CCIT Report #589 May 2006

Efficient Dynamic Aggregation

Yitzhak Birk, Idit Keidar, Liran Liss, and Assaf Schuster
Technion – Israel Institute of Technology
{birk@ee,idish@ee,liranl@tx,assaf@cs}.technion.ac.il

June 24, 2006

Abstract

We consider the problem of *dynamic aggregation* of inputs over a large graph. A dynamic aggregation algorithm must continuously compute the result of a given aggregation function over a dynamically changing set of inputs. To be efficient, such an algorithm should refrain from sending messages when the inputs do not change, and should perform *local* communication whenever possible.

We show an instance-based lower bound on the efficiency of such algorithms, and provide two algorithms matching this bound. The first, MultI-LEAG, re-samples the inputs at intervals that are proportional to the graph size, and is extremely message efficient. The second, DynI-LEAG, more closely monitors the aggregate value by sampling it more frequently, at the cost of slightly higher message complexity.

1 Introduction

We consider the problem of continuous monitoring of an aggregation function over a set of dynamically changing inputs on a large graph. We term this problem *dynamic aggregation*. For example, the inputs may reflect sensor readings of temperature or seismic activity, or load reported by computers in a computational grid. The aggregation function may compute the average temperature, or whether the percentage of sensors that detect an earthquake exceeds a certain threshold, or the maximum computer load. It is desirable to seek *local* solutions to this problem, where input values and changes thereof do not need to be communicated over the entire graph.

Since virtually every interesting aggregation function has some input instances on which it cannot be computed without global communication, a priori, it is not clear whether it is possible to do better. Nevertheless, in a recent work [1], we have shown that when computing an aggregation function on a large graph for fixed (in time) inputs, it is in many cases possible to reach the correct result without global communication. Specifically, while *some* problem instances trivially require global communication, many instances can be computed locally, i.e., in a number of steps that is independent of the graph size. We have introduced

a classification of instances according to a measure called *Veracity Radius* (VR), which captures the degree to which a problem instance is amenable to local computation. The VR is computed by looking at the *r-neighborhood* of a node v, which is the set of all nodes within radius r from v. Roughly speaking, the VR identifies the minimum neighborhood radius r_0 such that for all neighborhoods with radius $r \geq r_0$ the aggregation function yields the same value as the entire graph. (The formal VR definition allows some slack in the environments over which the aggregate function is computed.) VR provides a tight lower bound on computation time. In addition, [1] presents an efficient aggregation algorithm, I-LEAG, which achieves the lower bound up to a constant factor.

However, the results of [1] are restricted to the computation of a *static* aggregation instance, and do not directly extend to dynamic aggregation. If I-LEAG is to be used in a dynamic setting, the entire computation must be periodically invoked anew, even if no inputs change. In particular, all nodes must periodically send messages to their neighbors, which can lead to considerable waste of resources, especially when input changes are infrequent.

In this paper, we extend the results of [1] to deal with dynamic aggregation. We focus on algorithms that continuously compute the result of a given aggregation function at each node in the graph, and satisfy the following requirements: (1) the algorithm's output converges to the correct result in finite time once all input changes cease; and (2) once the algorithm has converged, no messages are sent as long as the input values persist.

In Section 3, we derive a lower bound on computation time for dynamic aggregation algorithms satisfying the above requirements. We show that if the algorithm has converged for some input instance I^{old}, and subsequently the inputs change to some instance I^{new}, then the computation of I^{new} must take a number of steps that is proportional to the maximum between the VRs of I^{old} and I^{new}. The lower bound is proven for both the time until the correct result is observed at all nodes (called *output stabilization time*) and the time until no messages are sent (called *quiescence time*).

We provide two efficient dynamic aggregation algorithms that achieve this lower bound up to a constant factor. Our algorithms employ the basic principles of I-LEAG, but are more involved as they need to refrain from sending information when there are no changes.

In Section 4, we focus on a scenario wherein it suffices to update output reflecting the aggregation result periodically, e.g., every few minutes. For this setting, we present MultI-LEAG, which operates in a multi-shot fashion: the inputs are sampled at regular intervals, and the correct (global) result relative to the last sample is computed before the next sample is taken. The sampling interval is proportional to the graphs diameter. MultI-LEAG selectively caches values according to the previous instance's VR to avoid sending messages when the inputs do not change. After every sample, MultI-LEAG reaches both output-stabilization and quiescence in time proportional to the lower bound, which can be considerably less than the sampling interval. MultI-LEAG is very efficient, and does not send more messages than necessary.

In Section 5, we focus on a scenario wherein the output must reflect the correct aggregation value promptly. That is, the input must be sampled very frequently, e.g., at intervals in the order of a single-hop message latency between a pair of neighbors in the graph, and not

proportional to the graph's diameter as in MultI-LEAG. For this setting, we present DynI-LEAG, which invokes multiple MultI-LEAG instances that are active concurrently. Although MultI-LEAG phases have different durations, DynI-LEAG manages to carefully pipeline a combination of complete and partial MultI-LEAG instances to achieve $O(\log^2(\mathit{diameter}))$ memory usage per node. Note that DynI-LEAG inspects multiple input samples during the time frame in which MultI-LEAG conducts a single sample. The corresponding lower bound on algorithms that operate in this mode reflects not only two instances, I^{old} and I^{new} as described above, but rather all instances sampled within a certain time window.

There is a tradeoff between our two algorithms: whereas MultI-LEAG delivers correct results corresponding to relatively old snapshots, DynI-LEAG closely tracks the aggregate result at the expense of a somewhat higher message complexity. Nevertheless, the total number of messages sent in both algorithms depends only on the actual number of input changes and on the VR values of recent instances but not on the system size.

Related work Following the proliferation of large-scale distributed systems such as sensor networks [2, 3], peer-to-peer systems [4], and computational grids [5], there is growing interest in methods for collecting and aggregating the massive amount of data that these systems produce, e.g., [6, 7, 8, 9, 10]. The semantics of validity for dynamic aggregation have been discussed in [11]. However, most of this work has not dealt with locality.

The initial work on using an "instance-based" approach to solve seemingly global problems in a local manner has focused on self-stabilization [12, 13, 14]. Instance-local solutions have also been proposed for distributed error confinement [15] and location services [16]. The first work that demonstrated instance-local aggregation algorithms by means of an empirical study is [17, 18]. Only recently, instance-local aggregation has been formalized [1]. However, this work did not consider dynamic scenarios.

2 Preliminaries

Model and Problem Definition Given a set D, we denote a multi-set over D by $\{d_1^{n_1}...d_m^{n_m}\}$, where $d_i \in D$ and $n_i \in \mathbb{N}$ indicates the multiplicity of d_i. We denote the set of multi-sets over D by $\mathbb{N}^D$. An *aggregation function* is a function $F{:}\mathbb{N}^D \to R$, where R is a discrete totally-ordered set, and F satisfies the following: (i) *convexity*: $\forall X, Y \in \mathbb{N}^D$: $F(X \cup Y) \in \Big[F(X), F(Y)\Big]$; and (ii) *onto* (in singletons): $\forall r \in R, \exists x \in D$: $F(x) = r$. Many interesting functions have these properties, e.g., min, max, majority, median, rounded average (with a discrete range) and consensus (e.g., by using OR/AND functions).

We model a distributed system as a fixed undirected graph $G = G(V, E)$. Computation proceeds in synchronous rounds in which each node can communicate with its immediate neighbors. A graph G and aggregation function F define the *aggregation problem* $P_{G,F}$ as follows: Every node v has an input value $I_v \in D$, which can change over time, and an output register $O_v \in R \cup \{\perp\}$. Initially, $O_v = \perp$ and v only knows its own input. We denote by $I(t)$ the input assignment (of all nodes) at time t. For a set of nodes $X \subseteq V$, we denote by I_X the multi-set induced by the projection of I on X, e.g., $I_V = I$. Assume that there

exists a time t_0 such that $\forall t \geq t_0$: $I(t) = I(t_0)$. An algorithm A *solves* $P_{G,F}$ if it has finite output-stabilization and quiescence times after t_0, and its final outputs are $\forall v \in V$: $O = F(I(t_0))$.

For a multi-shot algorithm, given two consecutive sampled input assignments I^{old} and I^{new}, we denote by $OS_A(I^{old}, I^{new})$ and $Q_A(I^{old}, I^{new})$ the output-stabilization and quiescence times, respectively, following I^{new}. In the general case, we denote by $OS_A(\mathcal{I})$ and $Q_A(\mathcal{I})$ the output-stabilization and quiescence times for an infinite input sequence $\mathcal{I}$ in which the inputs do not change after some time t_0.

Finally, we note that every aggregation function can be represented as a tuple $F = \langle \widehat{R}, F_I, F_{agg}, F_O \rangle$, where: $\widehat{R}$ is some internal representation, and $F_I{:}D \to \widehat{R}$, $F_{agg}{:}\widehat{R}^n \to \widehat{R}$ and $F_O{:}\widehat{R} \to R$ are functions such that for every set of nodes $V = \{v_1, ..., v_n\}$ and an input assignment I:

$$F(I_V) = F_O\Big(F_{agg}\big(\{F_I(I_v) \mid v \in V\}\big)\Big).$$

In many cases, the internal representation $\widehat{R}$ can be extremely compact. For example, for computing OR, it can be a single bit, and for simple majority voting, the number of "yes" and "no" votes.

Graph Notions Let $G = G(V, E)$ be a graph. Denote G's diameter and radius by $Diam(G)$ and $Rad(G)$, respectively. We use the following graph-theoretic notation:

Cluster A *cluster* is a subset $S \subseteq V$ of vertices whose induced subgraph $G(S)$ is connected.

Distance For every two nodes $v_1, v_2 \in V$, the distance between v_1 and v_2 in G, $dist(v_1, v_2)$, is the length of the shortest path connecting them.

Neighborhood The $r-$neighborhood ($r \in \mathbb{R}^+$) of a node v, $\Gamma_r(v)$, is the set of nodes $\{v' \mid dist(v, v') \leq r\}$. $\widehat{\Gamma}(v) = \Gamma_1(v) - \{v\}$ denotes the neighbors of a node v. For a cluster S: $\Gamma_r(S) = \bigcup_{v \in S} \Gamma_r(v)$ and $\widehat{\Gamma}(S) = \Gamma_1(S) - S$.

3 Lower Bound

In [1], we introduced an inherent metric for locality, the *Veracity Radius* (VR), which is defined as follows. A K-bounded *slack function*, is a non-decreasing continuous function $\alpha{:}\mathbb{R}^+ \to \mathbb{R}^+$ such that $\alpha(r) \in [\frac{r}{K}, r]$, for some $K \geq 1$. Given a graph G and an aggregation function F, the VR (parameterized by a slack function α) of an input instance I is:

$$VR_\alpha(I) \triangleq min\{r \in \mathbb{R}^+ \mid \forall r' \geq r, v \in V, S \subseteq V \text{ s.t. } \Gamma_{\alpha(r')}(v) \subseteq S \subseteq \Gamma_{r'}(v){:}\ F(I_S) = F(I)\}.$$

Simply speaking, VR identifies the minimum neighborhood radius r_0 such that for all neighborhood-like environments with radius $r \geq r_0$ (i.e., all subgraphs S that include an $\alpha(r)$-neighborhood and are included in an r-neighborhood), the aggregation function yields the same value as the entire graph. If $F(I_v) = F(I)$ for every $v \in V$, then $VR(I) = 0$ and I is called a *trivial* input assignment.

Given an aggregation problem $P_{G,F}$, we proved in [1] that for every $r \geq 0$, every slack function α and every deterministic algorithm A that solves P, there exists an assignment I with $VR_\alpha(I) \leq r$ for which $OS_A(I) \geq min\{\lfloor \alpha(r) \rfloor, Rad(G)\}$. A similar bound was also proven for quiescence. However, this *single-shot* lower bound is overly restrictive for dynamic systems because it ignores previous instances. We now show that for dynamic aggregation, in which an algorithm is not allowed to send messages after it converges, both current and previous instances are inherent to computation time. Due to lack of space, the proofs are deferred to Appendix A.

For multi-shot algorithms, in which convergence is guaranteed following every input sample, it suffices to consider only the two latest input samples:

Theorem 3.1 (Multi-shot Lower Bound)**.** *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{old}, r^{new} \geq 0$ such that $\alpha(r^{old}), \alpha(r^{new}) \leq Rad(G)$, and every deterministic multi-shot algorithm A that solves F, there exist two consecutive input samples I^{old}, I^{new} such that $VR_\alpha(I^{old}) \leq r^{old}$, $VR_\alpha(I^{new}) \leq r^{new}$, and $OS_A(\{I^{old}, I^{new}\}) \geq max\{\lfloor \alpha(r^{old})/6 \rfloor, \lfloor \alpha(r^{new}) \rfloor\}$. The same holds for quiescence.*

For algorithms that do not necessarily converge between consecutive samples, the multi-shot lower bound implies that the effects of an input assignment may impact algorithm performance during multiple future samples; the duration of these effects is proportional to the input's VR:

Corollary 3.2 (Dynamic Lower Bound)**.** *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{old}, r^{new} \geq 0$ such that $\alpha(r^{old}), \alpha(r^{new}) \leq Rad(G)$, every constant $C \geq 1$, and every deterministic algorithm A that solves F, there exist an input sequence $\mathcal{I}$ and time t_0 such that: (1) $\forall r > r^{old}$: for every $t \in [t_0 - C \cdot r, t_0)$, $VR(I(t)) < r$; (2) $VR(I(t_0)) \leq r^{new}$; and (3) $\forall t \geq t_0$: $I(t) = I(t_0)$; for which $OS_A(\mathcal{I}) \geq max\{\lfloor \alpha(r^{old})/6 \rfloor, \lfloor \alpha(r^{new}) \rfloor\}$. The same holds for quiescence.*

Finally, we note that for output-stabilization, these bounds are nearly tight: in Appendix B, we show how *full information* (FI) protocols, in which every node broadcasts all input changes to all other nodes, achieve $O(max\{\lfloor \alpha(r^{old}) \rfloor, \lfloor \alpha(r^{new}) \rfloor\})$ output-stabilization (for both multi-shot and ongoing operation) albeit at high memory usage and communication costs. Nevertheless, eventual quiescence is still guaranteed.

4 MultI-LEAG: An Efficient Multi-shot Aggregation Algorithm

We now introduce MultI-LEAG, an efficient aggregation algorithm that operates in a multi-shot fashion. MultI-LEAG is quiescent and maintains fixed outputs when the input does not change, while leveraging the veracity radius of the inputs to reach fast quiescence and output stabilization when changes do occur. This enables MultI-LEAG to achieve an extremely low communication complexity, which depends only on the number of changes and the VR of the previous and current input samples, rather than on graph size.

Let $G = G(V, E)$ be a graph, and let $\Lambda_\theta = \lceil \log_\theta(Diam(G)) \rceil$. In order to operate, MultI-LEAG requires a (θ, α)-local partition hierarchy of G, which was first defined in [1] and utilized by the I-LEAG algorithm:

Definition 4.1 ((θ, α)-Local Partition Hierarchy (LPH))**.** *Let $\theta \geq 2$ and let α be a slack function. A (θ, α)-local partition hierarchy of a graph G is a triplet $\langle \{\mathcal{S}_i\}, \{\mathcal{P}_i\}, \{\mathcal{T}_i\} \rangle, 0 \leq i \leq \Lambda_\theta$, where:*

- *$\{\mathcal{S}_i\}$ is a set of partitions, in which for every cluster $S' \in \mathcal{S}_{i-1}$ there exists a cluster $S \in \mathcal{S}_i$ such that $S' \subseteq S$. The topmost level, $\mathcal{S}_{\Lambda_\theta}$, contains a single cluster equal to V. Denote by $S_i(v)$ the cluster $S \in \mathcal{S}_i$ such that $v \in S$.*
- *$\{\mathcal{P}_i\}$ is a set of pivot sets. $\mathcal{P}_i$ includes a single pivot for every cluster $S \in \mathcal{S}_i$. For every $p \in \mathcal{P}_i$, denote $Sub_{i-1}(p) = \{p' \in P_{i-1} \mid p' \in S_i(p)\}$.*
- *$\{\mathcal{T}_i\}$ is a set of forests. For every $p \in \mathcal{P}_i$, $\mathcal{T}_i$ contains a directed tree $T_i(p)$ whose root is p and whose leaves are either $Sub_{i-1}(p)$ or the nodes in $S_0(p)$ if $i = 0$. For every $i > 0$, denote by $\widetilde{T}_i(p)$ the* logical tree *formed by concatenating $T_i(p)$ and $\widetilde{T}_{i-1}(p')$ at every $p' \in Sub_{i-1}(p)$, where $\forall p' \in \mathcal{P}_0$: $\widetilde{T}_0(p') = T_0(p')$.*

In addition, the following conditions must hold for every $p \in \mathcal{P}_i$, $S_i(p) \in \mathcal{S}_i$, and $T_i(p) \in \mathcal{T}_i$: (1) $\Gamma_{\alpha(\theta^i)}(p) \subseteq S_i(p) \subseteq \Gamma_{\theta^i}(p)$; (2) $T_i(p) \subseteq S_i(p)$; (3) the height of $\widetilde{T}_i(p)$ is at most θ^i.

This definition of an LPH is identical to [1], except for the additional second condition. Although we can do without it, it greatly simplifies the presentation. Note that this condition also implies that clusters must be connected within themselves (i.e., clusters are not weak).

An LPH can be computed once per graph, and used for any duration and any aggregation function. We next introduce two notions that link an aggregation problem and an LPH for it, which are closely related to VR:

Cluster in conflict Let $P_{G,F}$ be an aggregation problem. Given an input assignment I and an LPH for G, for every level $i > 0$, a cluster $S \subseteq \mathcal{S}_i$ is *in conflict* if at least two of the level-$(i-1)$ clusters that constitute S have different aggregate results. Level-0 clusters are always considered in conflict.

Veracity Level (VL) Let $P_{G,F}$ be an aggregation problem. Given an input assignment I and an LPH for G, a node v's *Veracity Level* is defined as:

$$VL_v(I) \triangleq max\{i \in [0, \Lambda_\theta] \mid S_i(v) \text{ is in conflict}\}.$$

It directly follows from convexity that the aggregate result of any level-i cluster whose nodes' VL is i, equals the global outcome. We denote by $VL(I)$ the maximum VL over all nodes.

MultI-LEAG is presented in Algorithm 1. It is provided with an LPH, and uses two procedures, do-phase and converge-cast, which are depicted in Algorithms 2 and 3, resp. Code in gray only applies to the DynI-LEAG algorithm presented in the next section, which

Algorithm 1 (MultI-LEAG) for node $v \in V$

Parameters: $F{:}\mathbb{N}^D \to R$, (θ, α)-local hierarchy $\langle\{\mathcal{S}_i\}, \{\mathcal{P}_i\}, \{\mathcal{T}_i\}\rangle, 0 \leq i \leq \Lambda_\theta$ of $G(V, E)$
Input: $I_v \in D$
Output: $O_v \in R \cup \{\perp\}$ initially $\perp$
Definitions: $\mathcal{P}_{i-1} \triangleq V$, $\mathit{Tree}^+ \triangleq \bigcup_{i,p\in\mathcal{P}_i} T_i(p)$ ignoring edge directions (i.e., $\mathit{Tree}^+ \subseteq E$),
$\widehat{S}_i(v) \triangleq S_i(v) \cup \{w \in \widehat{\Gamma}(S_i(v)) \mid \exists u \in S_i(v)\text{: } (u, w) \in \mathit{Tree}^+\}$, $\mathit{Phases} \triangleq \{-1, 0, ..., \Lambda_\theta\}$
Variables:
$\forall u \in \widehat{\Gamma}(v)$: $O_v^u \in R \cup \{\perp\}$ initially $\perp$,
VP_v, $VP_v^{new} \in \mathit{Phases}$ initially 0,
$\mathit{Conf}_v(i){:}\mathit{Phases} \to \{\mathsf{true}, \mathsf{false}\}$, initially false,
$\mathit{Agg}_v(i){:}\mathit{Phases} \to \widehat{R} \cup \{\perp\}$ initially $\perp$,
$\mathit{Agg}_v^{sent}(i){:}\mathit{Phases} \to \widehat{R} \cup \{\perp\}$ initially $\perp$,
$\mathit{Agg}_v^{recv}(i, p){:}\mathit{Phases} \times V \to \widehat{R} \cup \{\perp\}$ initially $\perp$

Synchronous phases:

```
1: loop    /* forever */
2:    Agg_v(−1) ← F_I(I_v)    /* read changes in input */
3:    ∀i: Conf_v(i) ← false
4:    VP_v ← VP_v^new
5:    for phase i = 0 to Λ_θ do
6:       do-phase(i)
```

also uses these procedures. Apart from its input I_v and output register O_v, every node v holds the following variables: O_v^u, the output of every neighbor $u \in \widehat{\Gamma}(v)$, VP_v and VP_v^{new}, v's veracity phase (corresponding to v's VL) in the previous and current input samples, resp. Additionally, for every level i in which v is a pivot, v holds the following mappings: Conf_v, a boolean indicating if $S_i(v)$ is in conflict; Agg_v, the internal aggregate representation of the input values in $S_i(v)$; Agg_v^{sent}, the last value of Agg_v sent to v's pivot in the next level; and Agg_v^{recv}, the last internal representation received from every $p' \in \mathit{Sub}_{i-1}(v)$.

MultI-LEAG operates in cycles (the outer loop). A cycle begins by sampling the input instance and ends with all nodes holding the correct aggregate result matching the sampled inputs. Within a cycle, MultI-LEAG executes Λ_θ synchronous phases that correspond to the levels of the partition hierarchy, calling do-phase each time. (A timer ensures that the next phase is not started before all nodes complete the current phase.) It is convenient to think of do-phase as a sequential operation that takes place concurrently in every cluster S of the current level. Informally, for every phase i and cluster $S \in \mathcal{S}_i$, do-phase operates in one of two modes. The first is to react according to S's conflict state: if S is in conflict, explicitly compute its aggregate result and assign it to the output of all nodes in S. The second is to merely propagate input *changes* in S, if any exist, to S's pivot.

The decision regarding which mode to use, from a node v's perspective, is as follows. Let j be v's VL in the *previous* input, I^{old}. Until phase j is reached, we just propagate changes if there are any, and otherwise do nothing. At phase j, we additionally verify that all nodes in $S_j(v)$ hold the correct output according to the *current* input, I^{new}; if they do

not, we multicast the correct output to them. Subsequently, we reactively handle conflicts as they occur. Note that for every phase i higher than v's current VL, $S_i(v)$ does not incur conflicts. Thus, Multi-LEAG achieves $O(max\{VR(I^{old}), VR(I^{new})\})$ output stabilization and quiescence times (Theorem D.2). In any case, no messages are sent when there are no input changes.

Had we chosen to operate in conflict detection mode at all times, the resulting protocol would closely resemble I-LEAG [1], and would send messages for every non-trivial input (because at least one cluster would suffer a conflict) regardless of whether any inputs change, which is unacceptable.

We now describe MultI-LEAG's operation in more detail. For every node v, VP_v equals v's VL according to the *previous* input instance, and remains unchanged until the end of the cycle. VP_v^{new} is gradually updated to reflect the current VL, and is only used to set VP_v in the next cycle. Therefore, for facility of exposition, we currently ignore the $Conf_v$ mapping and the update-vp message handler, which are responsible for updating VP_v^{new}. For every phase i, $p \in \mathcal{P}_i$, and $S_i(p) \in \mathcal{S}_i$, we distinguish among the following cases:

$\forall v \in S_i(p)$: $i < VP_v$ **(change propagation)** Every pivot $p' \in Sub_{i-1}$ sends any changes in $Agg_{p'}(i-1)$ to p (lines 7-9). Every such update is saved in $Agg_p^{recv}(i, p')$ by the change message handler. After all updates are accepted (this is ensured by the **wait** statement in line 11), $Agg_p(i)$ is recalculated (line 13).

$\forall v \in S_i(p)$: $i = VP_v$ **(change propagation and output validation)** Initially, we update $Agg_p(i)$ as described above. Next, we ensure that the output of every $v \in S_i(p)$ equals $F(I_{S_i(p)})$. As previous phases (which follow the first case) have not altered $S_i(p)$'s outputs at all, every $v \in S_i(p)$ holds the same output, which equals the aggregate result according to the previous instance. Therefore, it is sufficient to check only p's output. If $O_p \neq F_O(Agg_p(i))$ (line 15), then $S_i(p)$'s correct aggregate result is multicast to $\widehat{S}_i(v)$ and assigned by the output handler. Specifically, every $v \in S_i(p)$ updates O_v, and every neighbor u of v such that $u \in S_i(p)$ or $(u, v) \in Tree^+$ updates O_u^v. Otherwise, the outputs of all nodes in $S_i(p)$ remain unaltered.

$\forall v \in S_i(p)$: $i > VP_v$ **(conflict detection)** Assuming that all nodes within a level-$(i-1)$ cluster have the same output (see previous case), conflicts are detected without communication by comparing outputs of neighboring nodes along $T_i(p)$, which know each other's output. Detected conflicts are reported to p and handled by the conflict handler. In this case, p issues a converge-cast call (see Algorithm 3 and explanation below) to explicitly update $Agg_p(i)$. Finally, $S_i(p)$'s aggregate result, $F_O(Agg_p(i))$, is multicast to $\widehat{S}_i(v)$ as in the previous case.

Note that according to VL's definition, no other cases are possible.

To show how VP_v^{new} is gradually adjusted to reflect the current input instance, we begin by describing $Conf_v$, which records cluster conflict states. At the beginning of a cycle, $Conf_v$ maps trivially to false in all nodes. In every phase i and pivot $p \in \mathcal{P}_i$, $Conf_p(i)$ is assigned

Algorithm 2 (do-phase procedure) for node $v \in V$

Function do-phase(i,t)

1: **set timer** to $5\theta^i$
2: let $p \in \mathcal{P}_i$ s.t. $v \in S_i(p)$
3: **if** $i > VP_v^v(t)$ **then** /* fall back to I-LEAG */
4: **if** $v \in T_i(p) \ \wedge \ \exists u \in \widehat{\Gamma}(v)$ s.t. u is v's parent in $T_i(p)$ and $O_v^v(t) \neq O_v^u(t)$ **then**
5: send $\langle$conflict,i,p,t$\rangle$ to u
6: **else** /* $i \leq VP_v^v(t)$ */
7: **if** $v \in Sub_{i-1}(p) \ \wedge \ Agg_v^{sent}(i-1) \neq Agg_v(i-1,t)$ **then** /* propagate changes */
8: $Agg_v^{sent}(i-1) \leftarrow Agg_v(i-1,t)$
9: forward $\langle \mathsf{change}, i, v, Agg_v(i-1,t), p\rangle$ towards p in $T_i(p)$
10: **if** $v = p$ **then**
11: **wait** until timer $< 4\theta^i$ /* wait for all updates to arrive */
12: **if** $\exists p', p'' \in Sub_{i-1}(v)$ s.t. $F_O(Agg_v^{recv}(i,p')) \neq F_O(Agg_v^{recv}(i,p''))$ **then** $Conf_v(i,t) \leftarrow \mathsf{true}$
13: $Agg_v(i,t) \leftarrow F_{agg}(\ \{Agg_v^{recv}(i,p') \mid p' \in Sub_{i-1}(v)\}\ )$
14: **if** $i = VP_v^v(t)$ **then** /* reached VL of previous cycle: update output and VP */
15: **if** $O_v^v(t) \neq F_O(Agg_v(i,t))$ **then** multicast $\langle \mathsf{output}, i, v, F_O(Agg_v(i,t)), t\rangle$ to $\widehat{S}_i(v)$
16: **if** $i > 0 \ \wedge \ Conf_v(i,t) = \mathsf{false}$ **then** multicast $\langle \mathsf{update\text{-}vp}, i, v, 0, t\rangle$ to $T_i(v)$
17: **wait** until timer expires

Message handlers:

upon receiving the first $\langle$conflict,i,p,t$\rangle$ message:
if $v = p$ **then**
$Conf_v(i,t) \leftarrow \mathsf{true}$, $Agg_v(i,t) \leftarrow$ converge-cast(i) /* see Algorithm 3 */
multicast $\langle \mathsf{output}, i, v, F_O(Agg_v(i,t)), t\rangle$ to $\widehat{S}_i(p)$
multicast $\langle \mathsf{update\text{-}vp}, i, v, i, t\rangle$ to $T_i(v)$
else forward message to v's parent in $T_i(p)$

upon receiving a $\langle \mathsf{change}, i, p', \widehat{R}, p\rangle$ message:
if $v = p$ **then** $Agg_v^{recv}(i,p') \leftarrow \widehat{R}$
else forward message to v's parent in $T_i(p)$

upon receiving a $\langle \mathsf{output}, i, p, val, t\rangle$ message:
wait until timer expires
if $v \in S_i(p)$ **then** $O_v^v(t) \leftarrow val$
$\forall u \in \widehat{\Gamma}(v)$: **if** $u \in S_i(p)$ **then** $O_v^u(t) \leftarrow val$

upon receiving a $\langle \mathsf{update\text{-}vp}, i, p, l, t\rangle$ message:
if $i = 0$ **then**
if $v \in S_i(p)$ **then** $\forall u \in \widehat{\Gamma}(v)$ s.t. $u \notin S_i(p) \ \wedge \ (u,v) \in Tree^+$: send $\langle \mathsf{update\text{-}vp}, 0, p, l, t\rangle$ to u
wait until timer expires,$\forall u \in \Gamma(v)$ s.t. $u \in S_i(p)$: $VP_v^{new,u}(t) \leftarrow l$
else if $v \in \mathcal{P}_{i-1}$ **then**
if $l = 0 \ \wedge \ Conf_v(i-1,t) = \mathsf{true}$ **then** $l \leftarrow (i-1)$
multicast $\langle \mathsf{update\text{-}vp}, i-1, v, l, t\rangle$ to $T_{i-1}(v)$

Algorithm 3 (converge-cast RPC) for node $v \in V$

Function converge-cast$(i,t) \rightarrow \widehat{R}$
  if $i > VP_v^v(t) \ \wedge \ Conf_v(i,t) =$ false **then**
    for all $p' \in Sub_{i-1}(v)$ **parallel do**
      $tmp(p') \leftarrow p'$.converge-cast$(i-1,t)$ /* p' is reached via $T_i(v)$ */
    $Agg_v(i,t) \leftarrow F_{agg}(\ \{tmp(p') \mid p' \in Sub_{i-1}(v)\}\ )$
  return $Agg_v(i,t)$

true if $S_i(p)$ is in conflict. This is done either by examining updated aggregate results if $i \leq VP_p$ (line 12), or by receiving a conflict message if $i > VP_p$.

When a new cycle begins, VP_v^{new} is equal to VP_v. Subsequently, it is updated by update-vp messages, which are initiated by pivot nodes and flooded along their logical trees. Specifically, at phase i, a pivot $p \in \mathcal{P}_i$ changes VP_v^{new} for every node $v \in S_i(p)$ in two cases. If $i > VP_p$ and $S_i(p)$ is in conflict (i.e., a conflict message is received by p at level i), p increases VP_v^{new} to i. Alternatively, if $i > 0$, $i = VP_p$ and $S_i(p)$ is *not* in conflict (line 16), p decreases VP_v^{new} to the highest level for which v's cluster had a conflict so far. This is done by sending the first update-vp messages with a VP value (the last parameter) of 0. When a descendent pivot p' of p at some level $j < i$ receives a 0 VP value and its cluster is in conflict, it replaces this value with j for the rest of the subtree.

Thus, for any node v, VP_v^{new} can be lowered at most once when phase VP_v is reached (by v's pivot in level VP_v), and possibly increased one or more times in subsequent phases. At the end of the cycle, VP_v^{new} equals the input's VL, and is assigned to VP_v.

The converge-cast procedure is described in Algorithm 3 using remote procedure call (RPC) semantics. At every phase j and pivot $p \in \mathcal{P}_j$, invoking p.converge-cast(j) aggregates the inputs of $S_j(p)$ recursively based on p's logical tree, $\widetilde{T}_j(p)$. Note that for every level $i < j$ and pivot $p' \in \mathcal{P}_i$, if $i \leq VP_{p'}$ then $Agg_{p'}(i)$ is already up to date because all input changes in $S_j(p')$ have already been accounted for during phase i. In addition, if $i > VP_{p'}$ but $Conf_{p'}(i) =$ true, then $Agg_{p'}(i)$ was updated by a prior converge-cast operation during conflict handling in phase i. Thus, $Agg_{p'}(i)$ needs to be recalculated only if $i \leq VP_{p'}$ and $Conf_v(i) =$ false.

MultI-LEAG's correctness is proven in Appendix C. In Appendix D, we show that MultI-LEAG achieves the mutli-shot lower bound (Theorem 3.1) up to a constant factor.

5 DynI-LEAG: An Efficient Dynamic Aggregation Algorithm

While MultI-LEAG is efficient in terms of communication and converges rapidly after sampling the inputs, its sampling interval is proportional to the graph diameter. Therefore, it is not suitable for applications in which fast output stabilization is desirable at all times. In this section, we present DynI-LEAG, an efficient aggregation algorithm with fast output-stabilization.

DynI-LEAG achieves this by concurrently invoking multiple MultI-LEAG cycles, one

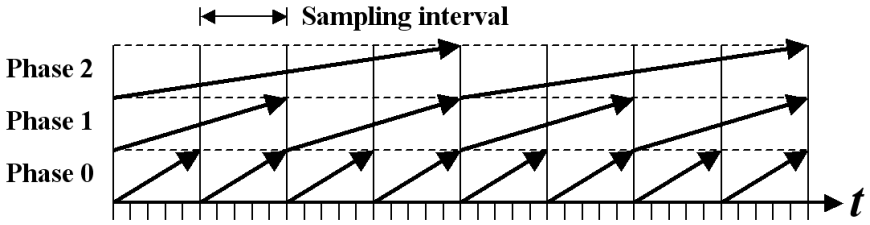

Figure 1: Ruler Pipelining for a 3-level LPH with $\theta = 2$.

per sample, and pipelining their phases. This is challenging, however, because phases have exponentially increasing durations. DynI-LEAG's samples occur frequently, at intervals reflecting the operation time of the first phase. Thus, invoking a full cycle upon each sample would create a number of concurrent cycles that is linear in the graph's diameter, which would lead to considerable resource (messages and memory) consumption. We overcome this challenge by invoking *partial* MultI-LEAG cycles, i.e., cycles that do not execute all phases, to ensure that at every level of the LPH only a single corresponding MultI-LEAG phase is executed at any given moment. This results in a "ruler-like" schedule that executes only $O(\log(Diam(G)))$ concurrent cycles, which we call *Ruler Pipelining*. Figure 1 illustrates ruler pipelining for an LPH with $\theta = 2$. As a consequence, DynI-LEAG requires only $O(\log^2(Diam(G)))$ memory per node (each MultI-LEAG instance has practically the same memory utilization as I-LEAG, which requires $O(\log(Diam(G)))$ memory for reasonable LPHs [1]), while the interval between two consecutive MultI-LEAG phases at the same level is only θ times that of an algorithm that requires $\Omega(Diam(G))$ memory.

A MultI-LEAG cycle ensures that its calculated output and VP values are correct only after it completes. Since this takes $O(Diam(G))$ time, yet another challenge is to select the proper output and VP (for new cycles) from among multiple ongoing cycles, while achieving output-stabilization and quiescence times proportional to the lower bound rather than the diameter.

DynI-LEAG is depicted in Algorithm 4, and uses the do-phase and converge-cast procedures (code in gray applies). To execute concurrent MultI-LEAG cycles, DynI-LEAG holds for every MultI-LEAG variable, except Agg_v^{sent} and Agg_v^{recv}, a mapping that associates each value the variable holds with a timestamp. This is also done for MultI-LEAG's output register, O_v, which is renamed to O_v^v to distinguish between the outputs of different cycles and the actual DynI-LEAG output. Note that the VP_v and VP_v^{new} variables are expanded to include a qualifier $u \in \widehat{\Gamma}(v)$, which enables nodes to hold the corresponding values of their neighbors. ($u = v$ designates v's values.) In addition, DynI-LEAG introduces one new variable, $t_v(i)$, which designates the starting time of the last level-i phase. Agg_v^{sent} and Agg_v^{recv} are not associated with timestamps since they can be perfectly pipelined, i.e., for every level i, $Agg_v^{sent}(i-1)$ and $Agg_v^{recv}(i)$ are only accessed by phase i. This enables DynI-LEAG to use partial cycles at no extra cost: each input change is communicated at most once to higher levels.

DynI-LEAG runs Λ_θ threads at each node, corresponding to the LPH levels, each of

Algorithm 4 (DynI-LEAG) for node $v \in V$

Parameters: F:$\mathbb{N}^D \to R$, (θ, α)-local hierarchy $\langle\{\mathcal{S}_i\},\{\mathcal{P}_i\},\{\mathcal{T}_i\}\rangle, 0 \le i \le \Lambda_\theta$ of $G(V,E)$
Input: $I_v \in D$
Output: $O_v \in R$ initially $\emptyset$
Definitions: $\mathcal{P}_{i-1} \triangleq V$, $Phases \triangleq \{-1, 0, ..., \Lambda_\theta\}$,
$Tree^+ \triangleq \bigcup_{i,p\in\mathcal{P}_i} T_i(p)$ ignoring edge directions (i.e., $Tree^+ \subseteq E$),
$\widehat{S}_i(p) \triangleq S_i(p) \cup \{v \in \widehat{\Gamma}(S_i(p)) \mid \exists u \in S_i(p)\text{: } (u,v) \in Tree^+\}$,
$\Delta(i) \triangleq \sum_{j=0}^{i} 5\theta^j$,
$LastCycle(i,t) \triangleq t - (t \bmod 5\theta^i) - \Delta(i)$
Variables:
$t_v(i)$:$Phases \to \mathbb{Z}$ initially 0,
$\forall u \in \Gamma(v)$: $O_v^u(t)$:$\mathbb{Z} \to R \cup \{\perp\}$ initially $\perp$,
$Conf_v(i,t)$:$Phases \times \mathbb{Z} \to \{\mathsf{true}, \mathsf{false}\}$ initially false,
$\forall u \in \Gamma(v)$: $VP_v^u(t)$, $VP_v^{new,u}(t)$:$\mathbb{Z} \to Phases$ initially 0,
$Agg_v(i,t)$:$Phases \times \mathbb{Z} \to \widehat{R} \cup \{\perp\}$ initially $\perp$,
$Agg_v^{sent}(i)$:$Phases \to \widehat{R} \cup \{\perp\}$ initially $\perp$,
$Agg_v^{recv}(i,p)$:$Phases \times V \to \widehat{R} \cup \{\perp\}$ initially $\perp$

1: **for all** $i \in [0, \Lambda_\theta]$ **parallel do**
2: **loop** /* forever */
3: **if** $i = 0$ **then**
4: $Agg_v(-1, t_v(i)) \leftarrow F_I(I_v)$ /* read input */
5: **for all** $u \in \Gamma_v$ **do**
6: $Candidates \leftarrow \{k \in [0,\Lambda_\theta] \mid VP_v^u(LastCycle(k, t_v(0))) \le k \ \wedge$
$VP_v^{new,u}(LastCycle(k, t_v(0))) = k\}$
7: $VP_v^u(t_v(0)), VP_v^{new,u}(t_v(0)) \leftarrow max(Candidates \cup \{0\})$
8: $O_v^u(t_v(0)) \leftarrow O_v^u(t')$ where $t' = LastCycle(VP_v^u(t_v(0)), t_v(0))$
9: do-bookkeeping$(t_v(0))$
10: $O_v \leftarrow O_v^v(t_v(0))$ /* adjust output */
11: **if** $t_v(i) \ge \Delta(i-1)$ **then** do-phase$(i, t_v(i) - \Delta(i-1))$ **else wait** for $5\theta^i$ time steps
12: $t_v(i) \leftarrow t_v(i) + 5\theta^i$
13: **barrier**$(t_v(i))$ /* synchronize all threads and message handlers that complete a phase at time $t_v(i)$ */

Function do-bookkeeping(t)
$T \leftarrow \{\ t' \mid \exists j \in [0,\Lambda_\theta]$ s.t. $t - (t \bmod 5\theta^j) - t' = \Delta(j)$ or $\Delta(j-1)\ \}$
$\forall j \in Phases, u \in \Gamma(v), t' \notin T$:$O_v^u(t') \leftarrow \perp$, $Conf_v(j,t') \leftarrow \mathsf{false}$, $VP_v^u(t')$, $VP_v^{new,u}(t') \leftarrow 0$,
$Agg_v(j,t') \leftarrow \perp$

which repeatedly calls the *do-phase* procedure (line 11) for the matching level. An individual MultI-LEAG cycle is identified by its starting time, which is also passed during *do-phase* invocations. For every level-i phase, $t_v(i)$ equals the current time when it starts and is incremented by the phase duration, $5\theta^i$, when it completes (line 12). The starting time of the corresponding cycle is found by subtracting from $t_v(i)$ the duration of previous phases,

$\Delta(i-1)$. Ruler pipelining is obtained as a direct outcome of this timing: the results of each completed level-i phase are either used in the level-$(i+1)$ phase that starts at the same time or ignored in the case of a partial cycle that ends at phase i. The barrier in line 13 eliminates data races between phases.

The crux of the algorithm is concentrated at the beginning of a new cycle (i.e., it is executed only by the thread handling phase 0), and consists of 4 operations: (1) sampling the input; (2) choosing the VP and initial output values for the new cycle; (3) estimating the output; and (4) performing some bookkeeping. The second operation is done both for a node itself and its neighbor information to ensure that neighboring nodes know each other's output upon starting the cycle.

To choose a VP value, we initially prepare a list of candidate levels. Level k is considered a candidate if $S_k(v)$ is known to have a conflict according to the most recent information. More formally, we look at the last cycle that completed phase k, i.e., the cycle that started at $\mathit{LastCycle}(k, t_v(0))$, where $t_v(0)$, at this point, is the current time. During a cycle, nodes can learn if their cluster at a certain level has a conflict by the reception (or absence) of update-vp messages during the corresponding phase. Specifically, upon completing phase k, if $VR_v^{new,v} = k$, then $S_k(v)$ has a conflict. However, as update-vp messages are only sent after a cycle completes its VP phase, this information is not available beforehand. Consequently, we only accept k as a candidate if both $VP_v^u(\mathit{LastCycle}(k, t_v(0))) \leq k$ and $VP_v^{new,u}(\mathit{LastCycle}(k, t_v(0))) = k$ hold. Next, we choose the highest candidate, where 0 is always considered a candidate. Both the initial output value and the DynI-LEAG's output estimate, O_v, are simply taken as the current output of the cycle corresponding to the chosen candidate. Thus, after the inputs stabilize, the choices of VP converge to VL and the outputs converge to the global aggregate result, thereby guaranteeing both quiescence and output-stabilization (Theorem C.9(D)).

Finally, the do-bookkeeping procedure ensures that every mapping that is never referenced again, i.e., the time its cycle has started corresponds to neither the current nor last phase of any level, is reset to its default value. Thus, every node has to maintain state for only $2\Lambda_\theta$ MultI-LEAG instances.

DynI-LEAG's correctness is proven in Appendix C. In Appendix D, we show that DynI-LEAG achieves the dynamic lower bound (Corollary 3.2) up to a constant factor.

References

[1] Y. Birk, I. Keidar, L. Liss, A. Schuster, and R. Wolff, "Veracity radius - capturing the locality of distributed computations," *To appear in PODC*, 2006.

[2] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," in *Proc. of the 5th Annual Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[3] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler, "Wireless sensor networks for habitat monitoring," in *Proc. of the ACM Workshop on Sensor Networks and Applications*, 2002.

[4] R. van Renesse, K. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Transactions on Computer Systems*, 2003.

[5] The Condor Project, `http://www.cs.wisc.edu/condor/`.

[6] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," *In Proceedings of the Sixth Annual Intl. Conf. on Mobile Computing and Networking*, August 2000.

[7] D. Kempe, A. Dobra, and J. Gehrke, "Computing aggregate information using gossip," *Proceedings of Fundamentals of Computer Science*, 2003.

[8] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani, "Estimating aggregates on a peer-to-peer network," tech. rep., Stanford University, Database group, 2003. Available from: `http://www-db.stanford.edu/~bawa/publications.html`.

[9] J. Considine, F. Li, G. Kollios, and J. Byers, "Approximate aggregation techniques for sensor databases," in *Proc. of ICDE*, 2004.

[10] J. Zhao, R. Govindan, and D. Estrin, "Computing aggregates for monitoring wireless sensor networks," in *Proc. of SNPA*, 2003.

[11] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani, "The price of validity in dynamic networks," in *Proc. of ACM SIGMOD*, 2004.

[12] S. Kutten and D. Peleg, "Fault-local distributed mending," *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, August 1995.

[13] S. Kutten and D. Peleg, "Tight fault-locality," in *Proc. of the 36th IEEE Symposium on Foundations of Computer Science*, October 1995.

[14] S. Kutten and B. Patt-Shamir, "Time-adaptive self-stabilization," *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pp. 149–158, August 1997.

[15] Y. Azar, S. Kutten, and B. Patt-Shamir, "Distributed error confinement," in *Proc. of the 22nd Annual Symp. on Principles of Distributed Computing*, July 2003.

[16] J. Li, J. Jannotti, D. D. Couto, D. Karger, and R. Morris, "A scalable location service for geographic ad hoc routing," in *Proc. of the 6th ACM Intl. Conf. on Mobile Computing and Networking*, August 2000.

[17] L. Liss, Y. Birk, R. Wolff, and A. Schuster, "A local algorithm for ad hoc majority voting via charge fusion," in *Proceedings of the Annual Conference on Distributed Computing (DISC)*, October 2004.

[18] R. Wolff and A. Schuster, "Association rule mining in peer-to-peer systems," in *Proc. of the IEEE Conference on Data Mining (ICDM)*, November 2003.

[19] D. Peleg, *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications, 2000.

A Lower Bound Proofs

Lemma A.1. *Let $P_{G,F}$ be an aggregation problem. For every slack function α and every $r \geq 0$ such that $6 \leq \alpha(r) \leq Rad(G)$, there exist two input assignments I, I' such that $VR_\alpha(I) \leq r$, $VR_\alpha(I') = 0$, and at least one node $v_0 \in G$ cannot distinguish between I and I' in fewer than $\lfloor \alpha(r)/6 \rfloor$ steps.*

Proof. Denote $r' = \lfloor \alpha(r)/6 \rfloor$. Choose $v_0 \in V$ such that $|\Gamma_{r'}(v_0)| = min_{v \in V} |\Gamma_{r'}(v)|$. Let $S = \Gamma_{r'}(v_0)$ and let $a, b \in D$ such that $F(a) \neq F(b)$ are consecutive values in R. Convexity ensures that $F(ab)$ is either $F(a)$ or $F(b)$. Without loss of generality, assume that it is $F(a)$. We construct I, I' as follows:

$$I_v = \begin{cases} b, & v \in S \\ a, & v \notin S \end{cases} \qquad I'_v = b, \quad v \in V$$

Obviously, $F(I') = F(b)$ and $VR_\alpha(I') = 0$.

We first claim that for every $v \in V$, $r'' \geq r$ and $T \subseteq V$ such that $\Gamma_{\alpha(r'')}(v) \subseteq T \subseteq \Gamma_{r''}(v)$, it holds that $F(I_T) = F(a)$. Denote the number of a and b values in T by n_1 and n_2, respectively. Clearly, $n_2 \leq |S|$. If $S \cap \Gamma_{r'}(v) = \emptyset$, we obtain that $n_1 \geq |S|$ because $|\Gamma_{r'}(v)| \geq |S|$. Otherwise, for every $w \in S$, it holds that $dist(w, v) \leq 3r'$. Since $5r' \leq \alpha(r) \leq Rad(G)$, there exists $v' \in T$ such that $dist(v, v') = 5r'$. Denote $S' = \Gamma_{r'}(v')$. For every $u \in S'$, it holds that $dist(u, v) \geq dist(v', v) - dist(u, v') \geq 4r'$, so $S \cap S' = \emptyset$. By noticing that $S' \subseteq \Gamma_{6r'}(v) \subseteq T$, we obtain once more that $n_1 \geq |S|$. By convexity, $F(I_T) = F(a^{n_1} b^{n_2}) = F((ab)^{n_2} \cup a^{n_1 - n_2}) = F(a)$.

As a result, it follows that $F(I) = F(a)$ and $VR_\alpha(I) \leq r$ by definition. Since all nodes in S have *exactly* the same input in both assignments, it is obvious that v_0 cannot distinguish between I and I' before $\lfloor \alpha(r)/6 \rfloor$ steps take place. ∎

Proof. (Theorem 3.1) If $\lfloor \alpha(r^{old})/6 \rfloor \leq \lfloor \alpha(r^{new}) \rfloor$, let I^{new} be the instance promised by the single-shot lower bound for $r = r^{new}$, let $r \in R$ such that $r \neq F(I^{new})$, and $\forall v \in V$: let $I^{old}_v = r$. Trivially, $VR_\alpha(I^{old}) \leq r^{old}$. Since $\lfloor \alpha(r^{new}) \rfloor \leq r^{new} \leq Rad(G)$, it follows from the single-shot lower bound that $OS_A(\{I^{old}, I^{new}\}) \geq \lfloor \alpha(r^{new}) \rfloor$.

Otherwise, note that if $\alpha(r^{old}) < 6$, the theorem holds trivially. Therefore, assume that $\alpha(r^{old}) \geq 6$ and let $I^{old} = I$ and $I^{new} = I'$ be the inputs and $v_0 \in V$ the node promised

by the construction of Lemma A.1 for $r = r^{old}$. We initially focus on output stabilization. If we would have set $I^{new} = I = I^{old}$, then A would not have changed the output of any node. Since v_0 cannot distinguish between I and I' in fewer than $\lfloor \alpha(r^{old})/6 \rfloor$ steps, v_0 cannot change its output before time $t = \lfloor \alpha(r^{old})/6 \rfloor$. Therefore, O_v must change from $F(I^{old})$ to $F(I^{new})$ at time t or later.

For quiescence, assume in contradiction that $Q_A\{I^{old}, I^{new}\} < \lfloor \alpha(r^{old})/6 \rfloor$. Therefore, v_0 cannot distinguish between the two cases, invalidating either output stabilization or correctness. ■

Proof. (Corollary 3.2) Let I^{old} and I^{new} be the corresponding input assignments for r^{old} and r^{new} according to Theorem 3.1. We build an infinite input sequence $\mathcal{I}$ as follows: $\forall t \in [0, t_0)$: $I(t) = I^{old}$, $\forall t \geq t_0$: $I(t) = I^{new}$, where t_0 is the time it takes A to converge on an infinite sequence of I^{old} assignments. Thus, the lower bound of Theorem 3.1 holds here because A essentially simulates a multi-shot algorithm with samples taken at $t = 0$ and $t = t_0$. ■

B Full Information Protocol (FI)

We present two full information protocols that broadcast any input changes to all nodes. The first, Multi-FI, operates in a multi-shot manner and samples the inputs approximately every diameter time. The second, Dyn-FI, is a dynamic algorithm that samples the inputs every time step.

B.1 Multi-FI

Multi-FI is presented in Algorithm 5. Multi-FI operates in cycles, each of which executes $Diam(G) + 1$ synchronous phases. In every phase i, we initially gather input changes (of v as well as those received from other nodes), record them in I_v^{sav}, and generate an up-to-date snapshot of the inputs up to radius i from v in I_v^{snap}. Subsequently, we calculate the aggregate result according to the snapshot in O_v^{new}, and update v's *veracity phase* VP_v that holds the last phase in which O_v^{new} has changed. The output is set according to O_v^{new} once we reach phase VP_v^{old}, which holds the final VP_v value of the previous cycle. Input changes are propagated to all nodes in a timely manner based on shortest path trees. (Note that no messages are sent in the last phase.) Thus, for every node u, $I_v^{sav}(u)$ at time t equals u's input as sampled at time $t - dist(u, v)$ as required for consistent snapshots.

Theorem B.1. *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{old}, r^{new} \geq 0$ such that $\alpha(r^{old}), \alpha(r^{new}) \leq Rad(G)$, and every two consecutive input samples I^{old}, I^{new} such that $VR_\alpha(I^{old}) \leq r^{old}$, $VR_\alpha(I^{new}) \leq r^{new}$, it holds that*

$$OS_{Multi-FI}(\{I^{old}, I^{new}\}) \leq max\{\lfloor \alpha(r^{old}) \rfloor, \lfloor \alpha(r^{new}) \rfloor\}.$$

Proof. Eventual quiescence is trivial. Consider a cycle with input I for which $VR_\alpha(I) = r$. It is straightforward from the algorithm that in phase i, I_v^{snap} holds the correct values of $I_{\Gamma_i(v)}$. Thus, according to the definition of VR, O_v^{new}'s stabilization time, VP_v, is $\lfloor \alpha(r) \rfloor$.

Algorithm 5 (Multi-FI) for node $v \in V$

Parameters: $F{:}\mathbb{N}^D \to R$, $G(V,E)$
Input: $I_v \in D$
Output: $O_v \in R \cup \{\perp\}$ initially $\perp$
Definitions: $Phases \triangleq \{0, ..., Diam(G)\}$, $\forall v \in V$: $SPT(v)$ denotes the shortest path tree rooted at v
Variables:
$M_v \in 2^{V \times D}$ initially $\emptyset$,
VP_v, $VP_v^{old} \in Phases$ initially 0,
$I_v^{sav}(u){:}V \to D \cup \{\perp\}$ initially $\perp$,
$I_v^{snap} \in D \cup \{\perp\}$ initially $\perp$,
$O_v^{new} \in R \cup \{\perp\}$ initially $\perp$

1: **loop** /* forever */
2: $O_v^{new} \leftarrow \perp$
3: **for** phase $i = 0$ to $Diam(G)$ **do**
4: $M_v \leftarrow$all messages received at this phase
5: **if** $i = 0 \ \wedge \ I_v \neq I_v^{sav}(v)$ **then** $M_v \leftarrow M_v \cup \{\langle v, I_v \rangle\}$ /* check for input changes */
6: $\forall \langle u, val \rangle \in M_v$: $I_v^{sav}(u) \leftarrow val$
7: $I_v^{snap} \leftarrow \{I_v^{sav}(u) \mid u \in V, dist(u,v) \leq i\}$ /* update input snapshot */
8: **if** $O_v^{new} \neq F(I_v^{snap})$ **then** /* update output snapshot and VP */
9: $O_v^{new} \leftarrow F(I_v^{snap})$
10: $VP_v \leftarrow i$
11: **if** $i \geq VP_v^{old}$ **then** $O_v \leftarrow O_v^{new}$ /* determine output */
12: $\forall \langle u, val \rangle \in M_v$: send $\langle u, val \rangle$ to downtree neighbors according to $SPT(u)$
13: $VP_v^{old} \leftarrow VP_v$

As a result, for I^{old} and I^{new}, we obtain that after sampling I^{new}, O_v^{new} stabilizes in $\lfloor \alpha(r^{new}) \rfloor$ time, while O_v reflects O_v^{new} after $\lfloor \alpha(r^{old}) \rfloor$ time, thereby achieving $max\{\lfloor \alpha(r^{old}) \rfloor, \lfloor \alpha(r^{new}) \rfloor\}$ output stabilization. Finally, we note that as long as the input does not change in subsequent cycles, O_v does not change either because O_v^{new} is assigned to O_v only after O_v^{new} stabilizes (according to the same input). ■

B.2 Dyn-FI

Dyn-FI is presented in Algorithm 6. Dyn-FI basically invokes multiple Multi-FI cycles in parallel, a cycle per time step, with an alternative mechanism for determining the output: it chooses the output among all concurrent cycles rather than just the last and current cycle (as Multi-FI does). Specifically, Dyn-FI continuously executes the following operations. First, we record the most recent input changes we heard of in I_v^{sav}. This information is then used to generate up-to-date snap-shots of all cycles that started at $t' \in [t - Diam(G), t]$ where t is the current time as follows. Fix some time t'; at time $t = t'$, $I_v^{snap}(t')$ includes only v's input; for every $t > t'$, the inputs sampled at t' of all nodes within a distance of $t - t'$ from

Algorithm 6 (Dyn-FI protocol) for node $v \in V$

Parameters: $F{:}\mathbb{N}^D \to R$, $G(V,E)$
Input: $I_v \in D$
Output: $O_v \in R \cup \{\perp\}$ initially $\perp$
Definitions: $Phases \triangleq \{0,...,Diam(G)\}$, $\forall v \in V$: $SPT(v)$ denotes the shortest path tree rooted at v
Variables:
$M_v \in 2^{V \times D}$ initially $\emptyset$,
$VP_v(t){:}\mathbb{Z} \to Phases$ initially 0,
$I_v^{sav}(u){:}V \to D \cup \{\perp\}$ initially $\perp$,
$I_v^{snap}(t){:}\mathbb{Z} \to \mathbb{N}^D$, initially $\emptyset$,
$O_v^{snap}(t){:}\mathbb{Z} \to R \cup \{\perp\}$ initially $\perp$,
$t \in \mathbb{Z}$, the current time (initially 0)

1: **loop** /* forever */
2: $M_v \leftarrow$ all received messages at this step
3: **if** $I_v \neq I_v^{sav}(v)$ **then** $M_v \leftarrow M_v \cup \{\langle v, I_v\rangle\}$ /* check for input changes */
4: $\forall \langle u, val\rangle \in M_v$: $I_v^{sav}(u) \leftarrow val$
5: **for all** $u \in V$ **do** /* update input snapshots */
6: $t' \leftarrow t - dist(u,v)$
7: $I_v^{snap}(t') \leftarrow I_v^{snap}(t') \cup \{I_v^{sav}(u)\}$
8: **for all** $t' \in [t - Diam(G), t]$ **do** /* update output snapshots and VPs */
9: **if** $O_v^{snap}(t') \neq F(I_v^{snap}(t'))$ **then**
10: $O_v^{snap}(t') = F(I_v^{snap}(t'))$
11: $VP_v(t') \leftarrow t - t'$
12: $i \leftarrow max\{j \in [0, Diam(G)] \mid VP_v(t-j) = j\}$ /* select phase for output */
13: $O_v \leftarrow O_v^{snap}(t-i)$
14: $VP_v(t') \leftarrow 0$, $I_v^{snap}(t') \leftarrow \emptyset$, $O_v^{snap}(t') \leftarrow \perp$, where $t' = t - Diam(G)$ /* cleanup */
15: $\forall \langle u, val\rangle \in M_v$: send $\langle u, val\rangle$ to downtree neighbors according to $SPT(u)$

v are added to $I_v^{snap}(t')$ (without changing the previous values in $I_v^{snap}(t')$). The aggregate result of $I_v^{snap}(t')$ is calculated in $O_v^{snap}(t')$, updating the corresponding output-stabilization time, $VP_v(t')$, accordingly. Finally, the output is chosen as the aggregate result of the oldest cycle that observed an output change in its most recent phase. Similar to Multi-FI, all input changes are pipelined to every node based on shortest path trees.

Theorem B.2. *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{old}, r^{new} \geq 0$ such that $\alpha(r^{old}), \alpha(r^{new}) \leq Rad(G)$, and every input sequence $\mathcal{I}$ and time t_0 such that: (1) $\forall r > r^{old}$: for every $t \in [t_0 - r, t_0)$, $VR(I(t)) < r$; (2) $VR(I(t_0)) = r^{new}$; and (3) $\forall t \geq t_0$: $I(t) = I(t_0)$; it holds that*

$$OS_{Dyn-FI}(\mathcal{I}) \leq max\{\lfloor \alpha(r^{old}) \rfloor, \lfloor \alpha(r^{new}) \rfloor\}.$$

Proof. Quiescence is trivial. For every $t' \in [t - Diam(G), t]$, it is straightforward from the algorithm that upon completing time step t, $I_v^{snap}(t') = \Gamma_{t-t'}(v)$. Thus, according to the

definition of VR, if $VR_\alpha(I(t')) = r$, then $O_v^{snap}(t')$ stabilizes with a value of $F(I(t'))$ at time $t' + \lfloor \alpha(r) \rfloor$, fixing $VP_v(t') = \lfloor \alpha(r) \rfloor$ as well.

Let $\widetilde{r} = max\{\lfloor \alpha(r^{old}) \rfloor, \lfloor \alpha(r^{new}) \rfloor\}$. According to assumption (1), after t_0, no cycle that reached a phase higher than $\widetilde{r}$ can be considered as a candidate for choosing the output. In addition, for every $t \geq t_0 + \widetilde{r}$, assumption (3) ensures that every cycle that reached a phase $\leq \widetilde{r}$ has sampled an input identical to $I(t_0)$. Consequently, it follows from assumption (2) that the oldest cycle that observed an output change in its most recent phase is the one that started at $t - \lfloor \alpha(r^{new}) \rfloor$. We conclude that $\forall t \geq t_0 + \widetilde{r}$: $O_v(t) = O_v^{snap}(t - \lfloor \alpha(r^{new}) \rfloor) = F(I(t_0))$. ■

C Correctness Proofs

An aggregation algorithm is considered correct if all nodes converge to the correct aggregate result and stop sending messages in finite time after all input changes have ceased. DynI-LEAG is constructed of pipelined, mutually-independent invocations of MultI-LEAG, which differ from MultI-LEAG only in the addition of cycle time stamps (e.g., $O_v^u(t)$ instead of O_v^u) and two minor symbolic name changes, i.e., O_v^v instead of O_v and VP_v^v instead of VP_v. The additional DynI-LEAG variables t_v and VP_v^u (that holds a neighbor's VP) do not influence a MultI-LEAG instance within a cycle. Therefore, we prove the correctness of both algorithms together using MultI-LEAG's variable names, while omitting the DynI-LEAG time stamps for brevity. When there are significant differences in the proofs of the algorithms (e.g., in situations where multiple MultI-LEAG invocations are considered in a DynI-LEAG execution), we present separate lemmas and theorems and explicitly mark them by M (for MultI-LEAG) or D (for DynI-LEAG).

For brevity, we denote the duration of phase i by $PT(i) = 5\theta^i$. (PT stands for phase time). In addition, we use the notation $VR(I(t))$ and $VR(t)$ interchangeably. In our proofs, we will make extensive use of the following definition:

Definition C.1 (Consistent Cycle)**.** *A cycle is said to be* consistent *if for every $i \in [0, \Lambda_\theta]$ and $S \in \mathcal{S}_i$, the following conditions are met at the beginning of the cycle:*

1. *If $\exists v \in S$ such that $VP(v) \geq i$, then $\forall v' \in S$: $VP(v') = VP(v)$, where $VP(v)$ represents the value assigned to VP_v in line 4 of MultI-LEAG or to $VP_v^v(t_v(0))$ in line 7 of DynI-LEAG.*

2. *If $\exists v \in S$ such that $VP(v) = i$, then all nodes in S have the same initial output. For MultI-LEAG, the initial output is the last value assigned to O_v by the previous cycle (or during initialization); for DynI-LEAG, the initial output is the value assigned to $O_v^v(t_v(0))$ in line 8.*

Lemma C.2. *For every phase i, all messages sent during i reach their destination by the end of the phase.*

Proof. Let $p \in \mathcal{P}_i$. According to the LPH properties, the height of $\widetilde{T}_i(p)$ is at most θ^i. Hence, it takes at most θ^i time for either a conflict message or a change message to reach its

intended pivot, $2\theta^i$ time to conduct a converge-cast [19], and $\theta^i + 1$ time for a multicast to span a cluster in $\mathcal{S}_i$ and its direct neighbors.

Since MultI-LEAG's messages are orchestrated by pivots, we distinguish between two cases. If $VP_p \leq i$, p can receive one or more change messages (sent concurrently at the beginning of the phase), which can optionally cause an output multicast and/or a concurrent update-vp multicast to be initiated after θ^i time has passed since the beginning of the phase. Thus, all such messages subside in $2\theta^i + 1$ time.

Otherwise, p can receive a conflict message (subsequent conflict messages are ignored), which in turn triggers a converge-cast. Upon converge-cast completion, output and update-vp multicasts are initiated concurrently. In this case, all messages subside in $4\theta^i + 1$ time. As the next phase does not start before $5\theta^i$ time has passed, we conclude that in all cases there are no messages in flight when the phase ends. ■

Lemma C.3. *At the end of every phase i, for every node $v \in V$ and every neighbor u of v such that $(u, v) \in Tree^+$: $O_u^v = O_v$.*

Proof. Let $p \in \mathcal{P}_i$ such that $v \in S_i(p)$. If $val = F(I_{S_i(p)})$ is explicitly calculated in phase i, Lemma C.2 ensures that both u and v receive an $\langle \mathsf{output}, i, p, val \rangle$ message before the end of the phase. (u receives it either as a member of $S_i(p)$ itself or as a neighbor of v because $(u, v) \in Tree^+$). In this case, v sets $O_v = val$, u sets $O_u^v = val$, and the lemma holds. Otherwise, both O_v and O_u^v are left unchanged. Thus, if $i > 0$, the lemma follows from our assumption on $i - 1$. If $i = 0$, we distinguish between two cases. For the first cycle, the lemma holds due to initialization. As for subsequent cycles, in MultI-LEAG the lemma holds due to the last phase of the previous cycle. In DynI-LEAG, both O_v^v and O_u^v are explicitly calculated in line 8 based on VP and output values of previous cycles. However, every change in both VP_v^v and O_v^v is always communicated to u and saved in VP_u^v and O_u^v, respectively, for any previous cycle. Consequently, these values are kept consistent at all times. Since the new output calculation is deterministic, we obtain the result. ■

Lemma C.4. *At the beginning of every phase i, it holds for every $p \in \mathcal{P}_i$ that $\forall p' \in Sub_{i-1}(p)$: $Agg_p^{recv}(i, p') = Agg_{p'}^{sent}(i - 1)$.*

Proof. The lemma holds trivially in the first cycle by initialization, and follows directly from lines 8-9 and Lemma C.2 in subsequent cycles. ■

Lemma C.5. *At the end of every phase i of a consistent cycle, it holds for every $p \in \mathcal{P}_i$ that:*

1. *if $VP_p \geq i$ then $Agg_p(i) = F_{agg}(\{F_I(I_v) \mid v \in S_i(p)\})$*
2. *if $VP_p \leq i$ then $\forall v \in S_i(p)$: $O_v = F(I_{S_i(p)})$.*
3. *$Conf_p(i) =$ true iff $S_i(p)$ is in conflict.*

Proof. By induction on phases. Let $p \in \mathcal{P}_i$. If $VP_p \geq i$, cycle consistency ensures that for every $p' \in Sub_{i-1}(p)$: $VP_{p'} \geq i$. Thus, it follows from the induction hypothesis that $Agg_{p'}(i-1) = F_{agg}(\{F_I(I_v) \mid v \in S_{i-1}(p')\})$. At the beginning of the phase, if $Agg^{sent}_{p'}(i-1) \neq Agg_{p'}(i-1)$, then p' sends to p an update message with $Agg_{p'}(i-1)$. Therefore, after θ^i time, it holds that $Agg^{recv}_p(i,p') = Agg_{p'}(i-1)$ either due to this message or due to Lemma C.4. In this case, (3) follows immediately from line 12. In addition, when $Agg_p(i)$ is subsequently calculated in line 13, we obtain that:

$$Agg_p(i) = F_{agg}(\{Agg^{recv}_p(i,p') \mid p' \in Sub_{i-1}(p)\} = F_{agg}(\{F_I(I_v) \mid v \in S_i(p)\},$$

proving (1). Note that this also establishes the induction base for (1) because $\forall v \in V$: $Agg_v(-1) = F_I(I_v)$.

For (2), we distinguish between two cases. If $VP_p = i$, we obtain from (1) that $F_O(Agg_p(i))= F(I_{S_i(p)})$. In addition, according to cycle consistency, $\forall u, v \in S_i(p)$: $O_u = O_v$ at the beginning of the cycle. Moreover, $\forall j < i$ and $p' \in \mathcal{P}_j$ such that $p' \in S_i(p)$, it holds that $VP_{p'} = i$ so the outputs of nodes in $S_i(p)$ were not changed in previous phases. (p' multicasts a new output to its cluster only if $VP_{p'} \leq j$.) Thus, if $O_p = F_O(Agg_p(i))$, the output of all nodes in $S_i(p)$ remains unchanged and (2) holds trivially. Otherwise, p explicitly multicasts $F(I_{S_i(p)})$ assigning it to the output register of every $v \in S_i(p)$ by the end of the phase (Lemma C.2). This case also establishes the induction base because VP values cannot be negative.

If $VP_p < i$, cycle consistency ensures that for all $v \in S_i(p)$: $VP_v < i$. Thus, according to the induction hypothesis, for every $p' \in Sub_{i-1}(p)$ and every $v \in S_{i-1}(p')$: $O_v = F(I_{S_{i-1}(p')})$. If $S_i(p)$ is in conflict, then at least one node in $T_i(p)$ reports a conflict to p following Lemma C.3 and the fact that $T_i(p)$ connects all pivots in $Sub_{i-1}(p)$. Consequently, $F(I_{S_i(p)})$ is calculated explicitly and assigned to the output register of every $v \in S_i(p)$ as before. Otherwise, every node in $S_i(p)$ has the same output, and the fact that $T_i(p) \subseteq S_i(p)$ ensures that no node will falsely report a conflict. Moreover, this output equals $F(I_{S_i(p)})$ by convexity. We conclude that both (2) and (3) hold. ∎

Lemma C.6. *At the end of every phase i of a consistent cycle, it holds that:*

1. *$\forall v$ s.t. $i \geq VP_v$: $VP^{new}_v = max\{j \leq i \mid S_j(v)$ has a conflict$\}$.*
2. *$\forall v$ s.t. $i < VP_v$: $VP^{new}_v = VP_v$.*

Proof. By induction on phases. Unless noted otherwise, line numbers and message handlers refer to the do-phase procedure (Algorithm 2). For the induction base, we note that upon starting phase 0, it holds that $VP^{new}_v = VP_v$. (In MUTLI-LEAG, this is true either due to initialization or due to line 4 of Algorithm 1; in DynI-LEAG, this is true due to line 7 of Algorithm 4.) Since level-0 pivots do not send update-vp messages, VP^{new}_v remains unchanged. Thus, if $VP_v > 0$, $VP^{new}_v = VP_v$ as required. Otherwise ($VP_v = 0$), $VP^{new}_v = 0$, which satisfies condition (1). (Recall that level-0 clusters are always considered to have a conflict.)

For the induction step, we assume that the lemma holds for phase $i-1$ and prove for phase i, building upon Lemma C.5(3). Denote v's pivots up to level i by $p_0, ..., p_i$. We distinguish among the following cases:

- $VP_v = i$. According to cycle consistency, $VP_{p_i} = VP_v$. Hence, if $S_i(v)$ has a conflict, then $Conf_{p_i}(i) =$ true so p_i does *not* send update-vp messages (line 16). According to the induction hypothesis, VP_v^{new} remains i as required. If $S_i(v)$ does not have a conflict, p_i multicasts an update-vp message M with $l = 0$. M is propagated along the logical tree $\widetilde{T}_i(p_i)$, optionally changing its l value by the update-vp handler while passing through pivots. Specifically, to reach v, M traverses the pivots p_{i-1},...,p_0 in descending order. Let $k = max\{j < i \mid S_j(v)$ has a conflict$\}$. For p_j such that $j > k$, M is unchanged. However, l is changed to k by p_k and remains so until is reaches v. Consequently, VP_v is set to k as required.

- $VP_v < i$. According to cycle consistency, $VP_{p_i} < i$. Thus, if $S_i(v)$ has a conflict, p_i receives a conflict message setting $Conf_{p_i}(i) =$ true, and multicasts in response an update-vp message M with $l = i$. Since $l \neq 0$, M reaches v unmodified, setting VP_v to i as required. Otherwise, p_i does not send update-vp messages during this phase and VP_v remains unchanged. Thus, the lemma follows from the induction hypothesis.

- $VP_v > i$. According to cycle consistency, $VP_{p_i} < i$. Thus, p_i does not send update-vp messages during this phase, VP_v remains unchanged, and the lemma follows from the induction hypothesis.

■

Lemma C.7. *(M) Every cycle is consistent.*

Proof. At $t = 0$, the lemma holds trivially by initialization. For subsequent cycles, we first consider the VP consistency condition. At the end of every cycle, it directly follows from Lemma C.6(1) that the VP^{new} values of all nodes obey the consistency condition. Since the VP values of every new cycle (after $t = 0$) are set according to the VP^{new} values of the previous cycle, we have the result. As for output consistency, Lemma C.5(3) guarantees that all nodes have the same outputs upon completing the last phase. As outputs remain unmodified when proceeding to the next cycle, the lemma holds. ■

Lemma C.7. *(D) Every cycle is consistent.*

Proof. By induction on cycles. For brevity, we refer to VP and output values at $t < 0$ as values of "negative" cycles. Therefore, the induction base is established by noticing that all cycles starting at $t \leq 0$ are consistent by initialization. For the induction step, let $t > 0$ be a time in which a cycle begins, i.e., $\forall v \in V$: $t_v(0) = t$ at t, let $k \in [0, \Lambda_\theta]$, and let $S \in S_k$. We first consider VP consistency. Assume that there exists a node $u \in S_k$ such that $VP_u^u(t) = j \geq k$. Therefore, j is u's highest ranking candidate for $VP_u^u(t)$ in line 7 of DynI-LEAG. Consider a node $v \in S_k$. Noticing that v and u also belong to the same level-j cluster, S', we first show that j is also a candidate of v, assuming that $j > 0$ (0 is always a candidate).

It holds that: (1) $VP_u^u(LastCycle(j,t)) \leq j$ and (2) $VP_u^{new,u}(LastCycle(j,t)) = j$. According to the induction hypothesis, cycle $LastCycle(j,t)$ is consistent. Therefore, (1) implies that

also $VP_v^v(LastCycle(j,t)) \leq j$. Since VP^{new} values are only adjusted by do-phase's update-vp handler at phase boundaries, $VP_u^{new,u}(LastCycle(j,t))$ represents the value set at the end of phase j of cycle $LastCycle(j,t)$. Consequently, it follows from (2) and Lemma C.6(1) that S' has a conflict in cycle $LastCycle(j,t)$, which mandates that $VP_v^{new,v}(LastCycle(j,t)) = j$. As a result, j is also a candidate of v.

Finally, by symmetry, if v has a candidate $l > j$ at t, it must also be u's candidate. Therefore, j must be v's highest ranking candidate, guaranteeing that $VP_v^v(t) = j$ as required.

For output consistency, assume that $\exists u \in S_k$ such that $VP_u^u = k$. Thus, according to VP consistency, $\forall v \in S_k$: $VP_v^v(t) = k$ so $O_v^v(t) = O_v^v(LastCycle(k,t))$. Since k is a valid VP candidate, it holds that $VP_v^v(LastCycle(k,t)) \leq k$. Therefore, Lemma C.5(2) ensures that $\forall u, v \in S_k$: $O_u^u(LastCycle(k,t)) = O_v^v(LastCycle(k,t))$ when $LastCycle(k,t)$ completed phase k. Since VP^{new} values are only adjusted by do-phase's output handler at phase boundaries, these outputs have not change as of time t, thereby completing the proof. ■

Lemma C.8. *Let C_1, C_2 be consecutive cycles. If C_1 and C_2 have the same input I, and for every node v in C_2: (1) $VP_v = VL_v(I)$ and (2) initially $O_v = F(I)$, then C_2 does not send messages nor change any outputs.*

Proof. Since the inputs have not changed between C_1 and C_2, any partial aggregate results, i.e., Agg_v variables, do not change either. Therefore, Lemma C.4 ensures that no update messages are sent in C_2. Hereafter, we limit the discussion to C_2.

We next show that no output or conflict messages are sent. Since outputs can only be changed by output messages, this also establishes output stabilization. An output message can be sent by the do-phase procedure in two cases: (1) by the conflict handler; and, (2) in line 15. Assume in contradiction that the first output message is sent in (1). However, as all nodes initially hold the same output no conflicts occur, so no conflict messages are sent. Consequently, this case is impossible. As for (2), a level-i pivot p for which $VP_p = i$ can multicast an output message only if $O_p \neq F_O(Agg_p(i))$. Assuming that the first output message is sent in (2), it holds that $O_p = F(I)$ when the condition above is checked. According to Lemma C.5(1), $Agg_p(i) = F_{Agg}(\{F_I(I_v) \mid v \in S_i(p)\})$, so $F_O(Agg_p(i)) = F(I_{S_i(p)})$. Since $VL_p(I) = i$, we reach a contradiction by observing that $F(I_{S_i(p)}) = F(I)$ by convexity.

Similarly, update-vp messages are sent by multicast operations either in the conflict handler or in line 16. The first case is ruled out as noted above. The second case occurs only if a level-i pivot p for which $VP_p = i$ detects that $Conf_p(i) =$ false. However, the fact that $VL_p(I) = i$ and Lemma C.5(3) ensure that $Conf_p(i) =$ true, excluding this case as well. ■

Theorem C.9. *(M) Let $P_{G,F}$ be an aggregation problem. Given a (θ, α)-LPH of G, MultI-LEAG correctly solves $P_{G,F}$ for every input instance.*

Proof. Assume that the inputs do not change after some time t_0, and let C_0 be the first cycle that starts after t_0. According to Lemma C.5(2), all nodes hold the correct global output after completing the last phase of this cycle. In addition, Lemma C.6(1) ensures that upon completing C_0, for every node v: $VP_v^{new} = VL_v$.

Consider the next cycle. Since its initial outputs are left unmodified and for every node v, VP_v is set according to C_0's latest VP_v^{new} value, it follows from Lemma C.8 that this cycle does not send messages nor change any outputs. The proof is completed by noticing that the same reasoning applies to every subsequent cycle. ■

Theorem C.9. *(D) Let $P_{G,F}$ be an aggregation problem. Given a (θ, α)-LPH of G, DynI-LEAG correctly solves $P_{G,F}$ for every input instance. Specifically, DynI-LEAG reaches quiescence and output-stabilization in at most $8PT(\Lambda_\theta)$ time after the last input change.*

Proof. Assume that all cycles after some time t_0 have the same input I. Let v be an arbitrary node, let $k = VL_v(I)$, and let C be a cycle that starts at some time $t \geq t_0$. For every $k' > k$, if $VP_v(t) \leq k'$, then upon completing phase k' or at any time afterwards, it follows from Lemma C.6(1) that $VP_v^{new,v}(t) \neq k'$. Consequently, C cannot support k' as a candidate VP (in line 7) for any subsequent cycle.

Let C_0 be the complete cycle that ends at $t_1 = t_0 + 3PT(\Lambda_\theta)) - (t_0 \bmod PT(\Lambda_\theta))$ and starts at $t'_0 = t_1 - \Delta(\Lambda_\theta) = \mathit{LastCycle}(\Lambda_\theta, t_0 + 3PT(\Lambda_\theta))$. Since

$$t'_0 = t_0 + 3PT(\Lambda_\theta)) - (t_0 \bmod PT(\Lambda_\theta)) - \Delta(\Lambda_\theta) > t_0,$$

neither C_0 nor any cycle that starts after it can support candidate VPs higher than k. Moreover, after t_1, no cycle that starts before C_0 can either because it is not the *LastCycle* of any level. As a consequence, for every cycle that starts after t_1, it holds that $VP_v^v \leq k$.

Now consider the complete cycle C_1 that starts at $t'_1 = t'_0 + 2PT(\Lambda_\theta)$. Since $t'_1 > t_1$, it holds that $VP_v^v(t'_1) \leq k$. Therefore, according to Lemma C.6(1), after C_1 completes phase k, $VP_v^{new,v} = k$ at all times. Moreover, because $\forall j > k$: $F(I_{S_j(v)}) = F(I)$ due to convexity, after C_1 completes phase k, it follows from Lemma C.5(2) that $O_v^v(t'_1) = F(I)$ at all times. This is true for every complete or partial cycle that starts after t'_1 and reaches phase k.

As a result, as soon as C_1 completes phase k, i.e., at $t_2 = t'_1 + \Delta(k)$, k becomes a valid VP candidate for v at all times, with an associated initial output value (as chosen in line 8) equal to $F(I)$. Thus, for every cycle that starts at $t \geq t_2$, it holds that $VP_v^v(t) = k$ and (initially) $O_v^v(t) = F(I)$ because k is the highest candidate VP at t.

Finally, we observe that every cycle starting after t_2 conforms to the conditions of Lemma C.8, so it does not send any messages nor change its initial outputs. Denote the complete cycle that starts at $t'_2 = t'_1 + 2PT(\Lambda_\theta)$ by C_2. Since $t'_2 > t_2$, it follows from Lemma C.2 that there cannot be any in-flight messages (due to previous cycles) after C_2 completes the last phase, which occurs at $t_3 = t'_2 + \Delta(\Lambda_\theta)$. Since

$$t_3 < t'_2 + 2PT(\Lambda_\theta) = t'_0 + 6PT(\Lambda_\theta) < t_0 + 8PT(\Lambda_\theta),$$

the theorem follows. ■

D Complexity Proofs

D.1 MultI-LEAG

Lemma D.1. *Let $P_{G,F}$ be an aggregation problem. Given a (θ,α)-LPH of G, for every non trivial input I: $VL(I) \leq \lceil \log_\theta(VR_\alpha(I)) \rceil$*

Proof. According to the definition of VR, for every integer $i \geq \lceil \log_\theta(VR_\alpha(I)) \rceil$, every $v \in V$, and every $S \subseteq V$ such that $\Gamma_{\alpha(\theta^i)}(v) \subseteq S \subseteq \Gamma_{\theta^i}(v)$, $F(I_S) = F(I)$. Thus, no two clusters in $\mathcal{S}_i$ can be in conflict, and we obtain that $VL(I) \leq \lceil \log_\theta(VR_\alpha(I)) \rceil$. ■

Theorem D.2. *Let $P_{G,F}$ be an aggregation problem. Given a (θ,α)-LPH of G, for every two consecutive cycles C^{old} and C^{new} with non-trivial input assignments I^{old} and I^{new}, respectively, MultI-LEAG's output stabilization and quiescence times for C^{new} are at most: $\left(\frac{5\theta^2}{\theta-1}\right)r$, where $r = max\{VR_\alpha(I^{old}), VR_\alpha(I^{new})\}$.*

Proof. Following Lemma C.6, upon completing C^{old}, $\forall v \in V$: VP_v^{new} equals $VL_v(I^{old})$. These values are taken as the nodes' VP in C^{new}. Therefore, in C^{new}, MultI-LEAG does not send any messages after phase $VP(I^{old})$ unless conflicts are detected. In addition, Lemma C.5(3) ensures that no conflicts are detected after phase $VP(I^{new})$. Noting that all messages sent during a phase subside by the end of the phase (Lemma C.2), we conclude that MultI-LEAG reaches quiescence by the end of phase $r = max\{VP(I^{old}), VP(I^{new})\}$. Since every phase i takes $5\theta^i$ time, we obtain from Lemma D.1 that:

$$Q_{I-LEAG}(I) \leq \sum_{i=0}^{r} 5\theta^i = 5\Big(\frac{\theta^{r+1}-1}{\theta-1}\Big) < \Big(\frac{5\theta^{(\log_\theta r+1)+1}-1}{\theta-1}\Big) < \Big(\frac{5\theta^2}{\theta-1}\Big)r.$$

As nodes change their outputs only in response to multicasts, $OS_{I-LEAG}(I) \leq Q_{I-LEAG}(I)$. ■

D.2 DynI-LEAG

Lemma D.3. *Denote $VP(t) = max\{VP_v^v(t) \mid v \in V\}$. If there exists a time t_0 and $k \in [1, \Lambda_\theta]$ such that $\forall t \geq t_0$: $VL(t) < k$ and $VP(t) \leq k$, then the following conditions hold:*

1. *$\forall t \geq t_0 + 3PT(k)$: $VP(t) < k$.*
2. *$\forall t \geq t_0 + 6PT(k)$: no messages are sent in phase k.*

Proof. Consider a cycle C that starts at $t \geq t_0 + 3PT(k)$, for some t_0 and k that uphold the lemma's assumptions. Let $t'_0 = LastCycle(k,t)$. Since *LastCycle* is a monotone non-decreasing function in t, we obtain that

$$t'_0 \geq LastCycle(k, t_0 + 3PT(k)) =$$

$$t_0 + 3LastCycle(k, t_0 + 3PT(k)) - (t_0 \bmod PT(k)) - \Delta(k) > t_0.$$

Consequently, $VL(t'_0) < k$ and $VP(t'_0) \leq k$. Following Lemma C.6(1), for every node v: $VP_v^{new,v}(t'_0) < k$, so k is not a candidate VP (in line 7) for C. Therefore, (1) holds according to our assumption on VP. Moreover, following our assumption on VL, C does not send messages at phase k because no conflicts are detected during this phase.

As for (2), note that any in-flight message at time $t \geq t_0 + 5PT(k)$ that was sent in phase k must belong to a cycle starting at $t - (t \bmod PT(k)) - \Delta(k-1) \geq t_0 + 3PT(k)$, contradicting our conclusions for C. ■

Lemma D.4. *If there exists a time t_0 and $k_0 \in [0, \Lambda_\theta]$ such that for every $k > k_0$ and every $t > t_0 - 6PT(k)$: $VL(t) < k$, then for every $k > k_0$:*

1. *$\forall t \geq t_0 - 3PT(k)$: $VP(t) < k$.*
2. *$\forall t \geq t_0$: no messages are sent in phase k.*

Proof. We show that the requirements for every level $k > k_0$ hold by induction from Λ_θ down to $k_0 + 1$. For the induction basis of $k = \Lambda_\theta$, note that $\forall t$: $VP(t) \leq k$ trivially. Thus, we can apply Lemma D.3 at $t_0 - 6PT(\Lambda_\theta)$ with respect to Λ_θ and obtain the result. For the induction step, we assume that the requirements hold for levels $\Lambda_\theta, ..., k+1$ and prove for level k. Since $\theta \geq 2$,

$$t_0 - 3PT(k+1) = t_0 - 3 \cdot 5\theta \cdot \theta^k \leq t_0 - 6 \cdot 5\theta^k = t_0 - 6PT(k).$$

Therefore, according to our assumptions and the fact that level $k+1$ upholds the requirements, we obtain that $\forall t \geq t_0 - 6PT(k)$: $VL(t) < k$ and $VP(t) \leq k$. The proof is completed by applying Lemma D.3 at $t_0 - 6PT(k)$ with respect to level k. ■

Lemma D.5. *For every $k^{old}, k^{new} \geq 0$, if all input changes cease at time t_0 and:*

1. *$\forall k > k^{old}$: for every $t \geq t_0 - 6PT(k)$, $VL(t) < k$*
2. *$VL(t_0) \leq k^{new}$*

then DynI-LEAG reaches both quiescence and output-stabilization by time $t_0 + 8 \cdot PT(\max\{k^{old}, k^{new}\})$.

Proof. Denote $l = max\{k^{old}, k^{new}\}$. Following Lemma D.4, for every $t \geq t_0$: $VL(t) \leq l$ and no messages are sent at phases higher than l. Therefore, after t_0, in every cluster of the level-l partition, $\mathcal{S}_l$, DynI-LEAG effectively behaves as if it were provided with an LPH with only l levels. As a result, the proof follows by applying Theorem C.9(D) with $\Lambda_\theta = l$. ■

Theorem D.6. *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{old}, r^{new} \geq 0$, and every input sequence $\mathcal{I}$ such that all input changes cease at time t_0 and:*

1. *$\forall r > r^{old}$: for every $t \geq t_0 - 30\theta \cdot r$, $VR_\alpha(t) < r$*

2. $VR_\alpha(t_0) = r^{new}$

then DynI-LEAG reaches both quiescence and output-stabilization by time $40\theta \cdot max\{r^{old}, r^{new}\}$.

Proof. Denote $k^{old} \triangleq \lceil \log_\theta r^{old} \rceil$, $k^{new} \triangleq \lceil \log_\theta r^{new} \rceil$. Let $k \in \mathbb{N}$ such that $k > k^{old}$, and denote $r \triangleq \theta^{k-1}$. Since $k \geq k^{old} + 1$, it follows that $r = \theta^{k-1} \geq \theta^{k^{old}} = \theta^{\lceil \log_\theta r^{old} \rceil} \geq r^{old}$. Therefore, by noticing that $30\theta \cdot r = 6 \cdot 5\theta^k = 6PT(k)$, it follows from our assumptions on $\mathcal{I}$ and Lemma D.1 that

$$\forall t \geq 6PT(k)\colon\ VL(t) \leq \log_\theta(VR_\alpha(t)) + 1 < \log_\theta r + 1 = k,$$

and

$$\forall t \geq t_0\colon\ VL(t) \leq \lceil \log_\theta(VR_\alpha(t)) \rceil \leq \lceil \log_\theta r^{new} \rceil = k^{new}.$$

Therefore, we obtain from Lemma D.5 that

$$OS_{DynI-LEAG}(\mathcal{I}) \leq 8PT(max\{k^{old}, k^{new}\}) = 40 \cdot \theta^{max\{k^{old}, k^{new}\}} \leq$$

$$40 \cdot max\{\theta^{\log_\theta r^{old}+1}, \theta^{\log_\theta r^{new}+1}\} \leq 40\theta \cdot max\{r^{old}, r^{new}\}.$$

■