their data. Thus it gives the end devices some sense of privacy and data protection. Examples of such data is personal pictures, medical data, hand-writing or speech recognition, etc.

In [8], a cryptographic protocol for FL is presented, under the assumption that any two users can communicate directly. The protocol of [8] is engineered to be robust against malicious users, and uses cryptographic machinery such as Diffie-Hellman key agreement and threshold secret sharing. We propose a way to do FL using only cloud storage, without requiring an active trusted central server. Here, we describe a simple scheme that is tailored to the fat-links scenario, assuming that users are “honest but curious.”

The idea is as follows. Each of the users has a vector of m weights. Weights are represented by non-negative integers from $\{0, 1, \dots, M-1\}$, so that user input is simply a vector in $(\mathbb{Z}_M)^m$. Let \mathbf{x}_i be the vector of user i . The goal of the computation is to compute $\sum_{i=0}^{n-1} \mathbf{x}_i$ (using addition over \mathbb{Z}_M) and store the result in the cloud. We assume that M is large enough so that no coordinate in the vector-sum exceeds M , i.e., that $\sum_{i=0}^{n-1} \mathbf{x}_i = \left(\sum_{i=0}^{n-1} \mathbf{x}_i \bmod M\right)$.

To compute this sum securely, we use basic multi-party computation in the CWC model. Specifically, each user i chooses a private random vector $\mathbf{z}_{i,j} \in (\mathbb{Z}_M)^m$ uniformly, for each of her neighbors j , and sends $\mathbf{z}_{i,j}$ to user j . Then each user i computes $\mathbf{y}_i = \mathbf{x}_i - \sum_{(i,j) \in E} \mathbf{z}_{i,j} + \sum_{(j,i) \in E} \mathbf{z}_{j,i}$, where addition is modulo M . Clearly, \mathbf{y}_i is uniformly distributed even if \mathbf{x}_i is known. Also note that $\sum_i \mathbf{y}_i = \sum_i \mathbf{x}_i$. Therefore all that remains to do is to compute $\sum_i \mathbf{y}_i$, which

can be done by invoking `cComb`, where the combining operator is vector addition over $(\mathbb{Z}_M)^m$. We obtain the following theorem from Theorem 11.

Theorem 20. *In a fat-links network, an FL iteration with vectors in $(\mathbb{Z}_M)^m$ can be computed in $O(Z_{\max} \log^2 n)$ rounds.*

Since the grain size of this operation is $O(\log M)$ bits, we can apply the pipelined version of `cComb` in case that the underlying topology is a cycle, to obtain the following.

Theorem 21. *In the uniform n -node wheel, an FL iteration with vectors in $(\mathbb{Z}_M)^m$ can be computed in $O(\sqrt{(m \log M)/b_c} + \log n)$ rounds, assuming that $b_c m \log M \leq b_\ell^2$ and $b_c \geq \log M$.*

5.2 File Deduplication with the Cloud

Deduplication, or Single-Instance-Storage (SIS), is a central problem for storage systems (see, e.g., [7, 17, 32]). Grossly simplifying, the motivation is the following: Many of the files (or file parts) in a storage system may be unknowingly replicated. The general goal of deduplication (usually dubbed dedup) is to identify such replications and possibly discard redundant copies. Many cloud storage systems use a dedup mechanism internally to save space. Here we show how the processing nodes can cooperate to carry out dedup without active help from the cloud, when the files are stored locally at the nodes (cf. serverless SIS [13]). We ignore privacy and security concerns here.

We consider the following setting. Each node i has a set of local files F_i with their hash values, and the goal is to identify, for each unique file $f \in \bigcup_i F_i$, a single owner user $u(f)$. (Once the operation is done, users may delete any file they do not own).

This is easily done with the help of `cComb` as follows. Let h be a hash function. For file f and processing node i , call the pair $(h(f), i)$ a *tagged hash*. The set $S_i = \{(h(f), i) \mid f \in F_i\}$ of tagged hashes of F_i is the input of node i . Define the operator $\tilde{\cup}$ that takes two sets S_i and S_j of tagged hashes, and returns a set of tagged hashes without duplicate hash values, i.e., if (x, i) and (x, j) are both in the union $S_i \cup S_j$, then only $(x, \min(i, j))$ will be in $S_i \tilde{\cup} S_j$. Clearly $\tilde{\cup}$ is associative and commutative, has a unit element (\emptyset) , and therefore can be used in the `cComb` algorithm. Note that if the total number of unique files in the system is m , then $s = m \cdot (H + \log n)$. Applying `cComb` with operation $\tilde{\cup}$ to inputs S_i , we obtain a set of tagged hashes S for all files in the system, where $(h(f), i) \in S$ means that user i is the owner of file f . Then we invoke `cCast` to disseminate the ownership information to all nodes. Thus dedup can be done in CWC in $O(Z_{\max} \log^2 n)$ rounds.

6 Conclusion and Open Problems

In this paper we have introduced a new model that incorporates cloud storage with a bandwidth-constrained communication network. We have developed

a few building blocks in this model, and used these primitives to obtain effective solutions to some real-life distributed applications. There are many possible directions for future work; below, we mention a few.

One interesting direction is to validate the model with *simulations and/or implementations* of the algorithms, e.g., implementing the federated learning algorithm suggested here.

A few algorithmic question are left open by this paper. For example, can we get good approximation ratio for the problem of combining in a general (directed, capacitated) network? Our results apply to fat links and the wheel topologies.

Another interesting issue is the case of *multiple cloud nodes*: How can nodes use them effectively, e.g., in combining? Possibly in this case one should also be concerned with privacy considerations.

Finally, *fault tolerance*: Practically, clouds are considered highly reliable. How should we exploit this fact to build more robust systems? and on the other hand, how can we build systems that can cope with varying cloud latency?

References

1. Adler, M., Gibbons, P., Matias, Y., Ramachandran, V.: Modeling parallel bandwidth: local versus global restrictions. *Algorithmica* **24**, 381–404 (1999)
2. Afek, Y., Giladi, G., Patt-Shamir, B.: Distributed computing with the cloud. arXiv e-prints, [arXiv:2109.12930](https://arxiv.org/abs/2109.12930), September 2021
3. Attiya, H., Welch, J.: *Distributed Algorithms*. McGraw-Hill, New York (1998)
4. Augustine, J., Hinnenthal, K., Kuhn, F., Scheideler, C., Schneider, P.: Shortest paths in a hybrid network model. In: *Proceedings of the SODA 2020*, pp. 1280–1299 (2020)
5. Awerbuch, B., Peleg, D.: Sparse partitions (extended abstract). In: *31st FOCS*, pp. 503–513. IEEE Computer Society (1990)
6. Baumann, N., Skutella, M.: Solving evacuation problems efficiently-earliest arrival flows with multiple sources. In: *Proceedings of the 47th FOCS*, pp. 399–410 (2006)
7. Bolosky, B., Corbin, S., Goebel, D., Douceur, J.: Single instance storage in windows 2000. In: *Proceedings of the 4th USENIX Windows Systems Symposium USENIX* (2000)
8. Bonawitz, K., et al.: Practical secure aggregation for privacy-preserving machine learning. In: *Proceedings of the CCS 2017*, pp. 1175–1191. ACM (2017)
9. Burkard, R.E., Dlaska, K., Klinz, B.: The quickest flow problem. *ZOR - Methods Models Oper. Res.* **37**, 31–58 (1993)
10. Censor-Hillel, K., Leitersdorf, D., Polosukhin, V.: Distance computations in the hybrid network model via oracle simulations. In: *Proceedings of the 38th STACS* (2021)
11. Cheng, Y., Liu, Y., Chen, T., Yang, Q.: Federated learning for privacy-preserving AI. *Commun. ACM* **63**(12), 33–36 (2020)
12. Culler, D., et al.: LogP: towards a realistic model of parallel computation. *SIGPLAN Not.* **28**(7), 1–12 (1993)
13. Douceur, J.R., Adya, A., Bolosky, W.J., Simon, P., Theimer, M.: Reclaiming space from duplicate files in a serverless distributed file system. In: *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pp. 617–624 (2002)

14. Dropbox. Prospectus. Filing to US Securities and Exchange Commission (2018). <https://www.sec.gov/Archives/edgar/data/1467623/000119312518055809/d451946ds1.htm>
15. Fleischer, L., Skutella, M.: Quickest flows over time. *SIAM J. Comput.* **36**(6), 1600–1630 (2007)
16. Fraigniaud, P.: Distributed computational complexities: are you Volvo-addicted or NASCAR-obsessed? In: Proceedings of the 30th PODC, pp. 171–172. ACM (2010)
17. Freeman, L.: Looking beyond the hype: Evaluating data deduplication solutions. Network Appliance Inc., September 2007. http://www-download.netapp.com/edm/TT/docs/Looking_beyond_hype_Dedupe.pdf
18. Friedman, R., Kliot, G., Kogan, A.: Hybrid distributed consensus. In: Baldoni, R., Nisse, N., van Steen, M. (eds.) OPODIS 2013. LNCS, vol. 8304, pp. 145–159. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03850-6_11
19. Gibbons, P., Matias, Y., Ramachandran, V.: Can shared-memory model serve as a bridging model for parallel computation? In: Proceedings of the 9th SPAA, pp. 72–83. ACM (1997)
20. Hegeman, J.W., Pemmaraju, S.V.: Lessons from the congested clique applied to MapReduce. *Theor. Comput. Sci.* **608**(P3), 268–281 (2015)
21. Hoppe, B., Tardos, É.: Polynomial time algorithms for some evacuation problems. In: SODA 1994 (1994)
22. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proceedings of the 21st SODA. Society for Industrial and Applied Mathematics (2010)
23. Kuhn, F., Schneider, P.: Computing shortest paths and diameter in the hybrid network model. In: Proceedings of the 39th PODC, pp. 109–118. ACM (2020)
24. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. *J. ACM* **27**(4), 831–838 (1980)
25. Lardinois, F.: Google Drive Will Hit a Billion Users This Week. TechCrunch, July 2018
26. Lee, C.C., Lee, D.T.: On a circle-cover minimization problem. *Inf. Process. Lett.* **18**(2), 109–115 (1984)
27. Li, M., Vitényi, P.: An Introduction to Kolmogorov Complexity and its Applications, 4th edn. Springer, Heidelberg (2019)
28. Linial, N.: Locality in distributed graph algorithms. *SICOMP* **21**, 193–201 (1992)
29. Lotker, Z., Patt-Shamir, B., Pavlov, E., Peleg, D.: Minimum-weight spanning tree construction in $O(\log \log(n))$ communication rounds. *SICOMP* **35**(1), 120–131 (2005)
30. Mansour, Y., Nisan, N., Vishkin, U.: Trade-offs between communication throughput and parallel time. *J. Complex.* **15**(1), 148–166 (1999)
31. McMahan, B., Moore, E., Ramage, D., Hampson, S., Agüera y Arcas, B.: Communication-efficient learning of deep networks from decentralized data. In: PMLR, vol. 54, pp. 1273–1282 (2017)
32. Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. *ACM Trans. Storage* **7**(4) (2012)
33. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. Society for Industrial and Applied Mathematics, Philadelphia (2000)
34. Skutella, M.: An introduction to network flows over time. In: Cook, W.J., Lovász, L., Vygen, J. (eds.) *Research Trends in Combinatorial Optimization*, pp. 451–482. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-76796-1_21
35. Valiant, L.G.: A bridging model for parallel computation. *Commun. ACM* **33**(8), 103–111 (1990)