Computation-Aware Data Aggregation

Bernhard Haeupler[*], D Ellis Hershkowitz[*], Anson Kahng[†], Ariel D. Procaccia[†]

Computer Science Department
Carnegie Mellon University
{haeupler,dhershko,akahng,arielpro}@cs.cmu.edu

Abstract

AllReduce is a fundamental primitive in distributed computing wherein a network computes a function of every nodes' input. However, while compute time is non-negligible in modern systems, standard models of distributed computing do not take compute time into account. Rather, most distributed models of computation only explicitly consider communication time.

In this paper, we introduce a model of distributed computation that considers *both* computation and communication so as to give a theoretical treatment of AllReduce. We study both the structure of and how to compute the fastest AllReduce schedule in this model. As our first result, we give a polynomial-time algorithm that computes the optimal schedule when the input network is a complete graph. Moreover, since one may want to run AllReduce over a pre-existing network, we also study AllReduce scheduling on arbitrary graphs. We demonstrate that this problem on arbitrary graphs is hard to approximate within a multiplicative 1.5 factor. Finally, we give an $O(\log n \cdot \log \frac{\text{OPT}}{t_m})$-approximation algorithm for this problem on arbitrary graphs, where n is the number of nodes and OPT is the length of the optimal schedule.

1 Introduction

Distributed systems drive much of the modern computing revolution. However, these systems are only as powerful as the abstractions which enable programmers to make use of them. A key such abstraction is AllReduce [Rab04, Gra03]. In AllReduce a programmer specifies some function they would like to apply to every private input of a node in a network; after AllReduce is run a node learns the result of this function applied to every nodes' input.[1] For example, if every node stored an integer value, a programmer could run an AllReduce to broadcast the sum or the largest value of every node in the network. The utility of modern systems is their ability to perform massive computations and so, often, applications of AllReduce consist of a function

[*] Supported in part by NSF grants CCF-1527110, CCF-1618280, CCF-1814603, CCF-1910588, NSF CAREER award CCF-1750808 and a Sloan Research Fellowship.

[†] Supported in part by NSF grants IIS-1350598, IIS-1714140, CCF-1525932, and CCF-1733556; by ONR grants N00014-16-1-3075 and N00014-17-1-2428; and by a Sloan Research Fellowship and a Guggenheim Fellowship.

[1] In practice in AllReduce, after a node learns this value, the value is then broadcast to all nodes throughout the network. Because this broadcast step only involves communication and because once one node has learned this value, it is well-understood how to spread said value in the network [Rav94, KP95, IRRS15], we focus just on the "compute" step. For ease of exposition we refer to this step as "AllReduce".

which is computationally-intensive to compute. A rigorous theoretical study of AllReduce, then, must take the cost of computation into account. At the same time, one cannot omit the cost of communication, as many applications of AllReduce operate on large datasets which take time to transmit over a network.

However, to our knowledge, all existing models of distributed computation—e.g., the CONGEST [Pel00], SINR [AD09], (noisy) radio network [KM98, CK85, CHHHZ17, CHHHZ18], congested clique [DKO14], dual graph [CHGK$^+$14], store-and-forward [LMR94, Rot13], LOCAL [Lin92], and telephone broadcast models [Rav94, KP95, IRRS15]—all only consider the cost of communication. Relatedly, while there has been significant applied research on communication-efficient AllReduce algorithms, there has been relatively little work that explicitly considers the cost of computation, and even less work that considers how to design a network to efficiently perform AllReduce [OKF14, KOF15, PY07, PY09, JLK12]. In this way, there do not seem to exist theoretical results for efficient AllReduce scheduling algorithms that consider both the cost of communication and computation.

Thus, we aim to provide answers to two theoretical questions in settings where both computation and communication are non-negligible:

1. *How should one structure a network to efficiently perform AllReduce?*
2. *How can one coordinate a fixed network to efficiently perform AllReduce?*

1.1 Our Model and Problem

The Token Network Model. So as to give formal answers to these questions we introduce the following simple distributed model, the Token Network Model. A Token Network is given by an undirected graph $G = (V, E)$, $|V| = n$, with parameters $t_c, t_m \in \mathbb{N}$ which describe the time it takes nodes to do computation and communication, respectively.[2]

Time proceeds in synchronous rounds during which nodes can compute on or communicate atomic tokens. Specifically, in any given round a node is busy or not busy. If a node is not busy and has at least one token it can *communicate*: any node that does so is busy for the next t_m rounds, at the end of which it passes one of its tokens to a neighbor in G. If a node is not busy and has two or more tokens, it can *compute*: any node that does so is busy for the next t_c rounds, at the end of which it combines (a.k.a. aggregates) two of its tokens into a single new token.[3] At a high level, this means that communication takes t_m rounds and computation takes t_c rounds.

The Token Computation Problem. We use our Token Network model to give a formal treatment of AllReduce scheduling. In particular, we study the Token Computation problem. Given an input Token Network, an algorithm for the Token Computation problem must output a schedule S which directs each node when to compute and when and with whom to communicate. A schedule is valid if after the schedule is run on the input Token Network where every node begins with a single token, there is one remaining token in the entire network; i.e., there is one node that has aggregated all the information in the network. We use $|S|$ to notate the length of S — i.e., the number of rounds S takes — and measure the quality of an algorithm by the length of the schedule that it outputs. For completeness, we give a more technical and formal definition in Appendix A.

[2] We assume $t_c, t_m = \text{poly}(n)$ throughout this paper.

[3] Throughout this paper, we assume for ease of exposition that the smaller of t_c and t_m evenly divides the larger of t_c and t_m, or equivalently that either t_c or t_m is 1.

Discussion of Modeling Choices. Our Token Network model and the Token Computation problem are designed to formally capture the challenges of scheduling distributed computations where both computation and communication are at play. In particular, combining tokens can be understood as applying some commutative, associative function to the private input of all nodes in a network. For instance, summing up private inputs, taking a minimum of private inputs, or computing the intersection of input sets can all be cast as instances of the Token Computation problem. We assume that the computation time is the same for every operation and that the output of a computation is the same size as each of the inputs as a simplifying assumption. We allow nodes to receive information from multiple neighbors as this sort of communication is possible in practice.

Lastly, our model should be seen as a so-called "broadcast" model [KM98] of communication. In particular, it is easy to see that our assumption that a node can send its token to only a single neighbor rather than multiple copies of its token to multiple neighbors is without loss of generality: One can easily modify a schedule in which nodes send multiple copies to one of equal length in which a node only ever sends one token per round. An interesting followup question could be to consider our problem in a non-broadcast setting.

1.2 Our Results

We now give a high-level description of our technical results.

Optimal Algorithm on Complete Graphs (Section 3). We begin by considering how to construct the optimal AllReduce schedule in the Token Network model for complete graphs for given values of t_c and t_m. The principal challenge in constructing such a schedule is formalizing how to optimally pipeline computation and communication and showing that any valid schedule needs at least as many rounds as one's constructed schedule. We overcome this challenge by showing how to modify a given optimal schedule into an efficiently computable one in a way that preserves its pipelining structure. Specifically, we show that one can always modify a valid optimal schedule into another valid optimal schedule with a well-behaved recursive form. We show that this well-behaved schedule can be computed in polynomial time. Stronger yet, we show that the edges over which communication takes place in this schedule induce a tree which generalizes Fibonacci trees [Hor83]. It is important to emphasize that this result has implications beyond producing the optimal schedule for a complete graph; it shows one optimal way to *construct* a network for AllReduce (if one had the freedom to include any edge), thereby suggesting an answer to the first of our two research questions.

Hardness and Approximation on Arbitrary Graphs (Section 4). We next consider the hardness of producing good schedules efficiently for arbitrary graphs and given values of t_c and t_m. We first show that no polynomial-time algorithm can produce a schedule of length within a multiplicative 1.5 factor of the optimal schedule unless P = NP. This result implies that one can only *coordinate* AllReduce over a pre-existing network so well.

Given that an approximation algorithm is the best one can hope for, we next give an algorithm which in polynomial time produces an approximately-optimal Token Computation schedule. Our algorithm is based on the simple observation that after $O(\log n)$ repetitions of pairing off nodes with tokens, having one node in each pair route a token to the other node in the pair, and then having every node compute, there will be a single token in the network. The difficulty in this approach lies in showing that one can route pairs of tokens in a way that is competitive with the length of the optimal schedule. We show that by considering the paths in G traced out by tokens sent by the optimal schedule, we can get a concrete hold on the optimal schedule. Specifically, we show that a polynomial-time algorithm based on our observation produces a valid schedule of length $O(\mathrm{OPT} \cdot \log n \cdot \log \frac{\mathrm{OPT}}{t_m})$ with high probability,[4] where OPT is the length of the optimal schedule. Using an easy bound on OPT, this can be roughly interpreted as an $O(\log^2 n)$-approximation algorithm. This result shows that AllReduce over a pre-existing network can be *coordinated* fairly well.

Furthermore, it is not hard to see that when $t_c = 0$ and $t_m > 0$, or when $t_c > 0$ and $t_m = 0$, our problem is trivially solvable in polynomial time. However, we show hardness for the case where $t_c, t_m > 0$, which gives a formal sense in which computation and communication cannot be considered in isolation, as assumed in previous models of distributed computation.

1.3 Terminology

For the remainder of this paper we use the following terminology. A token a *contains* token a' if $a = a'$ or a was created by combining two tokens, one of which contains a'. For shorthand we write $a' \in a$ to mean that a contains a'. A *singleton* token is a token that only contains itself; i.e., it is a token with which a node started. We let a_v be the singleton token with which vertex v starts and refer to a_v as v's singleton token. The *size* of a token is the number of singleton tokens it contains. Finally, let a_f be the last token of a valid schedule S; the *terminus* of S is the node at which a_f is formed by a computation.

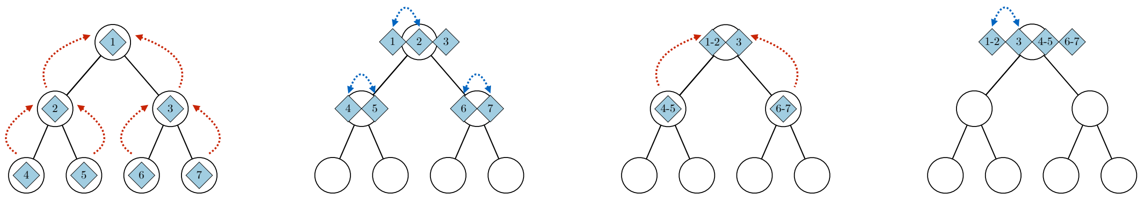
2 Related Work

The previous theoretical work which most closely resembles our own is a line of work in centralized algorithms for scheduling information dissemination [Rav94, KP95, IRRS15]. In this problem, an algorithm is given a graph and a model of distributed communication, and must output a schedule that instructs nodes how to communicate in order to spread some information. For instance, in one setting an algorithm must produce a schedule which, when run, broadcasts a message from one node to all other nodes in the graph. The fact that these problems consider spreading information is complementary to the way in which we consider consolidating it. However, we note that computation plays no role in these problems, in contrast to our Token Computation problem.

Of these prior models of communication, the model which is most similar to our own is the telephone broadcast model. In this model in each round a node can "call" another node to transmit information or receive a call from a single neighbor. Previous results have given a hardness of approximation of 3 [EK05] for broadcasting in this model and logarithmic as well as sublogarithmic approximation algorithms for broadcasting [EK06]. The two notable differences between this model and our own are (1) in our model nodes can receive information from multiple neighbors in a single round[5] and (2) again, in our model computation takes a non-negligible amount of time. Note, then, that even in the special case when $t_c = 0$, our model does not generalize the telephone broadcast model; as such we do not immediately inherit prior hardness results from the telephone broadcast

[4] Meaning at least $1 - 1/\mathrm{poly}(n)$ henceforth.

[5] See above for the justification of this assumption.



(a) Round 1 (b) Round 2 (c) Round 3 (d) Round 4

Figure 1: The naïve aggregation schedule on a binary tree for $t_c = t_m = 1$ and $n = 7$ after 4 rounds. tokens are represented by blue diamonds; a red arrow from node u to node v means that u sends to v; and a double-ended blue arrow between two tokens a and b means that a and b are combined at the node. Notice that the root gains an extra token every 2 rounds.

problem. Furthermore, (1) and especially (2) preclude the possibility of an easy reduction from our problem to the telephone broadcast problem.

There is also a great deal of related applied work; additional details are in Appendix B.

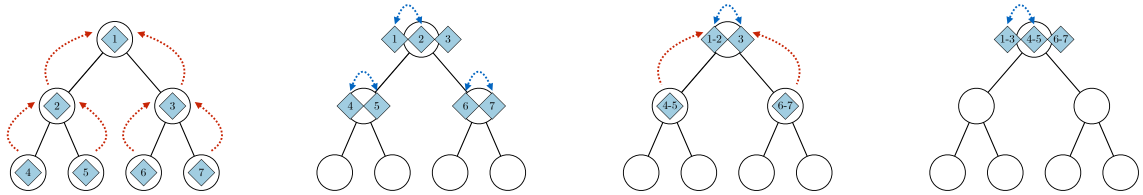
3 Optimal Algorithm for Complete Graphs

In this section we provide an optimal polynomial-time algorithm for the Token Computation problem on a complete graph. The schedule output by our algorithm ultimately only uses the edges of a particular tree, and so, although we reason about our algorithm in a fully connected graph, in reality our algorithm works equally well on said tree. This result, then, informs the design of an optimal network.

3.1 Binary Trees (Warmup)

We build intuition by considering a natural solution to Token Computation on the complete graph: naïve aggregation on a rooted binary tree. In this schedule, nodes do computations and communications in lock-step. In particular, consider the schedule S which alternates the following two operations until only a single node with tokens remains on a fixed binary tree: (1) every non-root node that has a token sends its token to its parent in the binary tree; (2) every v with at least two tokens performs one computation. Once only one node has any tokens, that node performs computation until only a single token remains. After $\log n$ iterations of this schedule, the root of the binary tree is the only node with any tokens, and thereafter only performs computation for the remainder of S. However, S does not efficiently pipeline communication and computation: after each iteration of (1) and (2), the root of the tree gains an extra token. Therefore, after $\log n$ repetitions of this schedule, the root has $\log n$ tokens. In total, then, this schedule aggregates all tokens after essentially $\log n(t_c + t_m) + \log n \cdot t_c$ rounds. See Figure 1.

For certain values of t_c and t_m, we can speed up naïve aggregation on the binary tree by pipelining the computations of the root with the communications of other nodes in the network. In particular, consider the schedule S' for a fixed binary tree for the case when $t_c = t_m$ in which every non-root node behaves exactly as it does in S but the root always computes. Since the root always computes in S', even as other nodes are sending, it does not build up a surplus of tokens as in S. Thus, this schedule aggregates all tokens after essentially $\log n(t_c + t_m)$ rounds when $t_c = t_m$, as shown in Figure 2.



(a) Round 1 (b) Round 2 (c) Round 3 (d) Round 4

Figure 2: The aggregation schedule on a binary tree for $t_c = t_m = 1$ and $n = 7$ after 4 rounds where the root pipelines its computations. Again, tokens are represented by blue diamonds; a red arrow from node u to node v means that u sends to v; and a double-ended blue arrow between two tokens a and b means that a and b are combined at the node. Notice that the root will never have more than 3 tokens when this schedule is run.

However, as we will prove, binary trees are not optimal even when they pipeline computation at the root and $t_c = t_m$. In the remainder of this section, we generalize this pipelining intuition to arbitrary values of t_c and t_m and formalize how to show a schedule is optimal.

3.2 Complete Graphs

We now describe our optimal polynomial-time algorithm for complete graphs. This algorithm produces a schedule which greedily aggregates on a particular tree, T_n^*. In order to describe this tree, we first introduce the tree $T(R, t_c, t_m)$. This tree can be thought of as the largest tree for which greedy aggregation aggregates all tokens in R rounds given computation cost t_c and communication cost t_m. We will overload notation and let $T(R)$ denote $T(R, t_c, t_m)$ for some fixed values of t_c and t_m. Let the root of a tree be the node in that tree with no parents. Also, given a tree T_1 with root r we define T_1 JOIN T_2 as T_1 but where r also has T_2 as an additional subtree. We define $T(R)$ as follows (see Figure 3 for an example):

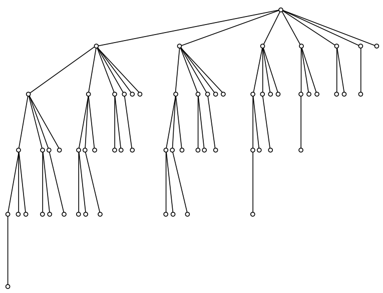


Figure 3: $T(16)$ for $t_c = 2$, $t_m = 1$.

$$T(R) := \begin{cases} \text{A single leaf} & \text{if } R < t_m + t_c \\ T(R - t_c) \text{ JOIN } T(R - t_c - t_m) & \text{otherwise} \end{cases}$$

Notice that for $t_c = t_m = 1$ the tree $T(R)$ is just a Fibonacci tree [Hor83] of recursion depth R; $T(R)$, then, can be thought of as a generalization of Fibonacci trees.[6]

Since an input to the TOKEN COMPUTATION problem consists of n nodes, and not a desired number of rounds, we define $R^*(n, t_c, t_m)$ to be the minimum value such that $|T(R^*(n, t_c, t_m))| \geq n$. We again overload notation and let $R^*(n)$ denote $R^*(n, t_c, t_m)$. Formally,

$$R^*(n) := \min\{R : |T(R)| \geq n\}.$$

[6] This echoes the fact that only graphs containing binomial heaps have $\log n$-length telephone broadcasts [EK05].

We let T_n^* denote $T(R^*(n))$. For ease of presentation we assume that $|T_n^*| = n$.[7]

The schedule produced by our algorithm will simply perform greedy aggregation on T_n^*. We now formally define greedy aggregation and establish its runtime on the tree $T(R)$.

Definition 1 (Greedy Aggregation)**.** Given an r-rooted tree, let the *greedy aggregation* schedule be defined as follows. In the first round, every node except for r sends its token to its parent. In subsequent rounds we do the following. If a node is not busy and has at least two tokens, it performs a computation. If a non-root node is not busy, has exactly one token, and has received a token from every child in previous rounds, it forwards its token to its parent.

Lemma 1. *Greedy aggregation on $T(R)$ terminates in R rounds.*

Proof. We will show by induction on $k \geq 0$ that greedy aggregation results in the root of $T(k)$ having a token of size $|T(k)|$ after k rounds. The base cases of $k \in [0, t_m + t_c)$ are trivial, as nothing needs to be combined. For the inductive step, applying the inductive hypothesis and using the recursive structure of our graph tells us that the root of $T(k + t_c)$ has a token of size $|T(k)|$ at its root in k rounds, and the root of the child $T(k - t_m)$ has a token of size $|T(k - t_m)|$ at its root in $k - t_m$ rounds. Therefore, by the definition of greedy aggregation, the root of $T(k - t_m)$ sends its token of size $|T(k - t_m)|$ to the root of $T(k + t_c)$ at time $k - t_m$, which means the root of $T(k + t_c)$ can compute a token of size $|T(k - t_m)| + |T(k)| = |T(k + t_c)|$ by round $k + t_c$. □

To build intuition about how quickly T_n^* grows, see Figure 6 for an illustration of $|T_n^*|$ as a function of n for specific values of t_c and t_m. Furthermore, notice that $T(R)$ and T_n^* are constructed in such a way that greedy aggregation pipelines computation and communication. We can now formalize our optimal algorithm, which simply outputs the greedy aggregation schedule on T_n^*, as Algorithm 1. The following theorem is our main result for this section.

Algorithm 1 OptComplete(t_c, t_m, n)

```
Input: $t_c$, $t_m$, $n$
Output: A schedule for Token Computation on $K_n$ with parameters $t_c$ and $t_m$
Arbitrarily embed $T_n^*$ into $K_n$
return Greedy aggregation schedule on $T_n^*$ embedded in $K_n$
```

Theorem 1. *Given a complete graph K_n on n vertices and any $t_m, t_c \in \mathbb{Z}^+$,* OptComplete *optimally solves* Token Computation *on the* Token Network *(K_n, t_c, t_m) in polynomial time.*

To show that Theorem 1 holds, we first note that OptComplete trivially runs in polynomial time. Therefore, we focus on showing that greedy aggregation on T_n^* optimally solves the Token Computation problem on K_n. We demonstrate this claim by showing that, given R rounds, $|T(R)|$ is the size of the largest solvable graph. Specifically, we will let $N^*(R)$ be the size of the largest complete graph on which one can solve Token Computation in R rounds, and we will argue that $N^*(R)$ obeys the same recurrence as $|T(R)|$.

First notice that the base case of $N^*(R)$ is trivially 1.

Lemma 2. *For $R \in \mathbb{Z}_0^+$ we have that $N^*(R) = 1$ for $R < t_c + t_m$.*

[7] If $|T_n^*| > n$, then we could always "hallucinate" extra nodes where appropriate.

Proof. If $R < t_c + t_m$, there are not enough rounds to send and combine a token, and so the Token Computation problem can only be solved on a graph with one node. □

We now show that for the recursive case $N^*(R)$ is always at least as large as $N^*(R - t_c) + N^*(R - t_c - t_m)$, which is the recurrence that defines $|T(R)|$.

Lemma 3. *For $R \in \mathbb{Z}_0^+$ we have that $N^*(R) \geq N^*(R - t_c) + N^*(R - t_c - t_m)$ for $R \geq t_c + t_m$.*

Proof. Suppose $R \geq t_c + t_m$. Let S_1 be the optimal schedule on the complete graph of $N^*(R - t_c)$ nodes with terminus v_{t1} and let S_2 be the optimal schedule on the complete graph of size $N^*(R - t_c - t_m)$ with corresponding terminus v_{t2}. Now consider the following solution on the complete graph of $N^*(R - t_c) + N^*(R - t_c - t_m)$ nodes. Run S_1 and S_2 in parallel on $N^*(R - t_c)$ and $N^*(R - t_c - t_m)$ nodes respectively, and once S_2 has completed, forward the token at v_{t2} to v_{t1} and, once it arrives, have v_{t1} perform one computation. This is a valid schedule which takes R rounds to solve Token Computation on $N^*(R - t_c) + N^*(R - t_c - t_m)$ nodes. Thus, we have that $N^*(R) \geq N^*(R - t_c) + N^*(R - t_c - t_m)$ for $R \geq t_c + t_m$. □

It remains to show that this bound on the recursion is tight. To do so, we case on whether $t_c \geq t_m$ or $t_c < t_m$. When $t_c \geq t_m$, we perform a straightforward case analysis to show that N^* follows the same recurrence as T_n^*. Specifically, we case on when the last token in the optimal schedule was created to show the following.

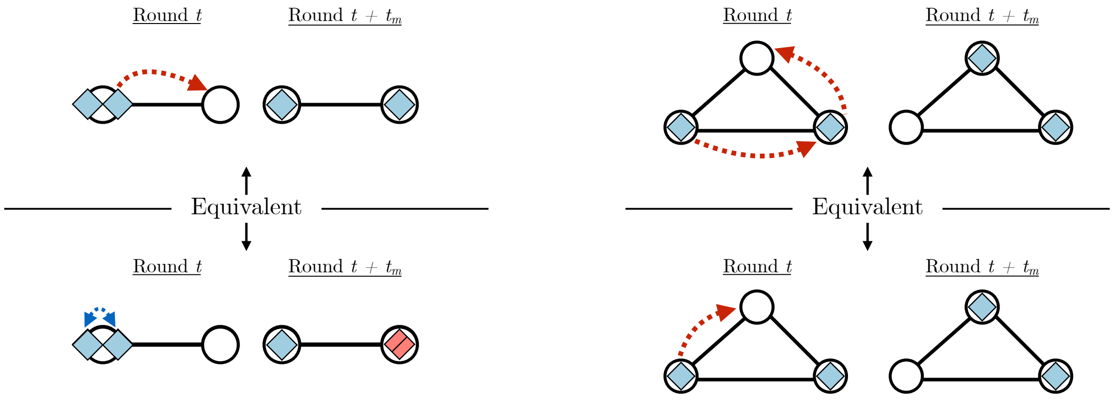
Lemma 4. *When $t_c \geq t_m$ for $R \in \mathbb{Z}_0^+$ it holds that $N^*(R) = N^*(R - t_c) + N^*(R - t_c - t_m)$.*

Proof. Suppose that $R \geq t_c + t_m$. By Lemma 3, it is sufficient to show that $N^*(R) \leq N^*(R - t_c) + N^*(R - t_c - t_m)$. Consider the optimal solution given R rounds. The last action performed by any node must have been a computation that combines two tokens, a and b, at the terminus v_t because, in an optimal schedule, any further communication of the last token increases the length of the schedule. We now consider three cases.

- In the first case, a and b were both created at v_t. Because both of a or b could not have been created at time $R - t_c$, one of them must have been created at time $R - 2t_c$ at the latest. This means that $N^*(R) \leq N^*(R - t_c) + N^*(R - 2t_c) \leq N^*(R - t_c) + N^*(R - t_c - t_m)$.
- In the second case, exactly one of a or b (without loss of generality, a) was created at v_t. This means that b must have been sent to v_t at latest at time $R - t_c - t_m$. It follows that $N^*(R) \leq N^*(R - t_c) + N^*(R - t_c - t_m)$.
- In the last case, neither a nor b was created at v_t. This means that both must have been sent to v_t at the latest at time $R - t_c - t_m$. We conclude that $N^*(R) \leq N^*(R - t_c - t_m) + N^*(R - t_c - t_m) \leq N^*(R - t_c) + N^*(R - t_c - t_m)$.

Thus, in all cases we have $N^*(R) \leq N^*(R - t_c) + N^*(R - t_c - t_m)$. □

We now consider the case in which communication is more expensive than computation, $t_c < t_m$. One might hope that the same case analysis used when $t_c \geq t_m$ would prove the desired result for when $t_c < t_m$. However, we must do significantly more work to show that $N^*(R) = N^*(R - t_c) + N^*(R - t_c - t_m)$ when $t_c < t_m$. We do this by establishing structure on the schedule which solves Token Computation on $K_{N^*(R)}$ in R rounds: we successively modify an optimal schedule in a way that does not affect its validity or length but which adds structure to the schedule.



(a) Combining insight

(b) Shortcutting insight

Figure 4: An illustration of the shortcutting and combining insights. Here, tokens are denoted by blue diamonds, and hallucinated tokens are denoted by striped red diamonds. As before, a red arrow from node u to node v means that u sends to v, and a double-ended blue arrow between two tokens a and b means that a and b are combined at the node. Notice that which nodes have tokens and when nodes have tokens are the same under both modifications (though in the combining insight, a node is only hallucinating that it has a token).

Specifically, we leverage the following insights — illustrated in Figure 4 — to modify schedules. *Combining insight*: Suppose node v has two tokens in round t, a and b, and v sends a to node u in round t. Node v can just aggregate a and b, treat this aggregation as it treats b in the original schedule and u can just pretend that it receives a in round $t + t_m$. That is, u can "hallucinate" that it has token a. Note that this insight crucially leverages the fact that $t_c < t_m$, since otherwise the performed computation would not finish before round $t + t_m$. *Shortcutting insight*: Suppose node v sends a token to node u in round t and node u sends a token to node w in a round in $[t, t + t_m]$. Node v can "shortcut" node u and send to w directly and u can just not send.

Through modifications based on these insights we show that there exists an optimal schedule where the last node to perform a computation never communicates, and every computation performed by this node computes on the token with which this node started. This structure, in turn, allows us to establish the following lemma, which asserts that when $t_c < t_m$ we have that $N^*(R)$ and $|T(R)|$ follow the same recurrence.

Lemma 5. *When $t_c < t_m$, for $R \in \mathbb{Z}_0^+$ it holds that $N^*(R) = N^*(R - t_c) + N^*(R - t_c - t_m)$.*

The proof of the lemma is relegated to Appendix D. We are now ready to prove the theorem.

Proof of Theorem 1. On a high level, we argue that the greedy aggregation schedule on $T(R)$ combines $N^*(R)$ nodes in R rounds and is therefore optimal. Combining Lemma 2, Lemma 4, and Lemma 5 we have the following recurrence on $N^*(R)$ for $R \in \mathbb{Z}_0^+$.

$$N^*(R) = \begin{cases} 1 & \text{if } R < t_c + t_m \\ N^*(R - t_c) + N^*(R - t_c - t_m) & \text{if } R \geq t_c + t_m \end{cases}$$

Notice that this is the recurrence which defines $|T(R)|$ so for $R \in \mathbb{Z}_0^+$ we have that $N^*(R) = |T(R)|$, and by Lemma 1, the greedy aggregation schedule on $T(R)$ terminates in R rounds.

Thus, the greedy aggregation schedule on $T(R)$ solves Token Computation on $K_{|T(R)|} = K_{N^*(R)}$ in R rounds, and therefore is an optimal solution for $K_{N^*(R)}$. Since T_n^* is the smallest $T(R)$ with at most n nodes, greedy aggregation on T_n^* is optimal for K_n and so OptComplete optimally solves Token Computation on K_n. Finally, the polynomial runtime is trivial. □

4 Hardness and Approximation for Arbitrary Graphs

We now consider the Token Computation problem on arbitrary graphs. Unlike in the case of complete graphs, the problem turns out to be computationally hard on arbitrary graphs. The challenge in demonstrating the hardness of Token Computation is that the optimal schedule for an arbitrary graph does not have a well-behaved structure. Our insight here is that by forcing a single node to do a great deal of computation we can impose structure on the optimal schedule in a way that makes it reflect the minimum dominating set of the graph. The following theorem formalizes this; its full proof is relegated to Appendix E.

Theorem 2. Token Computation *cannot be approximated by a polynomial-time algorithm within* $(1.5 - \epsilon)$ *for* $\epsilon \geq \frac{1}{o(\log n)}$ *unless* $P = NP$.

Therefore, our focus in this section is on designing an approximation algorithm. Specifically, we construct a polynomial-time algorithm, SolveTC, which produces a schedule that solves Token Computation on arbitrary graphs using at most $O(\log n \cdot \log \frac{\text{OPT}}{t_m})$ multiplicatively more rounds than the optimal schedule, where OPT is the length of the optimal schedule. Define the diameter D of graph G as $\max_{v,u} d(u, v)$. Notice that OPT/t_m is at most $(n-1)t_c/t_m + D$ since $\text{OPT} \leq (n-1)(t_c + D \cdot t_m)$: the schedule that picks a pair of nodes, routes one to the other then aggregates and repeats $n-1$ times is valid and takes $(n-1)(t_c + D \cdot t_m)$ rounds. Thus, our algorithm can roughly be understood as an $O(\log^2 n)$ approximation algorithm. Formally, our main result for this section is the following theorem whose lengthy proof we summarize in the rest of this section.

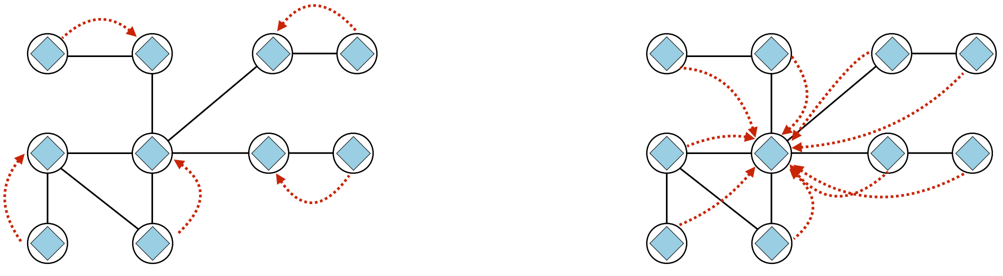
Theorem 3. SolveTC *is a polynomial-time algorithm that gives an* $O(\log n \cdot \log \frac{\text{OPT}}{t_m})$*-approximation for* Token Computation *with high probability.*

The rest of this section provides an overview of this theorem's lengthy proof. Our approximation algorithm, SolveTC, is given as Algorithm 2. SolveTC performs $O(\log n)$ repetitions of: designate some subset of nodes with tokens sinks and the rest of the nodes with tokens sources; route tokens at sources to sinks. If $t_c > t_m$, we will delay computations until tokens from sources arrive at sinks, and if $t_m \geq t_c$, we will immediately aggregate tokens that arrive at the same node.

4.1 Token Computation Extremes (Warmup)

Before moving on to a more technical overview of our algorithm, we build intuition by considering two extremes of Token Computation.

$t_m \ll t_c$. First, consider the case where $t_m \ll t_c$; that is, communication is very cheap compared to computation. As computation is the bottleneck here, we can achieve an essentially optimal schedule by parallelizing computation as much as possible. That is, consider a schedule consisting



(a) $t_m \ll t_c$

(b) $t_c \ll t_m$

Figure 5: An illustration of essentially optimal schedules for the extremes of the Token Computation problem. Dotted red arrows give the node towards which each node routes. In the case where $t_m \ll t_c$ one would repeat this sort of routing $O(\log n)$ times.

of $O(\log n)$ repetitions of: (1) each node with a token uniquely pairs off with another node with a token; (2) one node in each pair routes its token to the other node in its pair; (3) nodes that received a token perform one computation. This takes $O(t_c \cdot \log n)$ rounds to perform computations along with some amount of time to perform communications. But, any schedule takes at least $\Omega(t_c \cdot \log n)$ rounds, even if communication were free and computation were perfectly parallelized. Because the time to perform communications is negligible, this schedule is essentially optimal.

$t_c \ll t_m$. Now consider the case where $t_c \ll t_m$; that is, computation is very cheap compared to communication. In this setting, we can provide an essentially optimal schedule by minimizing the amount of communication that occurs. In particular, we pick a center c of the graph[8] and have all nodes send their tokens along the shortest path towards c. At any point during this schedule, it is always more time efficient for a node with multiple tokens to combine its tokens together before forwarding them since $t_c \ll t_m$. Thus, if at any point a node has multiple tokens, it combines these into one token and forwards the result towards c. Lastly, c aggregates all tokens it receives. This schedule takes $t_m \cdot r$ time to perform its communications, where r is the radius of the graph,[9] and some amount of time to perform its computations. However, because for every schedule there exists a token that must travel at least r hops, any schedule takes at least $\Omega(r \cdot t_m)$ rounds. Computations take a negligible amount of time since $t_c \ll t_m$, which means that this schedule is essentially optimal.

See Figure 5 for an illustration of these two schedules. Thus, in the case when $t_m \ll t_c$, we have that routing between pairs of nodes and delaying computations is essentially optimal, and in the case when $t_c \ll t_m$, we have that it is essentially optimal for nodes to greedily aggregate tokens before sending. These two observations will form the foundation of our approximation algorithm.

4.2 Approximation Algorithm

Recall that our approximation algorithm routes tokens from designated sources to designated sinks $O(\log n)$ times. Formally, the problem which our algorithm solves $O(\log n)$ times is as follows.

Definition 2 (Route and Compute Problem)**.** The input to the Route and Compute Problem

[8] The center of graph G is $\arg\min_v \max_u d(v,u)$ where $d(v,u)$ is the length of the shortest $u-v$ path.

[9] The radius of graph G is $\min_v \max_u d(v,u)$.

consists of a set $U \subseteq V$ and a set of directed paths $\vec{\mathcal{P}_U} = \{\vec{P_u} : u \in U\}$ where: (1) $u \in U$ has a token and is the source of $\vec{P_u}$; (2) every sink of every path $\vec{P_u}$ has a token; (3) if u and t_u are the sources and sinks of $\vec{P_u} \in \vec{\mathcal{P}_U}$, respectively, then neither u nor t_u are endpoints of any $\vec{P_{u'}} \in \vec{\mathcal{P}_U}$ for $u' \neq u$. A solution of cost C is a schedule of length C which, when run, performs computations on a constant fraction of tokens belonging to nodes in U.

SolveTC repeatedly calls a subroutine, GetDirectedPaths, to get a set of paths for which it would like to solve the Route and Compute Problem. It then solves the Route and Compute Problem for these paths, using RoutePaths$_m$ if $t_c \leq t_m$ or RoutePaths$_c$ if $t_c > t_m$. Below we give an overview of these procedures. The proofs of the lemmas in this section, as well as further details regarding SolveTC, are relegated to Appendix F.

Algorithm 2 SolveTC

```
Input: Token Computation instance given by graph $G = (V, E), t_c, t_m$
Output: A schedule for the input Token Computation problem
$W \leftarrow V$
for iteration $i \in O(\log n)$ do
    $\vec{\mathcal{P}_U} \leftarrow$ GetDirectedPaths$(W, G)$
    if $t_c > t_m$ then RoutePaths$_m(\vec{\mathcal{P}_U})$
    if $t_c \leq t_m$ then RoutePaths$_c(\vec{\mathcal{P}_U})$
    $W \leftarrow \{v : v \text{ has 1 token}\}$
```

4.2.1 Producing Paths on Which to Route

We now describe GetDirectedPaths. First, for a set of paths $\mathcal{P}$, we define the *vertex congestion* of $\mathcal{P}$ as $\text{con}(\mathcal{P}) = \max_v \sum_{P \in \mathcal{P}} (\#\text{ occurences of } v \in P)$, and the *dilation* of $\mathcal{P}$ as $\max_{P \in \mathcal{P}} |P|$.

Given that nodes in $W \subseteq V$ have tokens, GetDirectedPaths solves a flow LP which has a flow for each $w \in W$ whose sinks are $w' \in W$ such that $w' \neq w$. The objective of this flow LP is the vertex congestion. The flow for each $w \in W$ defines a probability distribution over (undirected) paths with endpoints w and w' where $w' \neq w$ and $w' \in W$. Given these probability distributions, we repeatedly sample paths by taking random walks proportional to LP values of edges until we produce a set of paths — one for each $w \in W$ — with low vertex congestion. Lastly, given our undirected paths, we apply another subroutine to direct our paths and fix some subset of nodes $U \subset W$ as sources such that $|U|$ is within a constant fraction of $|W|$. The key property of the LP we use is that it has an optimal vertex congestion comparable to OPT, the length of the optimal Token Computation schedule. Using this fact and several additional lemmas we can prove the following properties of GetDirectedPaths.

Lemma 6. *Given $W \subseteq V$, GetDirectedPaths is a randomized polynomial-time algorithm that returns a set of directed paths, $\vec{\mathcal{P}_U} = \{P_u : u \in U\}$ for $U \subseteq W$, such that with high probability at least $1/12$ of nodes in W are sources of paths in $\vec{\mathcal{P}_U}$ each with a unique sink in W. Moreover,*

$$\mathit{con}(\vec{\mathcal{P}_U}) \leq O\left(\frac{\text{OPT}}{\min(t_c, t_m)} \log \frac{\text{OPT}}{t_m}\right) \text{ and } \mathit{dil}(\vec{\mathcal{P}_U}) \leq \frac{8\text{OPT}}{t_m}.$$

4.2.2 Routing Along Produced Paths

We now specify how we route along the paths produced by GetDirectedPaths. If $t_c > t_m$, we run RoutePaths$_m$ to delay computations until tokens from sources arrive at sinks, and if $t_m \geq t_c$, we run RoutePaths$_c$ to immediately aggregate tokens that arrive at the same node.

Case of $t_c > t_m$ RoutePaths$_m$ adapts the routing algorithm of Leighton et al. [LMR94]—which was simplified by Rothvoß [Rot13]—to efficiently route from sources to sinks.[10] We let OPTRoute be this adaptation of the algorithm of Leighton et al. [LMR94].

Lemma 7. *Given a set of directed paths $\vec{\mathcal{P}}_U$ with some subset of endpoints of paths in $\vec{\mathcal{P}}_U$ designated sources and the rest of the endpoints designated sinks,* OPTRoute *is a randomized polynomial-time algorithm that w.h.p. produces a* Token Network *schedule that sends from all sources to sinks in $O(con(\vec{\mathcal{P}}_U) + dil(\vec{\mathcal{P}}_U))$.*

Given $\vec{\mathcal{P}}_U$, RoutePaths$_m$ is as follows. Run OPTRoute and then perform a single computation. As mentioned earlier, this algorithm delays computation until all tokens have been routed.

Lemma 8. RoutePaths$_m$ *is a polynomial-time algorithm that, given $\vec{\mathcal{P}}_U$, solves the* Route and Compute *Problem w.h.p. using $O(t_m(con(\vec{\mathcal{P}}_U) + dil(\vec{\mathcal{P}}_U)) + t_c)$ rounds.*

Case of $t_c \leq t_m$ Given directed paths $\vec{\mathcal{P}}_U$, RoutePaths$_c$ is as follows. Initially, every sink is *asleep* and every other node is *awake*. For $O(\text{dil}(\vec{\mathcal{P}}_U) \cdot t_m)$ rounds we repeat the following: if a node is not currently sending and has exactly one token then it forwards this token along its path; if a node is not currently sending and has two or more tokens then it sleeps for the remainder of the $O(\text{dil}(\vec{\mathcal{P}}_U) \cdot t_m)$ rounds. Lastly, every node combines any tokens it has for $t_c \cdot \text{con}(\vec{\mathcal{P}}_U)$ rounds.

Lemma 9. RoutePaths$_c$ *is a polynomial-time algorithm that, given $\vec{\mathcal{P}}_U$, solves the* Route and Compute *Problem w.h.p. using $O(t_c \cdot con(\vec{\mathcal{P}}_U) + t_m \cdot dil(\vec{\mathcal{P}}_U))$ rounds.*

By leveraging the foregoing results, we can prove Theorem 3; see Appendix F.3 for details.

5 Future Work

There are many promising directions for future work. First, as Section 4.1 illustrates, the extremes of our problem—when $t_c \ll t_m$ and when $t_m \ll t_c$—are trivial to solve. However, our hardness reduction demonstrates that for $t_c = 1$ and t_m in a specific range, our problem is hard to approximate. Determining precisely what values of t_m and t_c make our problem hard to approximate is open.

Next, it is not always the case that there exists a centralized coordinator to produce a schedule. We hope to give an analysis of our problem in a distributed setting as no past work in this setting takes computation into account. Even more broadly, we hope to analyze formal models of distributed computation in which nodes are not assumed to have unbounded computational resources and computation takes a non-trivial amount of time.

[10] Our approach for the case when $t_c > t_m$ can be simplified using techniques from Srinivasan and Teo [ST01]. In fact, using their techniques we can even shave the $\frac{\log \text{OPT}}{t_c}$ factor in our approximation. However, because these techniques do not take computation into account, they do not readily extend to the case when $t_c \leq t_m$. Thus, for the sake of a unified exposition, we omit the adaptation of their results.

We also note that there is a gap between our hardness of approximation and the approximation guarantee of our algorithm. The best possible approximation, then, is to be decided by future work.

Furthermore, we are interested in studying technical challenges similar to those studied in approximation algorithms for network design. For instance, we are interested in the problem in which each edge has a cost and one must build a network subject to budget constraints which has as efficient a Token Computation schedule as possible.

Lastly, there are many natural generalizations of our problem. For instance, consider the problem in which nodes can aggregate an arbitrary number of tokens together, but the time to aggregate multiple tokens is, e.g., a concave function of the number of tokens aggregated. These new directions offer not only compelling theoretical challenges but may be of practical interest.

References

[ACFP09] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella. Energy conservation in wireless sensor networks: a survey. *Ad Hoc Networks*, 7(3):537–568, 2009.

[AD09] Matthew Andrews and Michael Dinitz. Maximizing capacity in arbitrary wireless networks in the sinr model: Complexity and game theory. In *IEEE INFOCOM 2009*, pages 1332–1340. IEEE, 2009.

[ASSC02a] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[ASSC02b] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[BGMGM03] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani. Estimating aggregates on a peer-to-peer network. Technical report, Stanford InfoLab, 2003.

[CHGK$^+$14] Keren Censor-Hillel, Seth Gilbert, Fabian Kuhn, Nancy Lynch, and Calvin Newport. Structuring unreliable radio networks. *Distributed Computing*, 27(1):1–19, 2014.

[CHHHZ17] Keren Censor-Hillel, Bernhard Haeupler, D Ellis Hershkowitz, and Goran Zuzic. Broadcasting in noisy radio networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 33–42. ACM, 2017.

[CHHHZ18] Keren Censor-Hillel, Bernhard Haeupler, D Ellis Hershkowitz, and Goran Zuzic. Erasure correction for noisy radio networks. corr, vol. abs/1805.04165 (2018), 2018.

[CK85] Imrich Chlamtac and Shay Kutten. On broadcasting in radio networks-problem analysis and protocol design. *IEEE Transactions on Communications*, 33(12):1240–1246, 1985.

[DG08] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DKO14] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 367–376. ACM, 2014.

[DS14] I. Dinur and D. Steurer. Analytical approach to parallel repetition. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 624–633, 2014.

[EK05] M. Elkin and G. Kortsarz. A combinatorial logarithmic approximation algorithm for the directed telephone broadcast problem. *SIAM journal on Computing*, 35(3):672–689, 2005.

[EK06] M. Elkin and G. Kortsarz. Sublogarithmic approximation for telephone multicast. *Journal of Computer and System Sciences*, 72(4):648–659, 2006.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[Gra03] A. Grama. *Introduction to parallel computing.* Pearson Education, 2003.

[HHW18] B. Haeupler, D. E. Hershkowitz, and D. Wajc. Round- and message-optimal distributed part-wise aggregation. *arXiv preprint arXiv:1801.05127*, 2018.

[Hor83] Y. Horibe. Notes on Fibonacci trees and their optimality. *The Fibonacci Quarterly*, 21(2):118–128, 1983.

[IRRS15] J. Iglesias, R. Rajaraman, R. Ravi, and R. Sundaram. Rumors across radio, wireless, telephone. In *Proceedings of the Leibniz International Proceedings in Informatics (LIPIcs)*, volume 45, 2015.

[JLK12] N. Jain, J. M. Lau, and L. Kale. Collectives on two-tier direct networks. In *Proceedings of the European MPI Users' Group Meeting*, pages 67–77, 2012.

[KDG03] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 482–491, 2003.

[KM98] Eyal Kushilevitz and Yishay Mansour. Computation in noisy radio networks. In *SODA*, volume 98, pages 236–243, 1998.

[KOF15] B. Klenk, L. Oden, and H. Fröning. Analyzing communication models for distributed thread-collaborative processors in terms of energy and time. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 318–327, 2015.

[KP95] G. Kortsarz and D. Peleg. Approximation algorithms for minimum-time broadcast. *SIAM Journal on Discrete Mathematics*, 8(3):401–427, 1995.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

[Lin92] N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

[LMR94] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-shop scheduling in O(congestion + dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

[MAS06] D. Mosk-Aoyama and D. Shah. Computing separable functions via gossip. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 113–122, 2006.

[MYCR05] L. Marchal, Y. Yang, H. Casanova, and Y. Robert. A realistic network/application model for scheduling divisible loads on large-scale platforms. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 10–pp, 2005.

[OKF14] L. Oden, B. Klenk, and H. Fröning. Energy-efficient collective reduce and allreduce operations on distributed GPUs. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 483–492, 2014.

[Pel00] D. Peleg. Distributed computing. *SIAM Monographs on Discrete Mathematics and Applications*, 5, 2000.

[PY07] P. Patarasuk and X. Yuan. Bandwidth efficient all-reduce operation on tree topologies. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[PY09] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[Rab04] R. Rabenseifner. Optimization of collective reduction operations. In *Proceedings of the International Conference on Computational Science*, pages 1–9, 2004.

[Rav94] R. Ravi. Rapid rumor ramification: Approximating the minimum broadcast time. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 202–213, 1994.

[Rot13] T. Rothvoß. A simpler proof for O(congestion+dilation) packet routing. In *Proceedings of the International Conference on Integer Programming and Combinatorial Optimization*, pages 336–348, 2013.

[ST01] A. Srinivasan and C. Teo. A constant-factor approximation algorithm for packet routing and balancing local vs. global criteria. *SIAM Journal on Computing*, 30(6):2051–2068, 2001.

[VVR07] S. Viswanathan, B. Veeravalli, and T. G. Robertazzi. Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(10), 2007.

A Formal Model, Problem, and Definitions

Let us formally define the Token Computation problem. The input to the problem is a Token Network specified by graph $G = (V, E)$ and parameters $t_c, t_m \in \mathbb{N}$. Each node starts with a single token.

An algorithm for this problem must provide a schedule, $S : V \times [l] \to V \cup \{\mathsf{idle}, \mathsf{busy}\}$ where we refer to $|S| := l$ as the length of the schedule. Intuitively, a schedule S directs each node when to compute and when to communicate as follows:

- $S(v, r) = v' \neq v$ indicates that v begins passing a token to v' in round r of S;
- $S(v, r) = v$ indicates that v begins combining two token in round r of S;
- $S(v, r) = \mathsf{idle}$ indicates that v does nothing in round r;
- $S(v, r) = \mathsf{busy}$ indicates that v is currently communicating or computing.

Moreover, we define the number of computations that v has performed up to round r as $C_S(v, r) := \sum_{r' \in [r-t_c]} \mathbb{1}(S(v, r') == v)$, the number of messages that v has received up to round r as $R_S(v, r) := \sum_{r' \in [r-t_m]} \sum_{v' \neq v} \mathbb{1}(S(v', r') == v)$, and the number of messages that v has sent up to round r as $M_S(v, r) := \sum_{r' \in [r-t_m]} \sum_{v' \neq v} \mathbb{1}(S(v, r') == v')$. Finally, define the number of tokens a node has in round r of S as follows.

$$\mathsf{tokens}_S(v, r) := I(v) + R_S(v, r) - M_S(v, r) - C_S(v, r).$$

A schedule, S, is *valid* for Token Network (G, t_c, t_m) if:

1. Valid communication: If $S(v, r) = v' \neq v$ then $(v, v') \in E$, $S(v, r') = \mathsf{busy}$ for $r' \in [r+1, r+t_m]$ and $\mathsf{tokens}_S(v, r) \geq 1$;
2. Valid computation: If $S(v, r) = v$ then $S(v, r') = \mathsf{busy}$ for $r' \in [r+1, r+t_c]$ and $\mathsf{tokens}_S(v, r) \geq 2$;
3. Full aggregation: $\sum_{v \in V} \mathsf{tokens}_S(v, |S|) = 1$.

An algorithm solves Token Computation if it outputs a valid schedule.

B Deferred Related Work

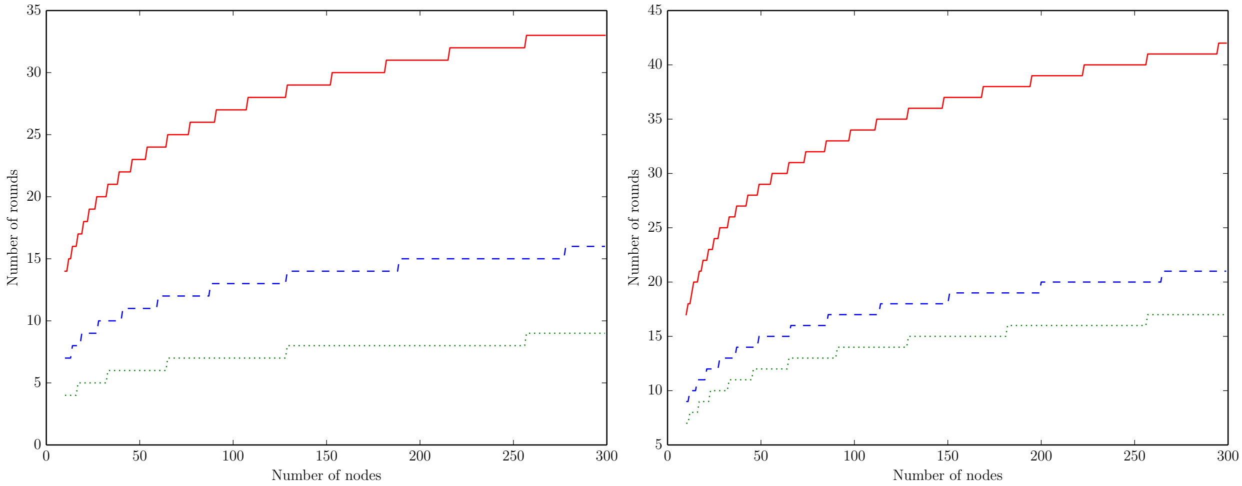
There is a significant body of applied work in resource-aware scheduling, sensor networks, and high-performance computing that considers both the relative costs of communication and computation, often bundled together in an energy cost. However, these studies have been largely empirical rather than theoretical, and much of the work considers distributed algorithms (as opposed to our centralized setting).

AllReduce in HPC There is much related applied work in the high-performance computing space on AllReduce [Rab04, Gra03]. However, while there has been significant research on communication-efficient AllReduce algorithms, there has been relatively little work that explicitly considers the cost of computation, and even less work that considers the construction of optimal topologies for efficient distributed computation. Researchers have empirically evaluated the performance of different models of communication [OKF14, KOF15] and have proven (trivial) lower bounds for communication without considering computation [PY07, PY09]. Indeed, to the best of our knowledge, the extent to which they consider computation is through an additive penalty that consists of a multiplicative factor times the size of all inputs at all nodes, as in the work of Jain et al. [JLK12]; crucially, this penalty is the same for any schedule and cannot be reduced via intelligent scheduling. Therefore, there do not seem to exist theoretical results for efficient algorithms that consider both the cost of communication and computation.

Resource-Aware Scheduling In the distributed computation space, people have considered resource-aware scheduling on a completely connected topology with different nodes having different loads. Although this problem considers computation-aware communication, these studies are much more empirical than theoretical, and only consider distributed solutions as opposed to centralized algorithms [VVR07, MYCR05].

Sensor Networks Members of the sensor networks community have studied the problem of minimizing an energy cost, which succinctly combines the costs of communication and computation. However, sensor networks involve rapidly-changing, non-static topologies [ASSC02a, ASSC02b], which means that their objective is not to construct a fixed, optimal topology, but rather to develop adaptive algorithms for minimizing total energy cost with respect to an objective function [ACFP09].

C Deferred Figures



(a) $t_c = 1$, $t_m = 2$

(b) $t_c = 2$, $t_m = 1$

Figure 6: An illustration of the optimal schedule length for different sized trees. The solid red line is the number of rounds taken by greedy aggregation with pipelining on a binary tree (i.e., $\lceil 2 \cdot t_c \cdot \log n + t_m \cdot \log n \rceil$); the dashed blue line is the number of rounds taken by greedy aggregation on T_n^*; and the dotted green line is the trivial lower bound of $\lceil t_c \cdot \log n \rceil$ rounds. Note that though we illustrate the trivial lower bound of $\lceil t_c \cdot \log n \rceil$ rounds, the true lower bound is given by the number of rounds taken by greedy aggregation on T_n^*.

D Proof of Lemma 5

Using our combining and shortcutting insights, we establish the following structure on a schedule which solves Token Computation on $K_{N^*(R)}$ in R rounds when $t_c < t_m$.

Lemma 10. *When $t_c < t_m$, for all $R \in \mathbb{Z}_0^+$, there exists a schedule, $\tilde{S}^*$, of length R that solves* Token Computation *on $K_{N^*(R)}$ such that the terminus of $\tilde{S}^*$, v_t, never communicates and every computation performed by v_t involves a token that contains v_t's singleton token, a_{v_t}.*

Proof. Let S^* be some arbitrary schedule of length R which solves Token Computation on $K_{N^*(R)}$; we know that such a schedule exists by definition of $N^*(R)$. We first show how to modify S^* into another schedule, S^*_{1-4}, which not only also solves Token Computation on $K_{N^*(R)}$ in R rounds, but which also satisfies the following four properties.

(1) v only sends at time t if v at time t has exactly one token for $t \in [R]$;

(2) if v sends in round t then v does not receive any tokens in rounds $[t, t + t_m]$ for $t \in [R]$;

(3) if v sends in round t then v is idle during rounds $t' > t$ for $t \in [R]$;

(4) the terminus never communicates.

Achieving property (1). Consider an optimal schedule S^*. We first show how to modify S^* to an R-round schedule S_1^* that solves Token Computation on $K_{N^*(R)}$ and satisfies property (1). We use our combining insight here. Suppose that (1) does not hold for S^*; i.e., a node v sends a token a_1 to node u at time t and v has at least one other token, say a_2, at time t. We modify S^* as follows. At time t, node v combines a_1 and a_2 into a token which it then performs operations on (i.e., computes and sends) as it does to a_2 in the original schedule. Moreover, node u pretends that it receives token a_1 at time $t + t_m$: any round in which S^* has u compute on or communicate a_1, u now simply does nothing; nodes that were meant to receive a_1 do the same. It is easy to see that by repeatedly applying the above procedure to every node when it sends when it has more than one token, we can reduce the number of tokens every node has whenever it sends to at most one. The total runtime of this schedule is no greater than that of S^*, namely R, because $t_c < t_m$. Moreover, it clearly still solves Token Computation on $K_{N^*(R)}$. Call the schedule S_1^*.

Achieving properties (1) and (2). Now, we show how to modify S_1^* into S_{1-2}^* such that properties (1) and (2) both hold. Again, S_{1-2}^* is of length R and solves Token Computation on $K_{N^*(R)}$. We use our shortcutting insight here. Suppose that (2) does not hold for S_1^*; i.e., there exists a v that receives a token a_1 from node u while sending another token a_2 to node u'. We say that node u is *bothering* node v in round t if node u communicates a token a_1 to v in round t, and node v communicates a token a_2 to node $u' \in V \setminus \{v, u\}$ in round $[t, t + t_m]$. Say any such pair is a *bothersome* pair. Furthermore, given a pair of nodes (u, v) and round t such that node u is bothering node v in round t, let the *resolution* of (u, v) in round t be the modification in which u sends its token directly to the node u' to which v sends its token. Note that each resolution does not increase the length of the optimal schedule because, by the definition of bothering, this will only serve as a shortcut; u' will receive a token from u at the latest in the same round it would have received a token from v in the original schedule, and nodes u' and v can pretend that they received tokens from v and u, respectively. However, it may now be the case that node u ends up bothering node u'. We now show how to repeatedly apply resolutions to modify S_1^* into a schedule S_{1-2}^* in which no node bothers another in any round t.

Consider the graph $B_t(S_1^*)$ where the vertices are the nodes in G and there exists a directed edge (u, v) if node u is bothering node v in round t in schedule S_1^*. First, consider cycles in $B_t(S_1^*)$. Note that, for any time t in which $B_t(S_1^*)$ has a cycle, we can create a schedule $\tilde{S}_1^*$ in which no nodes in any cycle in $B_t(S_1^*)$ send their tokens in round t; rather, they remain idle this round and pretend they received the token they would have received under S_1^*. Clearly, this does not increase the length of the optimal schedule and removes all cycles in round t. Furthermore, this does not violate property (1) because fewer nodes send tokens in round t, and no new nodes send tokens in round t.

Therefore, it suffices to consider an acyclic, directed graph $B_t(\tilde{S}_1)$. Now, for each round t, we repeatedly apply resolutions until no node bothers any other node during that round. Note that for every t, each node can only be bothering at most one other node because nodes can only send one message at a time. This fact, coupled with the fact that $B_t(\tilde{S}_1)$ is acyclic, means that $B_t(\tilde{S}_1)$ is a DAG where nodes have out-degree 1. It is not hard to see that repeatedly applying resolutions to a node v which bothers another node will decrease the number of edges in $B_t(\tilde{S}_1)$ by 1. Furthermore, because there are n total nodes in the network, the number of resolutions needed for any node v at time t is at most n.

Furthermore, repeatedly applying resolutions to $B_t(\tilde{S}_1)$ for times $t = 1, \ldots, R$ in order results in a schedule S^*_{1-2} with no bothersome pairs at any time t and that still satisfies property (1), and so schedule S^*_{1-2} satisfies properties (1) and (2). Since each resolution did not increase the length of the schedule we also have that S^*_{1-2} is of length R. Lastly, S^*_{1-2} clearly still solves Token Computation on $K_{N^*(R)}$.

Achieving properties (1) - (3). Now, we show how to modify S^*_{1-2} into S^*_{1-3} which satisfies properties (1), (2), and (3). We use our shortcutting insight here as well as some new ideas. Given S^*_{1-2}, we show by induction over k from 0 to $R - t_m$, where R is the length of an optimal schedule, that we can modify S^*_{1-2} such that if a node finishes communicating in round $R - k$ (i.e., begins communicating in round $R - k - t_m$), it remains idle in rounds $t' \in (R - k, R]$ in the modified optimal schedule. The base case of $k = 0$ is trivial: If a node communicates in round $R - t_m$, it must remain idle in round R because the entire schedule is of length R.

Suppose there exists a node v that finishes communicating in round $t = R - k$ but is not idle in some round $t' > R - k$ in S^*_{1-2}; furthermore, let round t' be the first round after t in which node v is not idle. By property (1), node v must have sent its only token away in round t, and therefore node v must have received at least one other token after round t but before round t'. We now case on the type of action node v performs in round t'.

- If node v communicates in round t', it must send a token it received after time t but before round t'. Furthermore, as this is the first round after t in which v is not idle, v cannot have performed any computation on this token, and by the inductive hypothesis, v must remain idle from round $t' + t_m$ on. Therefore, v receives a token a_u from some node u and then forwards this token to node u' at time t'. One can modify this schedule such that u sends a_u directly to u' instead of sending to v.

- If node v computes in round t', consider the actions of node v after round $t' + t_c$. Either v eventually performs a communication after some number of computations, after which point it is idle by the inductive hypothesis, or v only ever performs computations from time t' on.

  In round t', v must combine two tokens it received after time $t + t_m$ by property (1). Note that two distinct nodes must have sent the two tokens to v because, by the inductive hypothesis, each node that sends after round t remains idle for the remainder of the schedule. Therefore, the nodes u_1 and u_2 that sent the two tokens to v must have been active at times $t'_1, t'_2 > t$, where $t_1 \leq t_2$, after which they remain idle for the rest of the schedule. Call the tuple (v, u_1, u_2) a *switchable* triple. We can modify the schedule to make v idle at round t' by picking the node that first sent to v and treating it as v while the original v stays idle for the remainder of the schedule. In particular, we can modify S^*_{1-2} such that, without loss of generality, u_2 sends its token to u_1 and u_1 performs the computation that v originally performed in S^*_{1-2}. Note that this now ensures that v will be idle in round t' and does not increase the length of the schedule, as u_1 takes on the role of v. Furthermore, node u_1's new actions do not violate the inductive hypothesis: Either u_1 only ever performs computations after time t', or it eventually communicates and thereafter remains idle.

  We can repeat this process for all nodes that are not idle after performing a communication in order to produce a schedule S^*_{1-3} in which property (3) is satisfied.

First, notice that these modifications do not change the length of S^*_{1-2}: in the first case u' can still pretend that it receives a_u at time $t' + t_m$ even though it now receives it in an earlier round and in the second case u_2 takes on the role of v at the expense of no additional round overhead. Also, it is easy to see that S^*_{1-3} still solves Token Computation on $K_{N^*(R)}$.

We now argue that the above modifications preserve (1) and (2). First, notice that the modifications we do for the first case do not change when any nodes send and so (1) is satisfied. In the second case, because we switch the roles of nodes, we may potentially add a send for a node. However, note that we only require a node u_1 to perform an additional send when it is part of a switchable triple (v, u_1, u_2), and u_1 takes on the role of v in the original schedule from time t' on. However, because S^*_{1-2} satisfies (1), u was about to send its only token away and therefore only had one token upon receipt of the token from u_2. Therefore, because u_1 performs the actions that v performs in S^*_{1-2} from time t' on, and because at time t', both u_1 and v have exactly two tokens, (1) is still satisfied by S^*_{1-3}. Next, we argue that (3) is a strictly stronger condition than (2). In particular, we show that since S^*_{1-3} satisfies (3) it also satisfies (2). Suppose for the sake of contradiction that S^*_{1-3} satisfies (3) but not (2). Since (2) is not satisfied there must exist some node v that sends in some round t to, say node u, but receives a token in some round in $[t, t+t_m]$. By (3) it then follows that v is idle in all rounds after t. However, u also receives a token in round $t+t_m$. Therefore, in round $t+t_m$, two distinct nodes have tokens, one of which is idle in all rounds after $t+t_m$; this contradicts the fact that S^*_{1-3} solves Token Computation. Thus, S^*_{1-3} must also satisfy (2)

Achieving properties (1) - (4). It is straightforward to see that S^*_{1-3} also satisfies property (4). Indeed, by property (3), if the terminus ever sends in round $t < R - t_c$, then the terminus must remain idle during rounds $t' > t$, meaning it must be idle in round $R - t_c$ which contradicts the fact that in this round the terminus performs a computation. Therefore, $S^*_{1-4} = S^*_{1-3}$ satisfies properties (1) - (4), and we know that there exists an optimal schedule in which v_t is always either computing or idle.

Achieving the final property. We now argue that we can modify S^*_{1-4} into another optimal schedule $\tilde{S}^*$ such that every computation done at the terminus v_t involves a token that contains the original singleton token that started at the terminus. Suppose that in S^*_{1-4}, v_t performs computation that does not involve a_{v_t}. Take the first instance in which v_t combines tokens a_1 and a_2, neither of which contains a_{v_t}, in round t. Because this is the first computation that does not involve a token containing a_{v_t}, both a_1 and a_2 must have been communicated to the terminus in round $t - t_m$ at the latest.

Consider the earliest time $t' > t$ in which v_t computes a token a_{comb} that contains all of a_1, a_2, and a_{v_t}. We now show how to modify S^*_{1-4} into $\tilde{S}'$ such that v_t computes a token a'_{comb} at time t' that contains all of a_1, a_2, and a_{v_t} and is at least the size of a_{comb} by having nodes swap roles in the schedule between times t and t'. Furthermore, because the rest of the schedule remains the same after time t', this implies that $\tilde{S}'$ uses at most as many rounds as S^*_{1-4}, and therefore that $\tilde{S}'$ uses at most R rounds.

The modification is as follows. At time t, instead of having v_t combine tokens a_1 and a_2, have v_t combine one of them (without loss of generality, a_1) with the token containing a_{v_t}. Now, continue executing S^*_{1-4} but substitute a_2 for the token containing a_{v_t} from round t on; this is a valid substitution because v_t possesses a_2 at time t. In round t', v_t computes a token $a'_{comb} = a_{comb}$; the difference from the previous schedule is that the new schedule has one fewer violation of property (4), i.e., one fewer round in which it computes on two tokens, neither of which contains a_{v_t}.

We repeat this process for every step in which the terminus does not compute on the token containing a_{v_t}, resulting in a schedule $\tilde{S}^*$ in which the terminus is always combining a communicated token with a token containing its own singleton token. Note that these modifications do not affect properties (1) - (4) because this does not affect the sending actions of any node, and therefore $\tilde{S}^*$ still satisfies properties (1) - (4). It easily follows, then, that $\tilde{S}^*$ solves Token Computation on $K_{N^*(R)}$ in R rounds. Thus, $\tilde{S}^*$ is a schedule of length R that solves Token Computation on $K_{N^*(R)}$ in which every computation the terminus v_t does is on two tokens, one of which contains a_{v_t}, and, by (4), the terminus v_t never communicates. □

Having shown that the schedule corresponding to $N^*(R)$ can be modified to satisfy a nice structure when $t_c < t_m$, we can conclude our recursive bound on $N^*(R)$.

Lemma 5. *When $t_c < t_m$, for $R \in \mathbb{Z}_0^+$ it holds that $N^*(R) = N^*(R - t_c) + N^*(R - t_c - t_m)$.*

Proof. Suppose $R \geq t_c + t_m$. We begin by applying Lemma 10 to show that $N^*(R) \leq N^*(R - t_c) + N^*(R - t_c - t_m)$. Let v_t be the terminus of the schedule $\tilde{S}^*$ using R rounds as given in Lemma 10. By Lemma 10, in all rounds after round t_m of $\tilde{S}^*$ it holds that v_t is either computing on a token that contains a_{v_t} or busy because it did such a computation. Notice that it follows that every token produced by a computation at v_t contains a_{v_t}.

Now consider the last token produced by our schedule. Call this token a. By definition of the terminus, a must be produced by a computation performed by v_t, combining two tokens, say a_1 and a_2, in round $R - t_c$ at the latest. Since every computation that v_t does combines two tokens, one of which contains a_{v_t}, without loss of generality let a_1 contain a_{v_t}.

We now bound the size of a_1 and a_2. Since a_1 exists in round $R - t_c$ we know that it is of size at most $N^*(R - t_c)$. Now consider a_2. Since every token produced by a computation at v_t contains a_{v_t} and a_2 does not contain a_{v_t} it follows that a_2 must either be a singleton token that originates at a node other than v, or a_2 was produced by a computation at another node. Either way, a_2 must have been sent to v, who then performed a computation on a_2 in round $R - t_c$ at the latest. It follows that a_2 exists in round $R - t_c - t_m$, and so a_2 is of size no more than $N^*(R - t_c - t_m)$.

Since the size of a just is the size of a_1 plus the size of a_2, we conclude that a is of size no more than $N^*(R - t_c) + N^*(R - t_c - t_m)$. Since, $\tilde{S}^*$ solves Token Computation on a complete graph of size $N^*(R)$, we have that a is of size $N^*(R)$ and so we conclude that $N^*(R) \leq N^*(R - t_c) + N^*(R - t_c - t_m)$ for $R \geq t_c + t_m$ when $t_c < t_m$.

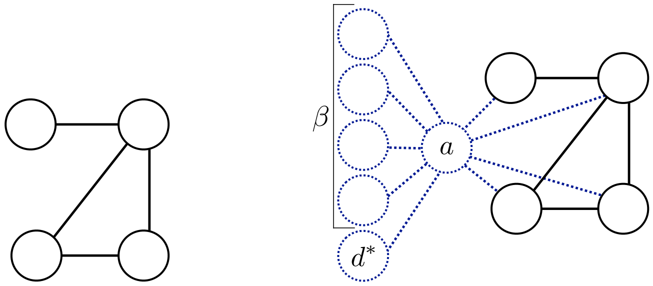
Lastly, since $N^*(R) \geq N^*(R - t_c) + N^*(R - t_c - t_m)$ for $R \geq t_c + t_m$ by Lemma 3, we conclude that $N^*(R) = N^*(R - t_c) + N^*(R - t_c - t_m)$ for $R \geq t_c + t_m$ when $t_c < t_m$. □

E Proof of Theorem 2

As a warmup for our hardness of approximation result, and to introduce some of the techniques, we begin with a proof that the decision version of Token Computation is NP-complete in Appendix E.1. We then prove the hardness of approximation result in Appendix E.2.

E.1 NP-Completeness (Warmup)

An instance of the decision version of Token Computation is given by an instance of Token Computation and a candidate ℓ. An algorithm must decide if there exists a schedule that solves Token Computation in at most ℓ rounds.



(a) G

(b) $\Psi(G, 1)$

Figure 7: An example of Ψ for a given graph G and $t_m = 1$. Nodes and edges added by Ψ are dashed and in blue. Notice that $|\beta| = \Delta + t_m = 3 + 1 = 4$.

We reduce from k-dominating set.

Definition 3 (k-dominating set)**.** *An instance of k-dominating set consists of a graph $G = (V, E)$; the decision problem is to decide whether there exists $\kappa \subseteq V$ where $|\kappa| = k$ such that for all $v \in V \setminus \kappa$ there exists $\nu \in \kappa$ such that $(v, \nu) \in E$.*

Recall that k-dominating set is NP-complete.

Lemma 11 (Garey and Johnson [GJ79])**.** *k-dominating set is NP-complete.*

Given an instance of k-dominating set, we would like to transform G into another graph G' in polynomial time such that G has a k-dominating set iff there exists a Token Computation schedule of some particular length for G' for some values of t_c and t_m.

We begin by describing the intuition behind the transformation we use, which we call Ψ. Any schedule on graph G in which every node only performs a single communication and which aggregates all tokens down to at most k tokens corresponds to a k-dominating set of G; in particular, those nodes that do computation form a k-dominating set of G. If we had a schedule of length $< 2t_m$ which aggregated all tokens down to k tokens, then we could recover a k-dominating set from our schedule. However, our problem aggregates down to only a single token, not k tokens. Our crucial insight, here, is that by structuring our graph such that a single node, a, must perform a great deal of computation, a must be the terminus of any short schedule. The fact that a must be the terminus and do a great deal of computation, in turn, forces any short schedule to aggregate all tokens in G down to at most k tokens at some point, giving us a k-dominating set.

Formally, Ψ is as follows. Ψ takes as input a graph G and a value for t_m and outputs $G' = (V', E')$. G' has G as a sub-graph and in addition has auxiliary node a where a is connected to all $v \in V$; a is also connected to dangling nodes $d \in \beta$, where $|\beta| = \Delta + t_m$, along with a special dangling node d^*.[11] Thus, $G' = (V \cup \{a, d^*\} \cup \beta, E \cup \{(a, v') : v' \in V' \setminus \{a\}\})$. See Figure 7.

We now prove that the optimal Token Computation schedule on $G' = \Psi(G, t_m)$ can be upper bounded as a function of the size of the minimum dominating set of G.

Lemma 12. *The optimal* Token Computation *schedule on $G' = \Psi(G, t_m)$ is of length at most $2t_m + \Delta + k^*$ for $t_c = 1$, where k^* is the minimum dominating set of G.*

[11] Δ is the max degree of G.

Proof. We know by definition of k^* that there is a dominating set of size k^* on G. Call this set κ and let $\sigma : V \rightarrow \kappa$ map any given $v \in V$ to a unique node in κ that dominates it. We argue that it must be the case that Token Computation requires at most $2t_m + t_c(\Delta + k^*)$ rounds on G' for $t_c = 1$. Roughly, we solve Token Computation by first aggregating at κ and then aggregating at a.

In more detail, in stage 1 of the schedule, every $d \in \beta$ sends to a, every node $v \in V$ sends to $\sigma(v)$ and a sends to d^*. This takes t_m rounds. In stage 2, each node does the following in parallel. Node d^* computes and sends its single token to a. Each $\nu \in \kappa$ computes until it has a single token and sends the result to a. Node a combines all tokens from $\beta \cup \{d^*\}$. Node d^* takes $1 + t_m$ rounds to do this. Each $\nu \in \kappa$ takes at most $\Delta + t_m$ rounds to do this. Node a takes $\Delta + t_m$ rounds to do this since a will receive d^*'s token after $t_m + 1$ rounds (and $\Delta \geq 1$ without loss of generality). Thus, stage 2, when done in parallel, takes $\Delta + t_m$ rounds. At this point a has $k^* + 1$ tokens and no other node in G' has a token. In stage 3, a computes until it has only a single token, which takes k^* rounds.

In total the number of rounds used by this schedule is $t_m + \Delta + t_m + k^* = 2t_m + \Delta + k^*$. Thus, the total number of rounds used by the optimal Token Computation schedule on G' is at most $2t_m + \Delta + k^*$. □

Next, we show that any valid Token Computation schedule on $G' = \Psi(G, t_m)$ that has at most two serialized sends corresponds to a dominating set of size bounded by the length of the schedule.

Lemma 13. *Given $G' = \Psi(G, t_m)$ and a* Token Computation *schedule S for G' where $|S| < 3t_m$, $t_c = 1$, $\kappa = \{v : v \in G,\ v \text{ sends to } a \text{ in } S\}$ is a dominating set of G of size $|S| - 2t_m - \Delta$.*

Proof. Roughly, we argue that a must be the terminus of S and must perform at most $|S| - 2t_m - \Delta$ computations on tokens from G, each of which is the aggregation of a node's token and some of its neighbors' tokens. We begin by arguing that a must be the terminus.

First, we prove that no $d \in \beta \cup \{d^*\}$ is the terminus of S. Suppose for the sake of contradiction that some $\bar{d} \in \beta \cup \{d^*\}$ is the terminus. Since our schedule takes fewer than $3t_m$ rounds, we know that every node sends a token that is not just the singleton token with which it starts at most once. Thus, a sends tokens that are not just the singleton token that it starts with at most once. Since $|\beta \cup \{d^*\} \setminus \{\bar{d}\}| = \Delta + t_m$ and a is the only node connected to these nodes, we know that every singleton token that originates in $\beta \cup \{d^*\} \setminus \{\bar{d}\}$ must travel through a. Moreover, since a sends tokens that are not just the singleton token that it starts with at most once, a must send all such tokens as a single token. It follows that a must perform at least $\Delta + t_m$ computations, but then our entire schedule takes at least $t_m + \Delta + t_m + t_m = 3t_m + \Delta > 3t_m$ rounds — a contradiction to our assumption that our schedule takes less than $3t_m$ rounds.

We now argue that no $v \in G$ is the terminus. Suppose for the sake of contradiction that some $\bar{v} \in V$ is the terminus. Again, we know that a sends tokens that are not just the singleton token that it starts with at most once. Thus, every token in $\beta \cup \{d^*\}$ must travel through a, meaning that a must perform $\Delta + t_m + 1$ computations. It follows that the schedule takes $t_m + \Delta + t_m + t_m + 1 > 3t_m$ rounds, a contradiction to our assumption that the schedule takes $< 3t_m$ rounds.

Thus, since no $d \in \beta \cup \{d^*\}$ and no $v \in G$ is the terminus, we know that a must be the terminus.

We now argue that a sends a token in the first round and this is the only time that a sends (i.e., the only thing that a sends is the singleton token that it starts with, which it sends immediately). Assume for the sake of contradiction that a sends a token that it did not start with. It must have taken at least t_m rounds for this token to arrive at a and at least an additional t_m rounds for a to send a token containing it. Moreover, since a is the terminus, a token containing this token must eventually return to a and so an additional t_m rounds are required. Thus, at least $3t_m$ rounds are required if a sends a token other than that with which it starts, a contradiction to the fact that our schedule takes $< 3t_m$ rounds.

Thus, since a is the terminus, our schedule solves Token Computation in fewer than $3t_m$ rounds, and no computations occur in the first t_m rounds, a does at most $|S| - t_m$ computations. Since a never sends any token aside from its singleton token, and a is the only node to which $\beta \cup \{d^*\}$ are connected, we know that a must combine all tokens of nodes in $\beta \cup \{d^*\}$, where a must take $\Delta + t_m$ rounds to do so. Thus, since a takes $\Delta + t_m$ rounds to aggregate tokens from $\beta \cup \{d^*\}$ and it performs at most $|S| - t_m$ computations in total, a must receive at most $|S| - 2t_m - \Delta$ tokens from G. It follows that $|\kappa| \leq |S| - 2t_m - \Delta$.

Since each token sent by a node in κ to a must be sent at the latest in round $|S| - t_m$ and since $|S| < 3t_m$, we have that every token sent by a node in κ is formed in fewer than $2t_m$ rounds. It follows that each such token is formed by tokens that travel at most 1 hop in G. Since every node in G must eventually aggregate its tokens at a, it follows that every node in G is adjacent to a node in κ. Thus κ is a dominating set of G, and as shown before $|\kappa| \leq |S| - 2t_m - \Delta$. □

Having shown that the optimal Token Computation schedule of $G' = \Psi(G, t_m)$ is closely related to the size of the minimum dominating set, we prove that Token Computation is NP-complete.

Theorem 4. *The decision version of* Token Computation *is NP-complete.*

Proof. The problem is clearly in NP. To show hardness, we reduce from k-dominating set. Specifically, we give a polynomial-time Karp reduction from k-dominating set to the decision version of Token Computation.

Our reduction is as follows. First, run $\Psi(G, t_m)$ for $t_m = \Delta + k + 1$ to get back G'. Next, return a decision version instance of Token Computation given by graph G' with $t_m = \Delta + k + 1$, $t_c = 1$ and $\ell = 2t_m + \Delta + k$. We now argue that G' has a schedule of length ℓ iff G has a k-dominating set.

- Suppose that G has a k-dominating set. We know that $k \geq k^*$, where k^* is the minimum dominating of G, and so by Lemma 12 we know that G' has a schedule of length at most $2t_m + \Delta + k^* \leq 2t_m + \Delta + k$.

- Suppose that G' has a Token Computation schedule S of length at most $2t_m + \Delta + k$. Notice that by our choice of t_m, we have that $|S| = 2t_m + \Delta + k < 3t_m$ and so by Lemma 13 we know that $\kappa = \{v : v \in G,\ v \text{ sends to } a \text{ in } S\}$ is a dominating set of G of size $|S| - 2t_m - \Delta$. Since $|S| \leq 2t_m + \Delta + k$ we conclude that $|\kappa| = |S| - 2t_m - \Delta \leq k$.

Lastly, notice that our reduction, Ψ, runs in polynomial time since it adds at most a polynomial number of vertices and edges to G. Thus, we conclude that k-dominating is polynomial-time reducible to the decision version of Token Computation, and therefore the decision version of Token Computation is NP-complete. □

E.2 Hardness of Approximation

We now show that unless P = NP there exists no polynomial-time algorithm that approximates Token Computation multiplicatively better than 1.5.

Recall that k-dominating set is $\Omega(\log n)$ hard to approximate.

Lemma 14 (Dinur and Steurer [DS14])**.** *Unless $P = NP$ every polynomial-time algorithm approximates minimum dominating set at best within a $(1 - o(1))(\log n)$ multiplicative factor.*

We prove hardness of approximation by using a $(1.5 - \epsilon)$ algorithm for Token Computation to approximate minimum dominating set with a polynomial-time algorithm better than $O(\log n)$. Similar to our proof of NP-completeness, given input graph G whose minimum dominating set we would like to approximate, we would like to transform G into another graph G' such that a $(1.5 - \epsilon)$-approximate Token Computation schedule for G' allows us to recover an approximately minimum dominating set.

One may hope to simply apply the transformation Ψ from the preceding section to do so. However, it is not hard to see that the approximation factor on the minimum dominating set recovered in this way has dependence on Δ, the maximum degree of G. If Δ is significantly larger than the minimum dominating set of G, we cannot hope that this will yield a good approximation to minimum dominating set. For this reason, before applying Ψ to G, we duplicate G a total of Δ/ϵ times to create graph G_α; this keeps Δ unchanged but increases the size of the minimum dominating set.[12] By applying Ψ to G_α instead of G to get back G'_α we are able to free our approximation factor from a dependence on Δ. Lastly, we show that we can efficiently recover an approximate minimum dominating set for G from an approximate Token Computation schedule for G'_α using our polynomial-time algorithm DSFromSchedule. Our full algorithm is given by MDSApx.

We first describe the algorithm — DSFromSchedule — we use to recover a minimum dominating set for G given a Token Computation schedule for $G'_\alpha = \Psi(G_\alpha, t_m)$. We denote copy i of G as G_i.

Algorithm 3 DSFromSchedule

```
**Input:** $G'_\alpha = \Psi(G_\alpha, t_m)$; a valid Token Computation schedule for $G'_\alpha$, $S$, of length $< 3t_m$; $\epsilon$
**Output:** A dominating set for $G$ of size $|S| - 2t_m - \Delta$
$\mathcal{K} \leftarrow \emptyset$
**for** $i \in \left[\frac{\Delta}{\epsilon}\right]$ **do**
    $\kappa_i \leftarrow \{v \in V_i : v \in G_\alpha \text{ sends to } a \text{ in } S\}$
    $\mathcal{K} \leftarrow \mathcal{K} \cup \{\kappa_i\}$
**return** $\arg\min_{\kappa_i \in \mathcal{K}} |\kappa_i|$
```

Lemma 15. *Given $G'_\alpha = \Psi(G_\alpha, t_m)$ and a valid* Token Computation *schedule S for G'_α where $|S| < 3t_m$, $t_c = 1$ and $\epsilon \in (0, 1]$,* DSFromSchedule *outputs in polynomial time a dominating set of G of size $\frac{\epsilon}{\Delta}(|S| - 2t_m - \Delta)$.*[13]

[12] Since the max degree of G and G_α are the same, throughout this section Δ will be used to refer to both the max degree of G and the max degree of G_α.

[13] Since this lemma allows for $\epsilon \in (0, 1]$, it may appear that we will be able to achieve an arbitrarily good approx-

Proof. Polynomial runtime is trivial, so we focus on the size guarantee. By Lemma 13 we know that $\kappa = \{v : v \in G_\alpha,\ v \text{ sends to } a \text{ in } S\}$ is a dominating set of G_α of size $|S| - 2t_m - \Delta$. Moreover, notice that $\kappa_i = \kappa \cap G_i$, and so it follows that κ_i is a dominating set of G_i, or equivalently G because G_i is just a copy of G. Thus we have that $\arg\min_{\kappa_i \in \mathcal{K}} |\kappa_i|$ will return a dominating set of G.

We now prove that $\arg\min_{\kappa_i \in \mathcal{K}} |\kappa_i|$ is small. Since each κ_i is disjoint we have $\sum_{i=1}^{\Delta/\epsilon} |\kappa_i| = |\kappa| \leq |S| - 2t_m - \Delta$. Thus, by an averaging argument we have that there must be some κ_i such that $|\kappa_i| \leq \frac{\epsilon}{\Delta}(|S| - 2t_m - \Delta)$. It follows that $\min_{\kappa_i \in \mathcal{K}} |\kappa_i| \leq \frac{\epsilon}{\Delta}(|S| - 2t_m - \Delta)$, meaning the κ_i that our algorithm returns is not only a dominating set of G but of size at most $\frac{\epsilon}{\Delta}(|S| - 2t_m - \Delta)$. □

Lastly, we combine Ψ with DSFromSchedule to get MDSApx, our algorithm for approximating minimum dominating set. Roughly, MDSApx constructs G'_α by applying Ψ to G_α, uses a $(1.5 - \epsilon)$ approximation to Token Computation to get a schedule to G'_α and then uses DSFromSchedule to extract a minimum dominating set for G from this schedule. MDSApx will carefully choose a t_m that is large enough so that the schedule produced by the $(1.5 - \epsilon)$ approximation for Token Computation is of length $< 3t_m$ but also small enough so that the produced schedule can be used to recover a small dominating set.

Algorithm 4 MDSApx

```
Input: Graph $G$; $(1.5 - \epsilon)$ Token Computation approximation algorithm $\mathcal{A}$
Output: An $O(1/\epsilon)$-approximation for the minimum dominating set of $G$
$\mathcal{D} \leftarrow \emptyset$
for $\hat{k} \in [n]$ do
    $G_\alpha \leftarrow \bigcup_{i=1}^{\Delta/\epsilon} G_i$
    $t_m \leftarrow \frac{1}{\epsilon}\left(\Delta + \frac{\hat{k}\Delta}{\epsilon}\right) + 1$; $t_c \leftarrow 1$
    $G'_\alpha \leftarrow \Psi(G_\alpha, t_m)$
    $S_{\hat{k}} \leftarrow \mathcal{A}\left(G'_\alpha, \frac{\hat{k}}{\epsilon}, t_m, t_c\right)$
    if $|S_{\hat{k}}| < 3t_m$ then
        $\kappa_{\hat{k}} \leftarrow$ DSFromSchedule$(G_\alpha, S, \epsilon)$
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{\kappa_{\hat{k}}\}$
return $\arg\min_{\kappa \in \mathcal{D}} |\kappa|$.
```

Lemma 16. *Given graph G and a $(1.5 - \epsilon)$-approximation algorithm for* Token Computation*, $\mathcal{A}$,* MDSApx *outputs in* $\operatorname{poly}\left(n, \frac{1}{\epsilon}\right)$ *time a dominating set of G of size $O\left(\frac{k^*}{\epsilon}\right)$, where k^* is the size of the minimum dominating set of G.*

Proof. By Lemma 15 we know that any set $\kappa_{\hat{k}} \in \mathcal{D}$ is a dominating set of G of size at most $\frac{\Delta}{\epsilon}\left(|S_{\hat{k}}| - 2t_m - \Delta\right)$. Thus, it suffices to show that $\mathcal{D}$ contains at least one dominating set of G, $\kappa_{\hat{k}}$ such that $S_{\kappa_{\hat{k}}}$ is small. We do so now.

Let k^* be the size of the minimum dominating set of G. We know that $k^* \leq n$ and so in some iteration of MDSApx we will have $\hat{k} = k^*$. Moreover, the minimum dominating set of G_α in this

imation for minimum dominating set. In fact, it might even seems as though we can produce a dominating set of size smaller than the minimum dominating set by simply letting ϵ be arbitrarily small. However, this is not the case. Intuitively, the smaller ϵ is, the larger G_α is and so the longer any feasible schedule S must be. Thus, decreases in ϵ are balanced out by increases in $|S|$ with respect to the size of our dominating set, $\frac{\epsilon}{\Delta}(|S| - 2t_m - \Delta)$.

iteration just is $\frac{\Delta k^*}{\epsilon}$ since G_α is just $\frac{\Delta}{\epsilon}$ copies of G. Consider this iteration. Let S^* be the optimal schedule for G'_α when $\hat{k} = k^*$. By Lemma 12 we know that $|S^*| \leq 2t_m + \Delta + \frac{k^*\Delta}{\epsilon}$. We now leverage the fact that that we chose t_m to be *large enough* so that $|S^*| < 3t_m$. In particular, combining the fact that $|S^*| \leq 2t_m + \Delta + \frac{k^*\Delta}{\epsilon}$ with the fact that $\mathcal{A}$ is a $(1.5 - \epsilon)$ approximation we have that

$$\begin{aligned}
|S_{k^*}| &\leq (1.5 - \epsilon)|S^*| \\
&\leq (1.5 - \epsilon)\left(2t_m + \Delta + \frac{k^*\Delta}{\epsilon}\right) \\
&= 3t_m - 2\,\epsilon\, t_m + (1.5 - \epsilon)\left(\Delta + \frac{k^*\Delta}{\epsilon}\right) \\
&= 3t_m - 2\,\epsilon\, t_m + (1.5 - \epsilon)\,\epsilon\,(t_m - 1) && \text{(By } t_m \text{ dfn.)} \\
&= 3t_m - \epsilon(0.5 + \epsilon)t_m - \epsilon(1.5 - \epsilon) \\
&< 3t_m. && (1)
\end{aligned}$$

Thus, since $|S_{k^*}| < 3t_m$ we know that $\kappa_{k^*} \in \mathcal{D}$. Lastly, we argue that $|\kappa_{k^*}| = O\left(\frac{k^*}{\epsilon}\right)$, thereby showing that $\arg\min_{\kappa \in \mathcal{D}} |\kappa|$, the returned dominating set of our algorithm, is $O\left(\frac{k^*}{\epsilon}\right)$.

We now leverage the fact that we chose t_m to be *small enough* to give us a small dominating set. Applying Lemma 15 we have that

$$\begin{aligned}
|\kappa_{k^*}| &\leq \frac{\epsilon}{\Delta}\left(|S_{k^*}| - 2t_m - \Delta\right) && \text{(By Lemma 15)} \\
&< \frac{\epsilon}{\Delta}(t_m - \Delta) && \text{(By Equation (1))} \\
&= \frac{\epsilon}{\Delta}\left(\frac{1}{\epsilon}\left(\Delta + \frac{k^*\Delta}{\epsilon}\right) + 1 - \Delta\right) && \text{(By } t_m \text{ dfn.)} \\
&= \left(1 + \frac{k^*}{\epsilon}\right) + \frac{\epsilon}{\Delta} - \epsilon \\
&= O\left(\frac{k^*}{\epsilon}\right)
\end{aligned}$$

Thus, we conclude that MDSApx produces an $O\left(\frac{k^*}{\epsilon}\right)$ minimum dominating set of G.

Lastly, we argue a polynomial in n and $1/\epsilon$ runtime of MDSApx. First we argue that each iteration requires polynomial time. Constructing G_α takes polynomial time since the algorithm need only create $\frac{\Delta}{\epsilon} = \text{poly}\left(n, \frac{1}{\epsilon}\right)$ copies of G. Running Ψ also requires polynomial time since it simply adds polynomially many nodes to G_α. $\mathcal{A}$ is polynomial by assumption and DSFromSchedule is polynomial by Lemma 15. Thus, each iteration takes polynomial time and since MDSApx has n iterations, MDSApx takes polynomial time in n and $1/\epsilon$. □

Given that MDSApx demonstrates an efficient approximation for minimum dominating set given a polynomial-time $(1.5 - \epsilon)$ approximation for Token Computation, we conclude our hardness of approximation.

Theorem 2. Token Computation *cannot be approximated by a polynomial-time algorithm within* $(1.5 - \epsilon)$ *for* $\epsilon \geq \frac{1}{o(\log n)}$ *unless* $P = NP$.

Proof. Assume for the sake of contradiction that P $\neq$ NP and there existed a polynomial-time algorithm $\mathcal{A}$ that approximated Token Computation within $(1.5 - \epsilon)$ for $\epsilon = \frac{1}{o(\log n)}$. It follows by Lemma 16 that MDSApx when run with $\mathcal{A}$ is a $o(\log n)$-approximation for minimum dominating set. However, this contradicts Lemma 14, and so we conclude that Token Computation cannot be approximated within $(1.5 - \epsilon)$ for $\epsilon \geq \frac{1}{o(\log n)}$. □

F Omitted Lemmas of the Proof of Theorem 3

F.1 Proof of Lemma 6

The goal of this section is to prove Lemma 6, which states the properties of GetDirectedPaths. To this end we will begin by rigorously defining the LP we use for GetDirectedPaths and establishing its relevant properties. We then formally define GetDirectedPaths, establish the properties of its subroutines and then prove Lemma 6.

F.1.1 Our Flow LP

The flow LP we use for GetDirectedPaths can be thought of as flow on a graph G "time-expanded" by the maximum length that a token in the optimal schedule travels. Given any schedule we define the *distance* that singleton token a travels as the number of times any token containing a is sent in said schedule. Let L^* be the furthest distance a singleton token travels in the optimal schedule. Given a guess for L^*, namely $\hat{L}$, we define a graph $G_{\hat{L}}$ with vertices $\{v_r : v \in V, r \in [\hat{L}]\}$ and edges $\{e = (u_r, v_{r+1}) : (u, v) \in E, r \in [\hat{L} - 1]\}$. We have a flow type for each $w \in W$, where $W = \{v : v \text{ has at least 1 token}\}$, which uses $\{w' : w' \in W \wedge w' \neq w\}$ as sinks. Correspondingly, we have a flow variable, $f_w(x_r, y_{r+1})$ for every $r \in [\hat{L} - 1]$, $w \in W$ and $(x, y) \in E$. The objective function of the LP is to minimize the maximum vertex congestion, given by variable z. Let $z(\hat{L})$ be the objective value of our LP given our guess $\hat{L}$. Formally, our LP is given in PathsFlowLP($\hat{L}$), where $\Gamma(v)$ gives the neighbors of v in G. See Figure 8 for an illustration of a feasible solution to this LP.

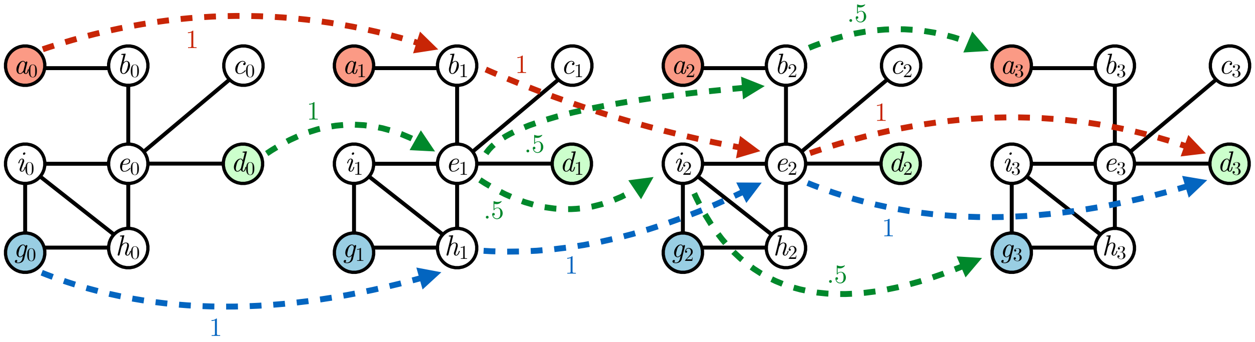


Figure 8: An illustration of non-zero flows for a feasible solution for PathsFlowLP(3) for graph G. Nodes a, d, and g are in W, and f_w is colored by w. For this feasible solution, $z = 2$.

$\min z$ s.t. (PathsFlowLP($\hat{L}$))

"Conserve flow across rounds"

$$\sum_{x' \in \Gamma(x)} f_w(x'_{r-1}, x_r) = \sum_{x'' \in \Gamma(x)} f_w(x_r, x''_{r+1}) \qquad \forall w \in W, x \notin W, r \in [\hat{L} - 1] \qquad (2)$$

"Every $w \in W$ is a source for f_w and not a sink for f_w"

$$\sum_{r \in [\hat{L}-1]} \left[\sum_{x' \in \Gamma(w)} f_w(w_r, x'_{r+1}) - \sum_{x' \in \Gamma(w)} f_w(x'_r, w_{r+1}) \right] \geq 1 \qquad \forall w \in W \qquad (3)$$

"w-flow ends at $w' \in W$ s.t. $w' \neq w$"

$$\sum_{w' \in W : w' \neq w} \sum_{u \in \Gamma(w')} f_w(u_{\hat{L}}, w'_{\hat{L}}) = 1 \qquad \forall w \qquad (4)$$

"z is the vertex congestion"

$$z \geq \sum_{w} \sum_{v \in \Gamma(v)} \sum_{r \in [D-1]} f_w(v'_r, v_{r+1}) \qquad \forall v \qquad (5)$$

"Non-negative flow"

$$f_w(x_r, y_{r+1}) \geq 0 \qquad \forall, x, y, r, w \in W \qquad (6)$$

F.1.2 Proof of the Key Property of our LP

The key property of our LP is that it has an optimal vertex congestion comparable to OPT. In particular, we can produce a feasible solution for our LP of cost 2OPT by routing tokens along the paths taken in the optimal schedule.

Lemma 17. $\min(t_c, t_m) \cdot z(2L^*) \leq 2\text{OPT}$.

The remainder of this section is a proof of Lemma 17. Consider a W as in Section 4.2.1 where $W \leftarrow \{v : v \text{ has at least 1 token}\}$ and the optimal schedule that solves Token Computation in time OPT.

We will prove Lemma 17 by showing that, by sending flow along paths taken by certain tokens in the optimal schedule, we can provide a feasible solution to PathsFlowLP($\hat{L}$) with value commensurate with OPT. For this reason we now formally define these paths, $\textsc{OptPaths}(W)$. Roughly, these are the paths taken by tokens containing singleton tokens that originate in W. Formally, these paths are as follows. Recall that a_w is the singleton token with which node w starts in the optimal schedule. Notice that in any given round of the optimal schedule exactly one token contains a_w. As such, order every round in which a token containing a_w is received by a node in ascending order as $r_0(w), r_1(w) \ldots$ where we think of w as receiving a_w in the first round. Correspondingly, let $v_i(w)$ be the vertex that receives a token containing a_w in round $r_i(w)$; that is $(v_1(w), v_2(w), \ldots)$ is the path "traced out" by a_w in the optimal schedule. For token a, let $C(a) \coloneqq \{a_{w'} : w' \in W \wedge a'_w \in a\}$ stand for all singleton tokens contained by token a that originated at a $w' \in W$. Say that token a is *active* if $|C(a)|$ is odd. Let $v_{L_w}(w)$ be the first vertex in $(v_1(w), v_2(w), \ldots)$ where an active token containing a_w is combined with another active token. Correspondingly, let $c(w)$ be the first round in which an active token containing a_w is combined with another active token. Say that a singleton token a_w is *pending* in round r if $r < c(w)$. We note the following behavior of pending singleton tokens.

Lemma 18. *In every round of the optimal schedule, if a token is active then it contains exactly one pending singleton token and if a token is inactive then it contains no pending singleton tokens.*

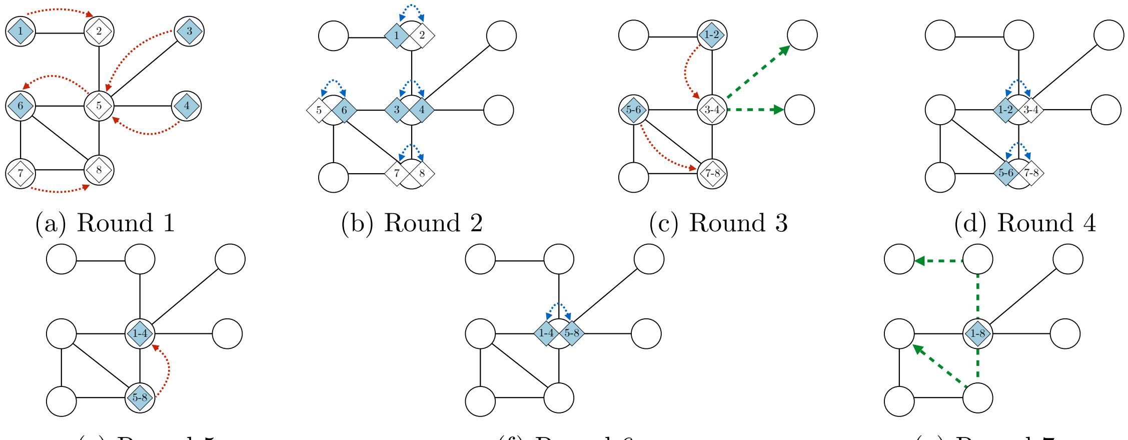
Proof. We prove this by induction over the rounds of the optimal schedule. As a base case, we note that in the first round of the optimal schedule a token is active iff it is a singleton node and every singleton node is pending. Now consider an arbitrary round i and assume that our claim holds in previous rounds. Consider an arbitrary token a. If a is not computed on by a node in this round then by our inductive hypothesis we have that it contains exactly one pending singleton token if it is active and no pending singleton tokens if it is not active. If a is active and combined with an inactive token, by our inductive hypothesis, the resulting token contains exactly one pending singleton token. Lastly, if a is active and combined with another active token by our inductive hypothesis these contain pending singletons a_w and a_u respectively such that $c(w) = c(u) = i$; it follows that the resulting token is inactive and contains no pending singleton tokens. This completes our induction. □

This behavior allows us to pair off vertices in W based on how their singleton tokens are combined.[14]

Lemma 19. *For each $w \in W$ there exists a unique $u \in W$ such that $u \neq w$ and $v_{L_w}(w) = v_{L_u}(u)$ and $c(w) = c(u)$.*

Proof. Consider the round in which a token containing a_w, say a, is combined with an active token, say b, at vertex $v_{L_w}(w)$. Recall that this round is notated $c(w)$. By Lemma 18 we know that a and b contain exactly one pending singleton token, say a_w and a_u respectively. Since both a and b are active in this round and b contains a_u we have $c(u) = c(w)$. Moreover, since both a and b are combined at the same vertex we have $v_{L_u}(u) = v_{L_w}(w)$. Lastly, notice that this u is unique since by Lemma 18 there is exactly one singleton token, a_u, contained by b such that $c(u) \leq c(w)$. □

[14] Without loss of generality we assume that $|W|$ is even here; if not, we can simply drop one element from W each time we construct $\textsc{OptPaths}(W)$.



(a) Round 1 (b) Round 2 (c) Round 3 (d) Round 4

(e) Round 5 (f) Round 6 (g) Round 7

Figure 9: An illustration of the optimal schedule and how $\textsc{OptPaths}(W)$ is constructed from it for a particular G. Active tokens are denoted by blue diamonds; inactive tokens are denoted by white diamonds; a dotted red arrow from node u to node v means that u sends to v; a double-ended blue arrow between two tokens a and b means that a and b are combined at the node; thick, dashed green lines give a path and its reversal in $\textsc{OptPaths}(W)$ (for a total of 4 paths across all rounds) where $(v_1(w), v_2(w), \dots v_{L_w}(w) = v_{L_u}(u), v_{L_u-1}(u), \dots, v_1(u)) = P \in \textsc{OptPaths}(\mathrm{W})$ drawn only in round $c(w)$. Furthermore, token a labeled with $\{v : a \text{ contains } a_v\}$ and $W = \{1, 3, 4, 6\}$.

Having paired off vertices in W, we can now define $\textsc{OptPaths}(W)$. Fix a w and let u be the vertex it is paired off with as in Lemma 19. We define $\textsc{OptPath}(w) \coloneqq (v_1(w), v_2(w), \dots v_{L_w}(w) = v_{L_u}(u), v_{L_u-1}(u), \dots, v_1(u))$. Lastly, define $\textsc{OptPaths}(W) = \bigcup_{w \in W} \textsc{OptPath}(w)$. See Figure 9 for an illustration of how $\textsc{OptPaths}(W)$ is constructed from the optimal schedule.

The critical property of $\textsc{OptPaths}(W)$ is that it has vertex congestion commensurate with OPT as follows.

Lemma 20. *con*$(\textsc{OptPaths}(W)) \leq \frac{2 \cdot \mathrm{OPT}}{\min(t_c, t_m)}$.

Proof. Call a pair of directed paths in $\textsc{OptPaths}(W)$ *complementary* if one path is $\textsc{OptPath}(w)$ and the other $\textsc{OptPath}(u)$ where u is to w as in Lemma 19. We argue that each pair of complementary paths passing through a given vertex v uniquely account for either t_c or t_m rounds of v's OPT rounds in the optimal schedule. Consider a pair of complementary paths, $P = (\textsc{OptPath}(w), \textsc{OptPath}(u))$, passing through a given vertex v. This pair of paths pass through v because in some round, say r_P, v sends a token containing a_u or a_w or v combines together tokens a and a' containing a_u and a_w respectively. Say that whichever of these operations accounts for P is *responsible* for P. Now suppose for the sake of contradiction that this operation of v in round r_P is responsible for another distinct pair P' of complementary paths, $\textsc{OptPath}(w')$ and $\textsc{OptPath}(u')$. Notice that a_w, $a_{w'}$, a_u and $a_{u'}$ are all pending in round r_P. We case on whether v's action is a communication or a computation and show that v's operation cannot be responsible for P' in either case.

- Suppose that v is responsible for P and P' because it performs a computation in r_P. It follows that v combines an active token a and another active token a' where without loss of generality $a_w, a_{w'} \in a$ and $a_{u'}, a_u \in a'$. However, it then follows that a is active and contains two pending singleton tokens, which contradicts Lemma 18.

- Suppose that v is responsible for P and P' because it performs a communication in r_P by sending token a. It follows that without loss of generality $a_w, a_{w'} \in a$. However, either a is active or it is not. But by Lemma 18 if a is active it contains 1 pending singleton token and if a is not active then it contains 0 pending singleton tokens. Thus, the fact that v sends a token containing two pending singleton tokens contradicts Lemma 18.

Thus, it must be the case that v's action in r_P is uniquely responsible for P.

It follows that each computation and communication performed by v uniquely corresponds to a pair of complementary paths (consisting of a pair of paths in OptPaths(W)) that passes through v. Since v performs at most $\text{OPT}/\min(t_c, t_m)$ operations in the optimal schedule, it follows that there are at most $\text{OPT}/\min(t_c, t_m)$ pairs of complementary paths in OptPaths(W) incident to v. Since each pair consists of two paths, there are at most $2 \cdot \text{OPT}/\min(t_c, t_m)$ paths in OptPaths(W) incident to v and so v has vertex congestion at most $2 \cdot \text{OPT}/\min(t_c, t_m)$ in OptPaths(W). Since v was arbitrary, this bound on congestion holds for every vertex. □

We now use OptPaths(W) to construct a feasible solution for PathsFlowLP($2L^*$). We let $\tilde{f}$ be this feasible solution. Intuitively, $\tilde{f}$ simply sends flow along the paths of OptPaths(W). More formally define $\tilde{f}$ as follows. For $w \in W$ and its corresponding path OptPath(w) $= (v_1(w), v_2(w), \ldots)$ we set $\tilde{f}_w(v_i, v_{i+1}) = 1$. We set all other variables of $\tilde{f}$ to 0 and let $\tilde{z}$ be the vertex congestion of OptPaths(W).

Lemma 21. *$(\tilde{f}, \tilde{z})$ is a feasible solution for* PathsFlowLP($2L^*$) *where $\tilde{z} \le 2\text{OPT}/\min(t_c, t_m)$.*

Proof. We begin by noting that every path in OptPaths(W) is of length at most $2L^*$: for each $w \in W$, OptPath(w) is the concatenation of two paths, each of which is of length no more than L^*. Moreover, notice that for each $w \in W$, the sink of OptPath(w) is a $w' \in W$ such that $w' \neq w$.

We now argue that $(\tilde{f}, \tilde{z})$ is a feasible solution for PathsFlowLP($2L^*$): each vertex v with incoming w-flow that is not in $W \setminus w$ sends out this unit of flow and so Equation (2) is satisfied; since each OptPath(w) is of length at most $2L^*$ and ends at a $w' \in W$ we have that every $w \in W$ is a source for f_w and not a sink for f_w, satisfying Equation (3); for the same reason, Equation (4) is satisfied; letting $\tilde{z}$ be the vertex congestion of OptPaths(W) clearly satisfies Equation (5); and flow is trivially non-zero.

Lastly, since $\tilde{f}$ simply sends one unit of flow along each path in OptPaths(W), our bound of $\tilde{z} \le 2\text{OPT}/\min(t_c, t_m)$ follows immediately from Lemma 20. □

We conclude that $\tilde{f}$ demonstrates that our LP has value commensurate with OPT.

Lemma 17. $\min(t_c, t_m) \cdot z(2L^*) \le 2\text{OPT}$.

Proof. Since *Lemma* 21 shows that $(\tilde{f}, \tilde{z})$ is a feasible solution for PathsFlowLP($2L^*$) with cost at most $2\text{OPT}/\min(t_c, t_m)$, our claim immediately follows. □

F.1.3 GetDirectedPaths Formally Defined

GetDirectedPaths solves our LP for different guesses of the longest path used by the optimal, samples paths based on the LP solution for our best guess, and then directs these paths. Formally, GetDirectedPaths is given in Algorithm 5, where $\xi := \lceil 2(n-1) \cdot (t_c + D \cdot t_m)/t_m \rceil$ is the range over which we search for L^*.

Algorithm 5 GetDirectedPaths(G, W)

```
**Input:** $W \subseteq V$ where $w \in W$ has a token
**Output:** Directed paths between nodes in $W$
$L \leftarrow \arg\min_{\hat{L} \in [\xi]} \left[ t_m \cdot \hat{L} + \min(t_c, t_m) \cdot t(\hat{L}) \right]$
$f_w^* \leftarrow$ PathsFlowLP$(L)$
$\mathcal{P}_W \leftarrow$ SampleLPPaths$(f_w^*, L, W)$
$\vec{\mathcal{P}_U} \leftarrow$ AssignPaths$(\mathcal{P}_W, W)$
**return** $\vec{\mathcal{P}_U}$
```

F.1.4 Sampling Paths from LP

Having shown that our LP has value commensurate with OPT and defined our algorithm based on this LP, we now provide the algorithm which we use to sample paths from our LP solution, SampleLPPaths. This algorithm produces a single sample by taking a random walk from each $w \in W$ where edges are taken with probability corresponding to their LP value. It repeats this $O(\log n)$ times to produce $O(\log n)$ samples. It then takes the sample with the most low congestion paths, discarding any high congestion paths in said sample. In particular, SampleLPPaths takes the sample $\mathcal{P}_W^i$ that maximizes $|Q(\mathcal{P}_W^i)|$ where $Q(\mathcal{P}_W^i) = \{P_w : P_w \in \mathcal{P}_W^i, \mathrm{con}(P_w) \leq 10 \cdot z(\hat{L}) \log \hat{L}\}$ for an input $\hat{L}$.

Algorithm 6 SampleLPPaths(f_w^*)

```
**Input:** $f_w^*$, solution to PathsFlowLP$(\hat{L})$; $\hat{L}$, guess of $L^*$; $W \subseteq V$
**Output:** Undirected paths between nodes in $W$
$\mathcal{C} \leftarrow \emptyset$
**for** sample $i \in O(\log n)$ **do**
    $\mathcal{P}_W^i \leftarrow \emptyset$
    **for** $w \in W$ **do**
        $v \sim f_w^*(w_1, v_2)$
        $P_w \leftarrow (w, v)$
        **while** $v \notin W$ **do**
            $v' \sim f_w^*(v_{|P_w|}, v'_{|P_w|+1})$
            $v \leftarrow v'$
            $P_w + = v$
        $\mathcal{P}_W^i \leftarrow \mathcal{P}_W^i \cup \{P_w\}$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{P}_W^i$
$\mathcal{P}_S \leftarrow Q(\arg\max_{\mathcal{P}_W^i \in \mathcal{C}} |Q(\mathcal{P}_W^i)|)$
**return** $\mathcal{P}_W$
```

The properties of SampleLPPaths are as follows.

Lemma 22. *For any fixed $W \subseteq V$, $\hat{L}$ and an optimal solution f_w^* to PathsFlowLP($\hat{L}$),* SampleLPPaths *is a polynomial-time randomized algorithm that outputs a set of undirected paths $\mathcal{P}_W$ such that $P_w \in \mathcal{P}_W$ is an undirected path with endpoints $w, w' \in S$ where $w \neq w'$. Also $|\mathcal{P}_W| \geq \frac{1}{3}|W|$ w.h.p., con$(\mathcal{P}_W) \leq z(\hat{L}) \cdot O(\log \hat{L})$, and dil$(\mathcal{P}_W) \leq \hat{L}$.*

Proof. Our proof consists of a series of union and Chernoff bounds over our samples. Consider an arbitrary $W \subseteq V$. Define T_w for $w \in W$ as the (directed) subgraph of $G_{\hat{L}}$ containing arc $(v, u) \in E_{\hat{L}}$ if for some r we have $f_w^*(x, y) > 0$. Notice that T_w is a weakly connected DAG where w has no edges into it: T_w does not contain any cycles since flow only moves from x_r to y_{r+1} for $x, y \in V$; by our flow constraints T_w must be weakly connected and w_r must have no edges into it for any r. Moreover, notice that P_w is generated by a random walk on T_w starting at w_1, where if the last vertex added to P_w was v, then we add u to P_w in step r of the random walk with probability $f_w^*(v_r, u_{r+1})$.

We first argue that every $P_w \in \mathcal{P}_W$ has endpoints $w, w' \in W$ for $w \neq w'$ and $\text{dil}(\mathcal{P}_W) \leq \hat{L}$. By construction, one endpoint of P_w is w. Moreover, the other endpoint of P_w will necessarily be a $w' \in W$ such that $w' \neq w$: by Equation (2) flow is conserved and by Equation (4) all flow from w must end at a point $w' \in W$ such that $w' \neq w$; thus our random walk will always eventually find such an w'. Moreover, notice that our random walk is of length at most $\hat{L}$ since T_w is of depth at most $\hat{L}$. Thus, every P_w is of length at most $\hat{L}$, meaning $\text{dil}(\mathcal{P}_W) \leq \hat{L}$.

Next, notice that, by the definition of Q, $\text{con}(\mathcal{P}_W) \leq z(\hat{L}) \cdot O(\log \hat{L})$ by construction since every element in $Q(\arg\max_{\mathcal{P}_W^i \in \mathcal{C}} |Q(\mathcal{P}_W^i)|)$ has $O(z(\hat{L}) \cdot O(\log \hat{L}))$ congestion.

Thus, it remains only to prove that $|\mathcal{P}_W| \geq \frac{1}{3}|W|$. We begin by arguing that for a fixed path P_w in a fixed set of sampled paths, $\mathcal{P}_W^i$ we have $\text{con}(P_w) \geq z(\hat{L}) \cdot O(\log \hat{L})$ with probability at most $\frac{1}{3}$. Consider a fixed path $P_w \in \mathcal{P}_W^i$ and fix an arbitrary $v \in P_w$. Now let X_{wv} stand for the random variable indicating the number of times that path P_w visits vertex w. without loss of generality we know that P_w contains no cycles (since if it did we could just remove said cycles) and so X_{sv} is either 1 or 0. By a union bound over rounds, then, we have $\mathbb{E}[X_{wv}] \leq \sum_r \sum_{u \in \Gamma(v)} f_W^*(u_r, v_{r+1}) \cdot \Pr(u \text{ taken in } (r-1)\text{th step}) \leq \sum_{u \in \Gamma(v)} \sum_r f_W^*(u_r, v_{r+1})$.

Now note that the congestion of a single vertex under our solution is just $\text{con}(v) = \sum_{w \in W} X_{wv}$. It follows that

$$\mathbb{E}[\text{con}(v)] = \sum_{w \in W} \mathbb{E}[X_{wv}] \leq \max_v \sum_w \sum_{u \in \Gamma(v)} \sum_r f_W^*(u_r, v_{r+1}) \leq z(\hat{L}).$$

Also notice that for a fixed v every X_{wv} is independent. Thus, we have by a Chernoff bound that that

$$\begin{aligned} \Pr(\text{con}(v) \geq z(\hat{L}) \cdot O(\log \hat{L})) &\leq \Pr\left(\sum_{w \in W} X_{wv} \geq \mathbb{E}\left[\sum_{w \in W} X_{wv}\right] \cdot O(\log \hat{L})\right) \\ &\leq \frac{1}{(\hat{L})^c} \end{aligned} \tag{7}$$

for c given by constants of our choosing. P_w is of length at most $\hat{L}$ by construction. Thus, by a union over $v \in P_w$ and Equation (7) we have that

$$\begin{aligned} \Pr\left(\text{con}(P_w) \geq z(\hat{L}) \cdot O(\log \hat{L})\right) &\leq \frac{1}{\hat{L}^{c-1}} \\ &\leq \frac{1}{3}. \end{aligned}$$

Thus, for a fixed path $P_w \in \mathcal{P}_W^i$ we know that this path has congestion at least $z(\hat{L}) \cdot O(\log \hat{L}))$ with probability at most $\frac{1}{3}$.

We now argue at least one of our $O(\log n)$ samples is such that at least $\frac{1}{3}$ of the paths in the sample have congestion at most $z(\hat{L}) \cdot O(\log \hat{L}))$. Let Y_{iw} be the random variable that is 1 if $P_w \in \mathcal{P}_W^i$ is such that $\text{con}(P_w) \geq z(\hat{L}) \cdot O(\log \hat{L}))$ and 0 otherwise. Notice that $\mathbb{E}[Y_{iw}] \leq \frac{1}{3}$ by the fact that a path has congestion at least $z(\hat{L}) \cdot O(\log \hat{L})$ with probability at most $\frac{1}{3}$. Now let $Z_i = \sum_{w \in W} Y_{iw}$ stand for the number of paths in sample i with high congestion. By linearity of expectation we have $\mathbb{E}[Z_i] \leq |W|\frac{1}{3}$. By Markov's inequality we have for a fixed i that $\Pr(Z_i \geq \frac{2}{3}|W|) \leq \Pr(Z_i \geq 2\,\mathbb{E}[Z_i]|W|) \leq \frac{1}{2}$. Now consider the probability that every sample i is such that more than $\frac{2}{3}$ of the paths have congestion more than $z(\hat{L}) \cdot O(\log \hat{L})$, i.e. consider the probability that for all i we have $Z_i \geq |W|\frac{2}{3}$. We have

$$\begin{aligned}\Pr\left(Z_i \geq |W|\frac{2}{3}, \forall i\right) &\leq \left(\frac{1}{2}\right)^{O(\log n)} \\ &= \frac{1}{\text{poly}(n)}.\end{aligned}$$

Thus, with high probability there will be some sample, i, such that $Z_i \leq |W|\frac{2}{3}$. It follows that with high probability $\max_{\mathcal{P}_W^i \in \mathcal{C}} |Q(\mathcal{P}_W^i)| \geq \frac{1}{3}|W|$ and since $\mathcal{P}_W = Q(\arg\max_{\mathcal{P}_W^i \in \mathcal{C}} |Q(\mathcal{P}_W^i)|)$, we conclude that with high probability $\mathcal{P}_W \geq \frac{1}{3}|W|$.

□

F.1.5 Directing Paths

Given the undirected paths that we sample from our LP, $\mathcal{P}_W$, we produce a set of directed paths $\vec{\mathcal{P}}_U$ using AssignPaths, which works as follows. Define G' as the directed supergraph consisting of nodes W and directed edges $E' = \{(w, w') : w' \text{ is an endpoint of } P_w \in \mathcal{P}_W)\}$. Let $\Gamma_{G'}(v) = \{v' : (v', v) \in E' \vee (v, v') \in E'\}$ give the neighbors of v in G'. For each node $w \in G'$ with in-degree of at least two we do the following: if v has odd degree delete an arbitrary neighbor of w from G'; arbitrarily pair off the neighbors of w; for each such pair (w_1, w_2) add the directed path $P_{w_1} \circ rev(P_{w_2})$ to $\vec{\mathcal{P}}_U$ where $rev(P_{w_2})$ gives the result of removing the last element of P_{w_2} (namely, w) and reversing the direction of the path; remove $\{w, w_1, w_2\}$ from G'. Since we remove all vertices with in-degree of two or more and every vertex has out-degree 1, the remaining graph trivially consists only of nodes with in-degree at most 1 and out-degree at most 1. The remaining graph, therefore, is all cycles and paths. For each cycle or path $w_1, w_2, w_3, \ldots$ add the path corresponding to the edge from w_i to w_{i+1} for odd i to $\vec{\mathcal{P}}_U$. We let U be all sources of paths in $\vec{\mathcal{P}}_U$ and we let P_u be the path in $\vec{\mathcal{P}}_U$ with source u.

The properties of AssignPaths are as follows.

Lemma 23. *Given $W \subseteq V$ and $\mathcal{P}_W = \{P_w : w \in W\}$ where the endpoints of P_w are $w, w' \in W$ for $w \neq w'$,* AssignPaths *in polynomial-time returns directed paths $\vec{\mathcal{P}}_U$ where at least $1/4$ of the nodes in W are the source of a directed path in $\vec{\mathcal{P}}_U$, each path in $\vec{\mathcal{P}}_U$ is of length at most $2 \cdot dil(\mathcal{P}_W)$ with congestion at most $con(\mathcal{P}_W)$ and each path in $\vec{\mathcal{P}}_U$ ends in a unique sink in W.*

Proof. When we add paths to $\vec{\mathcal{P}}_U$ that go through vertices of in-degree at least two, for every 4 vertices we remove we add at least one directed path to $\vec{\mathcal{P}}_U$ that is at most double the length of the longest a path in $\mathcal{P}_U$: in the worst case v has odd in-degree of 3 and we add only a single path. When we do the same for our cycles and paths for every 3 vertices we remove we add at least one directed path to $\vec{\mathcal{P}}_U$. Notice that by construction we clearly never reuse sinks in our directed paths. The bound on congestion and a polynomial runtime are trivial. □

F.1.6 Proof of Lemma 6

Finally, we conclude with the proof of Lemma 6.

Lemma 6. *Given $W \subseteq V$,* GetDirectedPaths *is a randomized polynomial-time algorithm that returns a set of directed paths, $\vec{\mathcal{P}_U} = \{P_u : u \in U\}$ for $U \subseteq W$, such that with high probability at least $1/12$ of nodes in W are sources of paths in $\vec{\mathcal{P}_U}$ each with a unique sink in W. Moreover,*

$$\textit{con}(\vec{\mathcal{P}_U}) \leq O\left(\frac{\text{OPT}}{\min(t_c, t_m)} \log \frac{\text{OPT}}{t_m}\right) \quad \textit{and } \textit{dil}(\vec{\mathcal{P}_U}) \leq \frac{8\text{OPT}}{t_m}.$$

Proof. The fact that GetDirectedPaths returns a set of directed paths, $\vec{\mathcal{P}_U}$, such that at least $1/12$ of nodes in W are sources in a path with a sink in W follows directly from Lemma 22 and Lemma 23.

We now give the stated bounds on congestion and dilation. First notice that $2L^* \in [\xi]$. Moreover, $2\text{OPT} \leq 2(n-1)(t_c + D \cdot t_m)$: the schedule that picks a pair of nodes, routes one to the other then aggregates and repeats $n-1$ times is always feasible and takes $(n-1)(t_c + D \cdot t_m)$ rounds. Thus, $2L^* \leq 2\frac{\text{OPT}}{t_m} \leq \xi$.

Thus, by definition of L we know that

$$\begin{aligned}
t_m \cdot L + \min(t_c, t_m) \cdot t(L) &\leq 2t_m \cdot L^* + \min(t_c, t_m) \cdot z(2L^*) \\
&\leq 2L^* + 2\text{OPT} && \text{(By Lemma 17)} \\
&\leq 4\text{OPT} && \text{(By dfn. of } L^*\text{)}
\end{aligned}$$

It follows, then, that $t_m \cdot L \leq 4\text{OPT}$ and so $L \leq \frac{4\text{OPT}}{t_m}$. Similarly, we know that $\min(t_c, t_m) \cdot z(L) \leq 4\text{OPT}$ and so $z(L) \leq \frac{4\text{OPT}}{\min(t_c, t_m)}$.

Lastly, by Lemma 22 we know that $\text{dil}(\mathcal{P}_W) \leq L \leq \frac{4\text{OPT}}{t_m}$ and $\text{con}(\mathcal{P}_W) \leq t(L) \cdot O(\log L) \leq O\left(\frac{\text{OPT}}{\min(t_c,t_m)} \cdot \log \frac{\text{OPT}}{t_m}\right)$. By Lemma 23 we get that the same congestion bound holds for $\vec{\mathcal{P}_U}$ and $\text{dil}(\vec{\mathcal{P}_U}) \leq \frac{8\text{OPT}}{\min(t_c,t_m)}$.

A polynomial runtime comes from the fact that we solve at most $(n-1)(t_c + D \cdot t_m) = \text{poly}(n)$ LPs and then sample at most $(n-1)(t_c + D \cdot t_m)$ edges $O(\log n)$ times to round the chosen LP. □

F.2 Deferred Proofs of Section 4.2.2

Lemma 7. *Given a set of directed paths $\vec{\mathcal{P}_U}$ with some subset of endpoints of paths in $\vec{\mathcal{P}_U}$ designated sources and the rest of the endpoints designated sinks,* OPTRoute *is a randomized polynomial-time algorithm that w.h.p. produces a* Token Network *schedule that sends from all sources to sinks in $O(\textit{con}(\vec{\mathcal{P}_U}) + \textit{dil}(\vec{\mathcal{P}_U}))$.*

Proof. Given a set of paths $\vec{\mathcal{P}_U}$, Rothvoß [Rot13] provides a polynomial-time algorithm that produces a schedule that routes along all paths in $O(\text{con}_E(\vec{\mathcal{P}_U}) + \text{dil}(\vec{\mathcal{P}_U})$ where $\text{con}_E(\mathcal{P}) =$

$\max_e \sum_{P \in \mathcal{P}} \mathbb{1}(e \in P)$ is the edge congestion. However, the algorithm of Rothvoß [Rot13] assumes that in each round a vertex can send a token along each of its incident edges whereas we assume that in each round a vertex can only forward a single token.

However, it is easy to use the algorithm of Rothvoß [Rot13] to produce an algorithm that produces a Token Network routing schedule using $O(\text{con}(\vec{\mathcal{P}_U}) + \text{dil}(\vec{\mathcal{P}_U}))$ rounds which assumes that vertices only send one token per round as we assume in the Token Network model as follows. Let G be our input network with paths $\vec{\mathcal{P}_U}$ along which we would like to route where we assume that vertices can only send one token per round. We will produce another graph G' on which to run the algorithm of Rothvoß [Rot13]. For each node $v \in G$ add nodes v_i and v_o to G'. Project each path $P \in \vec{\mathcal{P}_U}$ into G' to get $P' \in \vec{\mathcal{P}_U}'$ as follows: if edge (u, v) is in path $P \in \vec{\mathcal{P}_S}$ then add edge (u_o, v_i) and edge (v_i, v_o) to path P' in G'. Notice that $\text{con}(\vec{\mathcal{P}_U}) = \text{con}_E(\vec{\mathcal{P}_U}')$ and $\text{dil}(\vec{\mathcal{P}_U}) = 2\text{dil}(\vec{\mathcal{P}_U}')$. Now run the algorithm of Rothvoß [Rot13] on G' with paths $\vec{\mathcal{P}'_U}$ to get back some routing schedule S'.

Without loss of generality we can assume that S' only has nodes in G' send along a single edge in each round: every v_i is incident to a single outbound edge across all paths (namely (v_i, v_o)) and so cannot send more than one token per round; every v_o has a single incoming edge and so receives at most one token per round which, without loss of generality, we can assume v_o sends as soon as it receives (it might be the case that v_o collects some number of tokens over several rounds and then sends them all out at once but we can always just have v_o forward these tokens as soon as they are received and have the recipients "pretend" that they do not receive them until v_o would have sent out many tokens at once).

Now generate a routing schedule for G as follows: if v_o sends token a in round r of S' then v will send token a in round r of S. Since S only ever has vertices send one token per round, it is easy to see by induction over rounds that S will successfully route along all paths. Moreover, S takes as many rounds as S' which by [Rot13] we know takes $O(\text{con}(\vec{\mathcal{P}'_U}) + \text{dil}(\vec{\mathcal{P}'_U})) = O(\text{con}(\vec{\mathcal{P}_U}) + 2\text{dil}(\vec{\mathcal{P}_U})) = O(\text{con}(\vec{\mathcal{P}_U}) + \text{dil}(\vec{\mathcal{P}_U}))$. Thus, we let OPTRoute be the algorithm that returns S. □

Lemma 8. RoutePaths$_m$ *is a polynomial-time algorithm that, given* $\vec{\mathcal{P}_U}$*, solves the* Route and Compute *Problem w.h.p. using* $O(t_m(\mathit{con}(\vec{\mathcal{P}_U}) + \mathit{dil}(\vec{\mathcal{P}_U})) + t_c)$ *rounds.*

Proof. By Lemma 7, OPTRoute takes $t_m(\text{con}(\vec{\mathcal{P}_U}) + \text{dil}(\vec{\mathcal{P}_U}))$ rounds to route all sources to sinks. All sources are combined with sinks in the following computation and so RoutePaths$_m$ successfully solves the Route and Compute Problem since every source has its token combined with another token. The polynomial runtime of the algorithm is trivial. □

Lemma 9. RoutePaths$_c$ *is a polynomial-time algorithm that, given* $\vec{\mathcal{P}_U}$*, solves the* Route and Compute *Problem w.h.p. using* $O(t_c \cdot \mathit{con}(\vec{\mathcal{P}_U}) + t_m \cdot \mathit{dil}(\vec{\mathcal{P}_U}))$ *rounds.*

Proof. We argue that every source's token ends at an asleep node with at least two tokens and no more than $\text{con}(\vec{\mathcal{P}_U})$ tokens. It follows that our computation at the end at least halves the number of tokens.

First notice that if a vertex falls asleep then it will receive at most $\text{con}(\vec{\mathcal{P}_S})$ tokens by the end of our algorithm since it is incident to at most this many paths. Moreover, notice that every token will either end at a sink or a sleeping vertex and every sleeping vertex is asleep because it has two or more tokens. It follows that every token is combined with at least one other token and so our schedule at least halves the total number of tokens.

The length of our schedule simply comes from noting that we have $O(\text{dil}(\vec{\mathcal{P}_U}) \cdot t_m)$ forwarding rounds followed by $\text{con}(\vec{\mathcal{P}_U}) \cdot t_c$ rounds of computation. Thus, we get a schedule of total length $O(t_c \cdot \text{con}(\vec{\mathcal{P}_S}) + t_m \cdot \text{dil}(\vec{\mathcal{P}_S}))$. A polynomial runtime is trivial. □

F.3 Proof of Theorem 3

Theorem 3. SolveTC *is a polynomial-time algorithm that gives an* $O(\log n \cdot \log \frac{\text{OPT}}{t_m})$*-approximation for* Token Computation *with high probability.*

Proof. By Lemma 6 we know that the paths returned by GetDirectedPaths, $\vec{\mathcal{P}_U}$ are such that $\text{con}(\vec{\mathcal{P}_U}) \leq O\left(\frac{\text{OPT}}{\min(t_c,t_m)} \log \frac{\text{OPT}}{t_m}\right)$ and $\text{dil}(\vec{\mathcal{P}_U}) \leq \frac{8\text{OPT}}{t_m}$ and the paths returned have unique sinks and sources in W and there are at least $|W|/12$ paths w.h.p.

If $t_c > t_m$ then RoutePaths$_m$ is run which by Lemma 8 solves the Route and Compute Problem in $O(t_m \cdot \text{con}(\vec{\mathcal{P}_U}) + t_m \cdot \text{dil}(\vec{\mathcal{P}_U}) + t_c)$ rounds which is

$$\begin{aligned} &\leq O\left(t_m \cdot \frac{\text{OPT}}{\min(t_c, t_m)} \cdot \log \frac{\text{OPT}}{t_m} + t_m \cdot \frac{8\text{OPT}}{t_m} + t_c\right) \\ &= O\left(\text{OPT} \cdot \log \frac{\text{OPT}}{t_m} + t_c\right) \end{aligned}$$

If $t_c \leq t_m$ then RoutePaths$_c$ is run to solve the Route and Compute Problem which by Lemma 9 takes $O(t_c \cdot \text{con}(\vec{\mathcal{P}_U}) + t_m \cdot \text{dil}(\vec{\mathcal{P}_U}))$ rounds which is

$$\begin{aligned} &\leq O\left(t_c \cdot \frac{4\text{OPT}}{\min(t_c, t_m)} \cdot \log \frac{\text{OPT}}{t_m} + t_m \cdot \frac{8\text{OPT}}{t_m}\right) \\ &= O\left(\text{OPT} \cdot \log \frac{\text{OPT}}{t_m}\right) \end{aligned}$$

Thus, in either case, the produced schedule takes at most $O\left(\text{OPT} \cdot \log \frac{\text{OPT}}{t_m} + t_c\right)$ rounds to solve the Route and Compute Problem on at least $|W|/12$ paths in each iteration. Since solving the Route and Compute Problem reduces the total number of tokens by a constant fraction on the paths over which it is solved, and we have at least $|W|/12$ paths in each iteration w.h.p., by a union bound, every iteration reduces the total number of tokens by a constant fraction w.h.p. Thus, the concatenation of the $O(\log n)$ schedules produced, each of length $O(\text{OPT} \cdot \log \frac{\text{OPT}}{t_m} + t_c)$, is sufficient to reduce the total number of tokens to 1.

Thus, SolveTC produces a schedule that solves the problem of Token Computation in $O(\text{OPT} \cdot \log n \log \frac{\text{OPT}}{t_m} + t_c \cdot \log n)$ rounds. However, notice that $t_c \cdot \log n \leq \text{OPT}$ (since the optimal schedule must perform at least $\log n$ serialized computations) and so the produced schedule is of length $O(\text{OPT} \cdot \log n \log \frac{\text{OPT}}{t_m} + t_c \log n) \leq O(\text{OPT} \cdot \log n \log \frac{\text{OPT}}{t_m})$. Lastly, a polynomial runtime is trivial given the polynomial runtime of our subroutines. □