Distributed Computing (2024) 37:1–18
https://doi.org/10.1007/s00446-024-00460-w

Distributed computing with the cloud

Yehuda Afek[1] · **Gal Giladi**[1] · **Boaz Patt-Shamir**[2]

Received: 14 September 2022 / Accepted: 5 January 2024 / Published online: 1 February 2024

Abstract

We investigate the effect of omnipresent cloud storage on distributed computing. To this end, we specify a network model with links of prescribed bandwidth that connect standard processing nodes, and, in addition, passive storage nodes. Each passive node represents a cloud storage system, such as Dropbox, Google Drive etc. We study a few tasks in this model, assuming a single cloud node connected to all other nodes, which are connected to each other arbitrarily. We give implementations for basic tasks of collaboratively writing to and reading from the cloud, and for more advanced applications such as matrix multiplication and federated learning. Our results show that utilizing node-cloud links as well as node-node links can considerably speed up computations, compared to the case where processors communicate either only through the cloud or only through the network links. We first show how to optimally read and write large files to and from the cloud in general graphs using flow techniques. We use these primitives to derive algorithms for *combining*, where every processor node has an input value and the task is to compute a combined value under some given associative operator. In the special but common case of "fat links," where we assume that links between processors are bidirectional and have high bandwidth, we provide near-optimal algorithms for any commutative combining operator (such as vector addition). For the task of matrix multiplication (or other non-commutative combining operators), where the inputs are ordered, we present tight results in the simple "wheel" network, where procesing nodes are arranged in a ring, and are all connected to a single cloud node.

1 Introduction

In 2018 Google announced that the number of users of Google Drive is surpassing one billion [28]. Earlier that year, Dropbox stated that in total, more than an exabyte (10^{18} bytes) of data has been uploaded by its users [15]. 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 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 well advertised. A lesser known fact is that a cloud system also entails a massive component of *communication*, that makes it appear close to 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 bits per time unit to its 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 $\Omega(n)$ time units—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 $\Omega(n/\sqrt{n}) = \Omega(\sqrt{n})$ time units—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 $\tilde{\Theta}(\sqrt[4]{n})$ time units, as we show in this paper.

✉ Boaz Patt-Shamir
boaz@tau.ac.il

Yehuda Afek
afek@tauex.tau.ac.il

Gal Giladi
gal.giladi.cs@gmail.com

[1] School of CS, Tel Aviv University, 6997801 Tel Aviv, Israel

[2] School of EE, Tel Aviv University, 6997801 Tel Aviv, Israel

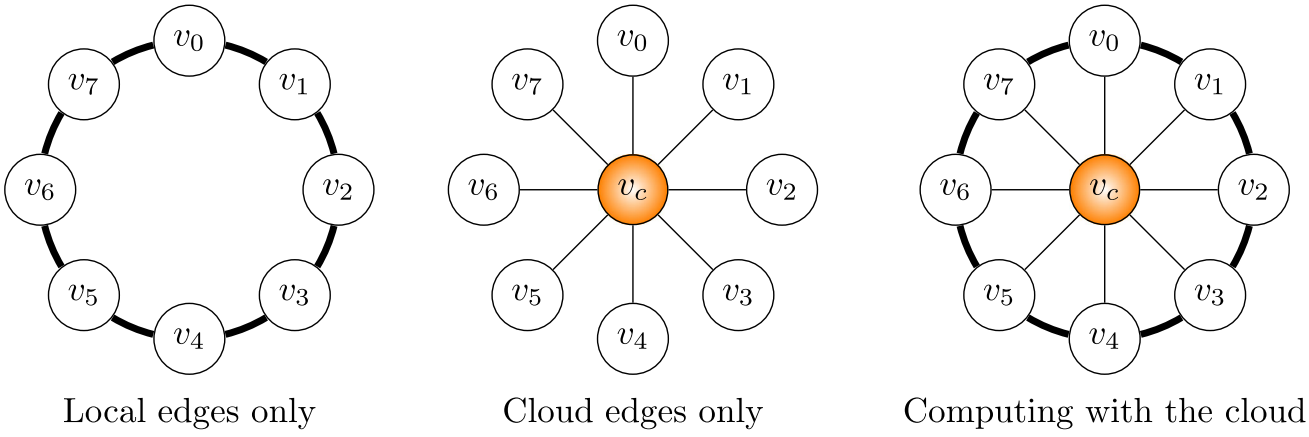

Fig. 1 Wheel topology with $n = 8$. The v_i nodes are processing nodes connected by a ring of high-bandwidth links. The cloud node v_c is connected to the processing nodes by lower-bandwidth links. All links are bidirectional and symmetric

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 to compute schedules that efficiently combine distributed inputs to compute various functions, such as vector addition and matrix multiplication. To this end, we first show how to implement (using dynamic flow techniques) primitive operations that allow for the efficient exchange of large messages (files) between processing nodes and cloud nodes. Finally, we show how to use the combining schedules to implement some popular applications, including federated learning [34] and file de-duplication (dedup) [35].

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).

The set of links connecting processing nodes is denoted by E_L ("local links"), and E_C ("cloud links") denotes 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 $w(e) > 0$ (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, based on CONGEST [36]. Executions proceed in synchronous rounds, 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 space; to write, a processing node v_i invokes `FW` with arguments that describe the filename f, a bit string S, and the location (index) within f that is the starting point for writing S. It is assumed that $|S| \leq w(v_i, v_c)$ bits (longer writes are implemented by a sequence of `FW` operations). To read, a processing node v_i invokes `FR` with arguments that describe 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., when an `FW` operation is executing, no other operation (read or write) referring to the same file locations is allowed to take place concurrently. Concurrent `FR` operations reading 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 fibers to Bluetooth or infrared links. Typically in this setting the local links have bandwidth much

[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 $w(v_i, v_c)$.

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 device 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

In this paper we address the question of how to efficiently move information around a CWC network (the model described above). To avoid confusion, we start by distinguishing between *schedules*, which specify what string of bits to send over each link in every step (cf. Definition 2.3), and *algorithms*, which compute these schedules. While the schedules are inherently parallel because they concern all links in each step, the algorithms we present in this paper are typically centralized. Our main objective is to find optimal or near-optimal schedules (i.e., whose timespan is close to the best possible). For the positive results, we present efficient (i.e., polynomial-time) algorithms that produce such schedules.

Our main results are algorithms to compute schedules that allow the users to combine values stored at nodes. These schedules use building blocks that facilitate efficient transmission of large messages between processing nodes and cloud nodes. These building block schedules, in turn, are computed in a straightforward way using dynamic flow techniques. Finally, we show how to use the combining schedules to derive new algorithms for federated learning and file deduplication (dedup) in the CWC model. More specifically, we provide implementations of the following tasks.

Basic cloud operations: Let $s \in \mathbb{N}$ be a given parameter. We use v_c to 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 $\otimes : \{0,1\}^s \times \{0,1\}^s \to \{0,1\}^s$. The requirement is to write to a cloud node v_c the s-bits string $S_1 \otimes S_2 \otimes \cdots \otimes S_n$. 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 $\overline{\mathbf{1}}$, such that $\overline{\mathbf{1}} \otimes S = S \otimes \overline{\mathbf{1}} = S$ 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. Note that we require that the result of applying $\otimes$ is as long as each of its operands.
- 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) [34]: 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 [34], 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., [35]. 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 schedules 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 [6].

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. 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 (operand), the second node holds the second input etc. The wheel is obtained by connecting the first and the last nodes for symmetry, and then connecting all to a cloud node.

Overview of techniques. As mentioned above, schedules for the basic file operations (cW, cR, cAW and cAR) are computed optimally using dynamic flow techniques, or more specifically, *quickest flow* (Sect. 2), which have been studied in numerous papers in the past (cf. [10, 37]). We present closed-form bounds on cW and cR for the wheel topology with general link bandwidths in Sect. 4.

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 (Theorems 3.7, 3.8, 3.10). 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 [36], and then combining the results in a computation-tree fashion. We perform the described procedure in near-optimal time.

Non-commutative operators are explored in the natural wheel topology. We present algorithms to compute schedules for wheel networks with *arbitrary* bandwidth (both cloud and local links). We prove an upper bound for cComb (Theorem 4.6) and a nearly-matching lower bound (Theorem 4.10).

Paper organization. In Sect. 2 we study the topology of basic primitives in the CWC model. In Sect. 3 we study combining schedules in general topologies in fat links networks. In Sect. 4 we consider combining for non-commutative operators in the wheel topology. In Sect. 5 we discuss application level usage of the CWC model, such as federated learning and deduplication. Conclusions and open problems are presented in Sect. 6.

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 non-comprehensive review.

Historically, distributed computing is split along the dichotomy of message passing vs shared memory [18]. 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 [31], and its derived CONGEST [36]. (Some models also include a broadcast channel, e.g. [2].) In both LOCAL and CONGEST, 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, and 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., [5]). 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.

The CONGESTED CLIQUE (CC) model [32] is a special case of CONGEST, where the underlying 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 [22]. Another model for computing in the cloud is the MPC model [24]. Recently, the HYBRID model [6] 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, 6, 11, 25, 26].

Another classical model of easily accessible shared memory is the PRAM [17], whose original focus was on detailed complexity of parallel computation. An early attempt to incorporate communication bandwidth constraints in PRAM, albeit indirectly, is due to Mansour et al. [33], who considered PRAM with m words of shared memory, p processors and input length n, in the regime where $m \ll p \ll n$. The LogP model [13] by Culler et al. aimed at adjusting the PRAM model to network-based realizations. Another proposal that models shared memory explicitly is the QSM model of Gibbons et al. [20], in which there is an explicit (typically uniform) limit on the bandwidth connecting processors

to shared memory. QSM is similar to our CWC model, except that there is no network connecting processors directly. A thorough comparison of bandwidth limitations in variants of the BSP model [38] and of QSM is presented by Adler et al. [1]. We note that LogP and BSP do not provide shared memory as a primitive object: the idea is to describe systems in a way that allows implementations of abstract shared memory.

Recently, Aguilera et al. [3] introduced the "m&m" model that combines message passing with shared memory. The m&m model assumes that processes can exchange messages via a fully connected network, and there are shared registers as well, where each shared register is accessible only by a subset of the processes. The focus in [3] is on solvability of distributed tasks in the presence of failures, rather than performance.

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 models computational bandwidth. 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 [38]), 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 Implementation of basic communication primitives in CWC

In this section we consider the basic operations of reading or writing to the cloud, by one or all processors. First, we give optimal results using standard dynamic flow techniques. While optimal, the resulting schedules are somewhat opaque in the sense that they do not give much intuition about the construction. We then consider the special case of networks with fat links, where we introduce the notion of "cloud clusters" that are both intuitive and can be used in a straightforward way to give good approximate solutions to the basic tasks.

We first review dynamic flows in Sect. 2.1, and then apply them to the CWC model in Sect. 2.2. Cloud clusters for fat-links networks are introduced in Sect. 2.3.

2.1 Dynamic flows

The concept of *quickest flow* [10], a variant of *dynamic flow* [37], is defined as follows. [2] A *flow network* consists of a directed weighted graph $G = (V, E, c)$ where $c : E \to \mathbb{N}$, with a distinguished source node and a sink node, denoted $s, t \in V$, respectively. A *dynamic flow* with *time horizon* $T \in \mathbb{N}$ and *flow value* F is a mapping $f : E \times [1, T] \to \mathbb{N}$ that specifies for each edge e and time step j, how much flow e carries between steps $j - 1$ and j, subject to the natural constraints:

- Edge capacities. For all $e \in E$, $j \in [1, T]$:

$$f(e, j) \leq c(e) \tag{1}$$

- Only arriving flow can leave. For all $v \in V \setminus \{s\}$, $j \in [1, T-1]$:

$$\sum_{i=1}^{j} \sum_{(u,v)\in E} f((u, v), i) \geq \sum_{i=1}^{j+1} \sum_{(v,w)\in E} f((v, w), i) \tag{2}$$

- No leftover flow at time T. For all $v \in V \setminus \{s, t\}$:

$$\sum_{j=1}^{T} \sum_{(u,v)\in E} f((u, v), j) = \sum_{j=1}^{T} \sum_{(v,w)\in E} f((v, w), j) \tag{3}$$

- Flow value (source).

$$\sum_{j=1}^{T} \left(\sum_{(s,v)\in E} f((s, v), j) - \sum_{(u,s)\in E} f((u, s), j) \right) = F \tag{4}$$

- Flow value (sink).

$$\sum_{j=1}^{T} \left(\sum_{(t,v)\in E} f((t, v), j) - \sum_{(u,t)\in E} f((u, t), j) \right) = -F \tag{5}$$

[2] We simplify the original definition to our context by setting all transmission times to 1.

Usually in dynamic flows, T is given and the goal is to maximize F. In the quickest flow variant, the roles are reversed:

Definition 2.1 Given a flow network, the **quickest flow** for a given value F is a dynamic flow f satisfying (1–5) above with flow value F, such that the time horizon T is minimal.

Theorem 2.1 *([10]) The quickest flow be computed in strongly polynomial time.*

The *Evacuation Problem* [23] is a variant of dynamic flow that we use, specifically the case of a single sink node [7]. In this problem, each node v has an initial volume of $F(v) \geq 0$ flow units, and the goal is to ship all the flow volume to a single sink node t in shortest possible time (every node v with $F(v) > 0$ is considered a source). Similarly to the single source case, shipment is described by a mapping $f : E \times [1, T] \to \mathbb{N}$ where T is the time horizon. The dynamic flow is subject to the edge capacity constraints (Eq. 1), and the following additional constraints:

- Only initial and arriving flow can leave. For all $v \in V, j \in [1, T-1]$:

$$F(v) + \sum_{i=1}^{j} \sum_{(u,v)\in E} f((u,v),i) \geq \sum_{i=1}^{j+1} \sum_{(v,w)\in E} f((v,w),i) \tag{6}$$

- Flow value (sources). For all $v \in V \setminus \{t\}$:

$$\sum_{j=1}^{T} \left(\sum_{(v,w)\in E} f((v,w),j) - \sum_{(u,v)\in E} f((u,v),j) \right) = F(v) \tag{7}$$

- Flow value (sink).

$$\sum_{j=1}^{T} \left(\sum_{(u,t)\in E} f((u,t),j) - \sum_{(t,w)\in E} f((t,w),j) \right) = \sum_{v\in V} F(v) \tag{8}$$

Formally, we use the following definition and result.

Definition 2.2 Given a flow network in which each node v has value $F(v)$, a solution to the evacuation problem is a dynamic flow f with multiple sources satisfying (1) and (6–8). The solution is *optimal* if the time horizon T of f is minimal.

Theorem 2.2 ([7]) *An optimal solution to the evacuation problem can be computed in strongly polynomial time.*

2.2 Using dynamic flows in CWC

In this section we show how to implement (i.e., compute schedules for) basic cloud access primitives using dynamic flow algorithms. These are the tasks of reading and writing to or from the cloud, invoked by a single node (cW and cR), or by all nodes (cAW and cAR). Our goal in all the tasks and algorithms is to find a schedule that implements the task in the minimum amount of time.

Definition 2.3 Given a CWC model, a **schedule** for time interval I is a mapping that assigns, for each time step in I and each link (u, v): a send (or null) operation if (u, v) is a local link, and a FW or FR (or null) operation if (u, v) is a cloud link.

We present optimal solutions to these problems in general directed graphs, using the quickest flow algorithm.

2.2.1 Serving a single node

Let us consider cW first. We start with a lemma stating the close relation between schedules as defined above and dynamic flows.

Lemma 2.3 *Let $G = (V, E, w)$ be a graph in the CWC model. There exists a schedule implementing cW_i from processing node i to cloud node v_c with string S of size s in T rounds if and only if there exists a dynamic flow of value s and time horizon T from source node i to sink node v_c.*

Proof Converting a schedule to a dynamic flow is trivial, as send and receive operations between processing nodes and FW and FR operations directly translate to a dynamic flow that transports the same amount of flow satisfying bandwidth constraints. For the other direction, let f be a dynamic flow of time horizon T and value s from node i to the cloud v_c. We construct a schedule implementing cW_i as follows.

First, we construct another dynamic flow f' which is the same as f, except that no flow leaves the sink node v_c. Formally, given a dynamic flow g, a node v and a time step t, let $\text{stored}_g(v, t)$ be the volume of flow stored in v at time t according to g. Flow f' is constructed by induction; At time step 1, f' is defined to be the same as f, except for setting $f'(e, 1) = 0$ for every edge e that leaves v_c. Let $t \in \{1, T-1\}$. For step $t+1$, f' is defined to be the same as f, except for capping the total flow that leaves any node v by $\text{stored}_{f'}(v, t)$, and setting $f'(e, 1) = 0$ for every edge e that leaves the sink.

Let $G_p = G - \{v_c\}$, and let $\text{stored}_g(G_p, t)$ denote $\sum_{v\in V_p} \text{stored}_g(v, t)$ for some dynamic flow g. Initially, $\text{stored}_f(G_p, 0) = \text{stored}_{f'}(G_p, 0) = s$, and by the induction, in every step $t \in \{1, T\}$, $\text{stored}_f(G_p, t) \geq \text{stored}_{f'}(G_p, t)$ due to flow that was not sent from v_c to G_p.

Also, in time step T, stored$_f(G_p, T) = 0$ since all flow was sent to the sink, and thus stored$_{f'}(G_p, T) = 0$ as well and stored$_{f'}(v_c, T) = s$ due to flow conservation. Therefore, f' is a dynamic flow with time horizon (at most) T and value s.

Given f', we construct a schedule $\mathcal{S}$ to implement cW_i by encoding flow using the operations of message send, message receive, and FW (no FR operations are required because in f' flow never leaves v_c). To specify which data is sent in every operation of the schedule, refer to all FW operations of the schedule. Let K be the number of FW operations in $\mathcal{S}$, let i_k be the node initiating the k-th call to FW and let l_k be the size of the message in that call. We assign the data transferred on each link during $\mathcal{S}$ so that when node i_k runs the k-th FW, it writes the l_k bits of S starting from index $s \cdot \sum_{j=1}^{k-1} l_j$. Note that processing nodes do not need to exchange indices of the data they transfer, as all nodes can calculate in preprocess time the schedule and thus "know in advance" the designated indices of the transferred data.

Correctness of the schedule S follows from the validity of f, as well as its time complexity. □

Theorem 2.4 *Given any instance $G = (V, E, w)$ of the CWC model, an optimal schedule realizing cW_i can be computed in polynomial time.*

Proof Consider a cW issued by a processing node i, wishing to write s bits to cloud node v_c. We construct an instance of quickest flow as follows. The flow network is G where w is the link capacity function, node i is the source and v_c is the sink. The requested flow value is s. The solution, computed by Theorem 2.1, is directly translatable to a schedule, after assigning index ranges to flow parts according to Lemma 2.3. Optimality of the resulting schedule follows from the optimality of the quickest flow algorithm. □

► *Remarks.*

- Note that in the presence of multiple cloud nodes, it may be the case that while writing to one cloud node, another cloud node is used as a relay station.
- Schedule computation can be carried out off-line: we can compute a schedule for each node i and for each required file size s (possibly consider only powers of $1+\epsilon$ for some $\epsilon > 0$), so that in run-time, the initiating node would only need to tell all other nodes which schedule to use.

Next, we observe that the reduction sketched in the proof of Theorem 2.4 works for reading just as well: the only difference is reversing the roles of source and sink, i.e., pushing s flow units from the cloud node v_c to the requesting node i. We therefore have the following theorem.

Theorem 2.5 *Given any instance of the CWC model, an optimal schedule realizing cR_i can be computed in polynomial time.*

2.2.2 Serving multiple nodes

Consider now operations with multiple invocations. Let us start with cAW (cAR is analogous, as above). Recall that in this task, each node has a (possibly empty) file to write to a cloud node. If all nodes write to the same cloud node, then using the evacuation problem variant of the quickest flow algorithm solves the problem (see Definition 2.2), Specifically, we have the following theorem.

Theorem 2.6 *Given any instance $G = (V, E, w)$ of the CWC model, an optimal schedule realizing cAW (cAR) in which every node i needs to write (read) a message of size s_i to (from) cloud node v_c can be computed in strongly polynomial time.*

Proof Similarly to Lemma 2.3, it is easy to see that there is such a schedule for cAW if and only if there is a dynamic flow solving the evacuation problem (Definition 2.2). Thus in order to solve cAW, we can construct a flow network as in Theorem 2.4, apply Theorem 2.2 to get the solution, and then translate it into a schedule. A schedule for cAR can be obtained by reversing the schedule for cAW, similarly to Theorem 2.5. □

However, in the case of multiple cloud nodes, we resort to the quickest *multicommodity* flow, defined as follows [37]. We are given a flow network as described in Sect. 2.1, but with k source-sink pairs $\{(s_i, t_i)\}_{i=1}^{k}$, and k *demands* $d_1, \ldots, d_k$. We seek k flow functions f_i, where f_i describes the flow of d_i units of commodity i from its source s_i to its sink t_i, subject to the usual constraints: the edge capacity constraints (1) applies to the sum of all k flows, and the node capacity constraints (2–3), as well as the source and sink constraints (4–5) are specified for each commodity separately.

It is known that determining whether there exists a feasible quickest multicommodity flow with a given time horizon T is NP-hard, but on the positive side, there exists an FPTAS to it [16], i.e., we can approximate the optimal T to within $1+\epsilon$, for any constant $\epsilon > 0$. Extending Theorem 2.6 in the natural way, we obtain the following result.

Theorem 2.7 *Given any instance of the CWC model and $\epsilon > 0$, a schedule realizing cAW or cAR can be computed in time polynomial in the instance size and ϵ^{-1}. The length of the schedule is at most $(1+\epsilon)$ times larger than the optimal length.*

2.3 Cloud clusters in fat-links networks

Recall that in the case of an s-fat-links network, all local links have bandwidth at least s, and all links are symmetric. For this case we develop a rather intuitive framework for the basic tasks, introducing the concept of cloud clusters.

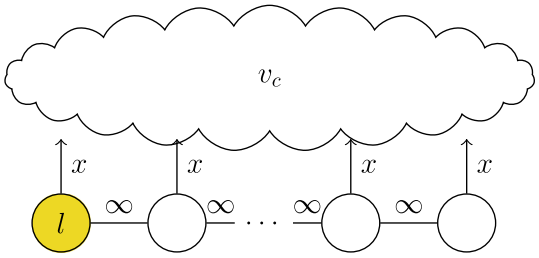

Fig. 2 A simple path example. The optimal distance to travel in order to write an s-bits file to the cloud would be $\sqrt{s/x}$

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 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. The schedules produced by 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 2.4 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 $\mathrm{diam}(B)$, is the maximum distance between any two nodes of B in the induced graph $G[B]$: $\mathrm{diam}(B) = \max_{u,v \in B} dist_{G[B]}(u, v)$.

We use the following definition for the network when ignoring the cloud. Note that the metric here is hop-based—w indicates link bandwidths.

Algorithm 1 cW_i

1: Construct a BFS spanning tree of B_i rooted at node i and assign for each index $1 \le x \le |B_i|$ 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 2.5 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.

Finally, we define the concept of cloud cluster of a node.

Definition 2.6 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

$$k_s(i) \stackrel{\text{def}}{=} \min(\{\mathrm{diam}(G_p)\} \cup \{k \mid (k+1) \cdot b_c(B_k(i)) \ge s\}) .$$

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 2.8 *Given a fat-links CWC system, Algorithm 1 solves the s-bits cW_i problem in $O(Z_i)$ rounds on B_i.*

Proof The algorithm broadcasts all s bits to all nodes in B_i, and then each node writes a subrange of the data whose size is proportional to its cloud bandwidth. Correctness is obvious. As for the time analysis: Steps 1–2 require $O(k(i))$ rounds. In the loop of steps 4–5, $b_c(B_i)$ bits are sent in every round, and thus it terminates in $O(s/b_c(B_i))$ rounds. The theorem follows from the definition of Z_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., [30]). 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 2.9 *Any algorithm solving cW_i in a fat-links CWC requires $\Omega(Z_i)$ rounds.*

Proof By definition, $Z_i = k(i) + s/b_c(B_i)$. Lemmas 2.10 and 2.11 show that each term of Z_i is a lower bound on the running time of any algorithm for cW_i. □

Lemma 2.10 *Any algorithm solving cW_i in a fat-links CWC system requires $\Omega(k(i))$ rounds.*

Proof Let A be an algorithm for cW_i that writes string S in t_A rounds. If $t_A \geq \mathrm{diam}(G_p)$ then $t_A \geq k(i)$ and we are done. Otherwise, we count the number of bits of S that can get to the cloud in t_A rounds. Since S is initially stored in i, in a given round t, only nodes in $B_{t-1}(i)$ can write pieces of S to the cloud. Therefore, overall, A writes to the cloud at most $\sum_{t=1}^{t_A} b_c(B_{t-1}(i)) \leq t_A \cdot b_c(B_{t_A-1}(i))$ bits. Hence, by assumption that A solves cW_i, we must have $t_A \cdot b_c(B_{t_A-1}(i)) \geq s$. The lemma now follows from definition of $k(i)$ as the minimal integer $\ell \leq \mathrm{diam}(G_p)$ such that $(\ell+1) \cdot b_c(B_\ell(i)) \geq s$ if it exists (otherwise, $k_c(i) = \mathrm{diam}(G_p)$ and $t_A > \mathrm{diam}(G_p)$). Either way, we are done. □

Lemma 2.11 *Any algorithm solving cW_i in a fat-links CWC system requires $\Omega(s/b_c(B_i))$ rounds.*

Proof Let A be an algorithm that solves cW_i in t_A rounds. If $k(i) = \mathrm{diam}(G_p)$ then B_i contains all processing nodes V_p, and the claim is obvious, as no more than $b_c(V_p)$ bits can be written to the cloud in a single round. Otherwise, $k(i) \geq s/b_c(B_i) - 1$ by Definition 2.6, and we are done since $t_A = \Omega(k(i))$ by Lemma 2.10. □

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

Theorem 2.13 *cR_i in a fat-links CWC requires $\Omega(Z_i)$ rounds.*

► *Remark:* The lower bound of Theorem 2.9 and the definition of cloud clusters (Definition 2.6) 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.

► *Large operands.* Suppose that a given CWC system S is s-fat-links, and we need to implement an operation (e.g., cW) with operand size $s' > s$. We can still use the algorithms from this section, at the cost of increasing the running time by a "scaling factor" of $\lceil s'/s \rceil$. The idea is to first compute a schedule for the operation in a system S' which is identical to S except that all link bandwidths are multiplied by the scaling factor $\lceil s'/s \rceil$. The schedule for S' is then emulated in S by executing each step of S' by $\lceil s'/s \rceil$ steps of S.

We note that in this case the resulting schedule of S may be non-optimal, because our optimality arguments do not apply to the emulation. More concretely, suppose that the schedule produced for a certain operation in S' is T steps long. Then, by the lower bounds in this section, we know that the operation cannot be implemented in $o(T)$ steps in S', implying the same lower bound for S (because S is less powerful than S'). The emulation outlined above shows that the operation can be implemented in S in $\lceil s'/s \rceil T$ steps, so the best implementation time in S may be anywhere in the intersection of $\Omega(T)$ and $O\left(\lceil s'/s \rceil T\right)$.

3 Computing and writing combined values

In this section we consider combining operations, reminiscent of convergecast in CONGEST [36]. Note that 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 from the number of bits leaving it).

In Sect. 3.1 we explain how to implement cComb in the general case using cAW and cAR. The implementation is simple and generic, but may be inefficient. We offer partial remedy in Sect. 3.2, where we present one of our main results: an algorithm to compute schedules for cComb in the special (but common) case, where $\otimes$ 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 $O(\log n)$ factor.

3.1 Combining non-commutative operators in general graphs

We now present algorithms to compute schedules 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 as their operands.

Theorem 3.1 *Let T_s be the time required for cAW (and cAR) when all files have size s. Then Algorithm 2 produces a schedule for cComb whose timespan is in $O(T_s \log n)$ rounds.*

Proof The idea is to do the combining over a binary "computation tree" by using the cloud to store the partial results.

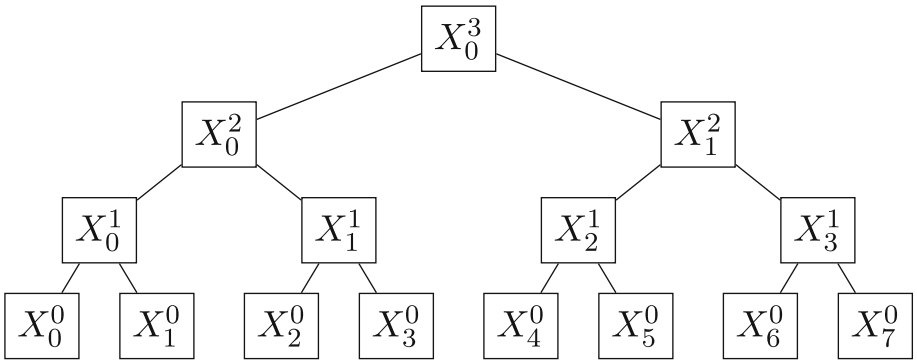

Fig. 3 Computation tree with $n = 8$. X_i^j denotes the result stored in node i in iteration j

Algorithm 2 High-level algorithm for cComb using cAW and cAR

1: $m := n$, $j := 0$
2: for all $0 \le i < n$ set $X_i^0 := S_i$
3: **while** $m > 1$ **do**
4: run cAW with inputs X_i^j for $0 \le i < m$ *// processor i holds X_i^j*
5: run cAR with inputs X_{2i}^j for $0 \le i < \lfloor m/2 \rfloor$ *// read left children*
6: **if** m is even **then**
7: run cAR with inputs X_{2i+1}^j for $0 \le i < \lfloor m/2 \rfloor$
8: **else**
9: run cAR with inputs X_{2i+1}^j for $0 \le i < \lfloor m/2 \rfloor - 1$; set $X_{m-1}^j := \overline{\mathbf{1}}$ *// take care of single-child parent*
10: **end if**
11: $m := \lceil m/2 \rceil$ *// processor i now holds X_{2i}^j and X_{2i+1}^j*
12: for all $0 \le i < m$, in parallel, node i calculates $X_i^{j+1} := X_{2i}^j \otimes X_{2i+1}^j$ locally
13: $j := j + 1$
14: **end while**
15: run cW from node 0 to write X_0^j to the cloud

The computation tree is defined as follows (see Fig. 3). Let X_i^j denote the i-th node at level j, as well as the value of that node. The leaves X_i^0 are the input values, and the value of an internal node X_i^{j+1} at level $j+1$ with left child X_{2i}^j and right child X_{2i+1}^j is $X_{2i}^j \otimes X_{2i+1}^j$. Pseudocode is provided in Algorithm 2. Correctness of the algorithm follows from the observation that after each execution of Step 4, there are m files of size s written in the cloud, whose product is the required output, and that m is halved in every iteration. If at any iteration m is odd, then node $\lceil m/2 \rceil - 1$ only needs to read one file, and therefore we set the other file that it reads to be $\overline{\mathbf{1}}$. When m reaches 1, there is only 1 file left, which is the required result.

As for the time analysis: Clearly, a single iteration of the **while** loop takes $3T_s = O(T_s)$ rounds.[3] There are $\lceil \log n \rceil$ iterations due to Step 11. Step 15 is completed in $O(T_s)$ rounds, and thus the total schedule time is $O(T_s \log n)$. □

[3] Note that when writing $\overline{\mathbf{1}}$ to the cloud, it can be encoded as a 0-bit string, which can only improve the runtime.

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 concurrent reads and disallows concurrent writes to the same memory location. We have the following result.

Theorem 3.2 *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.*

Proof First note that if there were n copies of the input file S in the cloud, then cCast and cAR would have been the exact same problem. The theorem follows from the observation that any algorithm for cAR with n inputs of size s in the cloud can be modified so that each invocation of FR with argument S_i is converted to FR with argument S (the input of cCast). □

3.2 Combining commutative operators in fat links network

In some cases, we can implement cComb more efficiently than the promise of Theorem 3.2. Specifically, if the combining operator is commutative and the network is an s-fat links network, we use the cloud clusters (as defined in Sect. 2.3) to do the combining in time which is at most a polylogarithmic factor larger than optimum. The saving is a result of using multiple concurrent cW and cR operations instead of cAW and cAR.

The high-level idea is simple. Assume that we are given a partition $\mathcal{C} = \{B_1, \ldots, B_k\}$ of the nodes set into clusters, i.e., $\cup_i B_i = V_p$ and $B_i \cap B_j = \emptyset$ for $i \ne j$. We first compute the result of combining within each cluster B_i using the local network and then combine the cluster results using the cloud operations. Below, we first explain how to implement cComb using any given partition $\mathcal{C}$ of the nodes. We later show how to find good partitions.

Consider first combining within clusters. We assume that we are given a distinguished *leader* node $r(B) \in B$ for each cluster $B \in \mathcal{C}$. We proceed as follows (see Algorithm 3). First we construct, in each cluster $B \in \mathcal{C}$, a spanning tree rooted at $r(B)$. We then apply convergecast using $\otimes$ over the tree. Clearly (see, e.g., [36]) we have:

Algorithm 3 Computing the combined result of cluster B at leader $r(B)$

Construct a BFS tree of B rooted at node $r(B)$.
Let h be the height of the tree, and let child(v) denote the children of a node v in the tree.
for $d := h$ **to** 2 **do**
  for all $i \in B$ at layer d of the tree, in parallel **do**
    if i is not a leaf **then**
      i computes $S'_i := S_i \otimes \bigotimes_{j \in \text{child}(i)} S'_j$
    else
      $S'_i := S_i$
    end if
    i sends S'_i to its parent node in the tree
  end for
end for
Node $r(B)$ computes $P_B := S_{r(B)} \otimes \left(\bigotimes_{j \in \text{child}(r(B))} S'_j\right)$

Algorithm 4 Computing the high level tree-nodes values

for $l := \lceil \log |\mathcal{C}| \rceil$ **to** 1 **do**
  for all tree-nodes y in layer l of the computation tree, in parallel **do**
    Let $B := \text{cl}(y)$
    if y is not a leaf **then**
      Let y_ℓ and y_r be the left and the right children of y, respectively.
      $r(B)$ invokes cR for $\text{vl}(y_\ell)$
      $r(B)$ invokes cR for $\text{vl}(y_r)$
      $r(B)$ computes $\text{vl}(y) := \text{vl}(y_\ell) \otimes \text{vl}(y_r)$
    else
      $\text{vl}(y) := P_B$ *// if y is a leaf its value is already stored at $r(B)$*
    end if
    $r(B)$ invokes cW for $\text{vl}(y)$
  end for
end for

Lemma 3.3 *Algorithm 3 computes $P_B = \bigotimes_{i \in B} S_i$ at node $r(B)$ in $O(\text{diam}(B))$ rounds.*

When Algorithm 3 terminates in all clusters, the combined result of every cluster is stored in its leader. We combine the cluster results by filling in the values of a virtual computation tree defined over the clusters (see Fig. 3). The leaves of the tree are the combined values of the clusters of $\mathcal{C}$, as computed by Algorithm 3. To fill 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 4 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 after this deletion the rightmost leaf is the only child of its parent, we delete the rightmost leaf and repeat 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 $\text{cl}(y)$. 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 4). 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. To analyze the running time we use the following definition.

Definition 3.1 Let $\mathcal{C}$ be a partition of the processing nodes into connected clusters, and let B be a cluster in $\mathcal{C}$. 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 **timespan of the partition $\mathcal{C}$**, denoted $Z(\mathcal{C})$, is the maximum timespan of its clusters.

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.

With these definitions, we can state the following result.

Lemma 3.4 *Let $\{P_1, \ldots, P_m\}$ be the values stored at the leaders of the clusters when Algorithm 4 is invoked. Then Algorithm 4 computes $\bigotimes_{i=1}^{m} P_i$ in $O(Z(\mathcal{C}) \cdot \log |\mathcal{C}|)$ rounds.*

Proof Computing all values in a tree layer requires a constant number of cW and cR invocations in a cluster, i.e., by Definition 3.1, at most $O(Z(\mathcal{C}))$ rounds of work in every layer. The number of layers is $\lceil \log |\mathcal{C}| \rceil$. The result follows. □

Combining Lemma 3.3 and Lemma 3.4, we can give an upper bound for any given cover $\mathcal{C}$.

Theorem 3.5 *Given a partition $\mathcal{C}$ of the processing nodes where all clusters have diameter at most $D_{\max}$, cComb can be solved in a fat-links CWC in $O\left(D_{\max} + Z(\mathcal{C}) \cdot \log |\mathcal{C}|\right)$ rounds.*

► *Remark.* We note that in Algorithm 4, 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. By using optimal schedules for cAW and cAR, the running-time can only improve beyond the upper bound of Theorem 3.5.

Algorithm 5 Computing a partition of the processing nodes

1: Let $\mathcal{C}_0 = \{B_i \mid i \in V_p\}$ be the set of all cloud clusters *// cf. Definition 2.6*
2: Let $\mathcal{C}_1 \subseteq \mathcal{C}_0$ be a maximal set of disjoint clusters from $\mathcal{C}_0$ *// any maximal set will do*
3: Let V_1 be shorthand for $\bigcup_{B\in\mathcal{C}_1} B$ *// V_1 denotes the set of nodes covered by $\mathcal{C}_1$*
4: **while** $V_1 \neq V_p$ **do**
5: **for** all $j \in V_p \setminus V_1$ in parallel **do**
6: **if** j has a neighbor in some $B \in \mathcal{C}_1$ **then**
7: $B \leftarrow B \cup \{j\}$ *// if more than one such B, choose arbitrarily*
8: **end if**
9: **end for**
10: **end while**
11: output $\mathcal{C}_1$

Partitioning the nodes. We now arrive at the problem of designing a good partition. This is done as follows (see Algorithm 5). We consider the set of all s-cloud clusters (recall that a cloud cluster is defined for each processing node in Definition 2.6). From these clusters we pick a maximal set $\mathcal{C}_1$ of disjoint clusters, say by the greedy algorithm. We then extend the clusters of $\mathcal{C}_1$ to cover all nodes while keeping them disjoint, in a breadth-first fashion: each iteration of the while loop (line 4) adds another layer of uncovered nodes to each cluster. We have the following:

Lemma 3.6 *The output of Algorithm 5 is a collection of disjoint clusters whose union is V_p. Furthermore, the diameter of the largest cluster in the output is at most $3 \cdot D_{\max}$, where $D_{\max}$ is the largest s-cluster diameter.*

Proof The disjointness of the sets in the output follows by induction: The initial clusters are disjoint by line 2, and thereafter, clusters are extended only by nodes which are not already members in other clusters in $\mathcal{C}_1$ (line 5). Clearly, by line 7, a node is added to a single cluster.

It is easy to see that when the algorithm terminates, the sets in the output cover all nodes: the distance of a node from $\mathcal{C}_1$ decreases by 1 in each iteration of the while loop. Finally, regarding the diameter of clusters in $\mathcal{C}_1$, note that by maximality of the input $\mathcal{C}_0$, each node not covered by $\mathcal{C}_1$ after the execution of line 2 is at distance at most $D_{\max}$ from some node in $\mathcal{C}_1$. Since the diameter of each cluster increases by at most 2 in each iteration of the while loop, and since there are at most $D_{\max}$ iterations, the lemma follows. □

Conclusion. We now arrive at our main result for this section. We use the following definition.

Definition 3.2 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 2.9).

Theorem 3.7 *Let $G = (V, E, w)$ be a CWC system with fat links.*

Then cComb *with a commutative combining operator can be solved in $O(Z_{\max} \log n)$ rounds.*

Proof Let $\mathcal{C}$ be the partition output by Algorithm 5. By Lemma 3.6, the diameter of any cluster in $\mathcal{C}$ is at most 3 times larger than the maximal diameter of any s-cloud cluster of the system. It follows from Lemma 3.3 that we can apply Algorithm 3 in each cluster $B \in \mathcal{C}$ and compute its combined value in time proportional to its diameter, which is bounded by $Z(B)$ (cf. Definition 2.6). Since the clusters are disjoint by Lemma 3.6, we can compute all these values in parallel in time $O(Z(\mathcal{C}))$. Finally, we invoke Algorithm 4, which produces the desired result in additional $O(Z(\mathcal{C}) \cdot \log |\mathcal{C}|)$ rounds by Lemma 3.4. □

Our algorithm produces schedules which are optimal to within a logarithmic factor, as stated in the following theorem.

Theorem 3.8 *Let $G = (V, E, w)$ be a CWC system with fat links.*

Then cComb *requires $\Omega(Z_{\max})$ rounds.*

Proof By reduction from cW. Let i be any processing node. Given a bit string S, assign $S_i = S$ as the input of node i in cComb, and for every other node $j \neq i$, assign $S_j = \overline{\mathbf{1}}$. Clearly, any algorithm for cComb that runs with these inputs solves cW_i with input S. The result follows from Theorem 2.9. □

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 partition 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 3.3). We obtain the following result.

Theorem 3.9 *Let $G = (V, E, w)$ be a CWC system with fat links. Then* cCast *can be performed in $O(Z_{\max})$ rounds.*

Finally, we note that since any algorithm for cCast also solves cR_i problem for every node i, we get from Theorem 2.13 the following result.

Theorem 3.10 *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).

Trivially, Theorem 3.5 applies in the non-commutative case if the ordering of the nodes happens to match an ordering induced by the algorithm, but this need not be the case in general. However, 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.

Our main result in this section is an algorithm for cComb in the wheel topology that works in time which is a logarithmic factor larger than optimal. In contrast to the result of Theorem 3.7 that applies any topology but requires fat links, here we analyze a particular topology with arbitrary bandwidths. We note that by using standard methods [27], the algorithm presented in this section 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 $0 \leq j < n$.

4.1 Cloud intervals and the complexity of cW and cR in the wheel topology

We start by defining cloud intervals. Cloud intervals in the wheel settings correspond to cloud clusters in general topologies, defined in Sect. 2.3.

Definition 4.1 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** $[i, i+k] \stackrel{\text{def}}{=} \{i, i+1, \ldots, i+k\} \subseteq V$ 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 $\phi(I) = \infty$.

For ease of presentation we consider the "one sided" case in which node i does not use one of its incident ring links. As we shall see, this limitation does not increase the time complexity by more than a constant factor.

Definition 4.2 Let i be a processing node in the wheel settings. We define the following quantities for clockwise intervals; counterclockwise intervals are defined analogously.

- $k_c(i)$ is the length of the smallest interval starting at i, for which the product of its size by the total bandwidth to the cloud along the interval exceeds s, i.e.,

$$k_c(i) = \min\left(\{n\} \cup \{k \mid (k+1) \cdot b_c([i, i+k]) \geq s\}\right) .$$

- $k_\ell(i)$ is the length of the smallest clockwise interval starting at node i, for which the bandwidth of the clockwise-boundary link bandwidth is smaller than the total cloud bandwidth of the interval, i.e.,

$$k_\ell(i) = \min\Big(\{n\} \cup \{k\} \mid w(i+k, i+k+1) < b_c([i, i+k])\Big) .$$

- $k(i) = \min\{k_c(i), k_\ell(i)\}$.
- $I_i = [i, i+k(i)]$. The interval I_i is called the (clockwise) **cloud interval** of node i.
- $Z_i = |I_i| + \frac{s}{\phi(I_i)} + \frac{s}{b_c(I_i)}$. Z_i is the **timespan** of the (clockwise) cloud interval of i.

The concept of cloud intervals is justified by the following results.

Theorem 4.1 *Given the cloud interval I_i of node i, Algorithm 1 solves the s-bits cW$_i$ problem in $O(Z_i)$ rounds.*

Proof The BFS tree of the interval would be a simple line graph, that is the whole interval. Step 2 of Algorithm 1 requires $O\left(|I_i| + \frac{s}{\phi(I_i)}\right)$ rounds: there are s bits to send over $\Theta(|I_i|)$ hops with bottleneck bandwidth $\phi(I_i)$. The rest of the time analysis is the same as in Theorem 2.8. □

We have the following immediate consequence.

Theorem 4.2 *Let Z_i^ℓ and Z_i^r denote the timespans of the counterclockwise and the clockwise cloud intervals of i, respectively. Then cW$_i$ can be solved in $O(\min(Z_i^\ell, Z_i^r))$ rounds.*

The upper bounds are essentially tight, as we show next. We start with one sided intervals.

Theorem 4.3 *In the wheel settings, any algorithm for cW$_i$ which does not use link $(i-1, i)$ requires $\Omega(Z_i)$ rounds.*

Proof We show that each term of Z_i is a lower bound on the running time of any algorithm solving cW$_i$. First, note that any algorithm for cW$_i$ that does not use edge $(i-1, i)$ requires $\Omega(k_c(i)) \geq \Omega(k(i)) = \Omega(|I_i|)$ rounds, due to the exact same arguments as in Lemma 2.10.

Next, we claim that any algorithm for cW$_i$ which does not use edge $(i-1, i)$ requires $\Omega(s/\phi(I_i))$ rounds. To see that note first that if $k(i) = 0$, then $\phi(I_i) = \infty$ and the claim is trivial. Otherwise, let $(j, j+1) \in E$ be any link in I_i with $w(j, j+1) = \phi(I_i)$. Note that $j - i < k(i)$ because $j+1 \in I_i$. Consider the total bandwidth of links emanating from the

interval $I' \stackrel{\text{def}}{=} [i, j]$. Since we assume that the link $(i-1, i)$ is not used, the number of bits that can leave I' in t rounds is at most $t \cdot \big(b_c(I') + w(j, j+1)\big)$. Notice that at least s bits have to leave I'. Observe that $b_c(I') \leq w(j, j+1)$, because otherwise we would have $k_\ell(i) = j - i$, contradicting the fact that $k(i) > j - i$. Therefore, any algorithm A that solves cW_i in t_A rounds satisfies

$$s \leq t_A \cdot \big(b_c(I') + w(j, j+1)\big) \leq 2t_A \cdot w(j, j+1) = 2t_A \cdot \phi(I_i) ,$$

and the claim follows.

Finally, we claim that any algorithm A for cW_i which does not use edge $(i-1, i)$ requires $\Omega(s/b_c(I_i))$ rounds. To see that, recall that $k(i) = \min(k_c(i), k_\ell(i))$. If $k(i) = n$ then I_i contains all processor nodes V_p, and the claim is obvious, as no more than $b_c(V_p)$ bits can be written to the cloud in a single round. Otherwise, we consider the two cases: If $k(i) = k_c(i)$, then $k(i) \geq s/b_c(I_i) - 1$ by definition, and we are done since $t_A = \Omega(k(i))$. Otherwise, $k(i) = k_\ell(i)$. Let us denote $w_R = w(i+k(i), i+k(i)+1)$. In this case we have $w_R < b_c(I_i)$. We count how many bits can leave I_i. In a single round, at most $b_c(I_i)$ bits can leave through the cloud links, and at most w_R bits can leave through the local links. Since A solves cW_i, we must have $s \leq t_A \cdot (b_c(I_i) + w_R) \leq 2t_A \cdot b_c(I_i)$, and hence $t_A = \Omega(s/b_c(I_i))$. □

Theorem 4.4 *Let Z_i^ℓ and Z_i^r denote the timespans of the counterclockwise and the clockwise cloud intervals of i, respectively. Then cW_i requires $\Omega(\min(Z_i^\ell, Z_i^r))$ rounds in the wheel settings.*

Proof Let T be the minimum time required to perform cW_i. Due to Lemma 2.3, we know that there is a dynamic flow mapping with time horizon T and flow value s from node i to the cloud. Let f be such a mapping. We assume that no flow is transferred *to* the source node i, as we can modify f so that these flow units would not be sent from i at all until the point where they were previously sent back to i. Let s_L and s_R be the total amount of flow that is transferred on links $(i-1, i)$ and $(i, i+1)$, respectively, and assume w.l.o.g. that $s_R \geq s_L$. Let f' be a new dynamic flow mapping which is the same as f, except that no flow is transferred on link $(i-1, i)$. Since f is a valid dynamic flow that transfers all s flow units from i to the cloud, f' has flow value at least $s - s_L$. Let A be a schedule derived from f'. The runtime of A is at most T rounds. Let A' be a schedule that runs A twice: A' would transfer $2(s - s_L)$ bits from node i to the cloud. Since $s \geq s_R + s_L$, we get that: $2(s - s_L) = 2s - 2s_L \geq s + s_L + s_R - 2s_L \geq s$, and thus A' solves cW_i without using link $(i-1, i)$. From Theorem 4.3, we get a lower bound for $2T$ of $\Omega(Z_i^r) = \Omega(\min(Z_i^\ell, Z_i^r))$. □

From Theorem 4.2 and Theorem 4.4 we get the following corollary for the uniform wheel:

Corollary 4.5 *Consider the uniform wheel topology, where all cloud links have bandwidth b_c, and all local links have bandwidth $b_\ell \geq b_c$. In this case cW can be solved in $\Theta\left(\frac{s}{b_\ell} + \min(\sqrt{\frac{s}{b_c}}, \frac{b_\ell}{b_c})\right)$ rounds. If $b_\ell < b_c$, the running time is $\Theta(s/b_c)$ rounds.*

Proof If $b_\ell < b_c$, $k_\ell(i) = 0$, $\phi(I_i) = \infty$ and the result follows. Otherwise, by definition we have $k_c(i) = \sqrt{s/b_c} - 1$ and $k_\ell(i) = b_\ell/b_c - 1$, hence $|I_i| = O(\min(\sqrt{s/b_c}, b_\ell/b_c))$. It follows that $b_c(I_i) = O(\min(\sqrt{s \cdot bc}, b_\ell))$. The result follows by noting that $\phi(I_i) = b_\ell$. □

Recall the example of Fig. 1: there we have $b_c = \sqrt{s}$ and $b_\ell \geq s^{3/4}$, and the running time is $O(s^{1/4})$.

► *Remark.* Notice that the same upper and lower bounds hold for the cR_i problem as well.

4.2 Combining in the wheel setting

We are now ready to adapt Theorem 3.5 to the wheel settings.

Definition 4.3 Given an n-node wheel, for each processing node i, let I_i be the cloud interval of i with the smaller timespan (clockwise or counter-clockwise). Define $j_{\max} = \text{argmax}_i \{|I_i|\}$, $j_c = \text{argmin}_i \{b_c(I_i)\}$, and $j_\ell = \text{argmin}_i \{\phi(I_i)\}$. Finally, define $Z_{\max} = |I_{j_{\max}}| + \frac{s}{\phi(I_{j_\ell})} + \frac{s}{b_c(I_{j_c})}$.

In words: $j_{\max}$ is the node with the longest cloud interval, j_c is the node whose cloud interval has the least cloud bandwidth, and j_ℓ is the node whose cloud interval has the narrowest bottleneck.

Our upper bound is as follows.

Theorem 4.6 *In the wheel settings, cComb can be solved in $O(Z_{\max} \log n)$ rounds.*

The general approach to prove Theorem 4.6 is similar to the one taken in the fat-links case: partition the nodes into clusters (intervals in the wheel case), compute combined values in the clusters using local links, and then complete the computation tree using cR and cW. The third stage is identical to the fat-links case, but the first two are not. We elaborate on them now.

► *Partitioning the nodes.* Given the set of cloud intervals, we construct a *cover*, i.e., a (not necessarily disjoint) set of cloud intervals whose union is V_p, the set of all processing nodes. We can do this so that every node is a member in a constant number of intervals in the cover.

Specifically, let $\mathcal{C}$ be the set of all cloud intervals I_i. We select a cover $\mathcal{C}' \subseteq \mathcal{C}$ such that every node is a member of

either one or two intervals of $\mathcal{C}'$. It is straightforward to find such a cover, say, by a greedy algorithm. In fact, a cover with a minimal *number of intervals* is found by the algorithm in [29] (in $O(n \log n)$ sequential time). The covers produced by [29] are sufficient for that matter, as the following lemma states.

Lemma 4.7 *Let $\mathcal{C}$ be a collection of intervals and denote $U = \bigcup_{I \in \mathcal{C}} I$. Let $\mathcal{C}' \subseteq \mathcal{C}$ be a minimal-cardinality cover of U, and let $\text{load}_{\mathcal{C}'}(i) = |\{I \in \mathcal{C}' : I \ni i\}|$. Then for all $i \in U$, $1 \leq \text{load}_{\mathcal{C}'}(i) \leq 2$.*

Proof Clearly $\text{load}_{\mathcal{C}'}(i) \geq 1$ for all $i \in U$ since $\mathcal{C}'$ is a cover of U. For the upper bound, first note that by the minimality of $|\mathcal{C}'|$, there are no intervals $I, I' \in \mathcal{C}'$ such that $I \subseteq I'$, because in this case I could have been discarded. This implies that the right-endpoints of intervals in $\mathcal{C}'$ are all distinct, as well as the left-endpoints. Now, assume for contradiction, that there exist three intervals $I, I', I'' \in \mathcal{C}'$ such that $I \cap I' \cap I'' \neq \emptyset$ (i.e., there is at least one node which is a member of all three). Then $I \cup I' \cup I''$ is a contiguous interval. Let $l = \min(I \cup I' \cup I'')$ and $r = \max(I \cup I' \cup I'')$. Clearly, for one of the three intervals, say I, no endpoint is l or r. But this means that $I \subseteq I' \cup I''$, i.e., we can discard I, in contradiction to the minimality of $|\mathcal{C}'|$. □

After selecting the minimal cover, we need to take care of node 0: We require that the interval that contains node 0 does not contain node $n-1$. If this is not the case after computing the cover, we split the interval $I_0 \in \mathcal{C}'$ that contains node 0 into two subintervals $I_0 = I_0^L \cup I_0^R$, where I_0^L is the part that ends with node $n-1$, and I_0^R is the part that starts with node 0.

► *Computing within intervals.* Given a cover $\mathcal{C}$, we compute the combined values within each cloud intervals in $\mathcal{C}$. We now explain how this is done. First, we require that each input S_i is associated with a single interval in $\mathcal{C}$. To this end, we use the rule that if a node i is a member in two intervals I and I', then its input S_i is associated with the interval I satisfying $\max(I) < \max(I')$, and a unit input $\overline{\mathbf{1}}$ is associated with i in the context of I', where $\overline{\mathbf{1}}$ is the unit (neutral) operand for $\otimes$. Intuitively, this rule means that the overlapping regions in an interval are associated with the "left" (counterclockwise) interval.

To do the computation, we apply a computation tree combining approach now *within* the intervals. Consider an interval I, and let $p = 2^{\lceil \log |I| \rceil}$. We map I to a complete binary tree with p leaves, where the leftmost $p - |I|$ leaves have the unit input $\overline{\mathbf{1}}$. These leaves are emulated by the leftmost node of I (the emulation is trivial). The actual computation proceeds in stages, where each stage ℓ computes all level-ℓ products in parallel. Let S_i^j denote the product of $S_i, \ldots, S_j$. The algorithm maintains the invariant that after S_i^j is computed, it is stored in node j. Initially, by assumption, for all $0 \leq i < n$, we have that S_i^i is stored at node i. The computation of a stage is performed as follows.

Let $S_i^j = S_i^k \otimes S_{k+1}^j$ be a product we wish to compute at level ℓ, and let S_i^k, S_{k+1}^j be the values held by its children. Note that $k + 1 - i = j - k = 2^{\ell-1}$. The algorithm forwards S_i^k from node k to node j, which multiplies it by (the locally stored) S_{k+1}^j, thus computing S_i^j, which is stored in node j for the next level. This way, the number of communication rounds is just the time required to forward s bits from k to j. Using pipelining, the number of rounds required is

$$j - k + \frac{s}{\phi([k, j])} = 2^{\ell-1} + \frac{s}{\phi([k, j])} \leq 2^{\ell-1} + \frac{s}{\phi(I)} . \tag{9}$$

We therefore have the following lemma.

Lemma 4.8 *Computing the combined value of an interval I can be done in $O\left(|I_{j_{\max}}| + \log |I_{j_{\max}}| \cdot \frac{s}{\phi(I_{j_\ell})}\right)$ rounds in the wheel settings, with P_I stored in the rightmost node of I for each cloud interval I.*

Proof We compute the values as described above. Correctness is obvious. Regarding time complexity, we conclude from Eq. 9 that the total time required to compute the product of all inputs of any interval I is at most

$$\sum_{\ell=1}^{\lceil \log |I| \rceil} \left(2^{\ell-1} + \frac{s}{\phi(I)}\right) \leq 2\left(|I| + \log |I| \cdot \frac{s}{\phi(I)}\right) . \tag{10}$$

□

Finally, we have to deal with the possible overlap of intervals in the cover. This is done by simple time-multiplexing: by Lemma 4.7, each node is contained in at most two intervals, and hence the set of even-numbered intervals (starting from the interval containing node 0) are disjoint, and so is the set of the odd-numbered intervals. Time multiplexing is done by activating all even-numbered intervals every even time slot, and activating the odd-numbered intervals at every odd time slot. This way we can compute the combined values of all intervals in parallel, with only a constant factor increase in the asymptotic complexity stated in Lemma 4.8.

► *Combining the interval values.* After computing the combined values of each interval of C, we combine these values in a computation tree fashion using cW and cR, as in Algorithm 4. Again, we have to deal with the overlap, and we do it using multiplexing. A slight complication here is due to the possible splitting of the cloud interval that contains node 0: the cR and cW operations still require the full cloud interval. To solve this problem, we multiplex the parallel invocations of cR and cW over four time slots: one of the even-numbered intervals excluding the interval containing node 0, two for the

interval containing node 0 (one for each of its sub-intervals), and one for odd-numbered intervals.

We have the following lemma.

Lemma 4.9 *Given the interval combined values, the root value of the computation tree is computed in* $O(Z_{\max} \cdot \log n)$ *rounds.*

Proof As explained above, by employing time multiplexing, we can run Algorithm 4 with a constant-factor slowdown. Thus, similarly to Lemma 3.4, completing the computation tree requires $O(Z_{\max}(C') \cdot \log |C'|)$ rounds. Noting that $|C'| \leq |C| \leq n$ and that all intervals in C' are cloud intervals, we get an upper bound of $O(Z_{\max} \cdot \log n)$ rounds. □

The proof of Theorem 4.6 can now be completed as follows. The correctness of the algorithm is derived from the general case (Theorem 3.5). As for the time analysis: The low levels of the algorithm (computing within intervals) require $O(Z_{\max} \log n)$ according to Lemma 4.8, noting that $|I_{j_{\max}}| \leq n$ and that $Z_{\max} = |I_{j_{\max}}| + \frac{s}{\phi(I_{j_\ell})} + \frac{s}{b_c(I_{j_c})}$ by definition. The high levels of the algorithm (completing the computation tree using cR and cW) require $O(Z_{\max} \cdot \log n)$ according to Lemma 4.9. All in all, the algorithm terminates in $O(Z_{\max} \cdot \log n)$ rounds. □

We close our treatment of the wheel topology with the lower bound.

Theorem 4.10 *Any algorithm for* cComb *in the wheel topology requires* $\Omega(Z_{\max})$ *rounds.*

Proof Similarly to Theorem 3.8, $\Omega(Z_i)$ is a lower bound for every node i, by reduction from cW_i. Recall that $Z_i = |I_i| + \frac{s}{\phi(I_i)} + \frac{s}{b_c(I_i)}$. For index $j_{\max}$ we get a lower bound of $\Omega(Z_{j_{\max}}) \in \Omega(|I_{j_{\max}}|)$. For index j_ℓ we get a lower bound of $\Omega(s/\phi\left(I_{j_\ell}\right))$. For index j_c we get a lower bound of $\Omega\left(s/b_c(I_{j_c})\right)$. Summing them all up, gives the desired lower bound. □

Remark. We note that if the combining operator can be applied to the operands in a piecewise fashion (as in vector addition, where a coordinate of the sum can be computed based only on the corresponding values of the coordinates of the summands), tighter pipelining is possible, yielding overall complexity which is essentially $O(Z_{\max} + \log n)$. Details can be found in [21].

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) [12, 34] is a distributed Machine Learning training paradigm, by which a 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 [9], a cryptographic protocol for FL is presented, under the assumption that any two users can communicate directly. The protocol of [9] 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 $\{0, 1, \ldots, M-1\}$, 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 3.7.

Theorem 5.1 *In a fat-links network, an FL iteration with vectors in* $(\mathbb{Z}_M)^m$ *can be computed in* $O(Z_{\max} \log n)$ *rounds.*

In the wheel case we obtain the following.

Theorem 5.2 *In the uniform n-node wheel, an FL iteration with vectors in* $(\mathbb{Z}_M)^m$ *can be computed in* $O(\sqrt{(m \log M)/b_c} \cdot \log n)$ *rounds, assuming that* $b_c m \log M \leq b_\ell^2$ *and* $b_c \geq \log M$.

Proof Using the notation of Sect. 3 and Sect. 4, the assumption implies that $s = m \log M$, and $Z_{\max} = O(\sqrt{s/b_c}) = O(\sqrt{(m \log M)/b_c})$. The result follows from Theorem 4.6. □

5.2 File deduplication with the cloud

Deduplication, or Single-Instance-Storage (SIS), is a central problem for storage systems (see, e.g., [8, 19, 35]). 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 [14]). 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 $\widetilde{\cup}$ 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 $S_i \,\widetilde{\cup}\, S_j$. Clearly $\widetilde{\cup}$ 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 $s = m \cdot (H + \log n)$, where H is the number of bits in a hash value. Applying cComb with operation $\widetilde{\cup}$ 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 $O(Z_{\max} \log n)$ 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 questions are left open by this paper. For example, can we get a 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?

Author Contributions All authors contributed equally.

Funding Open access funding provided by Tel Aviv University.

Declarations

Conflict of interest We are not aware of any conflicts of interest regarding the research presented in this paper.

References

1. Adler, M., Gibbons, P.B., Matias, Y., Ramachandran, V.: Modeling parallel bandwidth: local versus global restrictions. Algorithmica **24**, 381–404 (1999)
2. Afek, Y., Landau, G.M., Schieber, B., Yung, M.: The power of multimedia: combining point-to point and multi-access networks. In: Dolev, D. (ed.) Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing, 1988, pp. 90–104. ACM (1988)
3. Aguilera, M.K., Ben-David, N., Calciu, I., Guerraoui, R., Petrank, E., Toueg, S.: Passing messages while sharing memory. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC '18, pp. 51–60. Association for Computing Machinery, New York (2018)
4. Anagnostides, I., Gouleakis, T.: Deterministic distributed algorithms and lower bounds in the hybrid model. In: Gilbert, S. (ed.) 35th International Symposium on Distributed Computing, DISC 2021, volume 209 of LIPIcs, pp. 5:1–5:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
5. Attiya, H., Welch, J.: Distributed Algorithms. McGraw-Hill Publishing Company, Berkshire (1998)

6. Augustine, J., Hinnenthal, K., Kuhn, F., Scheideler, C., Schneider, P.: Shortest paths in a hybrid network model. In: Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, pp. 1280–1299 (2020)
7. Baumann, N., Skutella, M.: Solving evacuation problems efficiently–earliest arrival flows with multiple sources. In: 2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06), pp. 399–410 (2006)
8. Bolosky, B., Corbin, S., Goebel, D., Douceur, J.: Single instance storage in windows 2000. In: 4th USENIX Windows Systems Symposium (4th USENIX Windows Systems Symposium), Seattle, WA. USENIX Association (2000)
9. Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., Brendan McMahan, H., Patel, S., Ramage, D., Segal, A., Seth, K.: Practical secure aggregation for privacy-preserving machine learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, pp. 1175–1191, New York, NY, USA. Association for Computing Machinery (2017)
10. Burkard, R.E., Dlaska, K., Klinz, B.: The quickest flow problem. ZOR Methods Models Oper. Res. **37**, 31–58 (1993)
11. Censor-Hillel, K., Leitersdorf, D., Polosukhin, V.: Distance computations in the hybrid network model via oracle simulations. Technical Report arXiv:2010.13831 [cs.DC] (2020)
12. Cheng, Y., Liu, Y., Chen, T., Yang, Q.: Federated learning for privacy-preserving AI. Commun. ACM **63**(12), 33–36 (2020)
13. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: towards a realistic model of parallel computation. SIGPLAN Not. **28**(7), 1–12 (1993)
14. 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 22nd International Conference on Distributed Computing Systems, pp. 617–624 (2002)
15. Dropbox. Prospectus, 2018. Filing to US Securities and Exchange Commission. https://www.sec.gov/Archives/edgar/data/1467623/000119312518055809/d451946ds1.htm
16. Fleischer, L., Skutella, M.: Quickest flows over time. SIAM J. Comput. **36**(6), 1600–1630 (2007)
17. Fortune, S., Wyllie, J.: Parallelism in random access machines. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, New York, NY, USA. Association for Computing Machinery (1978)
18. Fraigniaud, P.: Distributed computational complexities: are you Volvo-addicted or NASCAR-obsessed? In: Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC '10, pp. 171–172, New York, NY, USA. Association for Computing Machinery (2010)
19. Freeman, L.: Looking beyond the hype: evaluating data deduplication solutions. http://www-download.netapp.com/edm/TT/docs/Looking_beyond_hype_Dedupe.pdf, September. Network Appliance, Inc (2007)
20. Gibbons, P.B., Matias, Y., Ramachandran, V.: Can shared-memory model serve as a bridging model for parallel computation? In: Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '97, pp. 72–83, New York, NY, USA. Association for Computing Machinery (1997)
21. Giladi, G.: Distributed computing with the cloud. Master's thesis, Tel Aviv University (2021)
22. Hegeman, J.W., Pemmaraju, S.V.: Lessons from the congested clique applied to MapReduce. Theor. Comput. Sci. **608**(P3), 268–281 (2015)
23. Hoppe, B., Tardos, É.: Polynomial time algorithms for some evacuation problems. In: SODA '94 (1994)
24. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '10, pp. 938–948, USA. Society for Industrial and Applied Mathematics (2010)
25. Kuhn, F., Schneider, P.: Computing shortest paths and diameter in the hybrid network model. In: Emek, Y., Cachin, C. (eds.) PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3–7 2020, pp. 109–118. ACM (2020)
26. Kuhn, F., Schneider, P.: Routing schemes and distance oracles in the hybrid model. In: Proceedings of the 36th international symposium on Distributed Computing (DISC). To appear (2022)
27. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. J. ACM **27**(4), 831–838 (1980)
28. Lardinois, F.: Google drive will hit a billion users this week. TechCrunch (2018). https://techcrunch.com/2018/07/25/google-drive-will-hit-a-billion-users-this-week
29. Lee, C.C., Lee, D.T.: On a circle-cover minimization problem. Inf. Process. Lett. **18**(2), 109–115 (1984)
30. Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and Its Applications, 4th edn. Springer, Cham (2019)
31. Linial, N.: Locality in distributed graph algorithms. SIAM J. Comput. **21**, 193–201 (1992)
32. Lotker, Z., Patt-Shamir, B., Pavlov, E., Peleg, D.: Minimum-weight spanning tree construction in $O(\log \log(n))$ communication rounds. SIAM J. Comput. **35**(1), 120–131 (2005)
33. Mansour, Y., Nisan, N., Vishkin, U.: Trade-offs between communication throughput and parallel time. J. Complex. **15**(1), 148–166 (1999)
34. McMahan, B., Moore, E., Ramage, D., Hampson, S., Arcas,B.A.: Communication-efficient learning of deep networks from decentralized data. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, (AISTATS), pp. 1273–1282 (2017)
35. Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. ACM Trans. Stor. **7**(4), 1–20 (2012)
36. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. Society for Industrial and Applied Mathematics, Philadelphia (2000)
37. Skutella, M.: An introduction to network flows over time. In: Cook, W.J., Lovász, L., Vygen, J. (eds.) Research Trends in Combinatorial Optimization. Springer, Berlin (2009)
38. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)