The Distributed Computing Column

BY

Stefan Schmid

Faculty of Computer Science, University of Vienna
Währinger Strasse 29, AT - 1090 Vienna
stefan_schmid@univie.ac.at

This distributed computing column features two articles:

1. Gopal Pandurangan, Peter Robinson, and Michele Scquizzato present a survey of the Distributed Minimum Spanning Tree (MST) problem, also covering lower bounds and recent results on the joint optimization of time and message complexity.

2. In the second column, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal propose, define, and analyze a generalization of causal consistency.

I would like to thank the authors for their contributions.
Enjoy!

The Distributed Minimum Spanning Tree Problem

Gopal Pandurangan[*] Peter Robinson[†] Michele Scquizzato[‡]

Abstract

This article surveys the distributed minimum spanning tree (MST) problem, a central and one of the most studied problems in distributed computing. In this problem, we are given a network, represented as a weighted graph $G = (V, E)$, and the nodes in the network communicate by message passing via the edges of G with the goal of constructing an MST of G in a distributed fashion, i.e., each node should identify the MST edges incident to itself. This article summarizes the long line of research in designing efficient distributed algorithms and showing lower bounds for the distributed MST problem, including the most recent developments which have focused on algorithms that are simultaneously round- and message-optimal.

[*]Department of Computer Science, University of Houston, Houston TX, USA. E-mail: gopalpandurangan@gmail.com. Supported, in part, by NSF grants CCF-1527867, CCF-1540512, IIS-1633720, and CCF-BSF-1717075, and by US-Israel Binational Science Foundation (BSF) grant 2016419.

[†]Department of Computing and Software, McMaster University, Hamilton, Canada. E-mail: peter.robinson@mcmaster.ca. Peter Robinson acknowledges the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

[‡]School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden. E-mail: mscq@kth.se. Supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 715672.

1 Introduction

The minimum-weight spanning tree (MST) problem is a classical and fundamental problem in graph theory, with a long line of research dating back to Borůvka's algorithm in 1926. The MST is an important and commonly occurring primitive in the design and operation of communication networks. Formally, the MST problem is as follows. Given an n-node connected (undirected) graph with edge weights, the goal is to compute a *spanning* tree of *minimum total weight*. Weights can be used to model fundamental network performance parameters such as transmission delays, communication costs, etc., and hence an MST represents a spanning tree that optimizes these parameters. One of the most common applications of MST is that it can serve as a backbone for efficient communication, e.g., it can be used naturally for *broadcasting*, i.e., sending a message from a (source) node to all the nodes in the network. Any node that wishes to broadcast simply sends messages along the spanning tree. The message complexity is then $O(n)$, which is optimal. The advantage of this method over flooding (which uses all the edges in the networks) is that redundant messages are avoided, as only the edges in the spanning tree are used. In particular, if weights model the cost or the delay for a message to pass through an edge of the network, an MST can minimize the total cost for a node to communicate with all the other nodes of the network.

Because of its fundamental importance in networks, the MST problem is also one of the central and most studied problems in network algorithms, specifically distributed network algorithms. In the distributed MST problem, the goal is to compute a MST in a distributed fashion efficiently, which is achieved by minimizing two fundamental performance measures used for distributed algorithms: message and time complexity. In a distributed network, each node (which represents a processor) is aware only of its incident edges (which represent communication links to its neighbors) and of their weights. That is, each node starts with only local knowledge, and at the end of the distributed algorithm each node should know which of its incident edges belong to the MST, i.e., the output knowledge is also local (e.g., a node need not know about the status of edges that are not incident to it). We will give formal details of the model assumptions in Section 2.

Besides an efficient communication backbone, a distributed MST algorithm can be also be used for leader election, a fundamental *symmetry breaking* problem in distributed computing. In the leader election problem, among many nodes (potentially all) nodes vying to be leader, the goal is to elect one unique leader. A distributed MST algorithm can construct a rooted tree (where parent-child relationships are known to each node), and the root can serve as the leader. The MST algorithms described in this article naturally construct a rooted tree.

In this article, we survey algorithms and lower bounds for the distributed MST problem. The rest of the article is organized as follows. In Section 2, we formally

describe the distributed computing model and its complexity measures and discuss some preliminaries on the distributed MST problem. In section 3, we give a quick historical survey and summarize the state of the art until 2016. In Section 4, we discuss three well-studied distributed MST algorithms. In Section 5, we present *singularly optimal* distributed MST algorithms that were discovered more recently, i.e., distributed MST algorithms that are simultaneously time- and message-optimal. In Section 6 we discuss lower bounds and describe the techniques used to show such results. We conclude with some open questions in Section 7.

2 Model, Definitions, and Preliminaries

We first describe the distributed computing model that we consider in this article. The model is called the CONGEST model (see, e.g., [47, 43]), which is now standard in the distributed computing literature.

A point-to-point communication network is modeled as an undirected weighted graph $G = (V, E, w)$, where the vertices of V represent the processors, the edges of E represent the communication links between them, and $w(e)$ is the weight of edge $e \in E$. Without loss of generality, we assume that G is connected. We use D to denote the hop-diameter (that is, the unweighted diameter) of G, and, in this article, by *diameter* we always mean hop-diameter. We also assume that the weights of the edges of the graph are all *distinct*. This implies that the MST of the graph is *unique*. [1] Each node hosts a processor with limited initial knowledge. Specifically, we make the common assumption that each node has unique identity numbers, and at the beginning of computation each vertex v accepts as input its own identity number and the weights of the edges incident to it. Thus, a node has only *local* knowledge. Specifically, we assume that each node u has ports (each port having a unique integer port number) and each incident edge of u is connected to one of u's distinct port. A node does not have any initial knowledge of the other endpoint of its incident edge (which node it is connected to or the port number that this node is connected to). This model is referred to as the *clean network model* in [47, 43] and is also sometimes called the KT_0 model, i.e., the initial (K)nowledge of all nodes is restricted (T)ill radius 0, c.f. [43, 47]. The KT_0 model is a standard model in distributed computing and used by many prior results on distributed MST (e.g., [3, 6, 15, 16, 36, 14, 10]), with a notable exception ([29], discussed in Section 3).

The vertices are allowed to communicate by sending messages to their neighbors through the edges of the graph G. Here we assume that communication is

[1] Even if the weights are not distinct, it is easier to make them distinct by using the unique node identifiers of the respective endpoints of the edges to break ties.

synchronous and occurs in discrete *rounds* (time steps). In each round, each node v can send an arbitrary message of $O(\log n)$ bits through each edge $e = (v, u)$ incident to v, and each message arrives at u by the end of this round. [2] The weights of the edges are at most polynomial in the number of vertices n, and therefore the weight of a single edge can be communicated in one time step. This model of distributed computation is called the CONGEST($\log n$) model or simply the CONGEST model [47, 43]. Some of the MST algorithms discussed are randomized and hence also assume that each vertex has access to the outcomes of an unbiased private coin flips.

The efficiency of distributed algorithms is traditionally measured by their time and message (or, communication) complexities. *Time complexity* measures the number of synchronous rounds taken by the algorithm, whereas *message complexity* measures the total number of messages sent and received by all the processors during the execution of the algorithm. Both complexity measures crucially influence the performance of a distributed algorithm. As defined in [45], we say that a problem enjoys *singular optimality* if it admits a distributed algorithm whose time and message complexity are both optimal.

We make an assumption that simplifies our algorithms and analysis: we assume that all edge weights in the graph are distinct. It is easy to show that this implies that the MST is unique. This assumption is without loss of generality, because one can tag each edge weight (additionally) with the node IDs of the endpoints of the edge (which are unique as a pair).[3] This tagging can be used to break ties between edges having the same weight.

As in centralized MST algorithms, distributed MST algorithms also rely on two important properties of an MST: (1) the *cut property* and (2) the *cycle property* [49]:

1. **Cut property:** A cut in a graph is a partition of the vertex set into two disjoint sets. The cut property states that, given any cut in a graph, the *lightest*, i.e. minimum weight, edge crossing the cut belongs to the MST. (Recall that due to the assumption of unique edge weights, there is a unique lightest edge crossing the cut.)
2. **Cycle property:** Consider any cycle in the graph. The heaviest (i.e. maximum weight) edge in the cycle will not be in the MST.

[2] If unbounded-size messages are allowed—this is the so-called LOCAL model—the MST problem can be trivially solved in $O(D)$ time by collecting all the topology information in one node [43].

[3] Note that this tagging involves nodes knowing the IDs of their neighbors. This tagging can be done during the course of an MST algorithm.

3 Summary of Prior Research

Given the importance of the distributed MST problem, there has been significant work over the last 30 years on this problem and related aspects. The distributed MST problem has triggered a lot of research in distributed computing, leading to new techniques for the design and analysis of distributed algorithms as well as for showing lower bounds. We first briefly summarize research until 2016 and then discuss very recent results on singularly optimal algorithms.

3.1 Research Until 2016

Early Work. Distributed MST was one of the very first distributed computing problems that was studied. A long line of research aimed at developing efficient distributed algorithms for the MST problem started more than thirty years ago with the seminal paper of Gallager, Humblet, and Spira [15], which presented a distributed algorithm that constructs an MST in $O(n \log n)$ rounds exchanging a total of $O(m + n \log n)$ messages, where n and m denote the number of nodes and the number of edges of the network, respectively.[4] The message complexity of this algorithm is (essentially) optimal [35], but its time complexity is not. Hence further research concentrated on improving the time complexity, resulting in a first improvement to $O(n \log \log n)$ by Chin and Ting [6], a further improvement to $O(n \log^* n)$ by Gafni [14], and then to $O(n)$ by Awerbuch [3] (see also [13]). The $O(n)$ bound is existentially optimal in the sense that there exist graphs with diameter $\Theta(n)$ for which this is the best possible, even for randomized Monte-Carlo algorithms (see [35]).

This was the state of the art until the mid-nineties when Garay, Kutten, and Peleg [16] raised the question of whether it is possible to identify graph parameters that can better capture the complexity of distributed network computations. In fact, for many existing networks, the hop-diameter D is significantly smaller than the number of vertices n, and therefore it is desirable to obtain protocols whose running time is bounded in terms of D rather than in terms of n. (We note that $\Omega(D)$ is a lower bound on the running time of any distributed MST algorithm in a "universal" sense [35, 34]). Garay, Kutten, and Peleg [16] gave the first such distributed algorithm for the MST problem with running time $O(D + n^{0.614} \log^* n)$, which was later improved by Kutten and Peleg [36] to $O(D + \sqrt{n} \log^* n)$.[5] However, both these algorithms are not message-optimal,[6]

[4] The original algorithm has a message complexity of $O(m \log n)$, but it can be improved to $O(m + n \log n)$.

[5] The $\log^* n$ factor can, in all respective algorithms, be reduced to $\sqrt{\log^* n}$, by growing components to a size larger by a factor $\sqrt{\log^* n}$ in the respective first phase.

[6] In this paper, "optimal" means "optimal up to a polylog(n) factor."

as they exchange $O(m + n^{1.614})$ and $O(m + n^{1.5})$ messages, respectively.

The lack of progress in improving the result of [36], and in particular breaking the $\tilde{O}(\sqrt{n})$ barrier,[7] led to work on lower bounds for the distributed MST problems, starting out by the result of Peleg and Rubinovich [48] and the more recent extensions of [11, 7], which we discuss in more detail in Section 6.

State of the Art (until 2016). We now summarize the state of the art for distributed MST algorithms until 2016: There exist algorithms which are either time-optimal (i.e., they run in $\tilde{O}(D + \sqrt{n})$ time) or message-optimal (i.e., they exchange $\tilde{O}(m)$ messages), *but not simultaneously both*. Indeed, the time-optimal algorithms of [36, 10] (as well as the sublinear time algorithm of [16]) are not message-optimal, i.e., they require asymptotically much more than $\Theta(m)$ messages. In contrast, the known message-optimal algorithms for MST (in particular, [15, 3]) are not time-optimal, i.e., they take significantly more time than $\tilde{O}(D + \sqrt{n})$. In their 2000 SICOMP paper [48], Peleg and Rubinovich raised the question of whether one can design a distributed MST algorithm that is *simultaneously* optimal with respect to time and message complexity. In 2011, Kor, Korman, and Peleg [32] also raised this question and showed that distributed *verification* of MST, i.e., verifying whether a given spanning tree is MST or not, can be done in optimal messages and time, i.e., there exists a distributed verification algorithm that uses $\tilde{O}(m)$ messages and runs in $\tilde{O}(D+\sqrt{n})$ time. However, achieving these bounds for MST construction remained open until 2017.

Singular Optimality. As defined in [45], the main question that remained open for distributed MST is whether there is a distributed algorithm that is *singularly optimal* or if the problem exhibits a *time-message trade-off*:

- **Singularly optimal:** A distributed algorithm that is optimal with respect to both measures simultaneously. In this case we say that the problem enjoys *singular optimality*.
- **Time-message trade-off:** When the problem inherently fails to admit a singularly optimal solution, namely, algorithms with better time complexity necessarily incur higher message complexity, and vice versa. In this case we say that the problem exhibits a *time-message trade-off*.

The above question addresses a fundamental aspect in distributed algorithms, namely the relationship between the two basic complexity measures of time and messages. The simultaneous optimization of both time and message complexity has been elusive for several fundamental problems (including MST, shortest

[7] $\tilde{O}(f(n))$ and $\tilde{\Omega}(f(n))$ denote $O(f(n) \cdot \text{polylog}(f(n)))$ and $\Omega(f(n)/\text{ polylog}(f(n)))$, respectively.

paths, and random walks), and consequently research in the last three decades in distributed algorithms has focused mainly on optimizing either one of the two measures separately. However, in various modern and emerging applications such as resource-constrained communication networks and distributed computation of large-scale data, it is crucial to design distributed algorithms that optimize both measures *simultaneously* [30, 24].

Besides the prior work already mentioned, we now discuss other relevant work on distributed MST.

Other Distributed MST Algorithms. Elkin [10] showed that a parameter called *MST-radius* captures the complexity of distributed MST algorithms better. The MST-radius, denoted by $\mu(G, w)$, is a function of the graph topology as well as the edge weights and, roughly speaking, is the maximum radius each vertex has to examine to check whether any of its edges is in the MST. Elkin devised a distributed protocol that constructs the MST in $\tilde{O}(\mu(G, w) + \sqrt{n})$ time. The ratio between diameter and MST-radius can be as large as $\Theta(n)$, and consequently, on some inputs, this protocol is faster than the protocol of [36] by a factor of $\Omega(\sqrt{n})$. However, a drawback of this protocol (unlike the previous MST protocols [36, 16, 6, 14, 15]) is that it cannot detect the termination of the algorithm in that time (unless $\mu(G, w)$ is given as part of the input). On the other hand, it can be shown that for distributed MST algorithms that correctly terminate, $\Omega(D)$ is a lower bound on the running time [48, 34]. (In fact, [34] shows that for every sufficiently large n and every function $D(n)$ with $2 \leq D(n) < n/4$, there exists a graph G of $n' \in \Theta(n)$ nodes and diameter $D' \in \Theta(D(n))$ which requires $\Omega(D')$ rounds to compute a spanning tree with constant probability in the clean network model.) We also note that the message complexity of Elkin's algorithm is $O(m + n^{1.5})$.

Some specific classes of graphs admit efficient MST algorithms that beat the general $\tilde{\Omega}(D + \sqrt{n})$ time lower bound. Perhaps the first result in this direction is due to Khan and Pandurangan [28] who showed that one can obtain a distributed $O(\log n)$-approximate algorithm for MST for special classes of graphs such as unit disk graphs (that are used to model wireless networks) and randomly weighted graphs that runs in $\tilde{O}(D)$ time and $\tilde{O}(m)$ messages. More recently, faster distributed MST algorithms have been presented for planar graphs, graphs of bounded genus, treewidth, or pathwidth [17, 22, 23], and graphs with small mixing time [18]. Faster distributed MST algorithms have also been developed for networks with small constant diameter. An $O(\log n)$ time algorithm for networks with $D = 2$ was given in [38], and an $O(\log \log n)$ time algorithm for networks with $D = 1$ (i.e., complete networks) was given in [37]. In fact, the complete network case, and more in general, MST construction in the so-called *Congested Clique* model, has received significant attention in recent years. (The Congested

Clique is a simple model for overlay networks whereby, unlike the CONGEST model, even non-adjacent nodes can communicate directly.) The deterministic algorithm of Lotker et al. [37] (which improved significantly over an easy-to-construct $O(\log n)$-round algorithm) was later improved by Hegeman et al. [24], who presented an $O(\log \log \log n)$-round randomized Monte Carlo algorithm. The key technique for this result is linear graph sketching [1, 2, 40], which has since been applied in other distributed algorithms (see, e.g., [44]). This paper also gives lower bounds on the message complexity of MST construction in the Congested Clique: $\Omega(n^2)$ in the KT_0 model and $\Omega(n)$ in the KT_1 model respectively. It also gives a randomized algorithm that uses $\tilde{O}(n)$ messages and runs in $\tilde{O}(1)$ rounds in the KT_1 model. Building on the work of Hegeman et al., the time bound for MST construction in the Congested Clique has been improved to $O(\log^* n)$ rounds [19], and then, very recently, to $O(1)$ rounds [27]. All these algorithms are randomized and crucially rely on using graph sketches.

However, there exists a significant gap in the time complexity of the distributed MST algorithms between the cases of network diameters 2 and 3. In [38], it was shown that the time complexity of any distributed MST algorithm is $\Omega(\sqrt[4]{n}/\sqrt{B})$ for networks of diameter 3 and $\Omega(\sqrt[3]{n}/\sqrt{B})$ for networks of diameter 4. These asymptotic lower bounds hold for randomized algorithms as well. (On the other hand, $O(\log n)$ time suffices to compute an MST deterministically for graphs with diameter 2.)

Time Complexity. From a practical perspective, given that MST construction can take as much as $\Omega(\sqrt{n}/\log n)$ time even in low-diameter networks, it is worth investigating whether one can design distributed algorithms that run faster and output an approximate minimum spanning tree. The question of devising faster approximation algorithms for MST was raised in [48]. Elkin [11] later established a hardness result on distributed MST approximation, showing that *approximating* the MST problem on a certain family of graphs of small diameter (e.g., $O(\log n)$) within a ratio H requires essentially $\Omega(\sqrt{n/H \log n})$ time. Khan and Pandurangan [28] showed that there can be an exponential time gap between exact and approximate MST construction by showing that there exist graphs where any distributed (exact) MST algorithm takes $\Omega(\sqrt{n}/\log n)$ rounds, whereas an $O(\log n)$-approximate MST can be computed in $O(\log n)$ rounds. The distributed approximation algorithm of [28] is message-optimal but not time-optimal.

Das Sarma et al. [7] settled the time complexity of distributed approximate MST by showing that this problem, as well as approximating shortest paths and about twenty other problems, satisfies a time lower bound of $\tilde{\Omega}(D + \sqrt{n})$. This applies to deterministic as well as randomized algorithms, and to both exact and approximate versions. In other words, any distributed algorithm for computing a

H-approximation to MST, for any $H > 0$, takes $\tilde{\Omega}(D + \sqrt{n})$ time in the worst case.

Message Complexity. Kutten et al. [35] fully settled the message complexity of leader election in general graphs assuming the clean network (i.e. KT_0 model), even for randomized algorithms and under very general settings. Specifically, they showed that any randomized algorithm (including Monte Carlo algorithms with suitably large constant success probability) requires $\Omega(m)$ messages; this lower bound holds in a universal sense: Given any n and m, there exists a graph with $\Theta(n)$ nodes and $\Theta(m)$ edges for which the lower bound applies. Since a distributed MST algorithm can also be used to elect a leader (where the root of the tree is the leader, which can be chosen using $O(n)$ messages once a tree is constructed), the above lower bound applies to distributed MST constructions as well, for all $m \geq cn$, where c is a sufficiently large constant.

The above lower bound bound holds even if nodes have initial knowledge of n, m, and D. It also holds for synchronous networks, where all the nodes wake up simultaneously. Finally, it holds not only for the CONGEST model [47], where sending a message of $O(\log n)$ bits takes one unit of time, but also for the LOCAL model [47], where the number of bits carried in a single message can be arbitrary. On the other hand, it is known from [15, 3] that an MST can be constructed using $\tilde{O}(m)$ messages in synchronous networks.

The KT_1 Variant. It is important to point out that all the results discussed in this article (including the MST results [3, 6, 15, 16, 36, 14, 10]) assume the so-called *clean network model*, a.k.a. KT_0 [47] (cf. Section 2), where nodes do not have initial knowledge of the identity of their neighbors. However, one can assume a model where nodes do have such a knowledge. This model is called the KT_1 *model*. Although the distinction between KT_0 and KT_1 has clearly no bearing on the asymptotic bounds for the time complexity, it is significant when considering message complexity. Awerbuch et al. [4] show that $\Omega(m)$ is a message lower bound for MST in the KT_1 model, if one allows only (possibly randomized Monte Carlo) comparison-based algorithms, i.e., algorithms that can operate on IDs only by comparing them.

Awerbuch et al. [4] also show that the $\Omega(m)$ message lower bound applies even to non-comparison based (in particular, algorithms that can perform arbitrary local computations) *deterministic* algorithms in the CONGEST model that terminate in a time bound that depends only on the graph topology (e.g., a function of n). On the other hand, for *randomized non-comparison-based* algorithms, it turns out that the message lower bound of $\Omega(m)$ does not apply in the KT_1 model. In 2015, King et al. [29] showed a surprising and elegant result: in the KT_1 model one can give a randomized Monte Carlo algorithm to construct an MST in $\tilde{O}(n)$ mes-

sages ($\Omega(n)$ is a trivial message lower bound for the KT_1 model) and in $\tilde{O}(n)$ time. This algorithm is randomized and not comparison-based. While this algorithm shows that one can achieve $o(m)$ message complexity (when $m = \omega(n \operatorname{polylog} n)$), it is *not* time-optimal (it can take significantly more than $\tilde{\Theta}(D + \sqrt{n})$ rounds). In subsequent work, Mashreghi and King [39] presented another randomized, not comparison-based MST algorithm with round complexity $\tilde{O}(\text{Diam(MST)})$ and with message complexity $\tilde{O}(n)$. It is an open question whether one can design a randomized (non-comparison based) algorithm that takes $\tilde{O}(D + \sqrt{n})$ time and $\tilde{O}(n)$ messages in the KT_1 model. Very recently, Gmyr and Pandurangan [20] presented improved algorithms in the KT_1 model for MST and several other problems. For the MST problem, they showed that it can be solved in $\tilde{O}(D + n^{1-\delta})$ rounds using $\tilde{O}(\min\{m, n^{1+\delta}\})$ messages for any $\delta \in [0, 0.5]$. In particular, for $\delta = 0.5$ they obtain a distributed MST algorithm that runs in optimal $\tilde{O}(D + \sqrt{n})$ rounds and uses $\tilde{O}(\min\{m, n^{3/2}\})$ messages. Notice that this improves over the singularly optimal algorithms ([45, 12, 21]) for the KT_0 model.

3.2 Recent Results

In 2017, Pandurangan et al. [45] presented the first distributed MST algorithm for the CONGEST model which is simultaneously time- and message-optimal. The algorithm is randomized Las Vegas, and always returns the MST. The running time of the algorithm is $\tilde{O}(D + \sqrt{n})$ and the message complexity is $\tilde{O}(m)$, and both bounds hold with high probability.[8] This is the first distributed MST algorithm that matches *simultaneously* the time lower bound of $\tilde{\Omega}(D + \sqrt{n})$ [11, 7] and the message lower bound of $\Omega(m)$ [35], which both apply even to randomized Monte Carlo algorithms, thus closing a more than thirty-year-old line of research in distributed computing. In terms of the terminology introduced earlier, we can therefore say that the distributed MST problem exhibits singular optimality up to polylogarithmic factors.

The work of Pandurangan et al. [45] raised the open problem of whether there exists a *deterministic* time- and message-optimal MST algorithm. We notice that the algorithm of Pandurangan et al. is *randomized*, due to the use of the randomized cover construction of [10], even though the rest of the algorithm is deterministic. Elkin [12], building on the work of [45], answered this question affirmatively by devising a deterministic MST algorithm that achieves essentially the same bounds, i.e., it uses $\tilde{O}(m)$ messages and runs in $\tilde{O}(D + \sqrt{n})$ time. Another deterministic round- and message-optimal algorithm for MST appeared very recently in [21]. Table 1 summarizes the known upper bounds on the complexity of distributed MST.

[8] Throughout, with high probability (w.h.p.) means with probability $\geq 1 - 1/n^{\Omega(1)}$.

Reference	Time	Messages	Computation
Gallager et al. [15]	$O(n \log n)$	$O(m + n \log n)$	Deterministic
Awerbuch [3]	$O(n)$	$O(m + n \log n)$	Deterministic
Garay et al. [16]	$O(D + n^{0.614} \log^* n)$	$O(m + n^{1.614})$	Deterministic
Kutten and Peleg [36]	$O(D + \sqrt{n} \log^* n)$	$O(m + n^{1.5})$	Deterministic
Elkin [10]	$\tilde{O}(\mu(G, w) + \sqrt{n})$	$O(m + n^{1.5})$	Randomized
Pandurangan et al. [45]	$\tilde{O}(D + \sqrt{n})$	$\tilde{O}(m)$	Randomized
Elkin [12]	$\tilde{O}(D + \sqrt{n})$	$\tilde{O}(m)$	Deterministic
Haeupler et al. [21]	$\tilde{O}(D + \sqrt{n})$	$\tilde{O}(m)$	Deterministic

Table 1: Summary of upper bounds on the complexity of distributed MST. Notation $\tilde{O}(\cdot)$ hides polyogarithmic factors in n.

4 Overview of Classical MST Algorithms

The recent singularly optimal distributed MST algorithm of [45, 12] that we discuss in Section 5 build on prior distributed MST algorithms that were either message-optimal or time-optimal but not both. We now provide a brief overview of three well-known previous MST algorithms: (1) The Gallager-Humblet-Spira (GHS) algorithm; (2) The Pipeline algorithm; (3) Garay-Kutten-Peleg (GKP) algorithm. This overview is based on [43], to which we refer to for a more detailed description of these algorithms.

4.1 The Gallager-Humblet-Spira (GHS) algorithm

The first distributed algorithm for the MST problem was given by Gallager, Humblet, and Spira in 1983 [15].

We are given an undirected, connected, weighted graph $G = (V, E, w)$. Let n be the number of nodes and m be the number of edges of G. Let T be the (unique) MST on G. An *MST fragment* (or simply a *fragment*) F of T is defined as a connected subgraph of T, that is, F is a subtree of T. An *outgoing edge* of an MST fragment is an edge in E where one adjacent node to the edge is in the fragment and the other is not. The *minimum-weight outgoing edge (MOE)* of a fragment F is the edge with *minimum weight* among all outgoing edges of F. As an immediate consequence of the cut property of MSTs, the MOE of a fragment $F = (V_F, E_F)$ is an edge of the MST.

The synchronous GHS algorithm is essentially a distributed implementation of the classical Borůvka's MST algorithm [5]. The GHS algorithm operates in *phases*. In the first phase, it starts with each individual node as a fragment by itself and continues until there is only one fragment left. That is, at the beginning,

there are $|V|$ fragments, and at the end of the last phase, a single fragment which is exactly the sought MST. All fragments find their MOE simultaneously in parallel.

In each phase, the algorithm maintains the following invariant: Each MST fragment has a leader and all nodes know their respective parents and children. The root of the tree will be the leader. Initially, each node (a singleton fragment) is a root node; subsequently each fragment will have one root (leader) node. Each fragment is identified by the identifier of its root, called the fragment ID, and each node in the fragment knows its fragment ID.

4.1.1 One phase of GHS

We describe one phase of the GHS algorithm. Each fragment's operation is coordinated by the respective fragment's root (leader). Each phase consists of two major operations: (1) Find MOE of all fragments and (2) Merging fragments via their MOEs.

Find MOE of all fragments In a phase, all fragments find their MOE *simultaneously in parallel*. To find the MOE of a fragment, the root in the fragment broadcasts a message ("find MOE") to all nodes in the fragment using the edges in the fragment. Once a node receives "find MOE" message, it finds its minimum outgoing *incident* edge (i.e, the minimum weight outgoing edge among all the incident edges). To find the minimum weight outgoing incident edge, a node checks its neighbors in *increasing order of weight*. If the fragment ID of its neighbor is different from its own, then the edge is an outgoing edge. Note that since edges are checked in increasing order of weight, the first neighbor whose fragment ID is different from its own is the minimum outgoing incident edge. Also, note that the checking can be done (in increasing weight order) starting from the neighbor that was checked *last* in the previous phase. This is because, all edges that were checked earlier would belong to *the same fragment* and will continue to be in the same fragment until the end of the algorithm. Then, each node sends its minimum outgoing incident edge to the root by *convergecasting* the minimum; the root then finds the MOE, which is the minimum among all the edges convergecast. Note that the convergecast process uses the fragment (tree) edges only.

Merging fragments via their MOEs Next, fragments are merged via their MOEs. Recall that MOEs belong to the MST (by the cut property).

Once the leader finds the MOE, it broadcasts a "Merge" message to all its fragment nodes (the broadcast is sent along the tree edges); the message contains the MOE edge of the fragment. Hence upon receiving the MOE edge, a node knows whether it is the same as its minimum outgoing incident edge or not. If a

node is incident to the fragment's MOE edge, then the node attempts to combine with its neighbor (which belongs to a different fragment). It sends a "Request to combine" message to its neighbor. If the neighbor has also selected the *same* edge as its MOE then the two neighboring nodes agree to combine through this edge; i.e., both neighboring nodes receive "Request to combine" message from each other. (Otherwise, if only of them receives this message, it ignores it. However, the MOE edge is not ignored: the neighbor marks the edge over which it receives the message as the MOE edge of its neighboring fragment). The node with the higher identifier becomes the root of the combined fragment. The (combined) root broadcasts a "new-fragment" message through the fragment edges and the MOE edges chosen by all the fragments. Each node updates its parent, children, and fragment identifier (which will be the ID of the new root).

To see that the above process correctly combines the fragments, we ascribe a direction to all the MOEs (towards the outgoing way). This creates a "directed tree" of fragments (think of fragments as "super-nodes"). Note that since each fragment has only one outgoing edge, there can *at most one pair* of neighboring nodes (in this directed tree). One of these nodes in the pair will be the root of the combined fragment as described above.

Analysis of GHS algorithm It is easy to argue that the total number of phases is $O(\log n)$. This is because, in each phase, the total number of fragments is reduced by at least half: in the worst case, each MOE will be the MOE of both neighboring fragments and they combine into one.

We next argue that each phase takes $O(n)$ time. Hence, overall time complexity is $O(n \log n)$. This is because in each phase, both the major operations take $O(n)$ time (these include broadcast and convergecast), since they happen along the MST edges and the diameter of the MST can be as large as $\Theta(n)$.

We next argue that each phase takes $O(n)$ messages (for convergecast and broadcast) plus the messages needed to find the MOE. The latter takes a total of $O(m + n \log n)$ messages because in each phase a node checks its neighbor in increasing order of weight starting from the last checked node. Thus, except for the last checked node (which takes one message per phase) all other neighbors are checked at most once. Hence, the total message complexity is

$$\sum_{v \in V} 2d(v) + \sum_{i=1}^{\log n} \sum_{v \in V} 1 = O(m + n \log n).$$

4.2 The Pipeline Algorithm

Next we discuss the Pipeline algorithm due to Peleg [46], which is slightly better than the GHS algorithm, i.e., it runs in $O(n)$ rounds. Note that this is *existentially*

optimal, since we know that the diameter is a lower bound for MST (even for randomized algorithms) [35] and hence there exists graphs of diameter $\Theta(n)$, where any MST algorithm will require $\Omega(n)$ rounds.

The Pipeline algorithm is essentially an *upcast* algorithm, where we build a BFS tree over the graph and each node upcasts edges to the root of the BFS tree; the root ends up having (enough) global knowledge of the network topology and locally computes the MST and downcasts the MST edges to all nodes in the network. Of course, a naive upcast is for each node to send all its incident edges and this upcast can take $\Theta(m)$ rounds, since there are as many edges. The main idea of the Pipeline MST algorithm is to filter the number of edges broadcast so that the running time is reduced to $O(n)$ rounds. However, the message complexity of the Pipeline algorithm can be as much as $\Theta(n^2)$.

The pipeline algorithm uses the cycle property of MST to filter edges at intermediate nodes. Each node v, except the root r, maintains two lists of edges, Q and U. Initially, Q contains only the edges adjacent to v, and U is empty. At each round, v sends the *minimum-weight* edge in Q that does not create a cycle with the edges in U to its parent and moves this edge from Q to U. If Q is empty, v sends a terminate message to its parent. The parent after receiving an edge from a child, adds the edge in its Q list. A leaf node starts sending edges upwards at round 0. An intermediate node starts sending at the first round after it has received at least one message from each of its children.

4.2.1 Analysis

We give the high-level idea behind the correctness and running time.

Correctness The algorithm's correctness follows from the cycle property. Using the cycle property, since only non-MST edges are filtered (note that an edge at node v is not sent upward if it closes a cycle with edges in U, i.e., the already sent edges—since edges are sent in non-increasing order, the filtered edges are the respective heaviest edge in a cycle) it can be shown that the root receives all the MST edges (plus possible additional edges) required to compute the MST correctly.

Running time It is easy to show that the edges reported by each node to its parent in the tree are sent in non-decreasing weight order, and each node sends at most $n - 1$ edges upward to its parent. This is because if more than $n - 1$ edges are sent through a node, then at least one edge will form a cycle with the edges sent previously and will be filtered by the cycle property. To build the BFS tree, it takes $O(D)$ time. Since the depth of the BFS tree is D and each node sends at

most $n-1$ edges upward, the pipeline algorithm takes $O(D+n) = O(n)$ time. The analysis shows that there is not too much delay (more than n) overall before the root receives all the edges that it needs for computing the MST.

The complexity of the Pipeline algorithm is stated in Table 1; for a detailed proof we refer to [43].

4.3 The Garay-Kutten-Peleg (GKP) Algorithm

The GKP algorithm [36] runs in $O(D + \sqrt{n}\log^* n)$ time, where D is the diameter of the graph G, which is essentially optimal (up to logarithmic factors), due to the existence of the time lower bound of $\tilde{\Omega}(D + \sqrt{n})$ [48, 11, 7]. Note that in graphs with diameter smaller than $\sqrt{n}$, this algorithm can be much faster than the Pipeline algorithm.

We give a high-level overview of the GKP algorithm; for full details we refer to [43].

The GKP algorithm consists of two parts: it combines the *GHS algorithm* and the *Pipeline algorithm* in a judicious way.

4.3.1 First part: Controlled-GHS Algorithm

The first part, called the *controlled-GHS algorithm* is similar to the GHS algorithm, with the crucial property of ensuring that the diameter of fragments do not become too big. (Note that in the GHS algorithm, even after the first phase, there can be fragments with diameter as large as n). The controlled-GHS begins with each node as a singleton fragment. In every phase, as in the GHS algorithm, each fragment finds its MOE. However, not all fragments are merged along their MOE edges. Only a subset of MOE edges are selected for merging. A crucial property of the merging is the following: in every phase, the number of fragments is reduced by at least a factor of two, while the diameter is not increased by more than a constant factor. The controlled-GHS algorithm continues for about $\log(\sqrt{n})$ phases. At the end of the first part, the following property is guaranteed: the diameter of each fragment is $O(\sqrt{n})$ and there are at most $\sqrt{n}$ fragments. The first part of the algorithm takes $O(\sqrt{n}\log^* n)$ time.

We note that Controlled-GHS as implemented in the time-optimal algorithm of [36] is not message-optimal—the messages exchanged can be $\tilde{O}(m+n^{1.5})$; however, a modified version can be implemented using $\tilde{O}(m)$ messages [43].

4.3.2 Second part: Pipeline algorithm

The second part of the algorithm uses the Pipeline algorithm to find the remaining (at most) $\sqrt{n}-1$ MST edges (since there are $\leq \sqrt{n}$ fragments left). As in the

Pipeline algorithm, a breadth-first tree B is built on G. Let $r(B)$ be the root of B. Using the edges in B, root $r(B)$ collects weights of the *inter-fragment* edges, computes the minimum spanning tree T' of the fragments by considering each fragment as a super node. It then broadcasts the edges in T' to the other nodes using the breadth-first tree B. Since the depth of B is $O(D)$ and each node sends at most $\sqrt{n}$ edges upward, the Pipeline algorithm takes $O(D + \sqrt{n})$ time; this analysis is very similar to the one of the Pipeline algorithm (by replacing n with $\sqrt{n}$). Thus the overall time complexity of the GKP algorithm is $O(D + \sqrt{n}\log^* n)$. The message complexity can be shown to be $O(m + n^{1.5})$.

The overall algorithms consists of running the controlled-GHS first and then switching to the Pipeline algorithm after $O(\sqrt{n}\log^* n)$ rounds. Combining the two parts, the complexity bounds as stated in Table 1 follow.

5 Toward Singular Optimality: Round- and Message-Optimal Distributed MST Algorithms

As mentioned in Section 1, up until recently, all known distributed MST algorithms achieved either optimal time or optimal message complexity, but not both. In this section, we describe the recent progress toward achieving *singular optimality*, as defined in Section 1. We begin by describing the algorithm of [45] in Section 5.1, on which the subsequent work of Elkin is based [12], which is discussed in Section 5.2.

5.1 A Randomized Singularly-Optimal Algorithm

We now describe the algorithm of [45] that attains *singular optimality*, i.e., $\tilde{O}(m)$ messages and $\tilde{O}(D+\sqrt{n})$ time. The first part of the algorithm consists of executing Controlled-GHS, described in Section 4.3.1. At the end of this procedure, at most $O(\sqrt{n})$ distinct MST fragments remain, each of which has diameter $O(\sqrt{n})$; we call these *base fragments*.

The second part of the algorithm is different from the existing time-optimal MST algorithms. The existing time-optimal MST algorithms [36, 10], as well as the algorithm of [16], are not message-optimal since they use the Pipeline procedure of [46, 16].

The algorithm of [45] uses a different strategy to achieve optimality in both time and messages. The main novelty is how the (at most) $\sqrt{n}$ base fragments which remain at the end of the Controlled-GHS procedure are merge into one resulting fragment (the MST). Unlike previous time-optimal algorithms [36, 10, 16], this algorithm does not use the Pipeline procedure of [46, 16], since it is not

message-optimal. Instead, it continues to merge fragments and uses two main ideas to implement a Borůvka-style merging strategy efficiently. The first idea is a procedure to efficiently merge when D is small (i.e., $D = O(\sqrt{n})$) or when the number of fragments remaining is small (i.e., $O(n/D)$). The second idea is to use *sparse neighborhood covers* and efficient communication between fragments to merge fragments when D is large *and* the number of fragments is large. Accordingly, the second part of the algorithm can be conceptually divided into three phases, which are described next. The description in the remainder of this subsection closely follows Section 2 in [45].

5.1.1 Phase 1: When D is $O(\sqrt{n})$:

At the start of this phase, a BFS tree of the entire network is constructed, and then merging is performed as follows: Each base fragment finds its minimum-weight-outgoing edge (MOE) by convergecasting *within* each of its fragments. This takes $O(\sqrt{n})$ time and $O(\sqrt{n})$ messages per base fragment, leading to $O(n)$ messages overall. The $O(\sqrt{n})$ MOE edges are sent by the leaders of the respective base fragments to the root by *upcasting* (see, e.g., [47]). This takes $O(D + \sqrt{n})$ time and $O(D\sqrt{n})$ messages, as each of the at most $\sqrt{n}$ edges has to be sent along up to D edges to reach the root. The root merges the fragments and sends the renamed fragment IDs to the respective leaders of the base fragments by *downcast* (which has the same time and message complexity as upcast [47]). The leaders of the base fragments broadcast the new ID to all other nodes in their respective fragments. This takes $O(\sqrt{n})$ messages per fragment and hence $O(n)$ messages overall. Thus one iteration of the merging can be done in $O(D + \sqrt{n})$ time and using $O(D\sqrt{n})$ messages. Since each iteration reduces the number of fragments by at least half, the number of iterations is $O(\log n)$. At the end of this iteration, several base fragments may share the same label. In subsequent iterations, each base fragment finds its MOE (i.e., the MOE between itself and the other base fragments which do not have the same label) by convergecasting within its own fragment and (the leader of the base fragment) sends the MOE to the root; thus $O(\sqrt{n})$ edges are sent to the root (one per base fragment), though there is a lesser number of combined fragments (with distinct labels). The root node finds the overall MOE of the combined fragments and initiates the merging. Since the time and message complexity per merging iteration is $O(D + \sqrt{n})$ and $O(D\sqrt{n}) = O(n)$, respectively, the overall complexity is still within the sought bounds.

5.1.2 Phase 2: When D and the Number of Fragments are Large:

When D is large (e.g. $n^{1/2+\varepsilon}$, for some $0 < \varepsilon \leq 1/2$) and the number of fragments is large (e.g. $\Theta(\sqrt{n})$) the previous approach of merging via the root of the

global BFS tree does not work directly, since the message complexity would be $O(D\sqrt{n})$. A second idea addresses this issue by merging in a manner that respects *locality*. That is, we merge fragments that are close by using a *local* leader, such that the MOE edges do not have to travel too far. The high-level idea is to use a *hierarchy of sparse neighborhood covers* due to [10] to accomplish the merging. [9] Intuitively, a sparse neighborhood cover is a decomposition of a graph into a set of overlapping clusters that satisfy suitable properties.

The main intuitions behind using a cover are the following: (1) the clusters of the cover have relatively smaller diameter (compared to the strong diameter of the fragment and is always bounded by D) and this allows efficient communication for fragments contained within a cluster (i.e., the weak diameter of the fragment is bounded by the cluster diameter); (2) the clusters of a cover overlap only a little, i.e., each vertex belongs only to a few clusters; this allows essentially congestion-free (overhead is at most polylog(n) per vertex) communication and hence operations can be done efficiently in parallel across all the clusters of a cover. This phase continues till the number of fragments reduces to $O(n/D)$, when we switch to Phase 3. We next give more details on the merging process in Phase 2.

Communication-Efficient Paths. An important technical aspect in the merging process is constructing efficient communication paths between nearby fragments, which are maintained and updated by the algorithm in each iteration. The algorithm requires fragments to be communication-efficient, in the sense that there is an additional set of *short paths* between the fragment leader f and fragment members, which may use "shortcuts" through vertices that are not part of the fragment to reduce the distance. A fragment F is *communication-efficient* if, for each vertex $v \in F$, there exists a path between v and f (possibly including vertices in $V(G) \setminus V(F)$) of length $O(\mathsf{diam}_G(F) + \sqrt{n})$, where $diam_G(F)$ is the weak diameter of F.

It can be shown that, in each iteration, all fragments find their respective MOE in time $\tilde{O}(\sqrt{n} + D)$ and $\tilde{O}(m)$ messages. Note that it is difficult to merge all fragments along their respective MOE, as this might create long fragment-chains. Instead, [45] compute a maximal matching in the fragment graph $\mathcal{F}_i$ induced by the MOE edges to merge fragments in a controlled manner. The crucial part is to use time- and message communication between the fragment leader and the nodes in the fragment (while finding MOEs) as well as between fragment leaders of adjacent fragments, which is made possible by the existence of the communication-efficient paths.

[9] Neighborhood covers were used by Elkin [10] to improve the running time of the Pipeline procedure of his distributed MST algorithm; on the other hand, in [45] they are used to replace the Pipeline part entirely.

In more detail, [45] use a hierarchy of sparse neighborhood covers to construct communication-efficient fragments. Each cover in the hierarchy consists of a collection of clusters of a certain radius, where the lowest cover in the hierarchy contains clusters of radius $O(\sqrt{n})$ and subsequent covers in the hierarchy have clusters of geometrically increasing radii, whereby the last cover in the hierarchy is simply the BFS tree of the entire graph. Initially, it is easy to construct communication-efficient paths in base fragments, since they have strong diameter $O(\sqrt{n})$. In subsequent iterations, when merging adjacent fragments, the algorithm finds a cluster that is (just) large enough to contain both fragments. The neighborhood property of the cluster allows the algorithm to construct communication-efficient paths between merged fragments (that might take shortcuts outside the fragments, and hence have small *weak diameter*), assuming that the fragments before merging are efficient. Note that it is important to make sure that the number of fragments in a cluster is not too large in relation to the radius of the cluster—otherwise the message complexity would be high. Hence, a key invariant maintained through all the iterations is that the *cluster depth times the number of fragments that are contained in the cluster of such depth is always bounded by* $\tilde{O}(n)$. This invariant is maintained by making sure that the number of fragments per cluster reduces sufficiently to compensate for the increase in cluster radius: At the end of the phase, when the cluster radius is D, the number of fragments is $\tilde{O}(n/D)$.

5.1.3 Phase 3: When the Cluster Radius is D:

Phase 3 employs a merging procedure similar to the one used in Phase 1. Recall that, in Phase 1, in every merging iteration, each base fragment finds their respective MOEs (i.e., MOEs between itself and the remaining fragments) by convergecasting to their respective leaders; the leaders send at most $O(\sqrt{n})$ edges to the root by upcast. The root merges the fragments and sends out the merged information to the base fragment leaders. In Phase 3, it is possible to treat the $O(n/D)$ fragments remaining at the end of Phase 2 as the "base fragments" and repeat the above process. An important difference to Phase 1 is that the merging leaves the leaders of these base fragments intact: in the future iterations of Phase 3, each of these base fragments again tries to find an MOE, whereby only edges that have endpoints in fragments with distinct labels are considered as candidate for the MOE.

Note that the fragment leaders communicate with their respective nodes as well as the BFS root via the hierarchy of communication-efficient routing paths constructed in Phase 2; these incur only a polylogarithmic overhead. This takes $\tilde{O}(D + n/D)$ time (per merging iteration) since $O(n/D)$ MOE edges are sent to the root of the BFS tree via communication-efficient paths (in every merging iteration) and a message complexity of $\tilde{O}(D \cdot n/D) = \tilde{O}(n)$ (per merging iteration) since, in

each iteration, each of the $O(n/D)$ edges takes $\tilde{O}(D)$ messages to reach the root. Since there are $O(\log n)$ iterations overall, the overall bound follows.

5.2 Deterministic Singularly-Optimal Algorithms

Shortly after the work of Pandurangan et al. [45], Elkin[12] presented a new simultaneously optimal distributed MST algorithm which 1) is deterministic, thus answering an explicit question raised in [45], and 2) is simpler and allows for a simpler analysis [12].

The main novelty in Elkin's algorithm is to grow fragments up to (strong) diameter $O(D)$, as opposed to $O(\sqrt{n})$, in the first phase of the algorithm. This results in an $(O(n/D), O(D))$-MST forest as the base forest, as opposed to an $(O(\sqrt{n}), O(\sqrt{n}))$-MST forest. The algorithm in [45] is then executed on top of this base forest.

This simple change brings benefits to the case $D \geq \sqrt{n}$, for which now the complexities of finding minimum-weight outgoing edges and of their subsequent upcast to the root of the auxiliary BFS tree of the network are within the desired time and message bounds. Hence Phase 2 of Part 2 of Pandurangan et al.'s algorithm can be bypassed, and since this phase contains the sole randomized portion of the algorithm (that is, the randomized cover construction of [10]), the final result is a fully deterministic algorithm.

A different approach toward singular-optimality has been given recently by Haeupler et al. [21], who provide a new deterministic distributed algorithm for MST which is simultaneously time- and message-optimal, and also achieve the same goal for some other problems.

6 Time and Message Lower Bounds

In this section, we provide some intuition behind the complexity lower bounds that have been shown for the distributed MST problem.

The Peleg-Rubinovich Lower Bound. In a breakthrough result, Peleg and Rubinovich [48] showed that $\Omega(D + \sqrt{n}/B)$ time is required by any distributed algorithm for constructing an MST, even on networks of small diameter (at least $\Omega(\log n)$), when assuming that at most B bits can be sent across an edge in a round. As a consequence, this result established the asymptotic (near) time-optimality of the algorithm of [36]. Peleg and Rubinovich introduced a novel graph construction that forces any algorithm to trade-off dilation with communication bottlenecks. While the original lower bound of Peleg and Rubinovich applies to deterministic algorithms for exact MST computation, the more recent work of [11, 7] extend

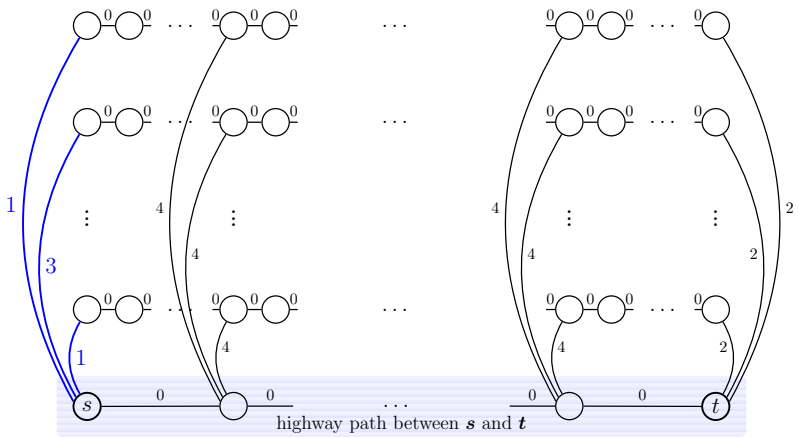

Figure 1: The simplified Peleg-Rubinovich lower bound graph of [48]. The shaded bottom path between vertices s and t represents the *highway path* and the rest of the other horizontal paths between the two nodes represent the vertex-disjoint *long paths*. The weight of each blue edge incident to s is chosen to be either 1 or 3. The *spoke edges* connect each highway vertex to the appropriate vertices of each long path and have weight 4.

this result by showing a lower bound of $\tilde{\Omega}(D + \sqrt{n})$ for randomized (Monte Carlo) and approximation algorithms as well.

The (simplified) lower bound graph construction considers two designated nodes, s and t, which are connected by $\Theta(\sqrt{n})$ vertex-disjoint *long paths*, each of length $\Theta(\sqrt{n})$, and one *highway path* consisting of $D = \Theta(n^{1/4})$ *highway vertices*, which determines the graph diameter. To ensure that the resulting graph does indeed have diameter $O(D)$, *spoke edges* are added from each i-th highway vertex to the respective (iD)-th vertex of each long path. Figure 1 depicts the weighted variant of the above graph; the assignment of the weights is explained below.

So far, the constructed graph $\bar{G}$ is unweighted and, as an intermediate step, [48] consider the *mailing problem* on $\bar{G}$, where node s is given as input a $\Theta(\sqrt{n})$-length bit string. An algorithm solves the mailing problem if eventually node t outputs the same bit string. The way $\bar{G}$ is constructed, one strategy is to send the input bits held by s in parallel along the $\Theta(\sqrt{n})$ long paths; another strategy is to send them along the (single) highway path of length $\Theta(n^{1/4})$ towards t, which means the bits are essentially forced to travel in sequence since each (highway) edge can carry at most B bits per round. Of course, we cannot make any assumptions on how exactly these bits are being sent from s to t: In fact, an algorithm is not bound to send these bits explicitly (i.e. one by one) along the edges but might attempt to somehow compress the bit string to speed up the process. To

rule out such shortcuts, [48] show that any algorithm that correctly solves the mailing problem, must ensure that the size of the possible state space of node t is large enough upon termination. That is, since there are $2^{\Theta(\sqrt{n})}$ choices for the input bit string given to s, the number of distinct states that node t can be in upon termination must also be at least $2^{\Theta(\sqrt{n})}$. Otherwise, there are at least 2 distinct input bit strings of s for which t is in the same state upon termination and hence t (wrongly) outputs the same bit string for both inputs. Initially, only s holds the input string and hence all other nodes (including t) have a state space of size 1. Taking into account the possibility of remaining silent, there are $2^B + 1$ possible ways that any given edge can be used in any given round. It follows that, after r rounds, the "information" delivered by the highway edges can increase the state space by a factor of at most $(2^B + 1)^r$. Given that $r \geq \Theta(\sqrt{n}/B)$ must hold upon termination, [48] obtain an $\Omega(\sqrt{n}/B)$ lower bound for the mailing problem. They also extend the above idea to graphs of diameter as small as $\Theta(\log n)$.

To obtain a time lower bound for the MST problem, Peleg and Rubinovich consider a weighted version of the graph, where all path edges have weight 0 and all spoke edges are given weight 4. The edges connecting t to the paths have weight 2, whereas the weights of edges incident to s have weight either 1 or 3. The crucial property of this construction is that node t must add to the MST edges its edge incident to the j-th slow path if and only if the weight of the edge that connects s to the j-th slow path is 3. See Figure 1 for an example. Intuitively speaking, this means that node t can only add the correct MST edges if there is an algorithm that solves the corresponding mailing problem, hence implying the sought $\Omega(D + \sqrt{n}/B)$ time lower bound.

The Das Sarma et al. [7] Lower Bound. The work of Das Sarma et al.[7] established that $\tilde{\Omega}(D + \sqrt{n})$ rounds is a fundamental time lower bound for several important distributed network problems including MST, shortest paths, minimum cut etc., and showed that this lower bound holds even for Monte-Carlo randomized algorithms. More importantly, they show that the bound holds for approximation algorithms as well, i.e., those that compute a solution that is within any non-trivial approximation factor to the optimal.

The approach of Das Sarma et al is to prove lower bounds by establishing an interesting connection between *communication complexity* and distributed computing. A key concept that facilitates this connection is the notion of *distributed verification*. In distributed verification, we are given a network G and a subgraph H of G where each vertex of G knows which edges incident on it are in H. The goal is to verify whether H has some properties, e.g., if it is a tree or if it is connected (every node knows at the end of the process whether H has the specified property or not). Das Sarma et al. [7] initiate a systematic study of dis-

tributed verification, and give almost tight lower bounds on the running time of distributed verification algorithms for many fundamental problems such as connectivity, spanning connected subgraph, and $s - t$ cut verification. They show applications of these results in deriving time lower bounds on the *hardness of distributed approximation* for many classical optimization problems including MST, shortest paths, and minimum cut.

The lower bound proofs consider the family of graphs (similar to the one in Figure 1) that was first used in [48, 10, 38]. However, while previous results [48, 10, 38, 31] rely on counting the number of states needed to solve the *mailing problem* (along with some sophisticated techniques for its variant, called *corrupted mailing problem*, in the case of approximation algorithm lower bounds), and use Yao's method [51] to derive lower bounds for randomized algorithms, the results of [7] used three main steps of simple reductions, starting from problems in communication complexity, as follows:

In the first step, they reduce the lower bounds of problems in the standard communication complexity model [33] to the lower bounds of the equivalent problems in the "distributed version" of communication complexity. Specifically, they prove the *Simulation Theorem* [7] which relates the *communication* lower bound from the standard communication complexity model [33] to compute some appropriately chosen function f, to the distributed *time* complexity lower bound for computing the same function in a specially chosen graph G. In the standard two-party communication complexity model[33], Alice and Bob can communicate directly (via a bidirectional edge of bandwidth one). In the distributed model, we assume that Alice and Bob are some vertices of G and they together wish to compute the function f using the communication graph G. The choice of graph G is critical and is the same as the graph used by Peleg and Rubinovich [48] (see Figure 1).

The connection established in the first step allows one to bypass the state counting argument and Yao's method, and reduces the task in proving lower bounds of *distributed verification problems* (see below) to merely picking the right function f to reduce from. The function f that is useful in showing randomized lower bounds for many problems (including MST) is the *set disjointness function* [33], which is the quintessential problem in the world of communication complexity with applications to diverse areas. Following a result well known in communication complexity [33], they show that the distributed version of the set disjointness problem has an $\tilde{\Omega}(\sqrt{n})$ lower bound on graphs of small diameter.

The second step is to reduce the concerned problem (such as MST or shortest paths) to an appropriate *verification problem* using simple reductions similar to those used in data streams [26]. The verification problem that is appropriate for MST is the *spanning connected subgraph verification* problem, where the goal is to verify if a given subgraph H of a graph G is spanning and connected.

Finally, in the third step, they reduce the verification problem to hardness of

distributed approximation for a variety of problems to show that the same lower bounds hold for approximation algorithms as well. For this, they use a reduction whose idea is similar to the one used to prove hardness of approximating TSP (Traveling Salesman Problem) on general graphs (see, e.g., [50]): Convert a verification problem to an optimization problem by introducing edge weights in such a way that there is a large gap between the optimal values for the cases where H satisfies (or does not satisfy) a certain property. This technique is surprisingly simple, yet yields strong unconditional hardness bounds — many hitherto unknown, left open (e.g., minimum cut) [9] and some that improve over known ones (e.g., MST and shortest path tree) [10]. As mentioned earlier, this approach shows that approximating MST by *any* factor needs $\tilde{\Omega}(\sqrt{n})$ time, while the previous result due to Elkin gave a bound that depends on α (the approximation factor), i.e. $\tilde{\Omega}(\sqrt{n/\alpha})$, using more sophisticated techniques.

To summarize, the lower bound proof for MST that applies to Monte Carlo randomized and approximation algorithms proceeds as follows. It starts by reducing the set disjointness problem in the standard two party communication complexity setting to the distributed version of the set disjointness problem which has to solved by communicating in a graph (the one in Figure 1). Then the distributed version is in turn reduced to the spanning connected subgraph verification problem which is then finally reduced to MST or approximate MST. Hence starting with the lower bound of the set disjointness problem (which is already known in communication complexity) we obtain a lower bound for MST.

The lower bound proof technique via this approach is quite general and conceptually straightforward to apply as it circumvents many technical difficulties by using well-studied communication complexity problems. Nevertheless, this approach yields tight lower bounds for many problems. More specifically, a significant advantage of this technique over previous approaches is that it avoids starting from scratch every time we want to prove a lower bound for a new problem. For example, extending the result from the mailing problem in [48] to the corrupted mailing problem in [10] requires some sophisticated techniques. The new technique, on the other hand, allows one to use known lower bounds in communication complexity to streamline such tasks. Another advantage of the approach by [7] is that extending a deterministic lower bound to a randomized one is sometimes technically difficult, when all that is available is a deterministic bound. By using connection to communication complexity, the technique of [7] allows one to obtain lower bounds for *Monte Carlo* randomized algorithms, while previous lower bounds hold only for Las Vegas randomized algorithms. The general approach of Das Sarma et al has been subsequently used to show many new lower bounds for other distributed graph algorithms.

A Simultaneous Lower Bound. The prior time and message lower bounds for MST are derived using two completely different graph constructions; the existing lower bound construction that shows one lower bound does not work for the other. Specifically, the existing graph constructions of [11, 7] used to establish the lower bound of $\tilde{\Omega}(D + \sqrt{n})$ rounds does not simultaneously yield the message lower bound of $\Omega(m)$; similarly, the existing lower bound graph construction of [35] that shows the message lower bound of $\Omega(m)$ (cf. Section 3.1) does not simultaneously yield the time lower bound of $\tilde{\Omega}(D + \sqrt{n})$.

In [45], the authors combine the two lower bound techniques — hardness of distributed symmetry breaking, used to show the message lower bound of $\Omega(m)$ for the KT_0 model [35], and communication complexity, used to show the lower bound on time complexity [7] — to show that, for any algorithm, there is some graph where *both* the message and time lower bounds hold.

7 Conclusion and Open Problems

This article surveyed the distributed MST problem, a cornerstone problem in distributed network algorithms. It has received a rich and continuing interest since the very first distributed MST algorithm was published more than 35 years ago.

On the algorithmic side, there has been a lot of work in trying to improve the time complexity and more recently both time and message complexity with the goal of obtaining singularly optimal algorithms. An important conceptual contribution that research in distributed MST introduced is the notion of efficiency with respect to specific graph parameters such as D, the network diameter. This notion led to development of new algorithms — starting with that of [16] and leading to the more recent singularly-optimal algorithms of [45, 12, 21]. More importantly, it also led to a deeper understanding of lower bounds in distributed network algorithms. The time lower bound of $\tilde{\Omega}(D + \sqrt{n})$ for MST first shown by Peleg and Rubinovich [48] led to similar lower bounds for distributed network algorithms for various other problems, including for randomized algorithms and approximation algorithms [7].

An open question is whether there exists a distributed MST algorithm with near-optimal time and message complexities in the KT_1 variant of the model, i.e., using $\tilde{O}(n)$ messages and $\tilde{O}(D + \sqrt{n})$ time. Very recently there has been some progress on this question [20]. Another interesting question is improving the performance of MST in special classes of graphs, e.g., in graphs of diameter two (where it is not known if one can get $o(\log n)$-round algorithms). There has been recent progress in this direction (cf. Section 3.1).

Currently, it is not known whether other important problems such as shortest paths, minimum cut, and random walks, enjoy singular optimality. These prob-

lems admit distributed algorithms which are (essentially) time-optimal but not message-optimal [41, 25, 8, 42]. Some work in this direction recently started to appear [21], but further research is needed to address these questions.

References

[1] K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 459–467, 2012.

[2] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM Symposium on Principles of Database Systems (PODS)*, pages 5–14, 2012.

[3] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 230–240, 1987.

[4] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990.

[5] O. Borůvka. O jistém problému minimálním (about a certain minimal problem). *Práce Mor. Prírodoved. Spol. v Brne III*, 3, 1926.

[6] F. Chin and H. Ting. An almost linear time and $O(n \log n + e)$ messages distributed algorithm for minimum-weight spanning trees. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 257–266, 1985.

[7] A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.

[8] A. Das Sarma, D. Nanongkai, G. Pandurangan, and P. Tetali. Distributed random walks. *J. ACM*, 60(1), 2013.

[9] M. Elkin. Distributed approximation: a survey. *SIGACT News*, 35(4):40–57, 2004.

[10] M. Elkin. A faster distributed protocol for constructing a minimum spanning tree. *J. Comput. Syst. Sci.*, 72(8):1282–1308, 2006.

[11] M. Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM J. Comput.*, 36(2):433–456, 2006.

[12] M. Elkin. A simple deterministic distributed MST algorithm, with near-optimal time and message complexities. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 157–163, 2017.

[13] M. Faloutsos and M. Molle. A linear-time optimal-message distributed algorithm for minimum spanning trees. *Distributed Computing*, 17(2):151–170, 2004.

[14] E. Gafni. Improvements in the time complexity of two message-optimal election algorithms. In *Proceedings of the 4th Symposium on Principles of Distributed Computing (PODC)*, pages 175–185, 1985.

[15] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[16] J. A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, 1998.

[17] M. Ghaffari and B. Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, MST, and min-cut. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 202–219, 2016.

[18] M. Ghaffari, F. Kuhn, and H.-H. Su. Distributed MST and routing in almost mixing time. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2017.

[19] M. Ghaffari and M. Parter. MST in log-star rounds of congested clique. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 19–28, 2016.

[20] R. Gmyr and G. Pandurangan. Time-message tradeoffs in distributed algorithms. In *Manuscript*, 2018.

[21] B. Haeupler, D. E. Hershkowitz, and D. Wajc. Round- and message-optimal distributed graph algorithms. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, 2018. To appear.

[22] B. Haeupler, T. Izumi, and G. Zuzic. Low-congestion shortcuts without embedding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 451–460, 2016.

[23] B. Haeupler, T. Izumi, and G. Zuzic. Near-optimal low-congestion shortcuts on bounded parameter graphs. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*, pages 158–172, 2016.

[24] J. W. Hegeman, G. Pandurangan, S. V. Pemmaraju, V. B. Sardeshmukh, and M. Scquizzato. Toward optimal bounds in the congested clique: graph connectivity and MST. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–100, 2015.

[25] M. Henzinger, S. Krinninger, and D. Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the 48th ACM Symposium on Theory of Computing (STOC)*, pages 489–498, 2016.

[26] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In J. M. Abello and J. S. Vitter, editors, *External memory algorithms*, pages 107–118. American Mathematical Society, Boston, MA, USA, 1999.

[27] T. Jurdzinski and K. Nowicki. MST in $O(1)$ rounds of congested clique. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2620–2632, 2018.

[28] M. Khan and G. Pandurangan. A fast distributed approximation algorithm for minimum spanning trees. *Distributed Computing*, 20(6):391–402, 2008.

[29] V. King, S. Kutten, and M. Thorup. Construction and impromptu repair of an MST in a distributed network with $o(m)$ communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 71–80, 2015.

[30] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson. Distributed computation of large-scale graph problems. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015.

[31] L. Kor, A. Korman, and D. Peleg. Tight Bounds For Distributed MST Verification. In *STACS*, pages 69–80, 2011.

[32] L. Kor, A. Korman, and D. Peleg. Tight bounds for distributed minimum-weight spanning tree verification. *Theory Comput. Syst.*, 53(2):318–340, 2013.

[33] E. Kushilevitz and N. Nisan. *Communication complexity*. Cambridge University Press, New York, NY, USA, 1997.

[34] S. Kutten, D. Nanongkai, G. Pandurangan, and P. Robinson. Distributed symmetry breaking in hypergraphs. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC)*, pages 469–483, 2014.

[35] S. Kutten, G. Pandurangan, D. Peleg, P. Robinson, and A. Trehan. On the complexity of universal leader election. *J. ACM*, 62(1), 2015.

[36] S. Kutten and D. Peleg. Fast distributed construction of small k-dominating sets and applications. *J. Algorithms*, 28(1):40–66, 1998.

[37] Z. Lotker, B. Patt-Shamir, E. Pavlov, and D. Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM J. Comput.*, 35(1):120–131, 2005.

[38] Z. Lotker, B. Patt-Shamir, and D. Peleg. Distributed MST for constant diameter graphs. *Distributed Computing*, 18(6):453–460, 2006.

[39] A. Mashreghi and V. King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN)*, 2017.

[40] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.

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

[42] D. Nanongkai, A. D. Sarma, and G. Pandurangan. A tight unconditional lower bound on distributed randomwalk computation. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 257–266, 2011.

[43] G. Pandurangan. *Distributed Network Algorithms*. 2018. `https://sites.google.com/site/gopalpandurangan/dna`.

[44] G. Pandurangan, P. Robinson, and M. Scquizzato. Fast distributed algorithms for connectivity and MST in large graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 429–438, 2016.

[45] G. Pandurangan, P. Robinson, and M. Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. In *Proceedings of the 49th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 743–756, 2017.

[46] D. Peleg. Distributed matroid basis completion via elimination upcast and distributed correction of minimum-weight spanning trees. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 164–175, 1998.

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

[48] D. Peleg and V. Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, 2000.

[49] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[50] V. V. Vazirani. *Approximation Algorithms*. Springer, July 2001.

[51] A. C.-C. Yao. Probabilistic Computations: Toward a Unified Measure of Complexity. In *FOCS*, pages 222–227, 1977.

Extending Causal Consistency to any Object Defined by a Sequential Specification

Achour Mostéfaoui[†], Matthieu Perrin[†], Michel Raynal[‡,⋆]

[†]LINA, Université de Nantes, 44322 Nantes, France
[‡]Univ Rennes, IRISA, 35042 Rennes, France
[⋆]Department of Computing, Polytechnic University, Hong Kong

Abstract

This paper presents a generalization of causal consistency suited to the family of objects defined by a sequential specification. As causality is captured by a partial order on the set of operations issued by the processes on shared objects (concurrent operations are not ordered), it follows that causal consistency allows different processes to have different views of each object history.

Keywords: Causality, Causal order, Concurrent object, Consistency condition.

1 Processes and Concurrent Objects

Let us consider a set of n sequential asynchronous processes p_1, ..., p_n, which cooperate by accessing shared objects. These objects are consequently called *concurrent* objects. A main issue consists in defining the correct behavior of concurrent objects. Two classes of objects can be distinguished according to way they are specified.

- The objects which can be defined by a sequential specification. Roughly speaking, this class of objects includes all the objects encountered in sequential computing (e.g., queue, stack, set, dictionary, graph). Different tools can be used to define their correct behavior (e.g., transition function, list of all the correct traces -histories-, pre and post-conditions, etc.).

It is usually assumed that the operations accessing these objects are *total*, which means that, whatever the current state of the object, an operation always returns a result.

As an example, let us consider a bounded stack. A pop() operation returns a value if the stack is not empty, and returns the default value $\perp$ if it is empty. A push(v) operation returns the default value $\top$ if the stack is full, and returns the default ok otherwise (v was then added to the stack). A simpler example is a read/write register, where a read operation always returns a value, and a write operation always returns ok.

- The objects which cannot be defined by a sequential specification. Example of such objects are Rendezvous objects or Non-blocking atomic commit objects [10]. These objects require processes to wait each other, and their correct behavior cannot be captured by sequences of operations applied to them.

In the following we consider objects defined by a sequential specification.

2 Strong Consistency Conditions

Strong consistency conditions are *natural* (and consequently easy to understand and use) in the sense that they require each object to appear as if it has been accessed sequentially. In a failure-free context, this can be easily obtained by using mutual exclusion locks bracketing the invocation of each operation.

Atomicity/Linearizability The most known and used consistency condition is *atomicity*, also called *linearizability*[1]. It requires that each object appears as if it was accessed sequentially, this sequence of operations S belonging to the specification of the object, and complying with the real-time order of their occurrences (which means that, if an operation op1 terminates before an operation op2 starts, op1 must appear before op2 in the sequence S).

Sequential consistency This consistency condition, introduced in [16], is similar to, but weaker than, linearizability, namely, it does not require the sequence of operations to comply with real-time order (which means that, while an operation op1 terminates before an operation op2 starts, op2 may appear before op1 in the sequence S).

[1] Atomicity was formally defined in [17, 18] for basic read/write objects. It was then generalized to any object defined by a sequential specification in [13]. We consider these terms as synonyms in the following.

Figure 1 presents an example of a sequentially consistent computation (which is not atomic) involving two read/write registers $R1$ and $R2$, accessed by two processes p_1 and p_2. The dashed arrows define the *causality* relation linking the read and write operations on each object (also called *read-from* relation when the object is a read/write register). It is easy to see that the sequence of operations made up of all the operations issued by p_2, followed by all the operations issued by p_1, satisfies the definition of sequential consistency.

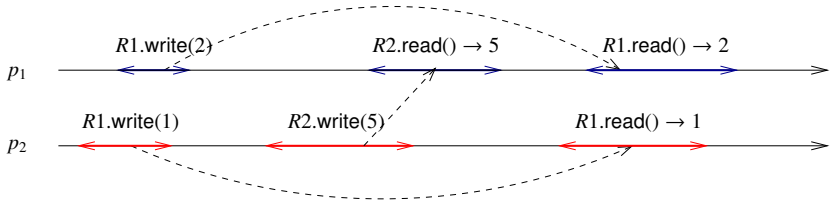

Figure 1: A sequentially consistent computation (which is not atomic)

Building objects satisfying a strong consistency condition in an asynchronous message-passing system Shared memories usually provide processes with objects built on top of basic atomic read/write objects or more sophisticated objects accessed by atomic operations such as Test&Set or Compare&Swap [11, 12, 23, 27]. This is no longer the case in message-passing systems where all the objects (except communication channels) have to be built from scratch [5, 22].

Constructions of sequentially consistent objects and atomic objects in failure-free message-passing systems can be found in [4, 5, 7, 21, 22]. These implementations rest on a mechanism which allows a total order on all operations to be built. This can be done by a central server, or a broadcast operation delivering messages in the same order at all the processes. Such an operation is usually called *total order broadcast* (TO-broadcast) or *atomic broadcast*. It is shown in [21] that, from an implementation point of view, sequential consistency can be seen as a form of lazy linearizability. The "compositional" power of sequential consistency is addressed in [8, 19].

The construction of objects satisfying a strong consistency condition (such as atomicity) in failure-prone message-passing systems is more difficult. A few objects including read/write registers, renaming objects, and snapshot objects, can be built in systems where, in each execution, a majority of processes do not crash [3]). A communication abstraction devoted to this class of objects has recently been proposed in [14]. This is no longer the case for a lot of common sequentially defined objects (e.g., as stacks and queues) which cannot be built in the presence of asynchrony and (even a minority of) process crashes [9]. Systems have

to be enriched with additional computing power (such as randomization or failure detectors) to allow such objects to be built (see, for example [24], for more developments on this subject).

3 Causal Consistency on Read/Write Objects (Causal Memory)

Causality-based consistency condition A causal memory is a set of read/write objects satisfying a consistency property weaker that atomicity or sequential consistency. This notion was introduced in [1]. It relies on a notion of *causality* similar to the one introduced in [15] for message-passing systems.

The main difference between causal memory and the previous strong consistency conditions lies in the fact that causality is captured by a partial order, which is trivially weaker than a total order. A total order-based consistency condition forces all the processes to see the same order on the object operations. Causality-based consistency does not. Each process can have its own view of the execution, their "greatest common view" being the causality partial order produced by the execution. Said differently, an object defined by a strong consistency condition is a *single-view* object, while an object defined by a causality-based consistency condition is a *multi-view* object (one view per process).

Another difference between a causality-based consistency condition and a strong consistency condition lies in the fact that a causality-based consistency condition copes naturally with process crashes and system partitioning.

Preliminary definitions As previously indicated, a causal memory is a set of read/write registers. Its semantics is based on the following preliminary definitions (from [1, 13]). To simplify the presentation and without loss of generality, we assume that (a) all the values written in a register are different, and (b) each register has an initial value written by a fictitious write operation.

- A *local (execution) history* L_i of a process p_i is the sequence of read and write operations issued by this process. If the operations $\mathsf{op1}$ and $\mathsf{op2}$ belong to L_i and $\mathsf{op1}$ appears before $\mathsf{op2}$, we say "$\mathsf{op1}$ precedes $\mathsf{op2}$ in p_i's process order". This is denoted $\mathsf{op1} \stackrel{i}{\rightarrow} \mathsf{op2}$.

- The *write-into relation* (denoted $\stackrel{wi}{\rightarrow}$) captures the effect of write operations on the read operations. Denoted $\stackrel{wi}{\rightarrow}$, it is defined as follows: $\mathsf{op1} \stackrel{wi}{\rightarrow} \mathsf{op2}$ if $\mathsf{op1}$ is the write of a value v into a register R and $\mathsf{op2}$ is a read operation of the register R which returns the value v.

- An *execution history* H is a partial order composed of one local history per process, and a partial order, denoted $\stackrel{po}{\rightarrow}$, defined as follows: $\mathsf{op1} \stackrel{po}{\rightarrow} \mathsf{op2}$ if
  - $\mathsf{op1}, \mathsf{op2} \in L_i$ and $\mathsf{op1} \stackrel{i}{\rightarrow} \mathsf{op2}$ (process order), or
  - $op1 \stackrel{wi}{\rightarrow} op2$ (write-into order), or
  - $\exists\, \mathsf{op3}$ such that $\mathsf{op1} \stackrel{po}{\rightarrow} \mathsf{op3}$ and $\mathsf{op3} \stackrel{po}{\rightarrow} \mathsf{op2}$ (transitivity).
- Two operations not related by $\stackrel{po}{\rightarrow}$ are said to be *independent* or *concurrent*.
- The projection of H on a register R (denoted $H|R$) is the partial order H from which are suppressed all the operations which are not on R.
- A *serialization* S of an execution history H (whose partial order is $\stackrel{po}{\rightarrow}$) is a total order such that, if $\mathsf{op1} \stackrel{po}{\rightarrow} \mathsf{op2}$, then $\mathsf{op1}$ precedes $\mathsf{op2}$ in S.

A remark on the partial order relation As we can see, the read-from relation mimics the causal send/receive relation associated with message-passing [15]. The difference is that zero, one, or several reads can be associated with the same write. In both cases, the (write-into or message-passing) causality relation is a global property (shared by all processes) on which is built the consistency condition. It captures the effect of the environment on the computation (inter-process asynchrony), while process orders capture the execution of the algorithms locally executed by each process.

Causal memory Let H_{i+w} be the partial order $\stackrel{po}{\rightarrow}$, from which all the read operations not issued by p_i are suppressed (the subscript $i + w$ means that only all the operations issued by p_i plus all write operations are considered).

As defined in [1], an execution history H is *causal* if, for each process p_i, there is a serialization S_i of H_{i+w} in which each read from a register R returns the value written in R by the most recent preceding write in R.

This means that, from the point of view of each process p_i, taken independently from the other processes, each register behaves as defined by its sequential specification. It is important to see, that different processes can have different views of a same register, each corresponding to a particular serialization of the partial order $\stackrel{po}{\rightarrow}$ from which the read operations by the other processes have been eliminated.

An example of a causal memory execution is depicted in Figure 2. Only one write-into pair is indicated (dashed arrow). As $R1.\mathsf{write}(1)$ and $R1.\mathsf{write}(2)$ are independent, each of the operations $R1.\mathsf{read}()$ by p_2 and p_3 can return any value, i.e., $u, v \in \{1, 2\}$. For the same reason, and despite the write-into pair on the register $R2$ involving p_1 and p_3, the operation $R1.\mathsf{read}()$ issued by p_3 can return

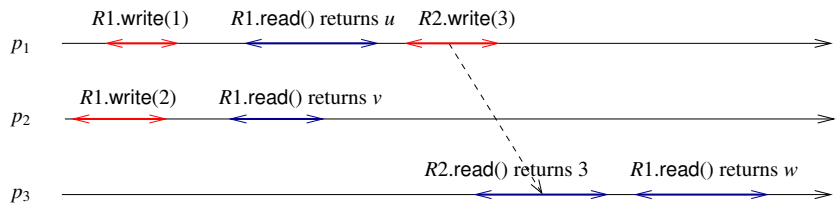

Figure 2: Example of an execution of a causal read/write memory

$w \in \{1, 2\}$. This shows that different processes can obtain different "views" of the same causal memory execution. Once a read returned a value, a new write-into pair is established.

Implementations of a causal read/write memory (e.g., [2]) rest on an underlying communication algorithm providing causal message delivery [6, 26]. It is shown in [1, 25] that, in executions that are data race-free or concurrent write-free, a causal memory behaves as a sequentially consistent read/write memory.

4 Causal Consistency for any Object

The problem Although it was introduced more than 20 years ago, it appears that, when looking at the literature, causal consistency has been defined and investigated only for read/write objects (the only exception we are aware of is [20]). This seems to be due to the strong resemblance between read/write operations and send/receive operations. Hence, the question: Is it possible to generalize causal consistency to any object defined by a sequential specification? This section answers positively this question.

Preliminary definitions The notations and terminology are the same as in the previous section, but now the operations are operations on any object O of a set of objects $\mathcal{O}$, each defined by a sequential specification.

Considering a set of local histories and a partial order $\stackrel{po}{\rightarrow}$ on their operations, let $Assignment_i(\stackrel{po}{\rightarrow})$ denote a function which replaces the value v returned by each operation $\mathsf{op}()$ not issued by p_i by a value v', possibly different from v, the only constraint being that v and v' belong to the same domain (as defined by the corresponding operation $\mathsf{op}()$). Let us notice that $Assignment_i(\stackrel{po}{\rightarrow})$ is not allowed to modify the values returned by the operations issued by p_i. Moreover, according to the domain of values returned by the operations, a lot of different assignments can be associated with each process p_i.

Given a partial order $\stackrel{po}{\rightarrow}$, and an operation op, the *causal past* of op with

respect to $\stackrel{po}{\rightarrow}$ is the set of operations $\{\mathsf{op}' \mid \mathsf{op}' \stackrel{po}{\rightarrow} \mathsf{op}\}$. A serialization S_i of a partial order $\stackrel{po}{\rightarrow}$ is said to be *causal past-constrained* if it is such that, for any operation op issued by p_i, only the operations of the causal past of op appear before op.

Causal consistency for any object Let $H = \langle L_1, \ldots, L_n \rangle$ be a set of n local histories (one per process) which access a set CO of concurrent objects, each defined by a sequential specification. H is *causally consistent* if there is a partial order $\stackrel{po}{\rightarrow}$ on the operations of H such that for any process p_i:

- $(\mathsf{op1} \stackrel{i}{\rightarrow} \mathsf{op2}) \Rightarrow (\mathsf{op1} \stackrel{po}{\rightarrow} \mathsf{op2})$, and
- $\exists$ an assignment $Assignment_i$ and a causal past-constrained serialization S_i of $Assignment_i(\stackrel{po}{\rightarrow})$ such that, $\forall\ O \in CO$, $S_i|O$ belongs to the sequential specification of O.

The first requirement states that the partial order $\stackrel{po}{\rightarrow}$ must respect all process orders. The second requirement states that, as far as each process p_i is concerned, the local view (of $\stackrel{po}{\rightarrow}$) it obtains is a total order (serialization S_i) that, according to some value assignment, satisfies the sequential specification of each object O.[2]

Let us remark that the assignments $Assignment_i()$ and $Assignment_j()$ associated with p_i and p_j, respectively, may provide different returned values in S_i and S_j for the same operation. Each of them represents the local view of the corresponding process, which is causally consistent with respect to the global computation as captured by the relation $\stackrel{po}{\rightarrow}$.

When the objects are read/write registers The definition of a causal memory stated in Section 3 is a particular instance of the previous definition. More precisely, given a process p_i, the assignment $Assignment_i$ allows an appropriate value to be associated with every read not issued by p_i. Hence, there is a (local to p_i) assignment of values such that, in S_i, any read operation returns the last written value. In a different, but equivalent way, the definition of a causal read/write memory given in [1] eliminates from S_i the read operations not issued by p_i.

While such operation eliminations are possible for read/write objects, they are no longer possible when one wants to extend causal consistency to any object defined by a sequential specification. This come from the observation that, while

[2]This definition is slightly stronger than the definition proposed in [20]. Namely, in addition to the introduction of the assignment notion, the definition introduced above adds the constraint that, if an operation op precedes an operation op' in the process order, then the serialization required for op must be a prefix of the serialization required for op'. On the other hand, it describes precisely the level of consistency achieved by Algorithm 1 presented below.

a write operation resets "entirely" the value of the object, "update" operations on more sophisticated objects defined by a sequential specification (such as the operations push() and pop() on a stack for example), do not reset "entirely" the value of the object. The "memory" captured by such objects has a richer structure than the one of a basic read/write object.

An example As an example illustrating the previous general definition of a causally consistent object, let us consider three processes p_1 p_2 and p_3, whose accesses to a shared unbounded stack are captured by the following local histories L_1, L_2, and L_3. In these histories, the notation $\mathsf{op}_i(a)r$ denotes the operation op() issued by p_i, with the input parameter a, and whose returned value is r.

- $L_1 = \mathsf{push}_1(a)\mathsf{ok},\ \mathsf{push}_1(\mathsf{c})\mathsf{ok}, \mathsf{pop}_1()c.$
- $L_2 = \mathsf{pop}_2()a,\ \mathsf{push}_2(b)\mathsf{ok}, \mathsf{pop}_2()b.$
- $L_3 = \mathsf{pop}_3()a,\ \mathsf{pop}_3()b.$

Hence, the question: Is $H = \langle L_1, L_2, L_3\rangle$ causally consistent? We show that the answer is "yes". To this end we need first to build a partial order $\stackrel{po}{\rightarrow}$ respecting the three local process orders. Such a partial order is depicted in Figure 3, where process orders are implicit, and the inter-process causal relation is indicated with dashed arrows (let us remind that this relation captures the effect of the environment –asynchrony– on the computation).

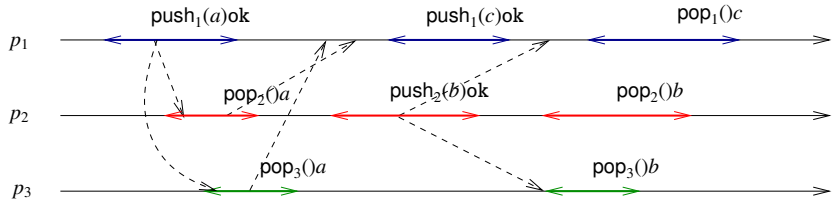

Figure 3: Example of a partial order on the operations issued on a stack

The second step consists in building three serializations respecting $\stackrel{po}{\rightarrow}$, S_1 for p_1, S_2 for p_2, and S_3 for p_3, such that, for each process p_i, there is an assignment of values returned by the operations pop() ($Assignment_i()$), from which it is possible to obtain a serialization S_i belonging to the specification of the stack. Such assignments/serializations are given below.

- $S_1 = \mathsf{push}_1(a)\mathsf{ok},\ \mathsf{pop}_3()a,\ \mathsf{pop}_2()\perp,\ \mathsf{push}_1(c)\mathsf{ok},\ \mathsf{push}_2(b)\mathsf{ok},$
  $\mathsf{pop}_1()c,\ \mathsf{pop}_2()b,\ \mathsf{pop}_3()\perp.$

- S_2 = $\mathsf{push}_1(a)\mathsf{ok}$, $\mathsf{pop}_2()a$, $\mathsf{push}_2(b)\mathsf{ok}$, $\mathsf{pop}_2()b$, $\mathsf{pop}_3()\perp$, $\mathsf{pop}_3()\perp$, $\mathsf{push}_1(c)\mathsf{ok}$, $\mathsf{pop}_1()c$.

- S_3 = $\mathsf{push}_1(a)\mathsf{ok}$, $\mathsf{pop}_3()a$, $\mathsf{pop}_2()\perp$, $\mathsf{push}_2(b)\mathsf{ok}$, $\mathsf{pop}_3()b$, $\mathsf{pop}_2()\perp$, $\mathsf{push}_1(c)\mathsf{ok}$, $\mathsf{pop}_1()c$.

The local view of the stack of each process p_i is constrained only by the causal order depicted in Figure 3, and also depends on the way it orders concurrent operations. As far as p_2 is concerned we have the following, captured by its serialization/assignment S_2. (The serializations S_1 and S_3 are built similarly.) We have considered short local histories, which could be prolonged by adding other operations. As depicted in the figure, due to the last causality (dashed) arrows, those operations would have all the operations in $L_1 \cup L_2 \cup L_3$ in their causal past.

1. Process p_2 sees first $\mathsf{push}_1(a)\mathsf{ok}$, and consequently (at the implementation level) updates accordingly its local representation of the stack.
2. Then, p_2 sees its own invocation of $\mathsf{pop}_2()$ which returns it the value a.
3. Then, p_2 sees its own $\mathsf{push}_2(b)$ and $\mathsf{pop}_2()$ operations; $\mathsf{pop}_2()$ returns consequently b.
4. Finally p_2 becomes aware of the two operations $\mathsf{pop}_3()$ issued by p_3, and the operations $\mathsf{push}_1(c)$ and $\mathsf{pop}_1()$ issued by p_1. To have a consistent view of the stack, it considers the assignment of returned values that assigns the value $\perp$ to the two operations $\mathsf{pop}_3()$, and the value c to the operations $\mathsf{pop}_1()$. In this way, p_2 has a consistent view of the stack, i.e., a view which complies with the sequential specification of a stack.

A universal construction Algorithm 1 is a universal construction which builds causally consistent objects from their sequential specification. It considers deterministic objects. This algorithm is built on top of any underlying algorithm ensuring causal broadcast message delivery [6, 26][3]. Let "co_broadcast MSG(a)" denote the causal broadcast of a message tagged MSG carrying the value a. The associated causal reception at any process is denoted "co-delivery". "?" denotes a control value unknown by the processes at the application level.

Each object O is defined by a transition function $\delta_O()$, which takes as input parameter the current state of O and the operation $\mathsf{op}(param)$ applied to O. It returns a pair $\langle r, new\_state\rangle$, where r is the value returned by $\mathsf{op}(param)$, and new_state is the new state of O. Each process p_i maintains a local representation of each object O, denoted $state_i[O]$.

[3] Interestingly, the replacement of the underlying message causal order broadcast by a message total order broadcast, implements linearizability.

```
when p_i invokes O.op(param) do
(1)  result_i ← ?;
(2)  co_broadcast OPERATION(i, O, op(param));
(3)  wait (result_i ≠ ?);
(4)  return (result_i).

when OPERATION(j, O, op(param)) is co-delivered do
(5)  ⟨r, state_i[O]⟩ ← δ_O(state_i[O], op(param));
(6)  if (j = i) then result_i ← r end if.
```

Algorithm 1: Universal construction for causally consistent objects (code for p_i)

When a process p_i invokes an operation op($param$) on an object O, it co-broadcasts the message OPERATION($i, O,$ op($param$)), which is co-delivered to each process (i.e., according to causal message order). Then, p_i waits until this message is locally processed. When this occurs, it returns the result of the operation.

When a process p_i co-delivers a message OPERATION($j, O,$ op($param$)), it updates accordingly its local representation of the object O. If p_i is the invoking process, it additionally locally returns the result of the operation.

5 Conclusion

This short article extended the notion of causal consistency to any object defined by a sequential specification. This definition boils down to causal memory when the objects are read/write registers.

The important point in causal consistency lies in the fact that each process has its own view of the objects, and all these views agree on the partial order on the operations but not necessarily on their results. More explicitly, while each process has a view of each object, which locally satisfies its object specification, two processes may disagree on the value returned by some operations. This seems to be the "process-to-process inconsistency cost" that must be paid when weakening consistency by considering a partial order instead of a total order. On another side and differently from strong consistency conditions, causal consistency copes naturally with partitioning and process crashes.

Acknowledgments

This work has been partially supported by the Franco-German DFG-ANR Project 40300781 DISCMAT (devoted to connections between mathematics and distributed

computing), and the French ANR project DESCARTES (devoted to layered and modular structures in distributed computing).

References

[1] Ahamad M., Neiger G., Burns J.E., Hutto P.W., and Kohli P., Causal memory: definitions, implementation and programming. *Distributed Computing*, 9:37-49 (1995)

[2] Ahamad M., Raynal M. and Thia-Kime G., An adaptive protocol for implementing causally consistent distributed services. *Proc. 18th Int'l Conference on Distributed Computing Systems (ICDCS'98)*, IEEE Press, pp. 86-93 (1998)

[3] Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121-132 (1995)

[4] Attiya H. and Welch J.L., Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91-122 (1994)

[5] Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages (2004)

[6] Birman K.P. and Joseph T.A., Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47-76 (1987)

[7] Cholvi V., Fernández A., Jiménez E., Manzano P., and Raynal M., A methodological construction of an efficient sequentially consistent distributed shared memory. *The Computer Journal*, 53(9):1523-1534 (2010)

[8] Ekström N. and Haridi S., A fault-tolerant sequentially consistent DSM with a compositional correctness proof. *Proc. 4th Int'l Conference on Networked Systems (NETYS'16)*, Springer LNCS 9944, pp. 183-192 (2016)

[9] Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)

[10] Gray J., Notes on database operating systems: an advanced course. *Springer LNCS* 60, pp. 10-17 (1978)

[11] Herlihy M. P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149 (1991)

[12] Herlihy M. P. and Shavit N., *The art of multiprocessor programming*. Morgan Kaufmann, 508 pages, ISBN 978-0-12-370591-4 (2008)

[13] Herlihy M.P. and Wing J.M, Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492 (1990)

[14] Imbs D., Mostéfaoui A., Perrin M. and Raynal M., Set-constrained delivery broadcast: definition, abstraction power, and computability limits. *Proc. 19th Int'l Conference on Distributed Computing and Networking (ICDCN'18)*, ACM Press, Article 7, 10 pages (2018)

[15] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565 (1978)

[16] Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691 (1979)

[17] Lamport L., On inter-process communications, part I: basic formalism. *Distributed Computing*, 1(2): 77-85 (1986)

[18] Misra J., Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153 (1986)

[19] Perrin M., Petrolia M., Mostéfaoui A., and Jard Cl., On composition and implementation of sequential consistency. *Proc. 30th Int'l Symposium on Distributed Computing (DISC'16*, Springer LNCS 9888, pp. 284-297 (2016)

[20] Perrin M., Mostéfaoui A., and Jard Cl., Causal consistency: beyond memory. *Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*, ACM Press, Article 26, 12 pages (2016)

[21] Raynal M., Sequential consistency as lazy linearizability. Brief announcement. *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, ACM press, pp. 151-152 (2002)

[22] Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, ISBN 978-3-642-38222-5 (2013)

[23] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[24] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach.* In print *Springer*, 550 pages, ISBN 978-3-319-94140-0 (2018)

[25] Raynal M. and Schiper A., From causal consistency to sequential consistency in shared memory systems. *Proc. 15th Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, Springer LNCS 1026, pp. 180-194 (1995)

[26] Raynal M., Schiper A. and Toueg S., The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343-350 (1991)

[27] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)

DISTRIBUTED COMPUTING *and* EDUCATION COLUMN
Special Issue

BY

JURAJ HROMKOVIC, STEFAN SCHMID

ETH Zurich, University of Vienna

In this column, Leslie Lamport makes the case for using the language of mathematics for describing algorithms. He argues that students should learn to think mathematically when writing code and programs.

The column is a special issue and collaboration of the distributed computing column and the education column.

Enjoy!

If You're Not Writing a Program, Don't Use a Programming Language

Leslie Lamport
Microsoft Research, Mountain View, CA, U.S.A.

Abstract

The need to handle large programs and to produce efficient compiled code adds complexity to programming languages and limits their expressiveness. Algorithms are not programs, and they can be expressed in a simpler and more expressive language. That language is the one used by almost every branch of science and engineering to precisely describe and reason about the objects they study: the language of mathematics. Math is useful for describing a more general class of algorithms than are studied in algorithm courses.

1 Introduction

I have worked with a number of computer engineers—both hardware and software engineers—and I have seen what they knew and what they didn't know. I have found that most of them do not understand some important basic concepts. These concepts are obscured by programming languages. They are better understood by a simple and powerful way of thinking about computation mathematically that is explained here.

This note discusses what is called *correctness* in the field of program verification: the requirement that each possible execution of a program satisfies some precisely defined property. For brevity, I will use "correctness" only in this sense, ignoring other common meanings of the word. For example, ease of use is not a correctness condition because it isn't precisely defined. That a program produces the right output 99.9% of the time is also not a correctness condition because it isn't an assertion about an individual execution. However, producing the right output *is* a correctness condition. A mathematical understanding of this concept of correctness is useful beyond the field of program verification. It provides a way of thinking that can improve all aspects of writing programs and building systems.

I will consider algorithms, not programs. It's fruitless to try to precisely distinguish between them, but we all have a general idea that an algorithm is a higher-level abstraction that is implemented by a program. For example, here is Euclid's algorithm for computing $GCD(M, N)$, the greatest common divisor of positive integers M and N:

> Let x equal M and y equal N. Repeatedly subtract the smaller of x and y from the larger. Stop when x and y have the same value, at which point that value is $GCD(M, N)$.

Implementing this algorithm in a programming language requires adding details that are not part of the algorithm. For example, should the types of M and N be single-precision integers, double-precision integers, or some class of objects that can represent larger integers? Should the program assume that M and N are positive or should it return an error if they aren't?

The algorithms found in textbooks and studied in algorithm courses are relatively simple ones that are useful in many situations. Most non-trivial programs implement one or more algorithms that are used only in that program. Coding is the task of implementing an algorithm in a programming language. However, programming is too often taken to mean coding, and the algorithm is almost always developed along with the code. The programmer is usually unaware of the existence of the algorithm she is developing. To understand why this is bad, imagine trying to discover Euclid's algorithm by thinking in terms of code rather than in terms of mathematics.

The term *algorithm* is generally taken to mean only an algorithm that might appear in a textbook on algorithms. The special-purpose algorithms implemented by programs and systems are usually called something else, such as high-level designs, specifications, or models. However, they differ from textbook algorithms only in being special purpose and often more complicated. I will call them algorithms to emphasize that they are fundamentally the same as what are conventionally called algorithms.

The benefit of thinking about algorithms mathematically is not limited to obviously mathematical problems like computing the GCD. Here's what I was told in an email from the leader of a team that built a real-time operating system by starting with a high-level design written in a mathematics-based language called TLA$^+$ [14]:

> The [TLA$^+$] abstraction helped a lot in coming to a much cleaner architecture (we witnessed first-hand the brainwashing done by years of C programming). One of the results was that the code size is about 10× less than in [the previous version].

The previous version of the operating system was flown in a spacecraft, where one assumes reducing code size was important. The common obsession with languages might lead readers to think this result was due to some magical features of TLA$^+$. It wasn't. It was due to TLA$^+$ letting users think mathematically.

This note is written for sophisticated readers, but the mathematical approach it presents is simple enough to be understood by undergraduates. The mathematics can be made as rigorous or as informal as we want. The discussion here is informal, using only simple examples. Not explained are how to make the mathematics completely formal and how to handle the large formulas that describe complex real-world algorithms. Also not explained is how to write and reason about those mathematical formulas in practice. That would require a book.

2 Behaviors and Properties

An execution of an algorithm is represented mathematically as a sequence of states, where a state is an assignment of values to variables. A sequence of states is called a *behavior*. For example, the following three-state behavior represents the one possible execution of Euclid's algorithm for $M = 12$ and $N = 18$

$$[x \leftarrow 12,\ y \leftarrow 18]\ ,\ \ [x \leftarrow 12,\ y \leftarrow 6]\ ,\ \ [x \leftarrow 6,\ y \leftarrow 6]$$

where $[x \leftarrow 12,\ y \leftarrow 18]$ is the state that assigns 12 to x and 18 to y. The simplicity and power of this way of representing executions has led me to find it the best one for studying correctness (as defined above).

A *property* is a predicate (Boolean-valued function) on behaviors. We say that a behavior b *satisfies* a property P, or that P *is true on* b, iff (if and only if) $P(b)$ equals TRUE. A correctness condition of an algorithm asserts that every behavior that represents an execution of the algorithm satisfies a property.

Partial correctness of Euclid's algorithm means that if it stops, then x and y both equal $GCD(M, N)$. The algorithm stops iff x and y have the same value. Therefore, partial correctness of the algorithm is expressed by the property that is true of a behavior iff every state of the behavior satisfies this condition:

> If x and y have the same value, then that value equals $GCD(M, N)$.

The state predicate (Boolean-valued function on states) that is true on a state iff the state satisfies this condition can be written as the formula

$$(x = y)\ \Rightarrow\ (x = GCD(M, N)) \tag{1}$$

where $\Rightarrow$ denotes logical implication. The property that is true on a behavior iff (1) is true on every state of the behavior is written as

$$\Box((x = y) \Rightarrow (x = GCD(M, N))) \tag{2}$$

For $M = 12$ and $N = 18$, property (2) is true on this five-state behavior:[1]

$$[x \leftarrow 1,\ y \leftarrow 37]\ ,\ \ [x \leftarrow 6,\ y \leftarrow 6]\ ,\ \ [x \leftarrow 42,\ y \leftarrow 7]\ ,$$
$$[x \leftarrow 6,\ y \leftarrow 6]\ ,\ \ [x \leftarrow 0,\ y \leftarrow 12]$$

Partial correctness of Euclid's algorithm means that property (2) is true of every behavior representing an execution of Euclid's algorithm. Amir Pnueli introduced $\Box$ into computer science as an operator of temporal logic [13], but we can consider it here to be an ordinary mathematical operator that maps state predicates to properties.

Observe that, like almost all mathematicians, I say that formula (2) *is* a property, which is a Boolean-valued function. Pedants and logicians might say that the property is the meaning of the formula, which is different from the formula itself. Like almost all mathematicians, I will ignore this distinction.

A property of the form $\Box I$ for a state predicate I is called an *invariance property*, and we say that $\Box I$ asserts that I is an *invariant*. Invariants play a crucial role in understanding algorithms.

Another important property of Euclid's algorithm is that it always terminates. The algorithm terminates when x equals y. Therefore, an execution that terminates is represented by a behavior containing a state in which the state predicate $x = y$ is true. A behavior contains such a state iff it is not the case that all of its states satisfy $x \neq y$—that is, iff the property $\Box(x \neq y)$ is not true on the behavior, which means that $\neg\Box(x \neq y)$ *is* true on the behavior. Hence a terminating behavior of the algorithm is one satisfying the property $\neg\Box(x \neq y)$.

3 Algorithms are Properties

We usually think of an algorithm as generating possible executions. For example, we can think of Euclid's algorithm generating an execution for each pair of positive integers M and N. (Nondeterminism is a more interesting source of multiple possible executions for an algorithm.) Each possible execution is represented mathematically by a behavior. We can represent an algorithm as the set of all these behaviors.

There is a natural correspondence between sets and predicates. A set S corresponds to the predicate, let's call it π_S, that is defined by letting $\pi_S(e)$ equal TRUE iff e is an element of S. Thus, instead of representing an algorithm by a set S of behaviors, we can represent it by the corresponding predicate π_S on behaviors. A predicate on behaviors is what we call a property, so corresponding to the set S of

[1] Note that a behavior is *any* sequence of states, not just one representing the execution of some algorithm.

behaviors representing executions of the algorithm is the property π_S that is true of a behavior iff the behavior represents an execution of the algorithm.

I will usually represent an algorithm with set S of behaviors as the property π_S because I find that to be the most helpful way of thinking about algorithms. This means that instead of thinking of an algorithm as a generator of executions, we think of it as a rule for determining if a sequence of states is an execution of the algorithm. (In computer science jargon, we think of the algorithm as a recognizer rather than a generator of executions.) This way of thinking about algorithms may strike you as either bizarre or an insignificant shift of viewpoint. However, I have found it to be quite helpful for understanding algorithms.

Algorithms are properties, but not every property is an algorithm. We use the term algorithm for a property that is satisfied by a set of behaviors representing the possible executions of what we think of as an algorithm.

We saw in Section 2 that a correctness condition of an algorithm asserts that every behavior representing an execution of the algorithm satisfies a property P. Since the algorithm is a property A, this correctness condition asserts that if a behavior satisfies A then it satisfies P — an assertion written mathematically as $A \Rightarrow P$.[2] The property P could be an algorithm that is a higher-level (more abstract) version of the algorithm A. In that case, we say that $A \Rightarrow P$ means that A *refines* P. Thus, our mathematical view of computation provides a natural definition of algorithm refinement as implication. Refinement is generalized in Section 6.1 to include data refinement [6].

4 Describing Algorithms Mathematically

4.1 State Machines

The practical way to precisely describe algorithms is with state machines. A state machine is usually described by a set of possible initial states and a next-state relation that determines the possible steps, where a step is a pair of successive states in a behavior. The possible executions of the state machine consist of all sequences $s_1, s_2, \ldots$ of states such that (1) s_1 is a possible initial state and (2) every step (s_i, s_{i+1}) satisfies the next-state relation. A Turing machine is obviously a state machine. An operational semantics of a programming language describes every program in the language as a state machine. Nondeterminism is represented by a next-state relation that is satisfied by more than one pair (s, t) of states for

[2] I am extending the operator $\Rightarrow$ on Boolean values to an operator on Boolean-valued functions. Such extensions are ubiquitous in mathematics. For example, if f and g are numerical-valued functions, then $f + g$ is the function defined by $(f + g)(x) = f(x) + g(x)$.

the same state s. We'll see in Section 4.4 that this definition of a state machine is incomplete, but it will suffice for now.

If a state is an assignment of values to variables, then an execution of a state machine is a behavior. The set of initial states can be represented as a state predicate and the next-state relation can be represented as a predicate on pairs of states. The state machine is represented as a property that is true on a behavior $s_1, s_2, \ldots$ iff these two conditions are satisfied:

SM1 The initial-state predicate is true on s_1.

SM2 The next-state predicate is true on every step (s_i, s_{i+1}).

4.2 State Machines in Mathematics

Let's represent Euclid's algorithm by a state machine. We described the algorithm above for a single pair of input values M and N, so we considered Euclid's algorithm for different values of M and N to be different algorithms. Let's do this in our mathematical representation of the algorithm, so we describe Euclid's algorithm for a single fixed pair M, N of positive integers. The algorithm has a single behavior; what that single behavior is depends on the (unspecified) values of the positive integers M and N.

The initial predicate is true on a state iff x has the value M and y has the value N. This state predicate can obviously be written

$$(x = M) \land (y = N) \tag{3}$$

A predicate on pairs of states is often written as a formula containing primed and unprimed variables, where an unprimed variable represents the value of the variable in the first state and a primed variable represents its value in the second state [5]. With this notation, the next-state predicate for Euclid's algorithm is

$$\begin{array}{lll} & (& (x > y) \\ & & \land\ (x' = x - y) \\ & & \land\ (y' = y) \qquad) \\ \lor & (& (y > x) \\ & & \land\ (y' = y - x) \\ & & \land\ (x' = x) \qquad) \end{array} \tag{4}$$

For example, the predicate (4) is true on the pair of states

$$(\ [x \leftarrow 18,\ y \leftarrow 12]\ ,\ [x \leftarrow 6,\ y \leftarrow 12]\ )$$

because that formula equals TRUE if $x = 18$, $y = 12$, $x' = 6$, and $y' = 12$. Formula (4) equals FALSE if the value of x equals the value of y. Hence, the next-state

predicate of Euclid's algorithm equals FALSE for any pair of states with $x = y$ true in the first state. This means that Euclid's algorithm halts in such a state.

Let's define $Init_E$ to equal the initial-state predicate (3) and $Next_E$ to equal the next-state predicate (4). These two formulas describe Euclid's algorithm—that is, they describe the property that is our mathematical representation of Euclid's algorithm. It's more convenient to describe this property with a single formula, which we now do.

First, define $\flat P$, for any state predicate P, to be the property that is true on a behavior iff P is true on the first state of the behavior. Then $\flat Init_E$ is the property that expresses condition SM1 of the state machine.

Next, extend the operator $\Box$ to predicates Q on pairs of states by letting $\Box Q$ be true on a behavior iff Q is true on every step of the behavior. Thus, $\Box Next_E$ is the property that expresses condition SM2 of the state machine.

The property that is Euclid's algorithm, which is the property satisfying conditions SM1 and SM2, is represented by the formula $\flat Init_E \land \Box Next_E$. For a state predicate P, it's customary to write $\flat P$ as simply P, determining from context whether P means the state predicate or the property $\flat P$. We thus write the property that is Euclid's algorithm as

$$Init_E \land \Box Next_E \tag{5}$$

Instead of defining Euclid's algorithm for a specific pair of integers, we could define it for an arbitrary pair of integers. The definition is the formula obtained by existentially quantifying formula (5) over all positive integers M and N — a formula I will write

$$\exists\, M, N \in \mathbf{Z}^+ \bullet (Init_E \land \Box Next_E)$$

where $\mathbf{Z}^+$ is the set of positive integers. Since M and N don't occur in $Next_E$, this formula can also be written

$$(\exists\, M, N \in \mathbf{Z}^+ \bullet Init_E) \land \Box Next_E$$

However, let's stick with our definition (5) of Euclid's algorithm for a single pair of positive integers M and N.

4.3 Proving Invariance

Let's now prove partial correctness of Euclid's algorithm, which is expressed as the property (2). We first consider the general problem of proving that an algorithm described by the formula $Init \land \Box Next$ satisfies an invariance property $\Box I$. We saw in Section 3 that this means proving the formula $Init \land \Box Next \Rightarrow \Box I$.

Since $\Box I$ asserts that the state predicate I is true on all states of a behavior, the natural way to prove $\Box I$ is by induction: proving that (i) I is true on the first state and (ii) for any j, if I is true on the j^{th} state then it's true on the $(j+1)^{\text{st}}$ state.

We can obviously prove (i) by proving $Init \Rightarrow I$. To prove (ii), it suffices to prove that for any pair (s, t) of states, if I is true on s and $Next$ is true on (s, t), then I is true on t. Let's define I' to be the formula obtained from I by priming all its variables. The formula I' represents the predicate on pairs of states that is true on (s, t) iff I is true on t. We can therefore prove (ii) by proving that I true on s and N true on (s, t) implies that I' is true on (s, t) — an assertion expressed by the formula $I \wedge Next \Rightarrow I'$. We can encapsulate all this in the following proof rule, which asserts that the truth of the two formulas above the line (the hypotheses) implies the truth of the formula below the line (the conclusion).

$$\frac{\begin{array}{l} Init \Rightarrow I \\ I \wedge Next \Rightarrow I' \end{array}}{Init \wedge \Box Next \Rightarrow \Box I} \tag{6}$$

A formula I satisfying the hypotheses of this rule is called an *inductive invariant* of the algorithm $Init \wedge \Box Next$. An inductive invariant is an invariant, but the converse isn't necessarily true. The invariant (1) of Euclid's algorithm is not an inductive invariant of the algorithm—for example, if $M = 12$ and $N = 18$, then the second hypothesis equals FALSE when $x = 16$ and $y = x' = y' = 8$. Define Inv_E to equal formula (1). To prove that Inv_E is an invariant, we find an inductive invariant I that implies it. The invariance of Inv_E then follows from the invariance of I and the following proof rule, which asserts the obvious fact that if a state predicate P implies a state predicate Q, then P true on all states of a behavior implies that Q is true on all states of the behavior.

$$\frac{P \Rightarrow Q}{\Box P \Rightarrow \Box Q} \tag{7}$$

The inductive invariant I for Euclid's algorithm is

$$GCD(x, y) = GCD(M, N)$$

The first hypothesis of rule (6) is trivially true. The truth of the second hypothesis follows from the observation that for any integers a and b, an integer divides both a and b iff it divides both a and $a - b$. That I implies Inv_E is obvious because $GCD(x, x)$ equals x.

As illustrated by Euclid's algorithm, partial correctness is an invariance property. The Floyd/Hoare method of proving partial correctness is based on proof rules (6) and (7), where the inductive invariant is written as a program annotation [3, 7].

What an algorithm does next is determined by its current state, not by what happened in the past. An algorithm does the right thing (for example, it terminates only with the right answer) because something is true of every state—that is, because it satisfies an invariance property. Understanding an algorithm requires understanding that invariant. Years of experience has shown that the one reliable method of proving the invariance of a state predicate is by finding an inductive invariant that implies it.

4.4 Termination

For $M = 12$ and $N = 18$, the formula $Init_E \wedge \Box Next_E$ is satisfied by the following three behaviors:

1. $[x \leftarrow 12,\ y \leftarrow 18]$
2. $[x \leftarrow 12,\ y \leftarrow 18],\ [x \leftarrow 12,\ y \leftarrow 6]$
3. $[x \leftarrow 12,\ y \leftarrow 18],\ [x \leftarrow 12,\ y \leftarrow 6],\ [x \leftarrow 6,\ y \leftarrow 6]$

Those are the three behaviors that satisfy conditions SM1 and SM2. (Behavior 1 trivially satisfies SM2 because it has no steps.) We consider behavior 3 to be the only correct one; we don't want Euclid's algorithm to allow behaviors 1 and 2, which stop prematurely.

A common approach is to modify the definition of the executions of a state machine by adding another condition to SM1 and SM2. Define a predicate N on pairs of states to be *enabled* on a state s iff there is some state t such that N is true on (s, t). The additional condition is:

SM3 The behavior does not end in a state in which the next-state predicate is enabled.

The definition of $\Box Next$ could be modified to imply SM3. While this approach is satisfactory for sequential algorithms that compute an answer and then stop, we'll see in Section 5.1 that it's a bad idea for more general algorithms, including concurrent ones.

Instead, we express the requirement that Euclid's algorithm not stop before it should by adding the requirement of weak fairness of the next-state relation—expressed by the formula WF($Next_E$). For any predicate N on pairs of states, WF(N) is true on a finite behavior iff the behavior doesn't end in a state in which N is enabled. It is true on an infinite behavior iff the behavior doesn't end with an infinite sequence of states such that (i) N is enabled on all states of the sequence and (ii) N is not true on any of that sequence's steps.

Euclid's algorithm terminates iff it eventually reaches a state in which $x = y$ is true, and we saw in Section 2 that this is asserted by the property $\neg\Box(x \neq y)$.

Thus, the assertion that every behavior of Euclid's algorithm stops is expressed mathematically by:

$$Init_E \wedge \Box Next_E \wedge \text{WF}(Next_E) \Rightarrow \neg\Box(x \neq y) \tag{8}$$

To prove (8) we prove two things:

1. $x + y$ is non-negative in all states of a behavior of the algorithm, which is expressed by:

$$Init_E \wedge \Box Next_E \Rightarrow \Box(x + y \geq 0) \tag{9}$$

2. Executing a step of the algorithm from a state with $x \neq y$ decreases $x + y$, which is implied by

$$(x \neq y) \wedge Next_E \Rightarrow (x' + y') < (x + y) \tag{10}$$

We've seen how to prove an invariance property like (9). Formula (10) follows from the fact that $(a - b) + b < a + b$ for any positive numbers a and b.[3] Since a non-negative integer can be decreased only a finite number of times before it becomes non-positive, and $Next_E$ is enabled iff $x \neq y$, (9) and (10) imply (8).

Termination of any sequential algorithm is proved in this way. In general, the formula $x + y$ is replaced by a suitable integer-valued function on states (usually called a *variance* function), and $x \neq y$ is replaced by the state predicate asserting that the next-state predicate is enabled.

5 Concurrent Algorithms

Many computer scientists seem to believe that a concurrent algorithm must be described by a collection of state machines that communicate in some way, each state machine representing a separate process. In fact, a concurrent algorithm is better understood as a single state machine. I will illustrate this with a simple example: the N-buffer producer/consumer algorithm.

5.1 The Two-Process Algorithm

In the standard description of the algorithm, a producer process sends a sequence of messages that are received by a consumer process. Messages are transmitted using an array of N message buffers numbered 0 through $N - 1$, with $N > 0$.

[3] A more rigorous proof requires showing that $(x > 0) \wedge (y > 0)$ is an invariant of Euclid's algorithm.

The producer sends messages by putting them in successive buffers starting with buffer 0, putting the i^{th} message in buffer $i - 1 \bmod N$; and the consumer removes messages in the same order from the buffers. The producer can put a message in a buffer only if that buffer is empty, and the consumer can remove a message only from a buffer that contains one.

The algorithm is described by the formula

$$Init_{PC} \ \wedge \ \Box Next_{PC} \ \wedge \ F_{PC} \tag{11}$$

where $Init_{PC}$ is the initial-state predicate, $Next_{PC}$ is the next-state predicate, and F_{PC} is a fairness property that is false on a behavior that ends before it should. I won't write the complete definitions of these formulas and will just briefly discuss $Next_{PC}$ and F_{PC}.

The obvious definition of $Next_{PC}$ has the form

$$Next_{PC} \ \equiv \ Send \ \vee \ Rcv$$

where $Send$ is true on a pair (s, t) of states iff t represents the state obtained from s by having the producer put its next message into a buffer; and similarly Rcv is true on a pair of states iff that pair represents the consumer removing its next message from a buffer.

There are three obvious choices for the property F_{PC}, leading to three different algorithms. We might want to require that messages keep getting sent and received forever. This is expressed by letting F_{PC} equal WF($Next_{PC}$). We can write the same algorithm by letting F_{PC} equal WF($Send$) $\wedge$ WF(Rcv). These two formulas are not equivalent, but they produce equivalent formulas (11) because this formula is true:

$$Init_{PC} \ \wedge \ \Box Next_{PC} \ \Rightarrow \ (\text{WF}(Next_{PC}) \ \equiv \ \text{WF}(Send) \wedge \text{WF}(Rcv))$$

Of course, we can't prove this formula without knowing the definitions of $Init_{PC}$, $Send$, and Rcv. However, it's possible to see from the definition of WF in Section 4.4 that it should be true because the informal description of the algorithm means $Init_{PC} \wedge \Box Next_{PC}$ should imply of a behavior that:

- $Send$ or Rcv is enabled in every state. (They can both be enabled.)
- The behavior can contain at most N consecutive steps that represent the sending of a message, and at most N consecutive steps that represent the receiving of a message.

The second obvious choice for F_{PC} requires that the producer should keep trying to send messages, but the consumer may stop receiving them. This is expressed by letting F_{PC} equal WF($Send$). Formula (11) then allows (but doesn't require)

a behavior to end only in a state in which all message buffers are full. The third obvious choice is to let F_{PC} equal WF(Rcv), allowing behaviors to end only in a state in which all the buffers are empty. Note that modifying the definition of $\Box Next_{PC}$ by adding condition SM3 of Section 4.4 would make it impossible to write the last two versions of the algorithm.

5.2 Another View of the Algorithm

Because producer/consumer is a two-process algorithm, it's natural to write $Next_{PC}$ as the disjunction $Send \lor Rcv$ of two formulas, each describing the steps that can be taken by one of the processes. Let's take a closer look at those formulas. Sensible definitions of $Send$ and Rcv will have the form

$$\begin{aligned} Send &\equiv \exists\, i \in \{0, \ldots, N-1\} \bullet S(i) \\ Rcv &\equiv \exists\, i \in \{0, \ldots, N-1\} \bullet R(i) \end{aligned}$$

where $S(i)$ describes a step in which the producer puts a message in buffer i, and $R(i)$ describes a step in which the consumer removes a message from buffer i. Define $SR(i)$ to equal $S(i) \lor R(i)$. Elementary logic shows that $Next_{PC}$ equals

$$\exists\, i \in \{0, \ldots, N-1\} \bullet SR(i) \tag{12}$$

Formula (12) looks like the next-state predicate of an N-process algorithm, where the processes are numbered 0 through $N-1$ and $SR(i)$ describes steps that can be performed by process number i. Process i can put a message into buffer i when it's empty, and it removes a message from that buffer when it contains one.

Is the producer/consumer algorithm a 2-process algorithm or an N-process algorithm? Mathematically, it's neither. The algorithm is the formula/property (11). We can view that formula as a 2-process algorithm or an N-process algorithm. Each view gives us a different way of thinking about the algorithm, enabling us to understand it better than we could with just one view.[4]

Formula (4), the next-state predicate of Euclid's algorithm, is also the disjunction of two formulas. This means we can view Euclid's algorithm not just as a uniprocess algorithm, but also as a 2-process algorithm. One process tries to subtract y from x, which it can do only when $y < x$ is true; the other process tries to subtract x from y, which it can do only when $x < y$ is true.

[4] I've ignored the fairness property, so you may be tempted to think that F_{PC} will tell us how many processes there are in the algorithm. It won't. For example, the algorithm we get by letting F_{PC} equal WF($Send$) is also obtained by letting it equal

$$\forall\, i \in \{0, \ldots, N-1\} \bullet \text{WF}(S(i))$$

This formula looks like the conjunction of fairness conditions on N separate processes.

All programming languages that I know of force you to think of the producer/consumer algorithm as either a 2-process algorithm or an N-process algorithm. Few algorithms are better understood by decomposing them into processes in different ways. But describing an algorithm with a programming language can limit our ability to understand it in other ways as well.

5.3 Safety and Liveness

What does correctness of the producer/consumer algorithm mean? Since the algorithm need not terminate, it doesn't satisfy any partial correctness condition or termination requirement. There's an endless variety of correctness properties that we might want to be satisfied by algorithms that need not terminate. Here are two that we might require of the version of the algorithm with F_{PC} equal to WF(Rcv).

PC1 The sequence of messages received by the consumer is a prefix of the sequence of messages sent by the producer.

PC2 Every message sent is eventually received.

PC1 is a *safety* property, which intuitively is a property asserting what *may* happen. PC2 is a *liveness* property, which intuitively is a property asserting what *must* happen. Here are the precise definitions. (Remember that every sequence of states is a behavior.)

- A safety property is one that is false on an infinite behavior iff it is false on some finite prefix of the behavior.

- A liveness property is one which, for any finite behavior, is true for some (possibly infinite) extension of that behavior.

It can be shown that any property is equivalent to the conjunction of a safety property and a liveness property. For any predicate I on states and predicate A on pairs of states, the formula $I \wedge \Box A$ is a safety property, and WF(A) is a liveness property.

The producer/consumer algorithm satisfies property PC1 iff $Init_{PC} \wedge \Box Next_{PC}$ implies that property. The fairness condition F_{PC} is irrelevant for safety properties. In general, for any algorithm $Init \wedge \Box Next \wedge F$, if F is the conjunction of formulas WF(A) and each of the predicates A implies $Next$, then the algorithm satisfies a safety property iff $Init \wedge \Box Next$ satisfies the safety property.[5]

[5]This is why we conjoin fairness conditions rather than other kinds of liveness conditions to $Init \wedge \Box Next$. If F were not of this form, the formula $Init \wedge \Box Next \wedge F$ would be hard to understand because the liveness property F could forbid steps allowed by $Next$.

5.3.1 Proving Safety Properties

An invariance property $\Box I$ is a safety property, and we've seen how to prove them. Stating PC1 as an invariance property would require being able to express the sequences of all messages that have been sent and received in terms of the formula's variables. Whether this is possible depends on the definitions of $Init_{PC}$, $Send$, and Rcv. If it's not possible, there are two ways to proceed.

The simplest approach is to add a history variable to the producer/consumer algorithm that records the sequences of messages sent and received, so PC1 can be expressed as an invariance property. A history variable is a simple kind of auxiliary variable—a variable that is added to an algorithm to produce a new algorithm that is the same as the original one if the value of the added variable is ignored [1]. (Auxiliary variables have other uses that I won't discuss.) In general, any safety property can be expressed as an invariance property by adding a history variable. However, this is often not a good approach because it encodes in an invariance proof the kind of unreliable behavioral reasoning that invariance proofs were developed to replace.

The second approach is to write a higher-level algorithm that obviously implies PC1, and then prove that the producer/consumer algorithm refines that algorithm. The higher-level algorithm will of course have the form $Init \wedge \Box Next$; the proof will use an invariance property $\Box Inv_{PC}$ of the producer/consumer algorithm and the following proof rule.

$$\frac{\begin{array}{l} Init_1 \;\Rightarrow\; Init_2 \\ Next_1 \wedge Inv_1 \wedge Inv_1' \;\Rightarrow\; Next_2 \end{array}}{Init_1 \wedge \Box Next_1 \wedge \Box Inv_1 \;\Rightarrow\; Init_2 \wedge \Box Next_2}$$

5.3.2 Proving Liveness Properties

Proving liveness properties like PC2 requires more than the simple counting-down argument used to prove termination. There is no single recipe for proving all liveness properties. Rigorous proofs are best done by generalizing $\Box$ to an operator on properties, where $\Box P$ asserts of a behavior that property P is true on all suffixes of that behavior. For example, $\Box\neg\Box(x \neq y)$ is true on an infinite behavior iff $x = y$ is true on infinitely many of its states. Weak fairness can be expressed with $\Box$, as can another important type of liveness property that I won't discuss called strong fairness [4].

At the heart of most liveness proofs are counting down arguments. The counting down argument used to prove termination of Euclid's algorithm is based on the fact that there is no infinite descending sequence $n_1 > n_2 > \ldots$ of natural numbers. Proving liveness properties of concurrent algorithms requires a generalization to counting down on a well-ordered set, which is a set S with partial

order $\succ$ containing no infinite descending sequence $s_1 \succ s_2 \succ \ldots$ of elements. For example, the set of all k-tuples of natural numbers, with the lexicographical ordering, is well ordered.

We can regard $\Box$ as an ordinary mathematical operator on properties. For completely formal reasoning, I find it better to use temporal logic and to regard $\Box$ as a temporal operator. The proofs of liveness one writes in practice can be formalized with a small number of temporal-logic axioms [9]. However, temporal logic is a modal logic and does not obey some important laws of traditional math, so it must be used with care. An informal approach that avoids temporal logic may be best for undergraduates.

6 Refinement

6.1 Data Refinement

Data refinement is a traditional method of refining sequential algorithms that compute an output as a function of their input [6]. An example of data refinement is refining Euclid's algorithm by representing the natural numbers x and y with bit arrays ax and ay of length k.

Mathematically, a k-bit array a is a function from $\{0, \ldots, k-1\}$ to $\{0, 1\}$. It represents the natural number $AtoN(a)$, defined by

$$AtoN(a) \equiv \sum_{i=0}^{k-1} a(i) \cdot 2^i$$

Let's write such a bit array a as $a(k-1) \ldots a(0)$, so $AtoN(01100)$ equals 12 for $k = 5$.

Let Alg_E be the formula $Init_E \wedge \Box Next_E \wedge WF(Next_E)$ that is Euclid's algorithm, and let Alg_{AE} be an algorithm whose variables are 5-bit arrays ax and ay. For any state s that assigns values to ax and ay, let $AEtoE(s)$ be the state that assigns the value $AtoN(ax)$ to x and $AtoN(ay)$ to y. For example,

$$AEtoE([ax \leftarrow 01100,\ ay \leftarrow 10010]) \ = \ [x \leftarrow 12,\ y \leftarrow 18]$$

We extend $AEtoE$ to behaviors by defining

$$AEtoE(s_1, s_2, \ldots) \ \equiv \ AEtoE(s_1),\ AEtoE(s_2),\ \ldots$$

For example, if b is the behavior

$$[ax \leftarrow 01100,\ ay \leftarrow 10010],\ [ax \leftarrow 01100,\ ay \leftarrow 00110],\ [ax \leftarrow 00110,\ ay \leftarrow 00110] \tag{13}$$

then $AEtoE(b)$ equals

$$[x \leftarrow 12,\ y \leftarrow 18],\ [x \leftarrow 12,\ y \leftarrow 6],\ [x \leftarrow 6,\ y \leftarrow 6] \tag{14}$$

We say that algorithm Alg_{AE} *refines* algorithm Alg_E under the *refinement mapping* $x \leftarrow AtoN(ax)$, $y \leftarrow AtoN(ay)$ iff, for every behavior b allowed by algorithm Alg_{AE}, the behavior $AEtoE(b)$ is allowed by algorithm Alg_E. For $M = 12$ and $N = 18$, algorithm Alg_E allows only the single behavior (14), so if Alg_{AE} refines Alg_E, it will allow only the single behavior (13).

The value of the state predicate $x < 2 \cdot y$ on a state $AEtoE(s)$ equals the value of the predicate $AtoN(ax) < 2 \cdot AtoN(bx)$ on the state s. For example, both the value of $x < 2 \cdot y$ on the state

$$[x \leftarrow AtoN(10010),\ y \leftarrow AtoN(01100)]$$

and the value of $AtoN(ax) < 2 \cdot AtoN(bx)$ on the state

$$[ax \leftarrow 10010,\ ay \leftarrow 01100]$$

equal $18 < 2 \cdot 12$ (which equals TRUE). In general, the value of any state predicate I on $AEtoE(s)$ is the same as the value on s of the formula obtained by substituting $AtoN(ax)$ for x and $AtoN(ay)$ for y in I. Mathematicians have no standard notation for such a formula obtained by substitutions; I'll write it

$$I \text{ WITH } x \leftarrow AtoN(ax),\ y \leftarrow AtoN(ay) \tag{15}$$

So, the value of I on $AEtoE(s)$ equals the value of formula (15) on s. Similarly, the value of the algorithm/formula Alg_E on a behavior $AEtoE(b)$ is the same as the value of

$$Alg_E \text{ WITH } x \leftarrow AtoN(ax),\ y \leftarrow AtoN(ay) \tag{16}$$

on the behavior b. We said above that Alg_{AE} refines Alg_E under the refinement mapping described by this WITH clause iff, for any behavior b satisfying Alg_{AE}, the behavior $AEtoE(b)$ satisfies Alg_E. We've just seen that the latter condition is equivalent to b satisfying (16). Therefore, Alg_{AE} refines Alg_E under this refinement mapping iff Alg_{AE} implies (16). In general, we have:

> **Definition** Algorithm Alg_1 implements algorithm Alg_2 under the refinement mapping $v_1 \leftarrow e_1, \ldots\ v_n \leftarrow e_n$ iff this formula is true:
>
> $$Alg_1 \ \Rightarrow\ (Alg_2 \text{ WITH } v_1 \leftarrow e_1,\ \ldots,\ v_n \leftarrow e_n)$$

"The brainwashing done by years of C programming" may lead one to think that there is little difference between the expression $x' = x - y$ in the next-state relation (4) of Euclid's algorithm and the C assignment statement `x = x-y`. However, expanding the definition of Alg_E shows that formula (16) is an algorithm whose next-state predicate contains the expression

$$(x' = x - y) \text{ WITH } x \leftarrow AtoN(ax),\ y \leftarrow AtoN(ay)$$

which equals

$$\sum_{i=0}^{k-1} ax'(i) \cdot 2^i \ = \ \sum_{i=0}^{k-1} ax(i) \cdot 2^i \ - \ \sum_{i=0}^{k-1} ay(i) \cdot 2^i$$

No programming language allows you to write anything resembling this formula.

Data refinement is described by substitution, which is a fundamental operation of mathematics. It cannot be properly understood in terms of the limited kinds of substitution provided by programming languages.

6.2 Step Refinement

Another kind of refinement is step refinement, in which a single step of a high-level algorithm is refined by multiple steps of a lower-level algorithm. Let's consider a very simple example.

Suppose we want to write a formula/algorithm/property representing a clock that displays the hour and minute, ignoring the relation between the display and physical time. We could write a formula Alg_{HM} containing the variables hr and min that represent the hour and minute displays. A behavior satisfying Alg_{HM} would contain this subsequence of three states:

$$[hr \leftarrow 4,\ min \leftarrow 58]\,,\ \ [hr \leftarrow 4,\ min \leftarrow 59]\,,\ \ [hr \leftarrow 5,\ min \leftarrow 0]$$

Suppose we also write a formula Alg_{HMS} describing a clock that represents the hour, minute, and second displays with variables hr, min, and sec.

If we ask for a clock that displays hours and minutes, without explicitly saying that it does not display seconds, then our request is satisfied by a clock displaying hours, minutes, and seconds. In mathematics, writing a formula like Alg_{HM} containing the variables hr and min doesn't imply that there is no variable sec. The formula simply says nothing about sec or any other variable besides hr and min. Therefore, the formula/property Alg_{HM} should be satisfied by the algorithm/property Alg_{HMS}. In other words, every behavior satisfying Alg_{HMS} should also satisfy Alg_{HM}. However, the way I've been writing our algorithms, a behavior satisfying Alg_{HMS} takes 60 steps to go from a state with $hr = 4$ and $min = 59$

to one with $hr = 5$ and $min = 0$, while a behavior satisfying Alg_{HM} does it in a single step. Therefore, I haven't been writing algorithms the way they should be written. Writing them properly requires a closer look at how mathematics is used to describe the world.

I defined a state to be an assignment of values to variables, and in the examples I've taken the variables to be the ones in the algorithm. Since writing the formula Alg_{HM} doesn't preclude the existence of variables other than hr and min, for what we are doing to make sense mathematically, a state should be an assignment of values to all possible variables. (Mathematicians assume that there are an infinite number of possible variables.) The formula Alg_{HM} is not an assertion about a universe consisting only of an hour-minute clock described by the variables hr and min. It's an assertion about a universe containing an hour-minute clock—a universe that might also contain Euclid's algorithm and the producer/consumer algorithm. A behavior represents a possible "execution" of this entire universe. The behavior satisfies formula Alg_{HM} iff it represents a universe in which the hour-minute clock is operating correctly. If Alg_{PC} is a specification of the producer/consumer algorithm, then a behavior satisfies $Alg_{HM} \wedge Alg_{PC}$ iff it represents a universe in which both the hour-minute clock and the producer/consumer algorithm are operating correctly.

It's obviously absurd for a specification of the hour-minute clock to require that, in a state with $hr = 4$ and $min = 59$, the next state of the entire universe must be one with $hr = 5$ and $min = 0$. It should allow multiple successive states with $hr = 4$ and $min = 59$ to precede a state with $hr = 5$ and $min = 0$ — perhaps trillions of them. This means that the next-state predicate for the hour-minute clock should have the form

$$Tick_{HM} \vee (hr' = hr \wedge min' = min) \tag{17}$$

where $Tick_{HM}$ describes how hr and min can change. Steps that leave hr and min unchanged (allowed by the second disjunct) are called *stuttering steps* of the algorithm. Steps allowed by Alg_{HMS} that change only sec are stuttering steps of Alg_{HM}, allowed by the next-state predicate (17). Therefore, Alg_{HMS} will imply Alg_{HM}.

All the formulas representing algorithms that we've written need to be modified to allow stuttering steps. Let's write formula (17) as $[Tick]_{\langle hr,min\rangle}$. We can then change the safety part (5) of Euclid's algorithm to $Init_E \wedge \Box[Next_E]_{\langle x,y\rangle}$ and the safety part of the producer/consumer algorithm (11) to $Init_{PC} \wedge \Box[Next_{PC}]_{\langle \ldots \rangle}$, where "..." is the list of all the algorithm's variables.

The next-state predicate (17) allows behaviors that, from some point on, contain only stuttering steps of the clock. Such a behavior represents one in which the clock stops. Since the entire universe need never stop, termination of any algorithm is represented by infinite stuttering. We can therefore simplify the math-

ematics by considering only infinite behaviors. Termination is still disallowed by fairness properties. The fairness condition WF($Tick_{HM}$) asserts that the hour-minute clock never stops, assuming that $Tick_{HM}$ does not allow steps that leave both hr and min unchanged.

In general, stuttering steps allow step refinement in which one step of a higher-level version of an algorithm is implemented by multiple steps of a lower-level version. One of those lower-level steps allows the higher-level step; the rest allow stuttering steps of the higher-level algorithm.

6.3 Proving Correctness by Refinement

As we have seen in Section 5.3.1, a correctness property of an algorithm is often best expressed as a higher-level algorithm. Proving correctness then means proving that the original algorithm refines the higher-level one. This usually involves both data refinement and step refinement. For example, an algorithm that refines Euclid's algorithm by representing integers with bit strings might refine a step of Euclid's algorithm with a sequence of steps that read or modify only a single bit at a time. The refinement mapping must be defined so that only one of those steps refines a step of Euclid's algorithm that modifies x or y. The rest must refine stuttering steps.

I expect that this kind of refinement sounds like magic to most readers, who won't believe that it can work in practice. Seeing that it is a straightforward, natural way to reason about algorithms requires working out examples, which I will not attempt here. I will simply report that among the refinement proofs I have written is a machine-checked correctness proof [11] of the consensus algorithm at the heart of a subtle fault-tolerant distributed algorithm by Castro and Liskov [2] that uses $3F + 1$ processes, up to F of which may be malicious (Byzantine). The proof shows that the Castro-Liskov consensus algorithm refines a version of the $2F + 1$ process Paxos consensus algorithm that tolerates F benignly faulty processes [10]. Steps of malicious processes, as well as many steps taken by the good processes to prevent malicious ones from causing an incorrect execution of Paxos, refine stuttering steps of the Paxos algorithm. I found that viewing the Castro-Liskov algorithm as a refinement of Paxos was the best way to understand it.

7 Conclusion

Algorithms are usually described with programming languages or languages based on programming-language concepts. The mathematical approach presented here can be viewed as describing algorithms semantically. It may seem impractical to

people used to thinking in terms of programming languages, whose semantics are so complicated. But programming languages are complicated because programs can be very large and must be executed efficiently. Algorithms are much smaller than programs, and they don't have to be executed efficiently.[6] This makes it practical to describe them in the much simpler and infinitely more expressive language of mathematics.

The informal mathematics I have used has not been rigorous. For example, $GCD(x, y) = GCD(M, N)$ is not really an inductive invariant of Euclid's algorithm. To make it inductive, we must conjoin the assertion that x and y are integers. A completely rigorous exposition might be inappropriate for undergraduates. However, their professors should understand how to reason rigorously about algorithms.

A simple formal basis for mathematics, developed about a century ago and commonly accepted by mathematicians today, is first-order logic and (untyped) set theory. To my knowledge, this is an adequate formalization of the mathematics used by scientists and engineers. (It has been found inadequate for formalizing the long, complicated proofs mathematicians can write.) Many computer scientists feel that types are necessary for rigor. Besides adding unnecessary complexity, types can introduce problems for mathematical reasoning that become evident only when one tries to provide a formal semantics for the language being used—something textbook writers rarely do.

The ideas put forth here are embodied in the TLA$^+$ specification language [8] mentioned in the introduction. TLA$^+$ is a formal language with tools that include a model checker and a proof checker. It was designed for describing concurrent algorithms, including high-level designs of distributed systems. Any attempt to formalize mathematics in a practical language requires choices of notation and underlying formalism that will not please everyone. Moreover, languages and tools that are better than TLA$^+$ for other application domains should be possible. But TLA$^+$ demonstrates that the approach described here is useful in engineering practice [12].

Today, programming is generally equated with coding. It's hard to convince students who want to write code that they should learn to think mathematically, above the code level, about what they're doing. Perhaps the following observation will give them pause. It's quite likely that during their lifetime, machine learning will completely change the nature of programming. The programming languages they are now using will seem as quaint as Cobol, and the coding skills they are learning will be of little use. But mathematics will remain the queen of science, and the ability to think mathematically will always be useful.

[6] Tools for checking an algorithm may have to execute it, but the execution need not be as efficient as that of a program implementing the algorithm.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. ACM, 1999.

[3] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.

[4] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.

[5] Eric C. R. Hehner. Predicative programming. *Communications of the ACM*, 27(2):134–151, February 1984.

[6] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[7] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[8] Leslie Lamport. TLA—temporal logic of actions. A web page, a link to which can be found at URL `http://lamport.org`. The page can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

[9] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[10] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[11] Leslie Lamport. Byzantizing paxos. In David Peleg, editor, *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer-Verlag, 2011.

[12] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, April 2015.

[13] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

[14] Eric Verhulst, Raymond T. Boute, José Miguel Sampaio Faria, Bernard H. C. Sputh, and Vitaliy Mezhuyev. *Formal Development of a Network-Centric RTOS*. Springer, New York, 2011.