Efficient Dynamic Aggregation[⋆]

Yitzhak Birk, Idit Keidar, Liran Liss, and Assaf Schuster

Technion – Israel Institute of Technology, Haifa 32000, Isreal
E-mail: {birk@ee, idish@ee, liranl@tx, assaf@cs}.technion.ac.il

Abstract. We consider the problem of *dynamic aggregation* of inputs over a large fixed 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 present 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, achieving quiescence between samplings, 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 fixed 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, whereby 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 one can do better. Nevertheless, we have recently shown that when computing an aggregation function on a large graph for fixed (in time) inputs, it is often possible to reach the correct result without global communication [1]. 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 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

⋆ This work was supported in part by a grant from the Israel Ministry of Science.

is computed by examining 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 for the entire graph. (The formal definition of VR 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.

The results of [1], however, 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. Specifically, 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 Sect. 3, we derive a lower bound on computation time for dynamic aggregation algorithms satisfying the above requirements. We show that if an algorithm has converged for some input I^{old}, and subsequently the inputs change to 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 (*output stabilization time*) and the time until no messages are sent (*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 messages when there are no changes.

In Sect. 4, we consider a scenario wherein it suffices to update the 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 graph diameter. MultI-LEAG selectively caches values according to the previous input'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 never exceeds the sampling interval and may be considerably shorter. We call this *sample-compute-output* cycle an iteration. MultI-LEAG is very efficient, and does not send more messages than necessary.

In Sect. 5, we consider a scenario wherein the output must reflect the correct aggregation value promptly. That is, the input is sampled very frequently, e.g., at intervals on the order of a single-hop message latency between neighboring nodes, and not proportional to the graph's diameter as in MultI-LEAG. For this setting, we present DynI-LEAG, which invokes multiple MultI-LEAG iterations in parallel. Although each MultI-LEAG iteration is comprised of several phases with different durations, DynI-LEAG manages to carefully pipeline a combination of complete and partial MultI-LEAG iterations 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 inputs, I^{old} and I^{new}, as described above, but rather all inputs 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 inputs 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., [6–10]. 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 [12–14]. Instance-local solutions have also been proposed for distributed error confinement [15], location services [16] and Minimum Spanning Tree [17]. The first work that demonstrated instance-local aggregation algorithms by means of an empirical study is [18, 19]. 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 an 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 *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 A, given two consecutive sampled input assignments I^{old} and I^{new}, we denote by $OS_A(I^{\text{old}}, I^{\text{new}})$ and $Q_A(I^{\text{old}}, I^{\text{new}})$ 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 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 inputs. We now show that for dynamic aggregation, in which an algorithm is not allowed to send messages after it converges, both current and previous inputs are inherent to computation time. Due to lack of space, the proofs are detailed in the full paper [20].

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

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

Finally, we note that for output-stabilization, these bounds are nearly tight: in [20], we show how *full information* (FI) protocols, in which every node broadcasts all input changes to all other nodes, achieve $O(max\{\lfloor \alpha(r^{\text{old}}) \rfloor, \lfloor \alpha(r^{\text{new}}) \rfloor\})$ output-stabilization time (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 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 (sometimes called cluster head) 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.

Apart from the second condition, this definition of an LPH is identical to [1], which provides general LPH construction algorithms. 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* [1]).

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.

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

```
Parameters: $F{:}\mathbb{N}^D \to R$, $(\theta, \alpha)$-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: $O_v \in R \cup \{\bot\}$ initially $\bot$
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., $Tree^+ \subseteq E$),
  $\widehat{S}_i(v) \triangleq S_i(v) \;\cup\; \{w \in \widehat{\Gamma}(S_i(v)) \mid \exists u \in S_i(v){:}\ (u, w) \in Tree^+\}$
Variables:
  $\forall u \in \widehat{\Gamma}(v){:}\ O_v^u \in R \cup \{\bot\}$ initially $\bot$,
  $VP_v$, $VP_v^{\text{new}} \in Phases$ initially 0,
  $Conf_v(i){:}Phases \to \{\mathsf{true}, \mathsf{false}\}$, initially true for $i = 0$ and false otherwise,
  $Agg_v(i){:}Phases \to \widehat{R} \cup \{\bot\}$ initially $\bot$,
  $Agg_v^{\text{sent}}(i){:}Phases \to \widehat{R} \cup \{\bot\}$ initially $\bot$,
  $Agg_v^{\text{recv}}(i, p){:}Phases \times V \to \widehat{R} \cup \{\bot\}$ initially $\bot$

Synchronous phases:
1: loop     /* forever */
2:    $Agg_v(-1) \leftarrow F_I(I_v)$     /* read changes in input */
3:    $\forall i > 0{:}\ Conf_v(i) \leftarrow$ false
4:    $VP_v \leftarrow VP_v^{\text{new}}$
5:    for phase $i = 0$ to $\Lambda_\theta$ do
6:       do-phase(i)
```

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 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 (used to compute v's VL as explained shortly) 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 $p' \in Sub_{i-1}(v)$.

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 $v \in Sub_{i-1}(p) \ \wedge\ Agg_v^{\text{sent}}(i-1) \neq Agg_v(i-1,t)$ then    /* send changes */
8:         $Agg_v^{\text{sent}}(i-1) \leftarrow Agg_v(i-1,t)$
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 changes to arrive */
12:        if $\exists p', p'' \in Sub_{i-1}(v)$ s.t. $F_O(Agg_v^{\text{recv}}(i,p')) \neq F_O(Agg_v^{\text{recv}}(i,p''))$ then
13:            $Conf_v(i,t) \leftarrow$ true
14:        $Agg_v(i,t) \leftarrow F_{agg}(\ \{Agg_v^{\text{recv}}(i,p') \mid p' \in Sub_{i-1}(v)\}\ )$
15:        if $i = VP_v^v(t)$ then    /* reached prev. VL: update output and VP */
16:            if $O_v^v(t) \neq F_O(Agg_v(i,t))$ then
17:                multicast $\langle \mathsf{output}, i, v, F_O(Agg_v(i,t)), t\rangle$ to $\widehat{S}_i(v)$
18:            if $i > 0 \ \wedge\ Conf_v(i,t) =$ false then multicast $\langle \mathsf{update\text{-}vp}, i, v, 0, t\rangle$ to $T_i(v)$
19: wait until timer expires
```

Message handlers:

```
upon receiving the first $\langle$conflict,i,p,t$\rangle$ message:
  if $v = p$ then
    $Agg_v(i,t) \leftarrow$ converge-cast($i, t$)    /* see Algorithm 3 */
    $Conf_v(i,t) \leftarrow$ true
    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 $Agg_v^{\text{recv}}(i,p') \leftarrow \widehat{R}$
  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. $u \in S_i(p)$: $VP_v^{\text{new},u}(t) \leftarrow l$
  else if $v \in \mathcal{P}_{i-1}$ then
    if $l = 0 \ \wedge\ Conf_v(i-1,t) =$ 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($i, t$) $\to \widehat{R}$
  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 $Agg_v(i, t)$
```

MultI-LEAG operates in iterations (the outer loop). An iteration begins by sampling the input and ends with all nodes holding the correct aggregate result matching the sampled inputs. Within an iteration, 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. (If not in conflict, do nothing.) 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 for the *current* input, I^{new}, 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 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 $O(max\{\, VR(I^{\text{old}}), VR(I^{\text{new}})\})$ output stabilization and quiescence times (Theorem 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, and remains unchanged until the end of the iteration. VP_v^{new} is gradually updated to reflect the current VL, and is only used to set VP_v in the next iteration. 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 $p' \in Sub_{i-1}$ sends changes in $Agg_{p'}(i-1)$ to p (lines 7-9). Every such update is saved in $Agg_p^{\text{recv}}(i, p')$ 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 14).

$\forall v \in S_i(p)$: $i = VP_v$ **(change propagation and output validation)** First, 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 input. Therefore, it is sufficient to check only p's output. If $O_p \neq F_O(Agg_p(i))$ (line 16), 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 $(u, v) \in \mathit{Tree}^+$ updates O_u^v. (Tree^+ denotes the union of all tree edges; see definition in Algorithm 1.) 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, we begin by describing Conf_v, which records cluster conflict states. At the beginning of an iteration, Conf_v maps trivially to false for every phase other than 0 in all nodes. In every phase i and pivot $p \in \mathcal{P}_i$, $\mathit{Conf}_p(i)$ is assigned 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 iteration 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 18), p decreases VP_v^{new} to the highest level for which v's cluster wan in 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 iteration, 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 $Conf_{p'}(i) = \mathsf{true}$, 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 $Conf_v(i) = \mathsf{false}$.

MultI-LEAG's correctness and complexity are proved in [20]. Specifically, we show that MultI-LEAG achieves the multi-shot lower bound (Theorem 1) up to a constant factor:

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

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 iterations, one 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 iteration upon each sample would create a number of concurrent iterations 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 iterations, i.e., iterations 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 iterations, 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 iteration 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 iteration 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 iterations) from among

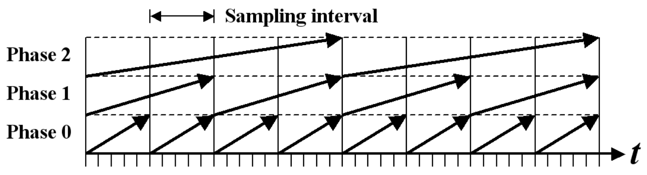

Fig. 1. Ruler Pipelining for a 3-level LPH with $\theta = 2$.

multiple ongoing iterations, 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 convergecast procedures (code in gray applies). To execute concurrent MultI-LEAG iterations, 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 time stamp. 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 iterations 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 time stamps since they can be perfectly pipelined, i.e., for every level i, $Agg_v^{\text{sent}}(i-1)$ and $Agg_v^{\text{recv}}(i)$ are only accessed by phase i. This enables DynI-LEAG to use partial iterations 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 which repeatedly calls the *do-phase* procedure (line 14) for the matching level. An individual MultI-LEAG iteration 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 15). The starting time of the corresponding iteration 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 iteration that ends at phase i. The barrier in line 16 eliminates data races between phases.

The crux of the algorithm is concentrated at the beginning of a new iteration (i.e., it is executed only by the thread handling phase 0), and comprises four operations: (1) sampling the input; (2) choosing the VP and initial output values for the new iteration; (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 iteration.

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

```
Parameters: $F{:}\mathbb{N}^D \to R$, $(\theta, \alpha)$-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., $Tree^+ \subseteq E$),
  $\widehat{S}_i(p) \triangleq S_i(p) \cup \{v \in \widehat{\Gamma}(S_i(p)) \mid \exists u \in S_i(p){:}\ (u, v) \in Tree^+\}$,
  $\Delta(i) \triangleq \sum_{j=0}^{i} 5\theta^j$,
  $LastIter(i, t) \triangleq t - (t \bmod 5\theta^i) - \Delta(i)$
Variables:
  $t_v(i){:}Phases \to \mathbb{Z}$ initially 0,
  $\forall u \in \Gamma(v){:}\ O_v^u(t){:}\mathbb{Z} \to R \cup \{\bot\}$ initially $\bot$,
  $Conf_v(i, t){:}Phases \times \mathbb{Z} \to \{\mathsf{true}, \mathsf{false}\}$ initially true for $i = 0$ and false otherwise,
  $\forall u \in \Gamma(v){:}\ VP_v^u(t)$, $VP_v^{\text{new},u}(t){:}\mathbb{Z} \to Phases$ initially 0,
  $Agg_v(i, t){:}Phases \times \mathbb{Z} \to \widehat{R} \cup \{\bot\}$ initially $\bot$,
  $Agg_v^{\text{sent}}(i){:}Phases \to \widehat{R} \cup \{\bot\}$ initially $\bot$,
  $Agg_v^{\text{recv}}(i, p){:}Phases \times V \to \widehat{R} \cup \{\bot\}$ initially $\bot$

 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:         $Candidates \leftarrow \{k \in [0, \Lambda_\theta] \mid VP_v^u(LastIter(k, t_v(0))) \le k\ \wedge$
              $VP_v^{\text{new},u}(LastIter(k, t_v(0))) = k\}$
 7:         $VP_v^u(t_v(0))$, $VP_v^{\text{new},u}(t_v(0)) \leftarrow max(Candidates \cup \{0\})$
 8:         $O_v^u(t_v(0)) \leftarrow O_v^u(t')$ where $t' = LastIter(VP_v^u(t_v(0)), t_v(0))$
 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
12:       do-phase($i, t_v(i) - \Delta(i-1)$)
13:     else
14:       wait for $5\theta^i$ time steps
15:     $t_v(i) \leftarrow t_v(i) + 5\theta^i$
16:     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 $\Delta(j-1)\ \}$
  $\forall j \in Phases, u \in \Gamma(v), t' \notin T{:}O_v^u(t') \leftarrow \bot$, $Conf_v(j, t') \leftarrow$ false,
      $VP_v^u(t')$, $VP_v^{\text{new},u}(t') \leftarrow 0$, $Agg_v(j, t') \leftarrow \bot$
```

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 be in conflict according to the most recent information. More formally, we look at the last iteration that completed phase k, i.e., the iteration that started at $LastIter(k, t_v(0))$, where $t_v(0)$, at this point, is the current time. During an iteration, nodes can learn if their cluster at a certain level is in conflict by the reception (or absence) of update-vp messages during the corresponding phase. Specifically, upon completing phase k, if $VR_v^{\mathrm{new},v} = k$, then $S_k(v)$ is in conflict. However, as update-vp messages are only sent after an iteration completes its VP phase, this information is not available beforehand. Consequently, we only accept k as a candidate if both $VP_v^u(LastIter(k, t_v(0))) \leq k$ and $VP_v^{\mathrm{new},u}(LastIter(k, t_v(0))) = k$ hold. Next, we choose the highest candidate, where 0 is always considered a candidate. Both the initial output value and DynI-LEAG's output estimate, O_v, are simply taken as the current output of the iteration 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 3).

Finally, the do-bookkeeping procedure ensures that every mapping that is never referenced again, i.e., the time its iteration 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 iterations.

DynI-LEAG's correctness and complexity are proved in [20]. Specifically, we show that DynI-LEAG achieves the dynamic lower bound (Corollary 1) up to a constant factor:

Theorem 3. *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{\mathrm{old}}, r^{\mathrm{new}} \geq 0$, and every input sequence $\mathcal{I}$ such that all input changes cease at time t_0 and:*

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

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

6 Conclusions

We provided two efficient algorithms, MultI-LEAG and DynI-LEAG, for dynamic aggregation in large graphs with fixed topologies. When the inputs are stable, the algorithms are quiescent and hence do not waste any resources from the communication network. When changes do occur, the performance of these algorithms is proportional to the *Veracity Radius* of the inputs at hand, which enables them to achieve optimal *instance-local* operation and resource utilization.

Consequently, these algorithms are extremely attractive for data aggregation tasks in dynamic, resource-constrained environments in which topological changes are infrequent compared to the sampling rate, be it for periodically obtaining the result according to the most recent sample in a very efficient manner (MultI-LEAG) or for closely tracking the monitored environment to capture global trends as fast as possible (DynI-LEAG).

References

1. Birk, Y., Keidar, I., Liss, L., Schuster, A., Wolff, R.: Veracity radius - capturing the locality of distributed computations. To appear in PODC (2006)
2. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: Tag: a tiny aggregation service for ad-hoc sensor networks. In: Proc. of the 5th Annual Symposium on Operating Systems Design and Implementation (OSDI). (2002)
3. Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D.: Wireless sensor networks for habitat monitoring. In: Proc. of the ACM Workshop on Sensor Networks and Applications. (2002)
4. van Renesse, R., Birman, K., Vogels, W.: 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. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: A scalable and robust communication paradigm for sensor networks. In Proceedings of the Sixth Annual Intl. Conf. on Mobile Computing and Networking (2000)
7. Kempe, D., Dobra, A., Gehrke, J.: Computing aggregate information using gossip. Proceedings of Fundamentals of Computer Science (2003)
8. Bawa, M., Garcia-Molina, H., Gionis, A., Motwani, R.: Estimating aggregates on a peer-to-peer network. Technical report, Stanford University, Database group (2003) Available from: `http://www-db.stanford.edu/~bawa/publications.html`.
9. Considine, J., Li, F., Kollios, G., Byers, J.: Approximate aggregation techniques for sensor databases. In: Proc. of ICDE. (2004)
10. Zhao, J., Govindan, R., Estrin, D.: Computing aggregates for monitoring wireless sensor networks. In: Proc. of SNPA. (2003)
11. Bawa, M., Gionis, A., Garcia-Molina, H., Motwani, R.: The price of validity in dynamic networks. In: Proc. of ACM SIGMOD. (2004)
12. Kutten, S., Peleg, D.: Fault-local distributed mending. Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (1995)
13. Kutten, S., Peleg, D.: Tight fault-locality. In: Proc. of the 36th IEEE Symposium on Foundations of Computer Science. (1995)
14. Kutten, S., Patt-Shamir, B.: Time-adaptive self-stabilization. Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (1997) 149–158
15. Azar, Y., Kutten, S., Patt-Shamir, B.: Distributed error confinement. In: Proc. of the 22nd Annual Symp. on Principles of Distributed Computing. (2003)
16. Li, J., Jannotti, J., Couto, D.D., Karger, D., Morris, R.: A scalable location service for geographic ad hoc routing. In: Proc. of the 6th ACM Intl. Conf. on Mobile Computing and Networking. (2000)
17. Elkin, M.: A faster distributed protocol for constructing a minimum spanning tree. In: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (SODA). (2004) 359–368
18. Liss, L., Birk, Y., Wolff, R., Schuster, A.: A local algorithm for ad hoc majority voting via charge fusion. In: Proceedings of the Annual Conference on Distributed Computing (DISC). (2004)
19. Wolff, R., Schuster, A.: Association rule mining in peer-to-peer systems. In: Proc. of the IEEE Conference on Data Mining (ICDM). (2003)
20. Birk, Y., Keidar, I., Liss, L., Schuster, A.: Efficient dynamic aggregation. CCIT Technical Report 589, EE Department, Technion (2006)