Information and Computation ••• (••••) ••••••

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Load balanced distributed directories ☆

Shishir Rai [a], Gokarna Sharma [a,*], Costas Busch [b], Maurice Herlihy [c]

[a] Department of Computer Science, Kent State University, Kent, OH, USA
[b] School of Computer and Cyber Sciences, Augusta University, Augusta, GA, USA
[c] Department of Computer Science, Brown University, Providence, RI, USA

ARTICLE INFO

Article history:
Received 19 May 2019
Received in revised form 22 August 2020
Accepted 31 October 2020
Available online xxxx

Keywords:
Distributed systems
Directories
Hierarchical clustering
Stretch
Congestion
Processing load
Load balancing
Approximation

ABSTRACT

We present LB-Spiral, a novel distributed directory protocol for shared objects, suitable for large-scale distributed shared memory systems. Each shared object has an owner node that can modify its value. The ownership may change by moving the object from one node to another in response to *move* requests. The value of an object can be read by other nodes with *lookup* requests. The distinctive feature of LB-Spiral is that it balances the processing load on nodes in addition to minimizing the communication cost in general network topologies. In contrast, the existing distributed directory protocols for general network topologies only minimize the communication cost. In particular, LB-Spiral achieves poly-log approximation for both load and communication cost in general networks with respect to the problem parameters. Simulation results show that the established theoretical results translate well in practice.

1. Introduction

Distributed directories are data structures that enable access to shared objects in a network. They support three basic operations: (i) *publish*, allowing a shared object to be inserted in the directory so that other nodes can find it; (ii) *lookup*, providing a read-only copy of the object to the requesting node; and (iii) *move*, allowing the requesting node to write the object locally after getting it.

Distributed directories are suitable for distributed systems where shared objects are moved to those nodes that need them [2]. Tasks operate on local shared objects and if remote shared objects are required, a task must communicate through the directory to the remote nodes. In the distributed setting, *cache-coherence* for the shared objects ensures that writing to an object automatically *locates* and *invalidates* other cached copies of that object. A *distributed directory protocol* (DDP) [3] is a distributed directory implementation which realizes a coherence mechanism. Any DDP guarantees each *lookup* and *move* operation to the shared object in a distributed directory is individually atomic.

DDPs have a long history of research. They have widely been used in distributed shared memory implementations in multi-cache systems [3–5]. DDPs have also been used to implement fundamental problems in distributed systems, including distributed queues [6], mobile object tracking [7], and distributed mutual exclusion [8]. Very recently, DDPs have been studied for implementing transactional memory [9,10] in large-scale distributed systems [2,11–14].

☆ A preliminary version of this article appears in the Proceedings of SSS'18 [1].

* Corresponding author.
E-mail addresses: srai@kent.edu (S. Rai), sharma@cs.kent.edu (G. Sharma), kbusch@augusta.edu (C. Busch), mph@cs.brown.edu (M. Herlihy).

https://doi.org/10.1016/j.ic.2021.104700
0890-5401/

In the literature, the performance of a DDP was typically evaluated with respect to the *communication cost*, the total distance traversed by all the messages in the network. The ratio of the actual communication cost to the optimal cost provides an approximation ratio known as *stretch*. The goal is to guarantee stretch as small as possible. Existing DDPs such as Arrow [6], Relay [12], Combine [11], Ballistic [2], and Spiral [13] focus only on minimizing the stretch, while several other proposed DDPs [3–5] do not even have stretch analysis.

Processing load can also significantly affect the DDP performance. *Processing load* is measured as the worst node utilization, namely, the maximum number of times operations for objects use a node in the distributed directory. Load minimization is very important because it allows to evenly utilize available network resources (processing power, energy, etc.), avoiding the chance to create bottlenecks due to some "hotspot" resources [2]. Here, we present a novel DDP for general network topologies that simultaneously balances the processing load (minimizes maximum processing load) and minimizes the communication cost (minimizes stretch). The only previously known DDP that controls load and stretch simultaneously is MultiBend [14], which however is only suitable for the restricted case of mesh (grid) topologies. Furthermore, most of the techniques developed for MultiBend [14] to achieve such guarantees do not extend to general network topologies.

1.1. Problem statement

We describe the problem with respect to a single shared object, since for each object we can apply the same DDP solution. Consider a network and a set $\mathcal{E} = \{r_0, r_1, \ldots, r_\ell\}$ of operations to the shared object (ℓ does not need to be known and the bounds are independent of ℓ). The initial operation r_0 is to *publish* the shared object in the directory while the remaining, $r_1, r_2, \ldots, r_\ell$ are *move* operations for the object. The objective is to design a DDP that arranges the operations r_i, $i > 0$, in a total order (or a "distributed queue") [6]. Each operation r_i, $i > 0$, has a source node s_i, denoting the previous owner node, and a destination node t_i that issues r_i which will become the new object owner. The destination node of an operation r_{i_1} is the source node of another operation r_{i_2} in the total order, where the total order is a permutation of the requests in $\mathcal{E}$ that preserves the real time ordering. For every request r_i, the distributed directory provides a path p_i from s_i to t_i along which the object is transferred. Ideally, the collection of the paths minimize the load and the stretch. Formally,

- *Load balancing:* Minimize the maximum node *processing load* $PL = \max_v |\{i : v \in p_i\}|$. The processing load PL can be compared to the optimal processing load PL^* that is attainable by any DDP to provide a (processing load) approximation ratio.
- *Stretch:* Minimize total communication cost $A(\mathcal{E}) = \sum_{i=1}^{\ell} |p_i|$, where $|p_i|$ is the total length of the path p_i. $A(\mathcal{E})$ can be compared to the optimal cost $A^*(\mathcal{E})$ from the optimal algorithm OPT that has complete knowledge about $\mathcal{E}$ to provide a request ordering with minimal stretch $A(\mathcal{E})/A^*(\mathcal{E})$. We are interested to minimize $\max_{\mathcal{E}} A(\mathcal{E})/A^*(\mathcal{E})$.

1.2. Contributions

Let $G = (V, E, w)$ be a network with nodes V, edge set E, and weight function w on the edges in E (formal definition in Section 2) and $\mathcal{Z}$ a distributed directory built on it. Previously known DDPs can be classified into three categories, depending on the distributed directory $\mathcal{Z}$ on which they run. Arrow [6] and Relay [12] run on top of a *spanning tree* built from the graph G as a directory $\mathcal{Z}$. Ballistic [2] and Combine [11] run on top of an *overlay tree* constructed modifying the spanning tree of G as a directory $\mathcal{Z}$. Spiral [13] and MultiBend [14] run on top of a *hierarchy of clusters based overlay* constructed using leaders of the clusters of some hierarchical clustering of G as a directory $\mathcal{Z}$. Therefore, a node *participates* on the directory $\mathcal{Z}$ in each of the three categories discussed above if it is one of the nodes on the spanning tree, overlay tree, or a leader node of a cluster in the hierarchy of clusters based overlay.

It is easy to see that the processing load of a node in all the aforementioned DDPs is $O(\ell)$ in the worst-case, with the exception of MultiBend [14] which minimizes simultaneously both stretch and processing load but only on mesh network topologies. This is because once the directory $\mathcal{Z}$ (spanning, overlay, or hierarchical overlay) is constructed, the participating nodes on the directory always process the requests.

We present LB-Spiral, a new DDP for shared objects, that is suitable for general networks, and is load balanced and at the same time maintains low stretch. LB-Spiral is based on a hierarchy of clusters based overlay as a directory $\mathcal{Z}$. We build $\mathcal{Z}$ based upon the distributed directory $\mathcal{Z}$ we used in the our previous DDP, Spiral [13]. The difference is that Spiral minimizes only the stretch, while LB-Spiral minimizes both the stretch and the processing load. Both the load and stretch guarantees hold in any *arbitrary execution*, meaning that an initial *publish* operation can be followed by a sequence of *move* requests that may be initiated at arbitrary moments of time. This dual optimization needs non-trivial changes in the directory $\mathcal{Z}$ used in Spiral. We prove the following result.

Theorem 1. LB-Spiral *guarantees* $O(\log^3 n \cdot \log D)$ *amortized stretch and* $O(\log n \cdot \log D)$ *approximation of the processing load on any node in any general network* G *for any arbitrary execution, where* n *is the number of nodes and* D *is the diameter of* G.

Theorem 1 states that the processing load approximation is independent of the number of operations ℓ. This is in sharp contrast to the existing DDPs in general networks where processing load of some nodes is linearly dependent on ℓ, in

the worst-case. This is due to the fact the nodes in the higher levels of the directory $\mathcal{Z}$ process almost all the requests issued by the graph nodes for the shared object. Therefore, for some nodes, the processing load is $\Omega(\ell)$ in the worst-case, making it a hotspot resource. At the same time, the stretch approximation is remained independent of ℓ. It was not known in the literature whether the stretch approximation remains independent of ℓ when the processing load is considered simultaneously with stretch. Therefore, to our knowledge, this is the first DDP which achieves this simultaneous dual performance characteristic.

Looking into how significant the guarantees provided by LB-Spiral are, the stretch of LB-Spiral is optimal within a poly-log factor compared to the $\Omega(\log n/\log\log n)$ lower bound of Alon *et al.* [15] for the sequential execution scenario of *move* operations. The universal TSP lower bounds, such as $\Omega(\sqrt{\log n/\log\log n})$ by Jia *et al.* [16] for Euclidean metrics, $\Omega(\sqrt[6]{\log n/\log\log n})$ by Hajiaghayi *et al.* [17] for $n \times n$ grid, and $\Omega(\log n)$ by Gorodezky *et al.* [18] for Ramanujan graphs, apply to the communication cost of concurrent execution scenarios of *move* operations. For the processing load, if a node issues κ requests then its minimum processing load is at least κ. Our result shows that irrespective of whether a node issues requests or not and how many requests other nodes issue, its load is no more than $O(\log D \cdot \log n)$ factor of the minimum processing load.

LB-Spiral also provides guarantees for *lookup* operations. For any individual *lookup* operation, it guarantees $O(\log^6 n)$ stretch[1] which is in contrast to the *move* stretch that is obtained combining the costs of a set of *move* operations. This *lookup* stretch guarantee is particularly useful for read-dominated workloads. In the analysis, we do not consider the processing load for *lookup* operations as they do not update the directory $\mathcal{Z}$. However, if needed, for balancing processing load for *lookup* accesses we can use the same techniques as for *move* operations. The *publish* cost in LB-Spiral is proportional to the diameter of the network and it is a fixed initial cost which is only considered once and compensated by the costs of the *move* operations issued thereafter. The same approach has been used in previous DDPs [2,11–14] for *publish* operations.

For special network topologies that satisfy bounded doubling dimension properties [2,19] we obtain improved theoretical bounds, where we can show that LB-Spiral has both amortized *move* stretch and processing load of $O(\log D)$ in arbitrary executions. Furthermore, the stretch for a *lookup* operation is only $O(1)$. The benefit here is that the polylog(n) factors in the stretch and load bounds of general topologies are avoided while moving to doubling dimension topologies.

The theoretical results developed are tested through simulation experiments to see how well they translate in practice. The findings show that LB-Spiral performs well in practice for both stretch and processing load. We implemented and experimented LB-Spiral in random networks of different sizes that are generated using the Erdős-Rényi model [20].

1.3. Relation between processing load and network congestion

Traditionally, load balancing in networks has been studied through *network congestion* which is measured as the worst edge utilization and corresponds to the maximum number of times the object requests use any actual edge in the network. The DDP MultiBend [14] has been shown to control edge congestion in addition to stretch, albeit only for the restricted case of mesh network topologies (i.e., d-dimensional grids, $d \geq 2$). The processing load we studied in this paper relates to node congestion. Processing load deals with minimizing the number of times an overlay node of G is accessed as part of the overlay structure which runs on top of G. On the other hand, edge congestion deals with minimizing the number of times any actual edge of G, overlay or not, is serving on the actual paths in G realizing requests between two subsequent nodes of the overlay structure $\mathcal{Z}$.

It is well-known that simultaneous minimization of stretch and edge congestion is impossible in general network topologies (this is folklore knowledge and an example supporting this knowledge is provided in Busch *et al.* [21]). Therefore, stretch and edge congestion were studied independently in the existing research for general network topologies [22]. For DDPs, typically only stretch is considered for minimization in general network topologies [13,11,23,24], and Racke [22] gave the first algorithm to minimize edge congestion on general networks (ignoring stretch minimization). *Our result shows that simultaneous minimization of the processing load and stretch is the best possible we could do towards making DDPs load balanced and possibly many other problems on general networks, circumventing the impossibility barrier on simultaneous minimization of stretch and edge congestion.*

1.4. Techniques

The idea in LB-Spiral is to use an hierarchy of clusters based overlay as a directory $\mathcal{Z}$. We borrow the directory $\mathcal{Z}$ used in our previous DDP Spiral [13] but modify it in a novel way so that processing load is minimized as well as stretch (Spiral only minimizes stretch). Consider the directory $\mathcal{Z}$ used in Spiral. There are $h + 1 = O(\log D)$ levels of clusters in $\mathcal{Z}$ such that cluster diameters increase exponentially with respect to the level. In each cluster, one node is chosen to act as a leader. The leader node of a cluster in a level is connected through a shortest path to the leader node of a cluster in immediate upper level and also the immediate lower level. In other words, the leader node of a cluster at level 0 is connected to a leader node of a cluster at level 1, the leader node of a cluster at level 1 is connected to a leader node of a cluster at level 2, and so on. This connection of the leader nodes (of two subsequent levels) provides an overlay on the hierarchy of clusters $\mathcal{Z}$

[1] This *lookup* stretch guarantee is obtained through the use of special-parent node concept, without which it can be $O(D \cdot \log^3 n)$.

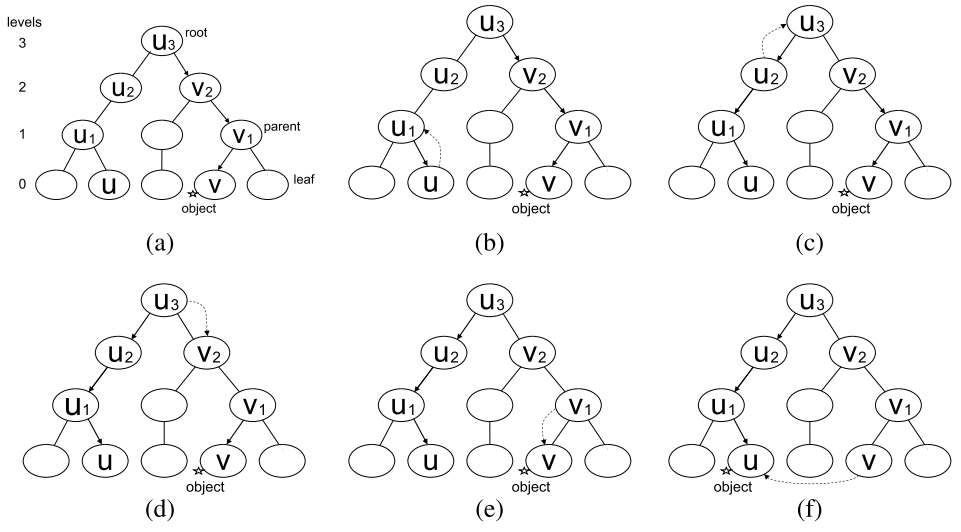

Fig. 1. An illustration of LB-SPIRAL for a *move* request issued by node u. Only leader nodes at respective clusters are shown. **a:** the creator node v issues a *publish* operation forming the initial downward directory path (based on the spiral path from v); **b:** node u issues a *move* request which follows a spiral path from u to the root, adjusting the pointers in subsequent levels to point towards u; **c:** the *move* request finds the downward directory path; **d:** the *move* request starts its down phase, deleting the old pointers of the directory path; **e:** the *move* request reaches previous owner node v; **f:** the object is moved from v to u making u the new owner node.

and is used to forward messages to different level clusters. Each node participates at all levels of the hierarchy, i.e., in each of the $h+1$ levels, there is at least a cluster each node $u \in G$ belongs to. This guarantees an existence of an overlay path from each node in level 0 up to the root node in level $h+1$. At the bottom level 0 each cluster consists of an individual node in G which is by default a leader. At the top level $h+1$ there is a single cluster for the whole graph G with a special leader node called *root*. At any intermediate level a node may belong to several clusters of that level. Only the bottom level nodes can issue *publish*, *lookup*, or *move* requests for the shared objects. The nodes in higher levels (which are also the graph nodes) implement the overlay structure and used to propagate the requests in the graph G. To minimize the load in addition to stretch in LB-SPIRAL, this directory $\mathcal{Z}$ is modified such that the leader is changed appropriately while serving a request so that the processing load of a node can be balanced at all times, irrespective of the number of operations $\mathcal{Z}$ serves. This leader change process only decides which node in a cluster becomes a new leader; the clusters connected in each level remain the same and they will never be changed. Therefore, the question is which node to of a cluster to make a new leader after change. Without this leader change, it can be shown that the processing load at some node in $\mathcal{Z}$ can be $\Omega(\ell)$, in the worst-case, where ℓ is the number of requests served by $\mathcal{Z}$.

Fig. 1 depicts how LB-SPIRAL works for a *move* operation. LB-SPIRAL maintains at all times a *directory path* which is directed from the root of $\mathcal{Z}$ to the bottom-level node of $\mathcal{Z}$ that *owns* the shared object. Particularly, this directory path points from level h leader to level $h-1$ leader, level $h-1$ leader to level $h-2$ leader, and so on. It ends at level 0 where level 1 leader points to it. The directory path is initialized by the first *publish* operation. The pointers are set when *publish* operation goes from the bottom level to the root level. After that, the directory path is updated whenever the object moves (changes ownership) from one node to another. To access the object, each bottom level node uses a *spiral path* to find and intersect the directory path. The spiral path of a bottom-level node $u \in \mathcal{Z}$ visits upward the leader nodes in all the clusters that it belongs to (note that u belongs at least a cluster in each level of $\mathcal{Z}$). While going up, the spiral path sets pointers to point to the immediate lower level leader from the current level leader. (The spiral path in G grows outwards from the origin as the level increases which gives the perception of a *spiral* formation.) It is guaranteed that a spiral path and the directory path intersect at some level of $\mathcal{Z}$ (in the worst-case at the highest level). Once they meet, a *move* operation forces the directory path to divert at the intersection point toward the new owner node (the origin of the spiral path). The existing directory path from the (met) level to the bottom-level node is now deleted. Thus, the directory path is maintained using the combinations of spiral path segments of *publish* and *move* operations. In fact, as soon as the object is created by some bottom level node, it publishes the object by following its spiral path towards the root, making each parent leader (we call this parent leader the parent node when we describe our algorithm in Section 4) pointing to its child leader (we call this child leader the child node in Section 4) and hence forming the *initial* directory path from root to a leaf node. The *initial* directory path is built on the spiral path from the creator node to the root. Fig. 1a shows the leaders in the cluster hierarchy after the successful publish operation of v with directory path from the root u_3 to v. When some node u issues a move request, the request goes upward following u's spiral path until it intersects the directory path to v (Figs. 1b–1c). While going up, the move request also sets downward links toward u. The move request resets the directory path it follows while descending towards the owner v (Figs. 1d–1e); the directory path now points to u. As soon as the move request reaches

v, the object is forwarded to u along some shortest path (Fig. 1f). A *lookup* operation is served similar to *move* without modifying the existing directory path.

For balancing the processing load, a node that initiates a *move* operation will become the leader of all the clusters it visits in the hierarchy until it intersects the directory path. That is, the leader of a cluster is changed to the node of that cluster that issued the request. Each affected cluster (in a level) is required to inform children and parent leaders, in immediate lower and higher levels, respectively, about the change on the leader node in the cluster and also transferring the directory path information from the old leader to the new leader, which we call *update overhead*. To bound the update overhead (which can be as much as $O(n)$ in the worst-case), the hierarchical clustering in the original Spiral protocol is modified appropriately so that in LB-Spiral a binary tree of clusters is formed between two subsequent levels of the hierarchy $\mathcal{Z}$. This helps to control the number of cluster leaders that need to be updated about the change on the cluster leader at any level. Particularly, informing only one child leader and only one parent leader will be enough. The new ideas on load balancing together with the approach of Spiral for stretch makes LB-Spiral to satisfy Theorem 1.

Although the techniques discussed above (leader change and binary tree of clusters to control update overhead) minimize *move* stretch and processing load, they may not be enough in minimizing the stretch for *lookup* operations. That means, the stretch for a *lookup* operation may still range from $O(\log^3 n)$ (in the best case) to $O(D \cdot \log^3 n)$ (in the worst case). We use a notion of *special-parent* node in the spiral path so that keeping the pointers on the special-parents while performing the *move* operations helps in guaranteeing *lookup* stretch of $O(\log^6 n)$ at all times. In other words, the factor of D is reduced to $O(\log^3 n)$ through the use of a spiral-parent. This *lookup* stretch is better when $D > \log^3 n$. If $D < \log^3 n$, then *lookup* can be executed without using special-parent concept to still guarantee $O(\log^6 n)$ stretch.

1.5. Related work

As mentioned earlier, the closest related works to ours are the previously known DDPs such as Arrow [6], Relay [12], Combine [11], Ballistic [2], Spiral [13], MultiBend [14], and many other directory algorithms [3–5]. Although these DDPs use some kind of overlay structures, their constructions, except MultiBend, are useful to minimize just the stretch. Although MultiBend simultaneously controls (edge) congestion and stretch, it is only tailored for mesh topologies.

Minimizing processing load is along the lines of research on *distributed hash table* protocols (DHTs) [25–28], where the load is minimized only for the nodes of G that participate as DHT protocol nodes. However, DHTs are different since they store key-value pairs by statically assigning keys (or objects) to nodes, whereas in DDPs objects are mobile.

The concept of LB-Spiral (also of Ballistic [2] and Spiral [13]) is similar to the approaches to locate nearest neighbors, tracking mobile users, compact routing, and related problems (e.g., [7,29–32]). However, these approaches provide efficient techniques only to locate copies and when the objects move autonomously (without being requested). The DDPs provide mechanisms that can make moving, looking up, and republishing of objects efficient and also avoid *race conditions* that might occur while synchronizing concurrent requests in distributed shared memory systems [2,13].

Finally, our study of minimizing processing load is different from existing studies where local *memory overhead* is considered for minimization in addition to stretch [29]. The memory overhead is minimized by distributing the storage of objects from the leader node to the other nodes in the cluster and later search them through embedding a De Bruijn graph in each cluster. It was shown [29] that the memory overhead can be just polylog(n) times the optimal. However, in these techniques, the worst processing load of a node (i.e., leader) is still linearly dependent on the total number of operations.

1.6. Paper organization

The rest of the paper is organized as follows. We describe the network model in Section 2. We describe the hierarchical clustering we use for LB-Spiral in Section 3. We then detail LB-Spiral in Section 4 and analyze it in Section 5. We discuss extensions to doubling-dimension graphs in Section 6 and simulation results in Section 7. Finally, we conclude in Section 8 with a short discussion.

2. Network model

We represent a distributed network as a weighted graph $G = (V, E, \mathfrak{w})$, with nodes (network machines) V, where $|V| = n$, edges (interconnection links between machines) $E \subseteq V \times V$, and edge weight function $\mathfrak{w} : E \to \mathbb{R}^+$. We assume that $\mathfrak{w}(u, u) = 0$ for any $u \in V$. A path p in G is a sequence of nodes, with respective sequence of edges connecting the nodes, such that $|p| = \sum_{e \in p} \mathfrak{w}(e)$. For convenience, we will treat paths as walks, which may possible visit a node multiple times. A *sub-path* of p is any subsequence of consecutive nodes in p; we may also refer to a sub-path as a *fragment* of p. We assume that G is connected, i.e., there is a path in G between any pair of nodes. Let dist(u, v) denote the shortest path length (distance) between nodes u and v in G. The *k-neighborhood* of a node $v \in G$ is the set of nodes which are within distance at most k from v (including v). The k-neighborhood essentially describes a circle with center v and radius k. The *diameter* D is the maximum shortest path distance over all pairs of nodes in G.

We assume that G represents a network in which nodes do not crash, it implements FIFO communication between nodes (i.e. no overtaking of messages occurs), and messages are not lost. The previous DDPs [2,6,12–14] (except Combine [11]) have the FIFO assumption. We also assume that, upon receiving a message, a node is able to perform a local computation and

Table 1
Commonly used notations.

Symbol	Meaning
$G, V, E, \mathfrak{w}$	Graph, nodes, edges, edge weight function
D	Diameter of G; i.e., diam(G)
p	A path in G
dist(u, v)	The distance between two nodes $u, v \in G$
$s(u, v)$	The distance between two nodes $u, v \in G$ that is shortest
X	A (node) cluster in G
diam(X)	The diameter of cluster X
h	$\lceil \log D \rceil + 1$
$Z(u)$	The set of clusters that $u \in G$ belongs to
$Z = \{X_1, \ldots, X_k\}$	A cover cluster set where each node $u \in G$ belongs to at least one cluster X_i, $1 \leq i \leq k$, i.e., $\lvert Z(u) \rvert \geq 1$
γ	The locality parameter
$\mathcal{Z} = \{Z_0, \ldots, Z_h\}$	A (σ, χ)-labeled cover hierarchy with $h+1$ levels built from Z
χ	The cluster labeling parameter
ξ	The shared object
r_i	*publish*, *move*, or *lookup* operation on ξ
$\mathcal{E}$	Set of publish, move or lookup operations
s_i	Previous owner node of ξ
t_i	New owner node of ξ
δ	The height of "dummy" tree T
$(h, 1)$	The highest level in $\mathcal{Z}$
$(0, \chi)$	The lowest level in $\mathcal{Z}$
sub-level (i, j)	Level i and label j
next(i, j)	sub-level immediately higher than (i, j) (i.e., sub-level $(i, j+1)$ or $(i+1, 1)$)
prev(i, j)	sub-level immediately lower than (i, j) (sub-level $(i, j-1)$ or $(i-1, \chi)$)
$X_{i,j}(u) \in Z_i(u)$	A cluster at sub-level (i, j) that u belongs to
$X_{i,\chi+\delta-1}(u) \in Z_i(u)$	The highest sub-labeled cluster at level i that u belongs to
$X_{i,1}(u) \in Z_i(u)$	The lowest sub-labeled cluster at level i that u belongs to
$\ell(X)$	the leader node of cluster X
$p(u)$	Spiral path of node u
$\text{parent}_{(i,j)}(x)$	If x is a leader node of a sub-level (i, j) cluster, then $\text{parent}_{(i,j)}(x)$ is the leader node of sub-level next(i, j) cluster
$\text{sparent}_{i,j}(x)$	If x is a leader node of a sub-level (i, j) cluster, then $\text{sparent}_{(i,j)}(x)$ is the leader node of a cluster at level $i+\phi$ (the value of ϕ is given in Section 4.6)

send a message in a single atomic step. LB-Spiral can be extended to accommodate non-FIFO communication and tolerate unreliable communication links (i.e., message losses) by adapting techniques used in Combine [11].

3. Hierarchical clustering

We describe a hierarchy of clusters based overlay built on top of G to run our load balanced DDP LB-Spiral which we present in Section 4. We also define spiral and directory paths that will be useful in LB-Spiral. Fig. 2 provides an illustration of the ideas used and developed in the construction. We first discuss the labeled cover hierarchy we built for our previous DDP Spiral. We then discuss why it is not suitable to optimize the processing load and the modifications we introduce to be able to balance the processing load. This section is heavy in definitions and terminologies. Therefore, the terminologies commonly used are summarized in Table 1.

3.1. Labeled cover hierarchy build for Spiral

We describe here the labeled cover hierarchy we built for our previous DDP Spiral [33]. A *cluster* is any set of nodes $X \subseteq V$. Particularly, a cluster X may have one or more nodes of G. The diameter of a cluster X is the maximum distance between any two nodes in X, i.e., $\text{diam}(X) = \max_{u,v \in X} \text{dist}(u, v)$, where distances are w.r.t. G. Some of the concepts are shown in Fig. 2. We define the *cover* cluster set as follows.

Definition 1 (***cover cluster set***). Consider a set of clusters $Z = \{X_1, X_2, \ldots, X_k\}$. If Z satisfies the condition that $u \in V$ is in at least one cluster in Z, then we call Z the *cover cluster set* of G.

The diameter of the cover cluster set Z is the maximum diameter of its clusters: $\text{diam}(Z) = \max_{X_i \in Z} \text{diam}(X_i)$. Let $Z(u)$ denote the set of clusters in the cover set Z that u belongs to. We have that $\lvert Z(u) \rvert \geq 1$. We define the *locality* of the cover cluster set Z as follows.

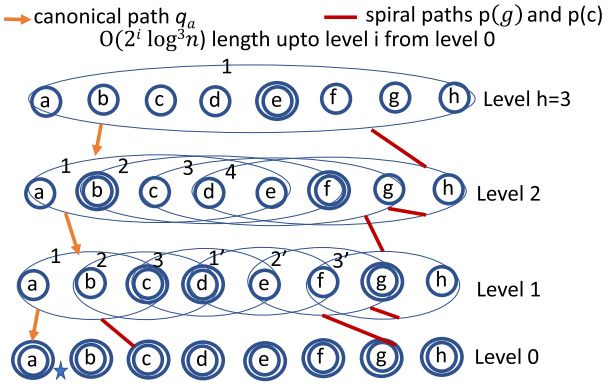

Fig. 2. Illustration of an example labeled cover hierarchy $\mathcal{Z}$ built on a graph G as well as the spiral and full and partial canonical paths. For simplicity, the construction considers 8 nodes a–h on a line. We assume that the distance between two consecutive nodes is 1. In level 0, each node is a cluster and a leader itself (nodes double circled) on that cluster. In level i, the nodes in the radius of 2^{i-1} are in a cluster. The clusters in each level are labeled (see the number of the oval shape representing a cluster with its labeling). A spiral path $p(g)$ is shown for node g from level 0 to level h such that it visits all the clusters in all the levels it belongs to in the increasing order of their labels. Particularly, $p(g)$ visits clusters 2', 3' (in level 1), 3 and 4 (in level 2), and 1 in level h. The (full) canonical path q_a for node a visits cluster 1 in all levels 1 to h. Node c has a partial canonical path with q_a from level h to level 1 and then spiral path $p(c)$ from level 1 to level 0. The length of the (canonical and spiral) paths is $O(2^i \cdot \log^3 n)$ from level 0 up to level i. Two concepts are not shown in this figure: (i) The shortest paths that are used to connect leader nodes of two consecutive clusters in the paths, and (ii) The binary tree of clusters between to consecutive levels.

Definition 2 (***cover locality***). We say that the cover cluster set Z has *locality* γ if for a node $u \in G$ there is some cluster $X \in Z(u)$ such that u belongs to X and X contains the γ-neighborhood of u, i.e., X contains all the nodes in G that are at distance $\leq \gamma$ from u.

A χ-*labeling* of the cover cluster set Z, for some positive integer χ, is an assignment of integer labels to its clusters, $\lambda(X_i) \in \{1, 2, \ldots, \chi\}$. We define *validity* of the χ-labeling of Z as follows.

Definition 3 (***valid labeling***). A χ-labeling of Z is *valid* if for each node $u \in V$ every cluster $X \in Z(u)$ that contains u has a different label, that is, if $X_i, X_j \in Z(u)$, $i \neq j$, then $\lambda(X_i) \neq \lambda(X_j)$.

Fig. 2 shows an example of a valid labeling so that no two clusters in $Z(u)$ at a level receive the same label. Labels are useful later in the DPP LB-Spiral to route the requests, particularly which cluster to pick (when there are multiple clusters to choose from) to forward the request from the current cluster.

Definition 4 (***labeled cover***). Z is a (σ, χ, γ)-*labeled cover* if Z is a cover cluster set with locality γ, $\text{diam}(Z) \leq \sigma\gamma$, and accepts a valid χ-labeling.

Consider the cover cluster set Z defined above. We define cover cluster set hierarchy $\mathcal{Z} = \{Z_0, \ldots, Z_h\}$, $h = \lceil \log D \rceil + 1$, with cover locality $\gamma_i = 2^{i-1}$, $1 \leq i \leq h$. We call $Z_i \in \mathcal{Z}$ the *level i cover* (i.e., cover cluster set Z for level i), and any cluster $X \in Z_i$ a *level i cluster*. We call $\mathcal{Z}$ labeled cover cluster hierarchy if each level i cover cluster set Z_i has a valid χ-labeling. Formally,

Definition 5 (***labeled cover hierarchy***). $\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_h\}$ is a (σ, χ)-*labeled cover hierarchy* for G when each Z_i, $1 \leq i \leq h$, is a (σ, χ, γ_i)-labeled cover cluster set with locality $\gamma_i = 2^{i-1}$, where $Z_0 = V$ (each node in V is a cluster) and $h = \lceil \log D \rceil + 1$.

We built a $(O(\log n), O(\log n))$-labeled cover hierarchy $\mathcal{Z}$ for our previous DDP Spiral, i.e., $\sigma = O(\log n)$ and $\chi = O(\log n)$ in Definition 5.

The construction of $(O(\log n), O(\log n))$-labeled cover hierarchy $\mathcal{Z}$ uses the concept of laminar partition hierarchies defined as follows. A *partition* of G is a cover consisting of disjoint clusters of nodes. A *laminar partition hierarchy* $\mathcal{P} = \{P_0, P_1, \ldots, P_{h'}\}$, where $h' = \lceil \log D \rceil$, has the following properties:

(i) $P_{h'}$ is a single cluster that consists of all nodes in V;
(ii) each P_i is a partition with cluster diameter at most 2^i;
(iii) each cluster in P_i is completely contained in some cluster in P_{i+1}, for $0 \leq i \leq h'-1$.

Node $v \in V$ is called *α-padded* w.r.t. $\mathcal{P}$ if $\alpha 2^i$-neighborhood of v is included in a cluster of P_i in every level i, where $0 \leq i \leq h'$. There exists a family of $l = O(\log n)$ laminar partition hierarchies $\mathcal{F} = \{\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^l\}$, such that every node $v \in V$ is $\Omega(1/\log n)$-padded in at least one of the partition hierarchies in $\mathcal{F}$ [34].

The $(O(\log n), O(\log n))$- labeled cover hierarchy we built transformed the family of laminar partition hierarchies $\mathcal{F}$ to an appropriate (σ, χ)-labeled cover hierarchy $\mathcal{Z}$. The cover cluster set for level i, $Z_i \in \mathcal{Z}$, is obtained from the union of all the level $j_i = i + \lfloor \log(c \log n) \rfloor$ partitions, namely, $Z_i = \bigcup_{\mathcal{P} \in \mathcal{F}} P_{j_i}$, for $1 \leq i \leq h$; in case $j_i > h'$, then, we use $Z_i = \bigcup_{\mathcal{P} \in \mathcal{F}} P_{h'}$. We set $Z_0 = V$, namely every node in level 0 is a cluster. Note that the padding of $\mathcal{F}$ we use in this construction is $1/(c \log n)$ for some constant c.

The locality of the cover cluster set Z_i is $\alpha 2^{j_i} = 2^{i-1} \cdot c \log n / c \log n = \gamma_i$, since a-padding implies that there is a cluster C in partition level j_i that includes a node u and its $\alpha 2^{j_i}$ neighborhood, and this cluster C appears in level i of $\mathcal{Z}$. Note that according to the definition of the partition, $\text{diam}(Z_i) \leq 2^{j_i} \leq 2^i \cdot c \log n \leq 2c\gamma_i \log n$. Therefore, we can set $\sigma = 2c \log n$.

We can get a χ-labeling of each cover Z_i as follows. If a cluster $X \in Z_i$ came from partition hierarchy $\mathcal{P}^k$ then it obtains label $\lambda(X) = k$. This implies that we will have $\chi = l = O(\log n)$ labels. The resulting labeling is valid, since for each level $Z_i \in \mathcal{Z}$, each cluster is obtained from a different partition hierarchy in $\mathcal{F}$, and thus we can not have any two clusters in $Z_i(u)$ with the same label. After all this, we perform normalization so that the resulting $(O(\log n), O(\log n))$-labeled cover hierarchy $\mathcal{Z}$ satisfies the following properties.

1. At level 0 each node in V belongs to exactly one cluster consisting of only itself.
2. Cover Z_h (highest level) consists of $\chi = O(\log n)$ copies of a cluster that contains all nodes V, where each copy is obtained from a different partition hierarchy in $\mathcal{F}$. We keep only one copy and remove the rest so that there is only one cluster at level h of the hierarchy.
3. In any level i, $1 \leq i \leq h-1$, of $\mathcal{Z}$ each node $u \in V$ belongs to exactly $\chi = O(\log n)$ clusters (one cluster from each partition hierarchy of $\mathcal{F}$); that is, $|Z_i(u)| = \chi$. (Some clusters could be identical.) Repeated clusters will be treated as different and will be assigned a different label.
4. Each cluster at level i, $0 \leq i < h$, is completely contained by a cluster at level $i+1$. This is due to the laminar decomposition property used in the construction.

3.2. Problems of the labeled cover hierarchy of Spiral for load balancing and modifications

As discussed in Section 1.4, to balance the processing load, a node that initiates a move operation becomes a leader in all the clusters of $\mathcal{Z}$ it visits following its spiral path until its spiral path intersects the directory path. This means that the old leaders of all the affected clusters have to be notified of this leader change. At a level, this needs informing all child and parent clusters in immediate lower and upper levels from that level about the change on the leader node and also transferring the directory path information from the old leader to the new leader. Minimizing this update overhead requires a small bound on the number of clusters (and respective leaders) that need to be informed. The $(O(\log n), O(\log n))$-labeled cover hierarchy $\mathcal{Z}$ construction used in Spiral discussed in Section 3.1 is not suitable for this as it does not bound the number of clusters in the cover cluster set Z_{i-1} (for level $i-1$) that are completely contained inside each cluster in the cover cluster set Z_i (for level i). We achieve the small bound on the number of clusters by forming a binary tree of clusters between two subsequent levels of $\mathcal{Z}$ so that only a constant number of clusters (in fact at most 2 each, parent cluster and child cluster) need to be informed about the leader change in the immediate upper and lower levels. Within a level, clusters are visited in the order of the sub-levels, and hence the binary tree not needed.

We now discuss how the binary tree of clusters is constructed (see Fig. 3) and $\mathcal{Z}$ is modified. Let CL_j be a cluster in the level i cover cluster set $Z_i \in \mathcal{Z}$. Let $W = \{CL^1_{j-1}, CL^2_{j-1}, \ldots, CL^w_{j-1}\}$ be the clusters in the level $i-1$ cover cluster set $Z_{i-1} \in \mathcal{Z}$ so that each cluster $CL^l_{j-1}, 1 \leq l \leq w$, is completely contained inside CL_j. We organize the clusters in W in a "dummy" *binary tree* T of clusters as follows. CL_j acts as the root cluster of T. Each cluster in $CL^l_{j-1} \in W$ acts as a leaf cluster of T, i.e., there will be w leaf clusters in T. In every level $m \geq 1$ of T merge two children clusters at level $m-1$ to obtain the parent cluster at level m. According to this construction, if there are Δ clusters at any level $m-1$ of T, then at level m of T, there will be at most $\lceil \Delta/2 \rceil$ clusters. Fig. 3 illustrates the binary tree T construction of clusters in the set $Z_{i-1} \in \mathcal{Z}$ that are completely contained inside a cluster in the set $Z_i \in \mathcal{Z}$.

Lemma 1. *If there are w clusters in the set $Z_{i-1} \in \mathcal{Z}$ completely contained inside a cluster CL_i in the set $Z_i \in \mathcal{Z}$, then a binary tree T can be embedded between levels $i-1$ and i with root of T being the cluster CL_i such that T has $O(\log w)$ levels.*

Proof. We have that $w \leq n$, T is a binary tree, and in each level m of the tree T, two children clusters at level $m-1$ are merged to form a cluster at level m. Therefore, there will be at most $O(\log w)$ levels in T. □

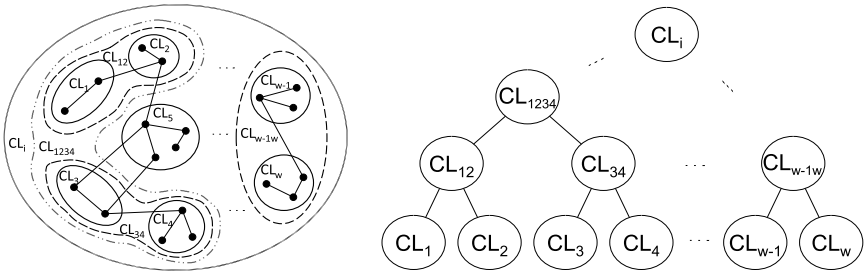

Fig. 3. An illustration of a binary tree embedded between a cluster CL_i at level i and the clusters at level $i-1$ that are completely contained inside CL_i.

Consider a level $i-1$ cluster $X_{i-1}(u)$ that a node $u \in G$ belongs to. In the tree T of clusters we built above, node u belongs to all the clusters in the path of tree T from $X_{i-1}(u)$ to the root cluster X_i at level i. This is because since $X_{i-1}(u)$ is completely contained inside X_i (due to the laminar construction that we use), u belongs to X_i and the clusters in each level of T are formed merging two clusters in the lower levels.

3.3. Overlay tree structure on the labeled cover hierarchy

We describe here how the labeled cover hierarchy is organized as an overlay tree structure. Let $\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_h\}$ be a (σ, χ)-labeled cover hierarchy. We need some definitions. Let $X_{i,j}(u) \in Z_i(u) \subseteq Z_i$ denote the cluster at level i, $1 \le i \le h-1$, that u belongs to and has label j; note that $Z_i(u)$ denotes all the clusters in the cover cluster set Z_i that u belongs to. We will refer to level i, label j, as the *sub-level* (i, j). Note that level i consists of χ sub-levels $(i,1), (i,2), \ldots, (i,\chi)$, for $1 \le i \le h-1$. Levels 0 and h are special cases which consist of a single sub-level each which for convenience we denote as $(0, \chi)$ and $(h, 1)$, respectively. We can order the sub-levels lexicographically so that $(i, j) < (i', j')$ if $i < i'$, or $i = i'$ and $j < j'$. We define the function next(i, j) (resp. prev(i, j)) to return the sub-level immediately higher (resp. lower) than (i, j).

This ordering can be extended also to the clusters that are organized in a binary tree T (Lemma 1) between two subsequent levels i and $i+1$. Let $X_{i,\chi}(u) \in Z_i(u)$ be a cluster at level i that u belongs to and has label χ and let $X_{i+1,1}(u) \in Z_{i+1}(u)$ be a cluster at level $i+1$ that u belongs to and has label 1. We assign levels to the clusters in the respective binary tree T in a path from $X_{i,\chi}(u)$ (a leaf of T) to $X_{i+1,1}(u)$ (the root of T) from $(i, \chi+1)$ to $(i, \chi+\delta-1)$, where $\delta \le \lceil \log n \rceil$ is the maximum height of T (Lemma 1). Since $X_{i,\chi}(u)$ is a leaf cluster in tree T, and it is merged with some other cluster to form a cluster in the next level, u belongs to each cluster in the path of T from the leaf cluster $X_{i,\chi}(u)$ up to the root cluster $X_{i+1,1}(u)$. Note also that according to the laminar construction we use, $X_{i,\chi}(u)$ is completely contained inside $X_{i+1,1}(u)$. Therefore, there will be $\chi+\delta-1$ sub-levels in each level i, summing the χ sub-levels of level i and the δ new levels introduced due to tree T. There may be the case the different trees T have different heights. We can normalize all such binary trees in the hierarchy $\mathcal{Z}$ to have the same height δ by repeating the root cluster if necessary.

In every cluster X we choose a designated *leader* node $\ell(X)$ arbitrary initially and changed later appropriately to balance the processing load. Denote the leader of cluster $X_{i,j}(u)$ as $\ell_{i,j}(u) = \ell(X_{i,j}(u))$. Since Z_h consists of a single sub-level it has a unique leader which we denote $\ell_{h,1}(u) = r$. Trivially, every node $u \in V$ is a leader of its own cluster at level 0, $\ell_{0,\chi}(u) = u$. For any pair of nodes $u, v \in V$, let $s(u, v)$ denote a shortest path from u to v.

We construct the overlay structure on $\mathcal{Z}$ as follows: for each node $u \in G$, connect the leader node $\ell(X_{i,j}(u))$ of sub-level (i, j) cluster $X_{i,j}(u)$ to the leader node $\ell(X_{\text{next}(i,j)}(u))$ of sub-level next(i, j) cluster $X_{\text{next}(i,j)}(u)$ using the shortest path $s(\ell(X_{i,j}(u)), \ell(X_{\text{next}(i,j)}(u)))$, $1 \le i \le h-1$, $1 \le j \le \chi$.

3.4. Construction time and information that nodes keep

This construction of $\mathcal{Z}$ is a one time process, which happens prior to LB-Spiral starts serving (shared) object requests. The modification on the construction during execution is only on which nodes act as leader nodes of the clusters. The leader change does not modify $\mathcal{Z}$ itself as the clusters, their localities, and their sub-labelings remain unchanged. The only change is which node in a cluster takes charge on processing the requests (i.e., becomes a leader). Furthermore, the construction time of $\mathcal{Z}$ is polynomial as labeled cover partitioning used in the construction in each level (of some locality) can be performed in polynomial time and there are $O(\log D)$ levels.

After the construction, each node maintains information related to $\mathcal{Z}$ and related to LB-Spiral. We discuss here what information a node maintains about $\mathcal{Z}$. We will discuss later in Section 4.7 what information a node maintains related to LB-Spiral, additionally to the information it maintains about $\mathcal{Z}$. Let $u \in G$ be a node in sub-level (i, j) cluster X. Node u maintains information on which nodes of G belong to X and how to reach to those nodes (i.e., the shortest paths from u to each other node in X). If u is a leader node in X, it additionally maintains information on which are the leader nodes of the next(i, j) cluster X_{next} and the prev(i, j) cluster X_{prev} and how to reach to them (this is the information u needs for the spiral path and canonical paths discussed next). It is assumed that leader nodes know how to reach to the leader nodes of the next(i, j) cluster X_{next} and the prev(i, j) cluster X_{prev}. In fact, we assume that, each node in a cluster at level (i, j)

knows the shortest path to reach to any node in the next(i, j) cluster X_{next} and any node in the prev(i, j) cluster X_{prev}. Therefore, when a leader node is changed in those clusters, the leader in the level (i, j) cluster immediately knows how to forward requests to those leaders. This information only has impact on the memory requirement (one time computation in the construction process), which we do not focus in this paper. In the algorithm description, for a leader node in level (i, j) cluster $X_{i,j}$, we use "parent" notion to denote the leader node of next(i, j) cluster X_{next} and "child" notion to denote the leader node of prev(i, j) cluster X_{prev}. Formal definition is given in Section 4. Furthermore, node u stores the information about its special-parent node (the details on which node acts as the special-parent of u and how u finds it becomes clear in Section 4.6).

3.5. Spiral paths

Let $\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_h\}$ be a (σ, χ)-labeled cover hierarchy with an overlay structure constructed by connecting the consecutive sub-level leaders. The spiral path $p(u)$ for each node $u \in V$ is built by visiting designated leader nodes in all the clusters that u belongs to starting from level 0 up to level h in $\mathcal{Z}$. Particularly, the spiral path visits the consecutive leader nodes in $\mathcal{Z}$, i.e., within each level, the clusters are visited according to the order of their labels, and between the levels, the clusters are visited based on the clusters in the binary tree, starting from leaf level going to the root. From an abstract point of view, the path forms a spiral which slowly unwinds outwards while it visits cluster leaders of higher levels which are possibly further away from u.

For any set of nodes $u_1, u_2, \ldots, u_k \in V$, let $s(u_1, u_2, \ldots, u_k)$ denote the concatenation of shortest paths $s(u_1, u_2)$, $s(u_2, u_3), \ldots, s(u_{k-1}, u_k)$. The spiral path $p(u)$ is formed by taking the concatenation of the shortest paths that connect the ascending sequence of leaders starting from node u (sub-level $(0, \chi)$) up to node r (sub-level $(h, 1)$). Formally,

Definition 6 (***spiral path***). The spiral path of node u is:

$$p(u) = s(u, \underbrace{\ell_{1,1}(u), \ldots, \ell_{1,\chi}(u)}_{\text{level 1}}, \underbrace{\ell_{1,\chi+1}(u), \ldots, \ell_{1,\chi+\delta-1}(u)}_{\text{between level 1 and 2}}, \underbrace{\ell_{2,1}(u), \ldots, \ell_{2,\chi}(u)}_{\text{level 2}}, \ldots,$$
$$\underbrace{\ell_{h-1,1}(u), \ldots, \ell_{h-1,\chi}(u)}_{\text{level } h-1}, \underbrace{\ell_{h-1,\chi+1}(u), \ldots, \ell_{h,\chi+\delta-1}(u)}_{\text{between level } h-1 \text{ and } h}, r).$$

We say that two paths *intersect* if they have a common node. We also say that two spiral paths intersect at level i if they visit the same leader at level i.

Lemma 2. *For any two nodes $u, v \in V$, their spiral paths $p(u)$ and $p(v)$ intersect at level* $\min\{h, \lceil \log(\text{dist}(u, v)) \rceil + 1\}$.

Proof. It is trivial to see that $p(u)$ and $p(v)$ intersect on level h at node r. Suppose $\iota = \lceil \log(\text{dist}(u, v)) \rceil + 1 \leq h$. From the definition of $\mathcal{Z}$ from Section 3.1, the clusters at level ι have locality $\gamma_\iota = 2^{\iota-1} \geq \text{dist}(u, v)$. Thus, some cluster $X \in Z_\iota(u)$ will contain v. Therefore, the paths $p(u)$ and $p(v)$ intersect at leader node $\ell(X)$. □

In the analysis of LB-Spiral, the directory path is obtained from fragments of spiral paths obtained from *move* operations. Such a fragmented path is actually a concatenation of shortest paths connecting leaders at successive sub-levels whose clusters share a common node. We will refer to such kind of path as *canonical*.

Definition 7 (***canonical path***). A canonical path q up to sub-level $(k, \iota) \leq (h, 1)$ is:

$$q = s(x_{0,\chi}, \underbrace{x_{1,1}, \ldots, x_{1,\chi}}_{\text{level 1}}, \underbrace{x_{1,\chi+1}, \ldots, x_{1,\chi+\delta-1}}_{\text{between level 1 and 2}}, \underbrace{x_{2,1}, \ldots, x_{2,\chi}}_{\text{level 2}},$$
$$\underbrace{x_{2,\chi+1}, \ldots, x_{2,\chi+\delta-1}}_{\text{between level 2 and 3}}, \ldots, \underbrace{x_{k,1}, \ldots, x_{k,\iota}}_{\text{level } k}),$$

such that for any two consecutive nodes $x_{i,j}$ and $x_{\text{next}(i,j)}$, where $(0, \chi) \leq (i, j) < (k, \iota)$, there is a node $y \in V$ with $x_{i,j} = \ell_{i,j}(y)$ and $x_{\text{next}(i,j)} = \ell_{\text{next}(i,j)}(y)$.

We will refer to $x_{0,\chi}$ and $x_{k,\iota}$ as the *bottom* and *top* nodes of q, respectively. The bottom node is always at level 0. A canonical path can be either *partial* when the top node is below level h (the root level) or *full* when the top node is the root r. A spiral path $p(u)$ is a full canonical path, and any prefix of it is a partial canonical path. Fig. 2 depicts some of these ideas.

Lemma 3. *For any canonical path q up to level k (and any sub-level (k, ι)) in $\mathcal{Z}$, $\text{length}(q) \leq c_3 2^{k+2} \log^3 n$, for some constant c_3.*

Proof. We first consider only the cost in traversing the sub-levels χ. Consider two consecutive nodes $x_{i,j}, x_{\text{next}(i,j)} \in q$, where $(0, \chi) < (i, j) < (k, \iota)$. From the definition of canonical paths, there is a node $y \in V$ with $x_{i,j} = \ell_{i,j}(y)$ and $x_{\text{next}(i,j)} = \ell_{\text{next}(i,j)}(y)$. Therefore,

$$\begin{aligned}\text{dist}(x_{i,j}, x_{\text{next}(i,j)}) &= \text{dist}(\ell_{i,j}(y), \ell_{\text{next}(i,j)}(y)) \\ &\leq \text{dist}(y, \ell_{i,j}(y)) + \text{dist}(y, \ell_{\text{next}(i,j)}(y)) \\ &\leq \text{diam}(X_{i,j}(y)) + \text{diam}(X_{\text{next}(i,j)}(y))\end{aligned}$$

There are two cases:

i. $\text{next}(i, j) = (i, j + 1)$: clusters $X_{i,j}(y)$ and $X_{\text{next}(i,j)}(y)$ are at the same level i. We have $\text{diam}(X_{i,j}(y)) \leq \sigma\gamma_i$ and $\text{diam}(X_{\text{next}(i,j)}(y)) \leq \sigma\gamma_i$. Since $\sigma = O(\log n)$, and $\gamma_i = 2^{i-1}$, we get $\sigma\gamma_i \leq c_1 2^{i-1} \log n$, for some constant c_1. Thus, $\text{dist}(x_{i,j}, x_{\text{next}(i,j)}) \leq c_1 2^i \log n$.
ii. $\text{next}(i, j) = (i + 1, 1)$: clusters $X_{i,j}(y)$ and $X_{\text{next}(i,j)}(y)$ are at levels i and $i + 1$, respectively. We have that $\text{diam}(X_{i,j}(y)) \leq \sigma\gamma_i \leq c_1 2^{i-1} \log n$ and $\text{diam}(X_{\text{next}(i,j)}(y)) \leq \sigma\gamma_{i+1} \leq c_1 2^i \log n$. Which gives $\text{dist}(x_{i,j}, x_{\text{next}(i,j)}) \leq c_1(2^{i-1} + 2^i) \log n \leq c_1 2^{i+1} \log n$.

We define the following subpaths of q:

$$\begin{aligned}q_i &= s(x_{i-1,\chi}, x_{i,1}, x_{i,2}, \ldots, x_{i,\chi}), \quad \text{for } 1 \leq i < k \\ q_k &= s(x_{k-1,\chi}, x_{k,1}, x_{k,2}, \ldots, x_{k,\iota})\end{aligned}$$

When we apply case ii for the first two nodes in q_i and case i for the remaining pairs of nodes, we obtain $\text{length}(q_i) \leq \chi c_1 2^i \log n$. Since $\chi = O(\log n)$, we have that $\chi \leq c_2 \log n$, for some constant c_2. Therefore, $\text{length}(q_i) \leq c_1 c_2 2^{i+1} \log^2 n$. Similarly, $\text{length}(q_k) \leq c_1 c_2 2^{k+1} \log^2 n$. Finally, we obtain:

$$\begin{aligned}\text{length}(q) &= \sum_{i=1}^{k} \text{length}(q_i) \leq c_1 c_2 (\log^2 n) \sum_{i=1}^{k} 2^{i+1} \\ &\leq c_1 c_2 2^{k+2} \log^2 n \leq c_3 2^{k+2} \log^2 n,\end{aligned}$$

for some constant $c_3 = c_1 c_2$.

Consider now the cost in traversing the δ levels in the tree T. Note that the cluster $X_{i-1,\chi}(u)$ and $X_{i,1}(u)$ have diameters such that $\text{diam}(X_{i-1,\chi}(u)) \leq \sigma\gamma_i \leq c_1 2^{i-1} \log n$ and $\text{diam}(X_{i,1}(u)) \leq \sigma\gamma_{i+1} \leq c_1 2^i \log n$. Which gives $\text{dist}(x_{i-1,\chi}, x_{i,1}) \leq c_1(2^{i-1} + 2^i) \log n \leq c_1 2^{i+1} \log n$. Since we have from Lemma 1 that T has $O(\log n)$ levels, $\text{length}(q)$ increases by at most a factor of $c_5 \cdot \log n$ due to the embedding of T between the clusters at two subsequent levels, where c_5 is some constant. Therefore, $\text{length}(q) \leq c_3 2^{k+2} \log^3 n$, for some constant $c_3 = c_4 \cdot c_5$. □

4. LB-Spiral Algorithm

We now present LB-Spiral, which is a load balanced DDP. We describe LB-Spiral for one shared object as it is typical in the DDP literature; multiple objects can be supported replicating the hierarchy for each object.

4.1. Overview of LB-Spiral

Consider some shared object ξ. LB-Spiral guarantees that at any time only one node holds the shared object ξ, which is the *owner* of the object. The owner is the only node that can modify (i.e., write) the object; the other nodes can only access the object for read.

LB-Spiral is implemented on the $(O(\log n), O(\log n))$-labeled cover hierarchy $\mathcal{Z}$ discussed in Section 3. Only the bottom level nodes of $\mathcal{Z}$ can issue requests (*publish*, *lookup*, and *move*) for ξ, while nodes in higher levels of $\mathcal{Z}$ are used to propagate the requests in G. The basic objective of LB-Spiral is to maintain a *directory path* in $\mathcal{Z}$ as in Spiral which is directed from the root node r to the bottom-level node that is the current owner of ξ. The directory path is updated whenever ξ moves from one node to another. Initially, the directory path is formed from the spiral path $p(v)$ of the object creator node v. As soon as the object ξ is created, v publishes ξ by visiting the leaders in its spiral path $p(v)$ towards the root r, making each parent leader node pointing to its child leader (Fig. 1a). These leader downward pointers correspond to path segments between consecutive leaders and the concatenation of these path segments from the root r down to v forms the initial directory path.

Algorithm 1: LB-SPIRAL.

```
1  When y receives m = ⟨v, up, publish⟩ from x:                    // publish
2    execute publish algorithm (Algorithm 2);
3  When y receives m = ⟨u, phase, lookup⟩ from x:                  // lookup
4    execute lookup algorithm (Algorithm 3);
5  When node y receives m = ⟨u, phase, move⟩ from node x:          // move
6    execute move algorithm (Algorithm 4);
```

A *move* request from a node u of G for the object ξ at the owner node v of G is served by following upwards leader ancestors in its spiral path $p(u)$ (up phase), setting downward links towards u until $p(u)$ intersects at x the directory path to the owner node v (Figs. 1b and 1c, where $x = u_3$). Then the *move* request follows a downward trajectory (down phase) deleting the links of the directory path while descending towards the owner node v (Figs. 1d and 1e); the directory path now points to the requesting node u. As soon as the *move* request reaches the owner node v, the object is forwarded from the previous owner node along some (shortest) path in the graph G (Fig. 1f). This process has resulted to a canonical directory path that consists of two spiral path fragments, a fragment of v's spiral path $p(v)$ between r and the intersection point x, and a fragment of u's spiral path $p(u)$ between x and u. Subsequent *move* operations may further fragment the directory path into multiple spiral path segments, but at all times a canonical directory path is maintained.

A *lookup* operation is served similar to a *move* operation but without modifying (adding or removing pointers) the directory path. A *lookup* operation fetches a copy of the shared ξ object from the current owner v to u. If a *move* operation later invalidates ξ from v, then the local copy of ξ at u is also invalidated.

The processing load is balanced by changing the leader of the clusters that the *move* request visits while it is in its up phase. Specifically, the originating node of the *move* request is selected as a leader in all the clusters it visits in its up phase. For the down phase, this is done only for *lookup* requests. Since the source node of a *lookup* request may not be in the clusters of the directory path in the down phase of a *lookup* request, we choose a node uniformly in random among the nodes in the cluster to act as a leader.

Concurrent *lookup* and *move* requests may be served through partial downward paths instead of the directory path. These requests are queued while the new directory path is being formed. For example, consider the scenario where a *lookup* operation is issued by a node w concurrently with the *move* operation of v. Suppose also that the *lookup* and *move* requests intersect in their up phase paths before their requests reach the directory path to u. Then the *lookup* request will descend down to v through a partial downward path while the *move* request ascends to x. The *lookup* will request the read-only copy of the object ξ from v. However, v may not have the copy of ξ yet. In this case, w's request is queued in v and it will be served when v receives ξ.

In the scenario where w's operation was a *move*, then two partial downward paths would coexist at the same time with the directory path until the up phases of u and v intersect. After that again two partial paths can coexist until the down phase of w reaches v and before the up phase of v reaches x. The result is that the *move* request from w will be queued after v. Similarly, multiple concurrent *move* operations temporarily lead to the formation of multiple partial downward paths to the origins of the requests. The *move* operations get queued in the origin nodes forming a distributed queue of *move* operations. Eventually, every move operation will be served by passing the object from the current owner at the head of the queue to the next node in the queue.

Finally, we would like to note that after several *move* operations the directory path may become highly fragmented. In such cases a *lookup* request may not find immediately the directory path to the shared object ξ, even if the *lookup* originates near ξ. In order to avoid this situation and guarantee efficiency, we introduce the notion of a *special parent* node, such that whenever a downward link is formed at a node z the special parent of z is also informed about z holding a downward pointer. The special parent is selected in such a way that any nearby lookup, close to z will either reach z or its special parent. The details will be provided in Section 4.6.

4.2. Detailed description of LB-SPIRAL

We now discuss LB-SPIRAL in detail. The pseudocode of LB-SPIRAL is given in Algorithm 1. We define the notion of parent node before giving details of *lookup* and *move* operations. We denote a *parent node* y of a node x in the spiral path $p(u)$ as $y = parent_{(i,j)}(x)$, i.e., if y is the sub-level (i, j) cluster leader in $p(u)$ then x is the leader of the immediate lower sub-level cluster leader in $p(u)$ (i.e., x is the leader of the sub-level prev(i, j) cluster). Note that the leader of a level 0 cluster is the node itself, since there is only one node in the level 0 cluster.

We formally detail how *publish*, *lookup*, and *move* operations are served in LB-SPIRAL separately below. We defer the discussion on leader change process used for load balancing to the next section (Section 4.3).

publish. The *publish*(ξ) operation issued by the creator node v of the object ξ assigns downward pointers along the edges of its spiral path $p(v)$, starting from v up to the root r and the pointers are directed toward v (Algorithm 2). After the operation finishes, there is a directed path from root r to v, with pointers from the leader in a cluster at sub-level (i, j) pointing to a leader in a cluster at sub-level prev(i, j), which is the initial directory path.

move. The *move*(ξ) operation issued by a node u is implemented in two phases: *up* and *down* (Algorithm 4). In the up phase, *move*(ξ) is sent from u upward in the hierarchy $\mathcal{Z}$ along the spiral path $p(u)$ towards the root r until it intersects

Algorithm 2: *publish*.

```
1  set y.link = x;
2  if y is not a leader node in the root cluster then
3     w ← parent node of y in the spiral path p(v);
4     CL_i ← cluster that w belongs to in the spiral path p(v);
5     select v as a leader of CL_i;
6     send m to v in CL_i;
```

Algorithm 3: *lookup*.

```
1  if m = ⟨u, up, lookup⟩ then                                   // lookup up phase
2     if y.link = ⊥ then
3        if y.slink list is empty then
4           w ← parent node of y in the spiral path p(u);
5           CL_i ← cluster that w belongs to in the spiral path p(u);
6           select u as a leader of CL_i;
7           send m to u in CL_i;
8        else
9           w ← node that first pointer of y.slink list points to;
10          CL_i ← cluster that w belongs to in the spiral path p(w);
11          select a random node z ∈ CL_i as a leader of CL_i;
12          send ⟨u,down,lookup⟩ to z in CL_i;
13    else
14       w ← node that pointer y.link points to;
15       CL_i ← cluster that w belongs to in the spiral path p(w);
16       select a random node z ∈ CL_i as a leader of CL_i;
17       send ⟨u,down,lookup⟩ to z in CL_i;
18 if m = ⟨u, down, lookup⟩ then                                 // lookup down phase
19    if y is a leaf node then send the read-only copy of ξ to u and remember u;
20    else
21       w ← node that pointer y.link points to;
22       CL_i ← cluster that w belongs to in the spiral path p(w);
23       select a random node z ∈ CL_i as a leader of CL_i;
24       send m to z in CL_i;
```

Algorithm 4: *move*.

```
1  if m = ⟨u, up, move⟩ then                                     // move up phase
2     assign oldlink ← y.link and set y.link = x;
3     add y in slink list of y's special parent;
4     if oldlink = ⊥ then
5        w ← parent node of y in the spiral path p(u);
6        CL_i ← cluster that w belongs to in the spiral path p(u);
7        select u as a leader of CL_i;
8        send m to u in CL_i;
9     else send ⟨u, down, move⟩ to oldlink;
10 if m = ⟨u, down, move⟩ then                                   // move down phase
11    if y is in the slink list in the special parent of y then erase y from slink;
12    if y is not a leaf node then oldlink ← y.link; y.link ← ⊥; send m to oldlink;
13    else send the writable copy of ξ to u;
14    invalidate(ξ) at node y and the read-only copies at other nodes;
```

Algorithm 5: Leader change on LB-SPIRAL.

```
1  select node u as the leader in CL_i;
2  copy information at old leader z to new leader u;
3  inform the parent and child of z about the new leader u;
```

at a node, say x, with the directory path. In the down phase, *move*(ξ) follows the directory path from node x to the object owner v (the directory path leads the *move* request to the owner v). The owner node v then sends the object ξ to the requesting node u along some shortest path in G. In the up phase, the *move* operation sets the directions of the edges in the fragment of $p(u)$ between u and x to point towards u. In the down phase, it deletes the downward pointers in the fragment of the directory path from x to v, making the new directory path point towards u. Through this process, when the *move*(ξ) operation from u reaches v in its down phase, u obtains a writable copy of ξ from v invalidating the old copy of ξ at v and modifying the directory path.

lookup. The *lookup*(ξ) operation issued by a node u is served similarly as of *move*(ξ) described above, but downward pointers are not added and existing downward pointers are not deleted in $\mathcal{Z}$, hence not modifying the existing directory

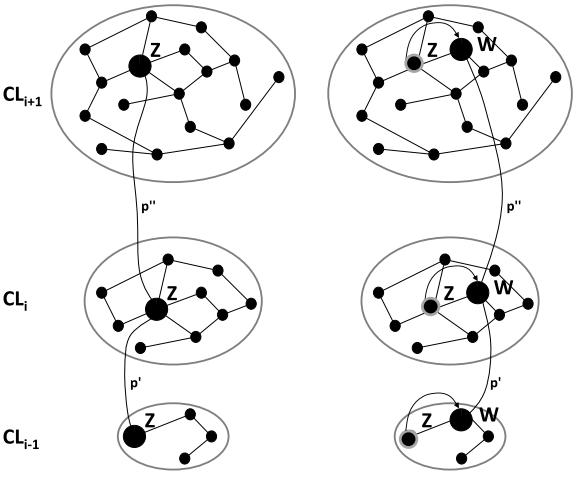

Fig. 4. An illustration of how a leader is selected in each cluster to balance the processing load. A *move* request from node w makes w the leader in each cluster it visits in its up phase and transfers the information from the old leader z to w.

path (Algorithm 3). Through this process, when the *lookup*(ξ) operation from u reaches v in its down phase, u obtains a read-only copy of ξ from v without invalidating ξ from v and without modifying the existing directory path. The object ξ is replicated to nodes u and v. If a *move* operation later invalidates ξ from v, then the copy of ξ at u is also invalidated.

4.3. Balancing the processing load in LB-Spiral

The description of LB-Spiral so far does not consider balancing the processing load of the nodes in G, i.e., the technique discussed above only minimizes the communication cost. Therefore, the technique above is same as in Spiral. We use the following technique to balance the processing load on the nodes of G. We describe separately below how we use the balancing technique to serve *publish*, *lookup*, and *move* requests.

publish. The *publish*(ξ) operation issued by the creator node v sets v as a leader in all the clusters in its spiral path $p(v)$ while going to the root cluster (including the root cluster) in $\mathcal{Z}$. Note that since the leaders are selected arbitrarily during the construction of $\mathcal{Z}$, the node v may not be the leader in all the clusters in the spiral path $p(v)$. If v is already a leader in some cluster in the spiral path $p(v)$, then leader change is not needed for that cluster. In each cluster in the spiral path $p(v)$, the downward pointers point from leader v in sub-level (i, j) cluster to leader v in sub-level prev(i, j) cluster.

move: The *move*(ξ) operation issued by a node u is served as follows. The node u sets itself as the leader in all the clusters in its spiral path $p(u)$ in its up phase until $p(u)$ intersects the directory path pointing to the owner v. In other words, the downward pointers point toward u in all the levels. This requires moving the information at the old leader z to the new leader u in all the clusters that go leader change. We discuss how this is done and the cost involved immediately after we describe *lookup* operations. Fig. 4 illustrates these ideas. The down phase needs no change and the exiting path is used.

lookup: The *lookup*(ξ) operation needs no leader change as it does not add or remove information in the directory $\mathcal{Z}$. However, if balancing is needed, then a *lookup* issued by a node u can set u as the leader in all the clusters similar as of *move*(ξ) in the up phase. In the down phase, it can pick a node uniformly at random in each cluster it visits in its down phase. Our analysis in Section 5 focuses on proving the processing load of the nodes of G considering only the *move* operations.

4.4. Overlead due to leader change

The use of leader selection procedure incurs extra cost to the actual cost of the *move* and *lookup* operations. This is because this procedure requires message exchanges between the old leader and the new leader within a cluster, and also with the parent and child clusters of the old leader to inform them about the new leader. The message exchange cost between the new and old leader is bounded by the diameter of the cluster they belong to, because these leaders communicate through the shortest path between them. Recall that we assume in the construction that each node in a cluster at any level knows the shortest paths to each node on its previous and next level clusters, which allows to immediately inform about the leader change in the leader nodes of those clusters. The cost to inform the parent and child clusters of the old leader is also bounded by the diameter of the cluster, in the worst-case, as the parent cluster is only 2 times large in diameter compared to the current cluster. Therefore, this message exchange only adds a constant factor increase in the costs of operations we account in the analysis given in Section 5. This is also observed in the simulation results presented in Section 7 as well. However the benefit of this additional overhead is that this step facilitates to control the processing load, since the

processing load on a leader node is always proportional to the number of requests that visit that leader. A leader selection approach we use in the spiral path plays major role in controlling processing load because it minimizes the overutilization of a node in serving the requests. Through our approach, a node in a cluster becomes a leader of that cluster if and only if the request (*move*, *lookup*, or *publish*) is issued by that node.

4.5. Handling concurrent executions

We observe that at any time a request updates information on three leader nodes, at levels prev(i, j), (i, j), and next(i, j), along the spiral path (or a directory path for the *lookup* operation). This is not a problem when the operations are sequential, i.e., a new operation (*lookup* or *move*) is issued only after the current operation finishes. However, in concurrent situations of *lookup* and *move* requests this might be a problem. Therefore, in the concurrent execution of *move* requests, we use the notion of a *conflict graph* for each level such that neighbors in the conflict graph cannot perform the leader change at the same time (that is, the leader change process is sequentialized for the affected clusters). That is, only the maximal independent set of leader nodes can perform the leader change concurrently. This sequentialization process does not hamper asymptotically the stretch and processing load bounds. The main reason is that the communication cost increase is within a factor of the length of the shortest paths between those leaders and processing load is measured with respect to how many requests are processed by a node. Again, this will increase the path lengths in the analysis only by a constant factor. Finally, note that the special parent node does not need to be locked because only one specific *slink* pointer value needs to be updated at any time.

4.6. Bounding the lookup cost in LB-Spiral

The ideal scenario for a *lookup* operation issued by a node w would be to find the directory path to the object owner node v at level $\log\lceil(\text{dist}(w, v))\rceil + 1$ leader node of $\mathcal{Z}$. In this case, compared to the shortest path distance dist(w, v) in G, the cost through the directory would be $O(\text{dist}(w, v) \cdot \log^3 n)$ (Lemma 3), giving $O(\log^3 n)$ competitive ratio for a *lookup* operation. However, a *lookup* request from any node $w \in G$ for the object ξ at the owner node $v \in G$ may not find the directory path to v at level $\log\lceil(\text{dist}(w, v))\rceil + 1$ leader node of $\mathcal{Z}$ where the spiral paths $p(w)$ and $p(v)$ intersect. This is because, after several *move* operations, the directory path may become highly fragmented and hence the directory path does not pass through the leader node at level $\log\lceil(\text{dist}(w, v))\rceil + 1$. Everything is not lost even in this case. This is because the construction of $\mathcal{Z}$ guarantees that a *lookup* operation from a node w always finds the directory path to the object owner node at the leader r on the highest level h. The impact of this is that the *lookup* cost using the directory is $O(D \cdot \log^3 n)$. Compared this cost to the minimum cost dist(w, v) may give the competitive ratio $O(D \cdot \log^3 n)$ for dist$(w, v) << D$ and competitive ratio $O(\log^3 n)$ for dist$(w, v) = O(D)$. The question is how to remove the dependence of *lookup* competitive ratio on D in all arbitrary situations.

The notion of a *special-parent node* helps to reduce the factor D in *lookup* competitive ratio to $O(\log^3 n)$, so that irrespective of dist(w, v), the *lookup* competitive ratio becomes $O(\log^6 n)$ (this proof is in the *lookup* cost analysis given in Theorem 3). Whenever a downward link is formed at a node z the special parent of z is also informed about z holding a downward pointer. The pointer information is stored in (removed from) a special-parent node in the up (down) phase of a *move* operation. Essentially, if y is a level i leader node in the spiral path $p(u)$ of node $u \in G$, the special-parent node for y is some ancestor leader node of y at level $\eta = i + \phi$ in the spiral path $p(u)$ (the definition below quantifies that $\phi = 4 + 3\log\log n + \log c_3$) and the proof of why this value of ϕ is enough is given in Lemma 4. The special-parent node definition takes into account how much locality γ_η is needed for any level η leader node to include the node that issues a *lookup* request and the level i leader node in the canonical directory path q towards the owner node.

Fig. 5 depicts how a special-parent provides efficient lookup. Node 5 is the current owner of the object ξ (denoted by bold star in the figure) and the directory path q to the current owner node 5 is $q_5 = s(r, \ldots, u_4, u_3, v_2, w_1, 5)$ which is shown in bold solid lines between the leaders in the hierarchy. Assume that the ownership change is in the order of $6 \to 1 \to 4 \to 5$. Therefore, the initial directory path was $q_6 = s(r, \ldots, u_4, w_3, w_2, x_1, 6)$ and 6 was the initial owner (that issued *publish* operation for ξ). The directory path to 1 was $q_1 = s(r, \ldots, u_4, u_3, u_2, u_1, 1)$ when *move* operation from 1 arrived to 6 first. The directory path q_1 is the concatenation of the fragment of the spiral path $p(6)$ of 6 from r to u_4 and the spiral path $p(1)$ of 1 from u_4 to 1. Similarly, when a *move* operation from 4 reached to 1 first, the directory path $q_4 = s(r, \ldots, u_4, u_3, v_2, w_1, 4)$, which is the concatenation of the spiral path $p(6)$ of 6 from r to u_4, the spiral path $p(1)$ of 1 from u_4 to u_3, and the spiral path $p(4)$ of 4 from u_3 to 4. Similarly, the directory path $q_5 = s(r, \ldots, u_4, u_3, v_2, w_1, 5)$ which is the concatenation of four spiral paths: the spiral path $p(6)$ of 6 from r to u_4, the spiral path $p(1)$ of 1 from u_4 to u_3, the spiral path $p(4)$ of 4 from u_3 to w_1, and the spiral path $p(5)$ of 5 from w_1 to 5. The initial directory path q_6 which was a full spiral path, is now become the directory path q_5 that is the combination of the spiral path fragments as new *move* requests are processed. Assume that the spiral path $p(7)$ of node 7 is $p(7) = s(7, x_1, w_2, x_3, w_4, \ldots, r)$ and of node 5 is $p(5) = s(5, w_1, w_2, x_3, w_4, \ldots, r)$. Assuming node 7 issues a *lookup* operation, it does not find the directory path at w_2 which is the common leader for 5 and 7 in their spiral paths $p(5)$ and $p(7)$.

In this paper, we assume that nodes know who their special-parents are. This allows the nodes to write (remove) the information directly. If the nodes do not know their special-parents, they could find using their spiral paths but that will increase the *move* stretch by a polylog(n) factor.

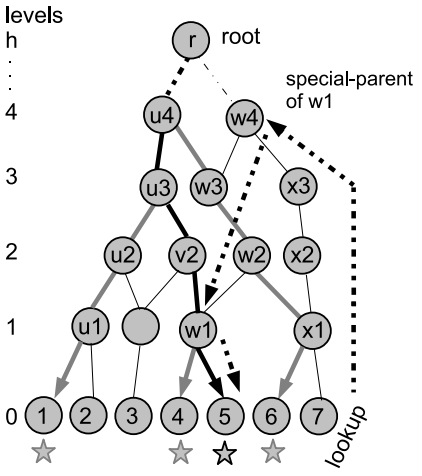

Fig. 5. Illustration of how a special-parent provides efficient *lookups*.

Definition 8 (***special-parent***). A *special-parent* node of a sub-level (i, j) leader node y in the spiral path $p(u)$, denoted as $\text{sparent}_{(i,j)}(y)$, is some leader node x in the spiral path $p(u)$ at level $\eta = i + \phi$, where $\phi = 4 + 3\log\log n + \log c_3$.

According to the definition above, for any level $i > h - \phi$, $\eta > h$. Therefore, for the leader node y in any of the level $i > h - \phi$, either the root (leader) node can be made its special-parent or the special-parent is not assigned at all. This way, the *lookup* will find the downward path at root r. Since each leader node y in any cluster of $\mathcal{Z}$ (except the root level cluster leader) is assigned a special-parent, y knows its special parent. The special-parent node of y maintains a special downward pointer, *slink*, to y. LB-Spiral maintains a list of *slink* pointers if one node is the special parent for the leaders of several clusters which, according to our $\mathcal{Z}$ construction, can happen. These special downward pointers are set (removed) when *move* operations are in the up (down) phase. The *slink* information is only set (and removed) for the nodes that are the special-parent of the nodes that are in the directory path towards the current owner of the object.

4.7. Information about LB-Spiral in each node

We discussed in Section 3.4 what information each node in G maintains about $\mathcal{Z}$. We discuss here what information each node maintains to successfully run LB-Spiral, in addition to the information they keep to maintain the directory information. Each leader node u of a sub-level (i, j) cluster in $\mathcal{Z}$ maintains data structure variables *link* and *slink*. Node u also maintains the leader information on $\text{prev}(i, j)$ and $\text{next}(i, j)$ clusters and the object ξ (when u is the current owner node for ξ). Since u may be a leader in all the clusters up to the root level starting from the bottom level, in the worst-case, it has $O(\log D \cdot \log n)$ copies of such variables, one copy for each sub-level.

5. Analysis of LB-Spiral

We give both the stretch and processing load analysis of LB-Spiral for sequential, concurrent (one-shot), and dynamic executions. However, the correctness proof of LB-Spiral is not discussed here as it can be easily proven by extending the correctness proofs of DDPs Ballistic [2], Combine [11], Spiral [13], and MultiBend [14].

5.1. Performance in sequential executions

In a sequential execution scenario the next request is initiated only after the current request completes. We first provide performance bounds for the communication costs of *publish*, *lookup* and *move* operations, and then we give the approximation of processing load of any node of G.

Theorem 2 (***publish cost***). *The publish operation in* LB-Spiral *has communication cost* $O(D \cdot \log^3 n)$.

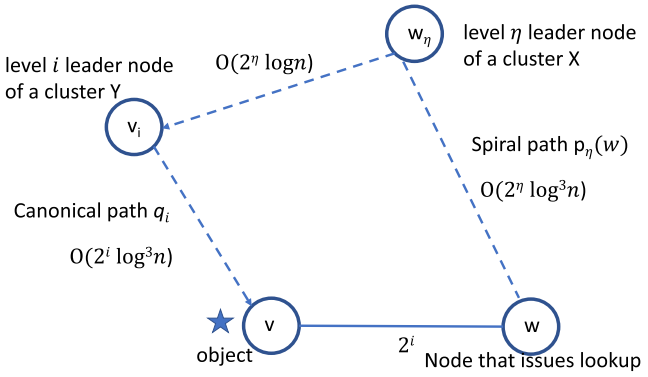

Fig. 6. Illustration of the proof of Lemma 4. The special-parent at level η should include level i leader node in the canonical path to the object and the lookup issuing node.

Proof. The theorem immediately follows from Lemma 3, by noticing that the number of levels in the hierarchy $\mathcal{Z}$ is $h = \lceil \log D \rceil + 1$, and that a spiral path is trivially a canonical path. □

The next lemma will be useful when we prove *lookup* stretch in Theorem 3.

Lemma 4. *If a node w issues a lookup request for the shared object ξ currently owned by a node v which is at distance $2^{i-1} < \text{dist}(w, v) \leq 2^i$ far from w, the spiral path $p(w)$ is guaranteed to either intersect with the directory path to v at level below $\eta = i + 4 + 3\log\log n + \log c_3$, or find a slink to the directory path for the object at level at most $\eta = i + 4 + 3\log\log n + \log c_3$, where $\eta \leq h = \lceil \log D \rceil + 1$.*

Proof. We prove this lemma for the second case of finding a *slink* to the directory path. The first case of intersecting with the directory path can be subsumed in the second case, since if the paths do not intersect up to level η then the second case becomes true. We use Fig. 6 for the ideas needed in this proof for the second case. Assume that v_i is the level i leader node in the canonical directory path q towards the owner node v of ξ and q_i is q's fragment up to level i. Assume also that $w_\eta = \ell(X)$ is the leader of the cluster X at level $\eta = i + 4 + 3\log\log n + \log c_3$, which has a *slink* information to v_i (set by some *move* request following Algorithm 4). For a *lookup* request issued by node w to find the *slink* to v_i, X must include v_i, since the spiral path $p(w)$ of w visits the leaders of all clusters that contain it. It suffices to show that the locality $\gamma_\eta = 2^{\eta-1}$ of X is at least the distance $\text{dist}(v_i, w)$ between v_i and w to guarantee that X contains w. As $\text{length}(q_i) \leq c_3 2^{i+2}\log^3 n$ (Lemma 3) and $\text{dist}(w, v) \leq 2^i$, $\text{dist}(v_i, w)$ is bounded by:

$$\begin{aligned}
\text{dist}(v_i, w) &\leq \text{length}(q_i) + \text{dist}(w, v) \\
&\leq c_3 2^{i+2}\log^3 n + 2^i \\
&\leq c_3 2^{i+3} \cdot \log^3 n \\
&\leq 2^{\log c_3} \cdot 2^{i+3} \cdot 2^{\log\log^3 n} \\
&\leq 2^{i+3+\log\log^3 n + \log c_3} \\
&\leq 2^{i+3+3\log\log n + \log c_3} \\
&\leq 2^{\eta-1} = \gamma_\eta,
\end{aligned}$$

with $\eta = i + 4 + 3\log\log n + \log c_3$. Therefore, the η level leader node we picked as a special-parent is guaranteed to find the directory path to the object ξ. □

Theorem 3 (***lookup stretch***). *The lookup stretch in* LB-Spiral *is $O(\log^6 n)$ in sequential executions.*

Proof. Suppose a node $w \in G$ issues a *lookup*(ξ) request r_L for a shared object ξ currently at an owner node $v \in G$ which is at distance $2^{i-1} < \text{dist}(w, v) \leq 2^i$ in G. We have from Lemma 4 that the spiral path $p(w)$ is guaranteed to either intersect with the directory path to v or find a *slink* to the directory path to v at level $\eta \leq i + 4 + 3\log\log n + \log c_3$. Therefore, we consider two cases: (i) $p(w)$ intersects directory path to v at level at most η and (ii) $p(w)$ find a *slink* to the directory path to v at level η. In the first case, the total cost of LB-Spiral, denoted as $A(r_L)$ for the *lookup* request r_L is the sum of the

length of the spiral path $p(w)$ up to level η, denoted as $p_\eta(w)$, from w in the bottom level and the length of the directory path q up to level η from node v, denoted as q_η. That is,

$$\begin{aligned} A(r_L) &= \text{length}(p_\eta(w)) + \text{length}(q_\eta) \\ &\leq 2 \cdot \text{length}(q_\eta) \\ &\leq 2 \cdot c_3 2^{\eta+2} \log^3 n \\ &= 2c_3 \cdot 2^{i+4+3\log\log n + \log c_3 + 2} \log^3 n \\ &\leq 128 \cdot (c_3)^2 \cdot 2^i \cdot \log^6 n, \end{aligned}$$

using Lemma 3.

In the second case, we have that

$$\begin{aligned} A(r_L) &= \text{length}(p_\eta(w)) + \text{dist}(x, v_i) + \text{length}(q_i) \\ &\leq 3 \cdot \text{length}(q_\eta) \\ &\leq 3 \cdot c_3 2^{\eta+2} \log^3 n \\ &\leq 3 \cdot c_3 \cdot 2^{i+4+3\log\log n + \log c_3 + 2} \log^3 n \\ &\leq 192 \cdot (c_3)^2 \cdot 2^i \cdot \log^6 n, \end{aligned}$$

again using Lemma 3, where $\text{dist}(x, v_i)$ is the distance from the leader node at level k to a node at level i to which *slink* at level k points to.

We have that the optimal communication cost $A^*(r_L) \geq 2^{i-1}$ as $\text{dist}(w, v) \leq 2^i$. Therefore, the stretch for the *lookup* operation is

$$\frac{A(r_L)}{A^*(r_L)} \leq \frac{192 \cdot (c_3)^2 \cdot 2^i \cdot \log^6 n}{2^{i-1}} = O(\log^6 n). \quad \square$$

We now give an amortized stretch analysis of LB-Spiral for *move* operations in sequential executions. As *move* requests are non-overlapping in sequential executions, the system attains *quiescent* configuration after a *move* request is served and until a next *move* request is issued. Define a sequential execution of a set $\mathcal{E}$ of $\ell + 1$ requests $\mathcal{E} = \{r_0, r_1, \ldots, r_\ell\}$ for the object ξ, where r_0 is the initial *publish* request and the rest are the subsequent *move* requests (we do not include *lookups* in $\mathcal{E}$ since they do not add or remove links in the directory $\mathcal{Z}$, and hence do not impact the performance of other *move* or *lookup* operations).

For the amortized stretch analysis define a two-dimensional array B of size $(k+1) \times (\ell+1)$, where $k+1$ and $\ell+1$ are the number of rows and columns of B, respectively. The $(k+1)$ rows of B can be denoted as $\{row_0, row_1, \ldots, row_k\}$, and the $\ell+1$ columns of B can be denoted as $\{col_0, col_1, \ldots, col_\ell\}$. Each location $[i, j]$ of the array B is initially $\perp$. We fix that $[0, 0]$ be the lower left corner element and $[k, \ell]$ be the upper right corner element in B. The levels visited by each request r_i in the hierarchy $\mathcal{Z}$ while searching for the object ξ are registered in the rows of column col_i. The maximum level reached by r_i before it finds the downward pointer in $\mathcal{Z}$ is called the *peak level* for r_i. We have that $h = k$. The peak level reached by r_0 (the *publish* request) is always h, the maximum level in $\mathcal{Z}$. Notice that r_0 is registered in all the locations of col_0 from 0 to k.

Our goal is to bound the stretch $\max_{\mathcal{E}} A(\mathcal{E})/A^*(\mathcal{E})$, where $A(\mathcal{E})$ denotes the total communication cost of serving requests in $\mathcal{E}$ using LB-Spiral and $A^*(\mathcal{E})$ denotes the optimal cost for serving requests in $\mathcal{E}$ through an optimal offline algorithm. We prove the following theorem for stretch $\max_{\mathcal{E}} A(\mathcal{E})/A^*(\mathcal{E})$ using array B.

Theorem 4 (***move stretch***). *The move stretch in* LB-Spiral *is* $O(\log^3 n \cdot \log D)$ *in sequential executions.*

Proof. Consider only the cost due to the up phase of each *move* request. The consideration of the down phase increases the cost by only a factor of 2. For any $c, d, 0 \leq c < d \leq \ell$, a *valid pair* $W^j_{(c,d)}$ of two non-empty entries in row_j, $0 \leq j \leq h$, is defined as $W^j_{(c,d)} = (row_j[c], row_j[d])$, such that $row_j[c] \neq \perp$ and $row_j[d] \neq \perp$, and if $d - c > 1$, then $\forall e, c+1 \leq e \leq d-1, row_j[e] = \perp$. That is, $W^j_{(c,d)}$ is a pair of two subsequent non-empty entries in a row. Denote by S_j the total count of the number of entries $row_j[i], 0 \leq i \leq \ell$, such that $row_j[i] \neq \perp$, and by W_j the total number of valid pairs $W^j_{(c,d)}$ in it. We have that $W_j = S_j - 1$.

We have from [13] that $A^*(\mathcal{E}) \geq \max_{1\leq h\leq k}(S_h - 1)2^{h-1}$. Similar to [13], we have that $A(\mathcal{E}) \leq \sum_{h=1}^{k} c_3(S_h - 1)2^{h+2}\log^3 n$. Since the execution $\mathcal{E}$ is arbitrary, $k = \lceil \log D \rceil + 1$, c_3 is a constant, and

$$\sum_{h=1}^{k} c_3(S_h - 1)2^{h+2}\log^3 n \leq k \cdot c_3 \max_{1\leq h\leq k}(S_h - 1)2^{h+2}\log^3 n,$$

we have that the *move* stretch is

$$\max_{\mathcal{E}} \frac{A(\mathcal{E})}{A^*(\mathcal{E})} \leq \frac{k \cdot c_3 \max_{1\leq h\leq k}(S_h - 1)2^{h+2}\log^3 n}{\max_{1\leq h\leq k}(S_h - 1)2^{h-1}} = O(\log^3 n \cdot \log D).$$

Note that the update overhead cost due to leader change while serving each *move* and *lookup* operation is not considered in the bound above. Nevertheless, the update overhead cost is within a constant factor of $A(\mathcal{E})$ and it does not change the *move* and *lookup* stretches asymptotically even if we include it in the *lookup* and *move* costs of LB-Spiral. The theorem follows. □

We now analyze the processing load of a node in LB-Spiral in sequential executions. We relate the processing load $PL(x)$ of a node $x \in G$ in LB-Spiral to the optimal load $PL^*(x)$ of that node to provide the approximation ratio. We prove the following theorem for processing load of any node of G for the sequential execution of *move* operations; we omit the *lookup* operations while computing processing load as they do not add or remove pointer information on the directory hierarchy $\mathcal{Z}$. The processing load for *lookup* operations can be established similarly as of *move* operations.

Theorem 5 (***processing load***). *The processing load approximation in* LB-Spiral *is* $O(\log n \cdot \log D)$ *for any node of* G.

Proof. Recall that every *move* request from its source node to its destination node is routed by LB-Spiral by selecting some paths. Specifically, these paths are spiral paths and they connect the leaders of the subsequent clusters in the hierarchy $\mathcal{Z}$ via shortest paths. Let x be any node in the graph G and $PL(x)$ be the load on x (the maximum number of times node x is used as a leader node of a cluster in the hierarchy $\mathcal{Z}$). Particularly, we bound the number of times a spiral path of a *move* operation passes through x, when x is a leader of some cluster in the hierarchy $\mathcal{Z}$.

Consider now the up phase of the *move* operations. We will deal with their down phase later. There are two possible scenarios:

- *A* move *request issued by x:* In this case, node x will become the leader of all the clusters in the spiral path $p(x)$ from the bottom level 0 to the peak level of that *move* operation.
- *A* move *request not issued by x:* In this case, node x does not become the leader of any cluster in the spiral path of that *move* operation.

Notice that there are $O(\log n)$ clusters in each level of the directory hierarchy $\mathcal{Z}$ and $O(\log n)$ levels of clusters in each binary tree embedded between any two levels. Moreover, there are $O(\log D)$ levels in the directory hierarchy $\mathcal{Z}$. This gives total $O(\log n \cdot \log D)$ clusters in the spiral path of any node in G from the bottom level to the top level of $\mathcal{Z}$. Therefore, for a *move* request issued by a node $x \in G$, x becomes the leader of $O(\log n \cdot \log D)$ clusters, in the worst-case, in the up phase. For a *move* request not issued by x, x does not become leader in any of the clusters in $\mathcal{Z}$.

Among ℓ requests in $\mathcal{E}$, suppose k of them were issued by x. In this case, $PL^*(x) \geq k$, as k *move* requests have to reach to x to get a copy of x. According to the argument above on the number of clusters in a spiral path, in LB-Spiral, the load of node x in the up phase is $PL(x) \leq k \cdot O(\log n \cdot \log D)$.

Consider now the down phase. Let q be a directory path from the top level to the bottom level. Each leader node on each cluster on this directory path q is visited only once. This is because, the downward pointer at that node is deleted in the down phase of a *move* operation passing through that node and that node does not become a leader again until it is a node that issued a *move* operation. Therefore, considering the down phase, the processing load $PL(x)$ increases only by a factor of 2. Since x is an arbitrary node in G, the processing load of a node of G in LB-Spiral is

$$\frac{PL(x)}{PL^*(x)} \leq \frac{k \cdot O(\log n \cdot \log D)}{k} = O(\log n \cdot \log D). \quad \square$$

5.2. Performance in one-shot executions

The performance analysis of LB-Spiral given in Section 5.1 does not apply to concurrent executions because the adversary is not allowed to gain by ordering the requests in a smarter way, i.e., the orderings provided by both LB-Spiral and OPT are the same. Concurrent executions can change the order of the requests in execution and hence affect the overall

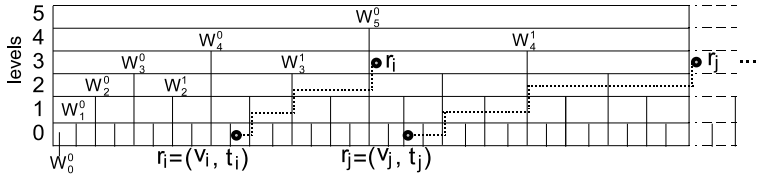

Fig. 7. Time windows at different levels used in the analysis of LB-Spiral for one-shot and dynamic executions.

performance of LB-Spiral. In one-shot executions, all requests come concurrently (at the same time) in the system. We study the following one-shot instance of concurrent execution. At time t as soon as a *publish* operation finished execution, $R \subseteq V$ nodes issue a *move* request each concurrently and no further requests occur. We calculate the total communication cost of all the requests including the publish operation (similar as of sequential execution) to provide the competitive ratio in one-shot situation.

For the performance analysis we assume that the network model is synchronous (LB-Spiral does not require synchrony for correctness), in addition to the assumptions of Section 2. We assume that a time unit is of duration required for a message sent by a node to reach a destination node that is a unit distance far from it. We define for level i *window* of time duration $\Phi(i) = 4c_3 2^i \log^3 n, 0 \leq i \leq h$ for some constant c_3 (Lemma 3), i.e., the longest distance traversed in level i (including sub-levels) following the canonical (or spiral) path. An example given in Fig. 7 shows the windows in different levels. At each level i a window represents the time that a node needs to reach and modify the pointers of all the leader nodes in level i. Thus, the division of time into fixed duration windows allows us to obtain upper bounds for the communication cost and also respective lower bounds. Moreover, $\Phi(.)$'s are aligned in such a way that $\Phi(i)$ and $\Phi(i-1)$ start at the same time, i.e., two windows of duration $\Phi(i-1)$ can be perfectly accommodated at a level i window $\Phi(i)$. We assume also that all requests proceed in rounds. A round is of duration $\Phi(h)$, where $h = \lceil \log D \rceil + 1$, and it has h overlapping aligned windows. According to this definition, in a round, there is 1 window for level h, 2 windows for level $h-1$, 4 windows for level $h-2$, and so on, so that there are $2^{(h-k)}$ windows for level k. In a window, each leader node in the canonical path (or spiral path) can exchange a message with each of its neighbors (parents or children). A leader node $w_{i,\chi+\delta-1}$ of the highest sub-level cluster at level i in a spiral path $p(w)$ forwards the request to a leader node $w_{i+1,1}$ of the lowest sub-level cluster at level $i+1$ at the end of its window $\Phi(i)$. Similarly, any request that arrives to a leader $w_{i,j}$ of a sub-level j cluster at level i is processed during $\Phi(i)$ and sent to higher sub-level cluster towards $w_{i,\chi+\delta-1}$.

The above assumption that the forwarding of requests to parent and child levels from the current level is done at the end of the window to make sure that the requests can reach and modify the pointers of all the leader nodes ($O(\log n)$ leaders one per cluster among $O(\log n)$ clusters) in the current level. In other words, the time windows impose a restriction to the protocol in the sense that they control when to forward requests to higher and lower levels from the current level. Therefore, time windows may add some additional delay in the upper bound cost but they do not affect the lower bound because lower bound computation can be done without such restriction.

Let us discuss briefly the execution of concurrent requests: At time zero, a node issues a *publish* operation r_0. As soon as the *publish* operation r_0 finishes at time t, l nodes issue one *move* request each concurrently, namely $\mathcal{E} = \{r_0, r_1, r_2, \ldots, r_l\}$. All the l *move* requests are forwarded to their parent nodes at level 1 at the end of window $\Phi(0)$, following their spiral paths. When level 1 cluster leaders in the respective spiral paths of the requesting nodes receive one request each, they simply forward it to the parent node at level 2 at the end of window $\Phi(1)$; if two requests are received at level 1, one will be forwarded to the parent node at level 2 following the spiral path of the forwarded request, while the other request will be "deflected" down to level 0 along the directory path formed by the previous request that was forwarded to level 2. For more than 2 requests, the above scenario occurs repetitively. There may be the case that current window $\Phi(k+1)$ is not yet expired when the requests in the window $\Phi(k)$ are ready to be sent (because $\Phi(k)$ expired). In this situation, we impose a restriction in message exchange between levels such that the messages from level k will be delayed until the current window at level $k+1$ (or level $k-1$ in the down phase) expires. Hence, the requests that need to be sent to level $k+1$ (or level $k-1$) from level k are sent as soon as a new window starts at level $k+1$ (or level $k-1$).

Denote by $A^*(\mathcal{E})$ the total communication cost of the optimal algorithm to serve all the requests in $\mathcal{E}$, and by $A(\mathcal{E})$ the total communication cost of the LB-Spiral to serve those requests, in concurrent executions. We will bound the stretch $\max_{\mathcal{E}} A(\mathcal{E})/A^*(\mathcal{E})$. For simplicity, we consider only the cost incurred by the up phase of each request; if we consider also the cost incurred by the down phase, the competitive ratio increases by a factor of 2 only. Moreover, similar to the sequential analysis, lets say $Q_k, 0 \leq k \leq h$, where $h = \lceil \log D \rceil + 1$, are the total number of requests in $\mathcal{E}$ (including the publish operation r_0) that reach level k, following their spiral paths, while searching for the directory path towards the shared object.

Lemma 5. *In one-shot concurrent execution $\mathcal{E}$, for the Q_k requests that reach level k in the hierarchy $\mathcal{Z}$, $A^*(\mathcal{E}) \geq \max_{1 \leq k \leq h} |Q_k - 1| \cdot 2^{k-1}$, where $h = \lceil \log D \rceil + 1$.*

Proof. The proof is in two steps. In the first step, we show that the optimal ordering of the one-shot concurrent requests that reach any level k in the hierarchy $\mathcal{Z}$ is related to the *Steiner tree* problem [35] of the *move* requests that reach that level. In the second step, we provide a lower bound for $A^*(\mathcal{E})$ based on the minimum Steiner tree cost to connect the source nodes of the Q_k requests that reach level k.

For the first step, assume that there are only Q_k requests in $\mathcal{E}$ issued by Q_k different nodes and all of them reach level k. Since all Q_k requests reached level k, we need to order the requests in Q_k one after another in a distributed queue with the minimum communication cost. As destination nodes are not known for requests beforehand, these nodes need to be found online while in execution. Therefore, in any algorithm, the source node of one request becomes the destination node of other request and the transactions that are currently executing in requesting nodes get the shared object one after another according to the distributed queue order. The minimum communication cost by any protocol to provide a distributed order is to connect them through a minimum Steiner tree. This is because the source nodes of Q_k requests in G are known in the beginning of execution and any connected subgraph T of G that connects those source nodes minimizing the sum of the lengths of T's edges is the optimal solution for the distributed order in any algorithm. There exist algorithms for the Steiner tree problem with the approximation less than 2 on the ratio of the cost of T returned by an algorithm and the cost of the optimal solution over all problem instances, e.g., [35].

We now bound the minimum possible cost for the Steiner tree T of Q_k requests. For any two requests r_i, r_j issued, respectively, by source nodes u, v and reached level k of the hierarchy $\mathcal{Z}$ in one-shot execution $\mathcal{E}$, $\text{dist}(u, v) \geq 2^{k-1}$. This is because every request r_i follows the spiral path of its source node until it meets some other request r_j at some level l, and if spiral paths of two nodes u, v meet at level l then, from Lemma 2, $\text{dist}(u, v) \geq 2^{l-1}$. Moreover, according to our time periods and the restriction imposed in forwarding the requests to parent and child levels from the current level, if they were at $\text{dist}(u, v) \leq 2^{l-1}$, then they would have reached only to levels $l-1$ or lower. Therefore, as we assumed that Q_k requests reach level k of $\mathcal{Z}$ (at the same time), the cost of the minimum Steiner tree to connect all Q_k requests is at least $|Q_k - 1| \cdot 2^{k-1}$. Moreover, we argue that this cost also holds when $|\mathcal{E}| > |Q_k|$ because all the requests follow their spiral paths until they meet other requests, and hence, Lemma 2 also applies in this case. Considering all the levels from 1 to h, it is safe to say that the optimal communication cost of any algorithm is at least bounded by the maximum cost of the Steiner tree at some level, i.e.,

$$A^*(\mathcal{E}) \geq \max_{1 \leq k \leq h} |Q_k - 1| \cdot 2^{k-1},$$

where $h = \lceil \log D \rceil + 1$. □

Lemma 6. *In one-shot concurrent execution $\mathcal{E}$,* LB-Spiral *has the communication cost $A(\mathcal{E}) \leq \sum_{k=1}^{h} c_3 \cdot |Q_k - 1| \cdot 2^{k+2} \log^3 n$, where c_3 is a constant, Q_k is the requests in $\mathcal{E}$ that reach level k in the hierarchy $\mathcal{Z}$, and $h = \lceil \log D \rceil + 1$.*

Proof. The total communication cost for each request that reaches level k of $\mathcal{Z}$ in LB-Spiral is bounded by $c_3 \cdot 2^{k+2} \log^3 n$ (Lemma 3). Therefore, for Q_k requests that reach level k, the cost is $\leq |Q_k - 1| \cdot c_3 2^{k+2} \log^3 n$ (note that one of them is always the *publish* request r_0; if it is not included in analysis, we can say $|Q_k|$ and the analysis follow). For all the levels from 1 to h, we can immediately have that

$$A(\mathcal{E}) \leq \sum_{k=1}^{h} c_3 \cdot |Q_k - 1| \cdot 2^{k+2} \log^3 n. \quad \square$$

Theorem 6. *The move stretch in* LB-Spiral *is $O(\log^3 n \cdot \log D)$ in concurrent (one-shot) executions. It achieves $O(\log n \cdot \log D)$ approximation on processing load on any node of G. Moreover, the publish operation has $O(D \cdot \log^3 n)$ cost and any lookup operation in* LB-Spiral *has $O(\log^6 n)$ stretch.*

Proof. The *move* stretch of $O(\log^3 n \cdot \log D)$ is now immediate comparing the costs $A^*(\mathcal{E})$ and $A(\mathcal{E})$ from Lemmas 5 and 6, respectively as in Theorem 4. The processing load is $O(\log n \cdot \log D)$ which again follows similarly as in Theorem 5. The *publish* cost is also $O(D \log^3 n)$ as in Theorem 2. The *lookup* stretch is also $O(\log^6 n)$ as the concurrent execution has no adverse impact on execution of the *lookup* operation. □

5.3. Performance in dynamic executions

The analysis in Sections 5.1 and 5.2 for LB-Spiral is for two extreme execution scenarios: (i) no two requests are executing simultaneously and (ii) all the requests are executing simultaneously. The performance of LB-Spiral can also be analyzed for requests that are initiated in arbitrary moments of time (i.e., dynamic executions), that is, this analysis can capture the execution scenarios where requests are neither completely sequential as considered in Section 5.1 nor completely concurrent as considered in Section 5.2. Furthermore, this analysis subsumes the sequential and one-shot cases. The idea here is

to use the dynamic analysis framework presented in [23]. We briefly describe that framework and argue why it works for LB-Spiral.

We identify a *move* request r_i by the tuple $r_i = (u, t)$ to capture a dynamic execution, where u is the leaf node in $\mathcal{Z}$ that initiates r_i and $t \geq 0$ is the time r_i is initiated. We denote by $\mathcal{E} = \{r_0 = (v_0, t_0), r_1 = (v_1, t_1), \ldots\}$ the arbitrary finite set of *move* requests (except that r_0 is a *publish* request) with each request r_i indexed according to its initiation time, i.e., $i < j$ implies $t_i \leq t_j$. To bound the competitive ratio of LB-Spiral due to the arbitrary set of *move* requests, we define for level k the time window $\Phi(k)$ as in the one-shot execution analysis and the requests are forwarded to the upper and lower levels at the end of the time window. In each level k, there are a consecutive set of windows with duration $\Phi(k)$. We name those windows $W_k^0, W_k^1, W_k^2, \ldots$. Fig. 7 depicts the time windows at different levels of $\mathcal{Z}$.

Suppose a request $r_i = (v_i, t_i)$ reaches level k of $\mathcal{Z}$ during time window W_k^p and $r_j = (v_j, t_j)$ reaches level k of $\mathcal{Z}$ during time window W_k^q. In the analysis, we need to bound their initiation time difference. Particularly,

Lemma 7. *Let $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ be two requests that reach level k in the respective time windows W_k^p and W_k^q, then $t_j - t_i \geq (q - p - 2) \cdot \Phi(k)$.*

The proof of Lemma 7 looks at what will be the maximum (time) delay for a request to reach level k after it is initiated. Since the request is forwarded to the next level at the end of the time window at the current level, the maximum delay can be shown as at most two time windows $\Phi(k)$ to reach level k after a request is initiated at time t.

In the analysis, we also need to bound the distance $\text{dist}(v_i, v_j)$ in G between the nodes v_i, v_j of two requests $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ that reach level k. The next lemma bounds $\text{dist}(v_i, v_j)$ for the case of r_i, r_j reaching to level $k-1$ and k inside the same time windows W_{k-1}^α and W_k^β.

Lemma 8. *Suppose $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ are two requests that reach level k. If r_i, r_j fall inside the same window W_{k-1}^α at level $k-1$ and also inside the same window W_k^β at level k, then $\text{dist}(v_i, v_j) \geq 2^{k-2}$.*

The proof intuition is as follows. As both requests fall inside the same window at level $k-1$ and then continue to the same window at level k, they must not have met each other at level $k-1$ (or lower). In other words, one request did not see the downward pointers previously set by another request, otherwise one request would have been diverted behind the another one following downward pointers. Moreover, in LB-Spiral, two requests can not follow each other after they see the downward pointers previously set. Therefore, these two requests must have been initiated from the nodes that are at distance at least $\text{dist}(v_i, v_j) \geq 2^{k-2}$ from each other in G so that they do not have a common leader at level $k-1$.

Suppose two requests $r_i = (v_i, t_i), r_j = (v_j, t_j)$ reach level k when $\text{dist}(v_i, v_j) < 2^{k-1}$, then the following lemma shows the existence of a third request $r_l = (v_l, t_l)$ that initiated between the initiation times of r_i, r_j satisfying certain distance property between either v_i or v_j from v_l.

Lemma 9. *Suppose $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ are two requests that reach level k. If $\text{dist}(v_i, v_j) < 2^{k-1}$, then there must exist a third request $r_l = (v_l, t_l)$ such that $t_i < t_l < t_j$ and either $\text{dist}(v_i, v_l) \geq 2^{k-4}$ or $\text{dist}(v_l, v_j) \geq 2^{k-4}$.*

The proof intuition is as follows. If $\text{dist}(v_i, v_j) < 2^{k-1}$ then r_j must meet r_i at level $k-1$, if the downward pointers set by r_i are not modified by any other request. However, if r_j meets r_i at level k, then r_j must have missed the downward pointers toward r_i in all the levels up to $k-1$. Thus, there must exist a third request $r_l = (v_l, t_l)$, which was initiated between the time r_i was initiated and the time r_j was initiated, which has deleted the downward pointers set by r_i. Suppose now that all requests with initiation time between the time r_i was initiated and the time r_j was initiated are at distance less than 2^{k-4} from r_i. All these intermediate requests are within distance less than $2 \cdot 2^{k-4} \leq 2^{k-3}$ from each other and they must meet each other at level $k-2$. Which implies that no request will reach level $k-1$. Therefore, r_l would not exist, a contradiction. Similarly, it cannot be that all the intermediate requests are within distance less than 2^{k-4} from r_j.

Lemma 9 provides a guarantee needed later in the lower bound analysis that if γ requests reach level k within the same time window, then at least $\gamma/2$ requests are initiated by the nodes with (pairwise) distance between them in G is at least 2^{k-4}. This is because, for every three requests in the same window, at least two of them satisfy this criteria.

With these basic results for time windows, we can proceed with the analysis. Denote by S_k^j the total number of requests in $\mathcal{E}$ that reach level k in $\mathcal{Z}$ inside some window W_k^j. We divide time windows into two categories, depending on S_k^j. We call the level k windows that have $S_k^j \geq 3$ the *dense windows* and the rest with $S_k^j < 3$ the *sparse windows*. The reason behind considering the windows with $S_k^j \geq 3$ and $S_k^j < 3$ separately is that we always need in the analysis at least $\lceil S_k^j/2 \rceil \geq 2$ requests inside any window that are at distance $\geq 2^{k-2}$ far from each other in graph G (Lemma 9). This helps in establishing a non-trivial lower bound for the communication cost. For $S_k^j < 3$, the goal is to transform those windows into dense windows and apply a similar analysis. Particularly, we perform a transformation such that there are exactly two requests in each window and the graph nodes that initiate them are at least 2^{k-1} far in G.

We divide the analysis into bounding the costs of serving the requests that fall into dense windows and sparse windows, respectively. Therefore, the total communication cost of LB-Spiral is the sum $A(\mathcal{E}) = A_D(\mathcal{E}) + A_S(\mathcal{E})$, where $A_D(\mathcal{E})$ ($A_S(\mathcal{E})$) is the cost of LB-Spiral in serving the requests in dense (sparse) windows. Given Lemmas 7–9, this cost analysis is easy for the dense windows. Particularly, for a dense window W_k^j, the communication cost of LB-Spiral is $\leq c \cdot S_k^j \cdot 2^{k+2} \log^3 n$ for some constant c. The optimal communication cost for any algorithm for the requests in W_k^j is at least $\lceil (S_k^j - 1)/2 \rceil \cdot 2^{k-2}$ (Lemma 9). After the cost analysis for individual dense time windows, the goal is to combine the costs of all the time dense windows. Observe that for any two requests that fall in two subsequent dense windows at level k, their initiation time difference $t_b - t_a \geq c \cdot 2^{k+2} \log^3 n$, for some constant c. Therefore, we can easily combine the upper bound costs of dense windows in any level k.

Extending the lower bound cost for a single dense window at level k to the combination of the dense windows is non-trivial. Nevertheless, we argue that the minimum cost of the dense window sequence of level k is at least the cost due to the *minimum cost Hamiltonian path* that visits each vertex of the dense sequence at that level exactly once. To provide intuition on why this argument is true, we use a notion of *directed dependency graph* in computing such Hamiltonian path.

Let $\mathcal{E}' = \{r_1, r_2, \ldots\} \subset \mathcal{E}$ be a subset of requests in $\mathcal{E}$. The directed dependency graph for the requests in $\mathcal{E}'$, denoted as $H(\mathcal{E}') = (V', E', \mathfrak{w}')$, has requests as vertices V', i.e., $|V'| = |\mathcal{E}'|$, a directed edge from any $r_i \in V'$ to any other $r_j \in V'$ such that both $(v_i, v_j) \in E'$ and $(v_j, v_i) \in E'$, and edge weight function $\mathfrak{w}' : E' \to \mathbb{R}^+$. Note that $H(\mathcal{E}')$ is a (bidirectional) directed complete graph – there are two directed edges between every pair of vertices. The edge weights are assigned such that

$$\forall i, j, \mathfrak{w}'(v_i, v_j) = \max\{\text{dist}(v_i, v_j), t_i - t_j\}.$$

The edge weights can be asymmetric, i.e., $\mathfrak{w}'(v_i, v_j)$ may be different than $\mathfrak{w}'(v_j, v_i)$. The time parameter included in the edge weight computation in $H(\mathcal{E}')$ plays a crucial role in the lower bound analysis since sometimes the time difference in $\mathfrak{w}'(v_i, v_j)$ translates to the communication cost (not the distance in G) as there is always a request that is searching for the predecessor node as soon as it is initiated.

It is easy to see from the construction of $H(\mathcal{E}')$ that each possible ordering for the requests in $\mathcal{E}'$ is given by a directed Hamiltonian path that visits each vertex of $H(\mathcal{E}')$ exactly once. This is because, irrespective of the algorithm used, it has to order the requests in a queue one after another, which a Hamiltonian path does by visiting the nodes of $H(\mathcal{E}')$ exactly once. Out of the possible orderings, the order which minimizes the ordering cost is the *lowest cost* directed Hamiltonian graph and any algorithm for the ordering must have cost at least the lowest cost Hamiltonian path. The existence of a Hamiltonian path in $H(\mathcal{E}')$ can be guaranteed at all times, since $H(\mathcal{E}')$ is a directed complete graph.

To establish the lower bound, the question that remains to answer is what would be the minimum length of a directed Hamiltonian path in $H(\mathcal{E}')$. For simplicity, we consider the directed dependency graph of the requests in all the dense windows at level k, denoted as $H_k(\mathcal{E}')$. We divide the vertices of $H_k(\mathcal{E}')$ into n_k groups $H_i, 1 \leq i \leq n_k$, each coming from a distinct dense window of level k. We order the group according to time from left to right. If we look at a particular group H_i, there are some directed edges between the vertices inside H_i, some directed edges going out to the other groups on the left and right sides of H_i, and some directed edges coming into H_i from the other groups on its left and right side. Without loss of generality, we can consider a sub-graph of H_i such that for any two vertices $u, v \in H_i$, $\text{dist}(u, v) \geq 2^{k-2}$. As in Lemma 9, there will be at least $\lceil S_k^i/2 \rceil$ vertices in each group H_i satisfying such criteria since we are considering the dense windows and there at least three requests inside each dense window satisfying the requirements of Lemma 9. Denote by P some directed Hamiltonian path on the sub-graph $H_k(\mathcal{E}')$ and by P^* the lowest cost Hamiltonian path among all P. According to our construction, some edges of P are between the vertices of a particular group H_i and some edges are between the vertices of groups $H_i, H_j, j \neq i$. Therefore, the cost of P, denoted as $A(P)$, is the cost $A(P_{int})$ of the edges between the vertices in H_i and the cost $A(P_{ext})$ of the edges between two different groups. $A(P_{ext})$ can further be divided into $A(P_{ext,left})$ and $A(P_{ext,right})$, where $A(P_{ext,left})$ is the cost due to the edges coming into H_i from (and going out from H_i to) the groups in the left of H_i and $A(P_{ext,right})$ is the cost due to the edges coming into H_i from (and going out from H_i to) the groups in the right of H_i.

Represent by

$$A(P^*) \geq A(P^*_{int}) + A(P^*_{ext,left}) + A(P^*_{ext,right})$$

the cost of the minimum cost Hamiltonian path P^* in any ordering. We can assign $A(P^*_{ext,right}) \geq 0$, therefore

$$A(P^*) \geq A(P^*_{int}) + A(P^*_{ext,left}).$$

Now let $\mathcal{W}_k$ be the set of all dense windows at level k. Pick every third window in $\mathcal{W}_k$ to make a set $\mathcal{W}_k^\zeta$ so that the requests in any two subsequent windows in $\mathcal{W}_k^\zeta$ do not overtake each other (This is due to Lemma 7, which guarantees that for any two windows $W_k^j, W_k^l \in \mathcal{W}_k^\zeta$, if $l > j$, then the requests in W_k^j have initiation time smaller than all the requests in W_k^l). According to this definition, there will be exactly three such sets (i.e., $1 \leq \zeta \leq 3$), and for the lower bound $A(P^*)$ we can consider only one set $\mathcal{W}_k^\zeta$. Let

$$M_k^\zeta = \sum_{1}^{|\mathcal{W}_k^\zeta|} S_k^i,$$

the total sum of the number of requests in all the windows in the dense window set $\mathcal{W}_k^\zeta$. After some further analysis, it can be shown that

$$\begin{aligned} A(P*) &\geq A(P^*_{int}) + A(P^*_{ext,left}) \\ &\geq 1/4 \cdot M_k^\zeta \cdot 2^{k-3}. \end{aligned}$$

We now bound the total cost of LB-Spiral in serving the requests in the dense windows in all the levels. Since there are 3 dense subsequences at level k, the total number of requests that reached level k inside dense windows is

$$\sum_{\zeta=1}^{3} M_k^\zeta \leq 3 \cdot \max_{1\leq\zeta\leq 3} M_k^\zeta.$$

Therefore, the total cost for all the levels is

$$\begin{aligned} A_D(\mathcal{E}) &\leq \sum_{k=1}^{h} \left(\sum_{\zeta=1}^{3} M_k^\zeta \right) \cdot \Phi(k) \\ &\leq \sum_{k=1}^{h} 3 \cdot \left(\max_{1\leq\zeta\leq 3} M_k^\zeta \right) \cdot \Phi(k) \\ &\leq \sum_{k=1}^{h} 3 \cdot \left(\max_{1\leq\zeta\leq 3} M_k^\zeta \right) \cdot c_3 2^{k+2} \log^3 n. \end{aligned}$$

The optimal communication cost for all the requests inside dense windows of all the levels is

$$A_D^*(\mathcal{E}) \geq 1/4 \cdot \max_{1\leq k\leq h} \left(\max_{1\leq\zeta\leq 3} M_k^\zeta \right) \cdot 2^{k-3},$$

using the cost $A(P^*)$ of for any level $1 \leq k \leq h$.

We now focus on bounding the costs $A_S(\mathcal{E})$ and $A_S^*(\mathcal{E})$ for executing the requests in the sparse windows in all the levels. Due to $S_k^j < 3$ requests inside each sparse window, it may not always be the case that these (at most) 2 requests satisfy the requirements for the lower bound derivation performed for dense windows. Therefore, the goal is to transform the sparse window scenario into a dense window case such that there are exactly two requests in each sparse window that are at least 2^{k-4} far in G. Note that in dense windows the distance lower bound was 2^{k-2}; here however it becomes 2^{k-4} (Lemma 9). The transformation given in [19] essentially defines another pair for each sparse window in level k for requests with distance $\text{dist}(v_i, v_j) < 2^{k-1}$ such that every two requests will have $\text{dist}(v_i, v_j) \geq 2^{k-1}$. After that the upper and lower bound costs similar to the dense window case will apply to the case of sparse windows. Finally, we have that the stretch of LB-Spiral is bounded by

$$\begin{aligned} \frac{A(\mathcal{E})}{A^*(\mathcal{E})} &= \frac{A_D(\mathcal{E}) + A_S(\mathcal{E})}{\max\{A_D^*(\mathcal{E}), A_S^*(\mathcal{E})\}} \\ &\leq \frac{2 \cdot \sum_{k=1}^{h} \cdot 3 \cdot \left(\max_{1\leq\zeta\leq 3} M_k^\zeta \right) \cdot c_3 2^{k+2} \log^3 n}{1/4 \cdot \max_{1\leq k\leq h} \left(\max_{1\leq\zeta\leq 3} M_k^\zeta \right) \cdot 2^{k-3}} \\ &\leq \frac{2 \cdot h \cdot \max_{1\leq k\leq h} \left(\max_{1\leq\zeta\leq 3} M_k^\zeta \right) \cdot c_3 2^{k+2} \log^3 n}{1/4 \cdot \max_{1\leq k\leq h} \left(\max_{1\leq\zeta\leq 3} M_k^\zeta \right) \cdot 2^{k-3}} \\ &= O(\log^3 n \cdot h) \\ &= O(\log^3 n \cdot \log D). \end{aligned}$$

Having the ratio $A(\mathcal{E})/A^*(\mathcal{E})$ above, we have the follow guarantees of LB-Spiral in dynamic executions.

Theorem 7. *The move stretch in* LB-SPIRAL *is* $O(\log^3 n \cdot \log D)$ *in dynamic executions. The processing load on any node of* G *is* $O(\log n \cdot \log D)$. *Moreover, the publish operation has* $O(D \cdot \log^3 n)$ *cost and any lookup operation has* $O(\log^6 n)$ *stretch.*

Proof. The *move* stretch of $O(\log^3 n \cdot \log D)$ is immediate from the analysis of the costs $A_D(\mathcal{E})$, $A_S(\mathcal{E})$, $A^*_D(\mathcal{E})$, and $A^*_S(\mathcal{E})$ above and their comparison since the ratio $\frac{A(\mathcal{E})}{A^*(\mathcal{E})}$ also applies to $\max_{\mathcal{E}} \frac{A(\mathcal{E})}{A^*(\mathcal{E})}$ due to our consideration of arbitrary $\mathcal{E}$ in establishing the ratio. The processing load is $O(\log n \cdot \log D)$ which again follows similarly as in Theorem 5. The *publish* cost is also $O(D \log^3 n)$ as in Theorem 2. The *lookup* stretch is also $O(\log^6 n)$ as the lookups can run concurrently with *move* operations. □

Proof of Theorem 1. Theorems 2–7 collectively prove Theorem 1 for LB-SPIRAL in any arbitrary (sequential, concurrent one-shot, or dynamic) execution. □

6. Improved results for constant doubling dimension graphs

If the metric of the underlying graph G has a constant doubling dimension (see [2,19]), we can improve both stretch and processing load for LB-SPIRAL. In particular, we can remove polylog(n) factors from the bounds. The *doubling dimension graph* is defined as follows: Let the space with radius δ of a point be called the ball centered at that point. A point set has doubling dimension ρ if any set of points that are covered by a ball of radius δ can be covered by 2^ρ balls of radius $\delta/2$. We say that a metric is doubling and has a low dimension if ρ is bounded by a small constant. The idea is to use the directory hierarchy $\mathcal{Z}$ suitable for doubling graphs. It was shown in [16,34] that $(O(1), O(1))$-labeled cover hierarchy is possible for small doubling graphs. In this section, we provide an alternative construction that essentially provides the same $(O(1), O(1))$-labeled cover hierarchy as in [16,34].

We perform the construction through the use of a maximal independent set algorithm, e.g., [36]. Formally, we define a sequence of *connectivity graphs* $I := \{I_0 = (V_0, E_0), I_1 = (V_1, E_1), \ldots, I_h = (V_h, E_h)\}$, where 0 is the lowest level and $h \leq \lceil \log D \rceil + 1$ is the highest level. At level 0, all nodes of G are in I_0, i.e., $V_0 = V$. Define E_ℓ to be the set of all edges in V_ℓ such that for each pair of nodes u, v in V_ℓ, $\text{dist}(u, v) \leq 2^{\ell+1}$. We define V_ℓ to be a maximal independent subset of $V_{\ell-1}$. V_h contains exactly one node, which is the root node r and E_h is the empty set.

We define the *default parent* and *update parent set* for each level ℓ node $w \in I_\ell$. The default parent of $w \in I_\ell$ is a node $w' \in I_{\ell+1}$ that is closest to w. Note that w' is within distance $2^{\ell+1}$ away from w. The update parent set of $w \in I_\ell$ is a subset of nodes in $I_{\ell+1}$ that are within distance $4 \cdot 2^{\ell+1}$ of w (including the default parent).

The directory $\mathcal{Z}$ is a layered node structure on I with the (overlay) edges between (default) parent-child pairs in every two consecutive level connectivity graphs I_ℓ and $I_{\ell+1}$. The (overlay) edge between two parent-child pairs may be a single edge or a path in G that connects those parent-child nodes in the original graph G (pick a shortest path if multiple such paths exist).

Consider node x in the bottom-level connectivity graph I_0. Denote by $default^\ell(x)$ the level-ℓ default parent of x. $default^\ell(x)$ is a recursive definition such that $default^0(x) = x$, and $default^\ell(x)$ is the default parent of $default^{\ell-1}(x)$. Denote by $updateparent^\ell(x)$ the update parent set of $default^{\ell-1}(x)$.

Lemma 10. *In constant-doubling networks, there are at most* $2^{3\rho}$ *number of update parent nodes in* $\mathcal{Z}$ *at level* $\ell + 1$ *for any node at level* ℓ, $1 \leq \ell \leq h$.

Proof. Pick a level ℓ node x in $\mathcal{Z}$. According to the construction of $\mathcal{Z}$, all the update parent sets of x in level $\ell + 1$ are within distance $4 \cdot 2^{\ell+1}$ from x. According to the definition of doubling dimension, all the nodes within distance $2^{\ell+1}$ are covered by 2^ρ balls of radius 2^ℓ. Which also means that all the nodes within distance $2 \cdot 2^{\ell+1}$ are covered by 2^ρ balls of radius $2 \cdot 2^\ell$. Therefore, all the nodes within distance $2 \cdot 2^{\ell+1}$ are covered by $(2^\rho)^2 = 2^{2\rho}$ balls of radius 2^ℓ, since each ball of radius $2 \cdot 2^\ell$ is covered by 2^ρ balls of radius 2^ℓ. Extending this definition recursively, the update parent sets of x in level $\ell + 1$ are covered by $(2^\rho)^3 = 2^{3\rho}$ balls of radius 2^ℓ, since they are within distance $4 \cdot 2^{\ell+1}$ from x. Moreover, any two nodes in the update parent set at level $\ell + 1$ are at least distance $2^{\ell+1}$ from each other since they are maximal independent sets at level ℓ. Therefore, x cannot have more that $2^{3\rho}$ level $\ell + 1$ update parents. □

We now define spiral path $p(u)$ for each node $u \in V$. $p(u)$ is formed by connecting the ascending sequence of update parent sets of node u starting from $updateparent^0(u) = u$ at level 0 to $updateparent^h(u) = r$ at the root level h (note that $default^\ell(u) \in updateparent^\ell(u)$ and hence $p(u)$ visits $default^\ell(u)$ in each level ℓ). Nodes in $updateparent^\ell(u)$ in each level ℓ are visited according to their node IDs in the increasing order. Thus, the highest ID node in $updateparent^\ell(u)$ is connected to the smallest ID node in $updateparent^{\ell+1}(u)$. This ordered visit of the nodes in each level resembles labeling of the clusters we used in Section 3.

Lemma 11. *For any two nodes $u, v \in V$, their spiral paths $p(u)$ and $p(v)$ intersect at level $\lceil \log(\text{dist}(u,v)) \rceil + 1$.*

The directory (or canonical) path can also be defined similarly as in Section 3.

Lemma 12. *For any canonical path q up to level k in $\mathcal{Z}$,* $\text{length}(q) \leq 2^{k+3\rho+6}$.

Proof. The distance between any level ℓ node and any level $\ell+1$ node is $2 \cdot 4 \cdot 2^{\ell+1}$. The distance between any two nodes in level $\ell+1$ is also $2 \cdot 4 \cdot 2^{\ell+1}$. Moreover, from Lemma 10, there are at most $2^{3\rho}$ nodes in the update parent set at level $\ell+1$. Therefore, the cost of traversing from the last node in ℓ to the last node in level $\ell+1$ is

$$\leq 2 \cdot 4 \cdot 2^{\ell+1} + 2^{3\rho} \cdot (2 \cdot 4 \cdot 2^{\ell+1}) \leq 2 \cdot 2^{3\rho} \cdot 2 \cdot 4 \cdot 2^{\ell+1} \leq 2^{3\rho} \cdot 2^{\ell+5}.$$

Combining the one level cost from level 0 up to level k, we have that,

$$\text{length}(q) \leq \sum_{\ell=0}^{k} (2^{3\rho} \cdot 2^{\ell+5}) \leq 2^{k+3\rho+6},$$

since $\text{length}(q)$ is the sum of the distances that increase by a factor 2 between two consecutive levels. □

The goal is now to run the operations of LB-Spiral on the directory $\mathcal{Z}$ developed analogously to Section 4. Therefore, in the small doubling graph G, we obtain the following theorem.

Theorem 8. *If the underlying topology G is a small doubling graph, the move stretch in* LB-Spiral *is $O(\log D)$ in any arbitrary (sequential, one-shot, or dynamic) execution. It achieves $O(\log D)$ approximation on processing load on any node of G. Moreover, the publish operation has $O(D)$ cost and any lookup operation has $O(1)$ stretch.*

Proof. The *publish* cost is $O(D)$ as the spiral path is now of length $O(2^k)$ for any level k and hence for the highest level $h = \lceil \log D \rceil + 1$ the length becomes $O(2^h) = O(2^{\lceil \log D \rceil + 1}) = O(D)$.

For the *lookup* stretch, Lemma 4 can be modified such that the spiral path $p(w)$ of a *lookup* issuing node w is guaranteed to either intersect with the directory path to the object owner node v that is at distance $\text{dist}(w,v) \leq 2^i$ or find a *slink* to the directory path for the object at level at most $\eta = i + \phi$, where $\phi = 3\rho + 8$. Let v_i be the level i leader node. As in Lemma 4, we need

$$\text{dist}(v_i, w) \leq \text{length}(q_i) + \text{dist}(w,v) \leq 2^{i+3\rho+6} + 2^i \leq 2^{i+3\rho+7} = 2^{\eta-1} = \gamma_\eta$$

with $\eta = i + 3\rho + 8$ for the η level leader node to have the *slink* information to v_i for the directory path to ξ at v. Therefore, the *lookup* cost is $A(r_L) \leq 3 \cdot \text{length}(q_\eta)$ as in Theorem 3. Which means,

$$A(r_L) \leq 3 \cdot 2^{\eta+3\rho+6} \leq 3 \cdot 2^{i+3\rho+8+3\rho+6} \leq 3 \cdot 2^{6\rho+14} \cdot 2^i.$$

The optimal cost $A^*(r_L) \geq 2^{i-1}$. Therefore, the stretch for the *lookup* operation is

$$\frac{A(r_L)}{A^*(r_L)} \leq \frac{3 \cdot 2^{6\rho+14} \cdot 2^i}{2^{i-1}} = O(1),$$

for a constant ρ.

For the *move* stretch, following the analysis as in Theorem 4, we have that

$$\max_{\mathcal{E}} \frac{A(\mathcal{E})}{A^*(\mathcal{E})} \leq \frac{h \cdot \max_{1 \leq k \leq h}(S_k - 1) 2^{k+3\rho+6}}{\max_{1 \leq k \leq h}(S_k - 1) 2^{k-1}} = O(\log D),$$

since ρ is a constant. For the one-shot and dynamic executions, the same stretch bound holds adapting the analysis.

For the processing load we can reduce the $O(\log n)$ factor to $O(1)$ since in each level there are only $O(2^{3\rho})$ sub-levels to visit instead of $O(\log n)$ sub-levels in the general topology. Therefore, the processing load approximation of a node of doubling graph G in LB-Spiral is

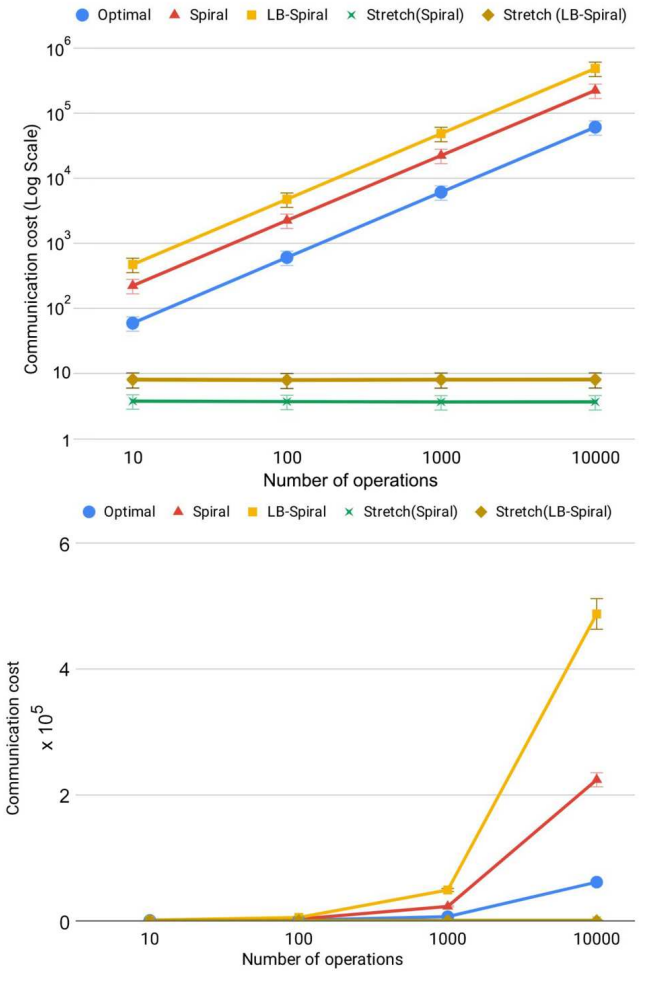

Fig. 8. The communication cost results of LB-Spiral and Spiral in executing 10–10,000 *move* operations in random networks of 128 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

$$\frac{PL(x)}{PL^*(x)} \leq \frac{k \cdot O(\log D)}{k} = O(\log D). \quad \square$$

7. Simulation results

Given nice theoretical performance guarantees of LB-Spiral in minimizing both stretch and processing load, we aim here to investigate how these properties translate in real world through experimental evaluation.[2] For the evaluation, we use the Erdős-Rényi model [20] and generate random graphs of different sizes, ranging from 64 nodes to 1,024 nodes. Particularly, we use the $G(n, \rho)$ variant of the Erdős-Rényi model [20] where a graph G is constructed connecting nodes randomly such that each edge is included in G with probability $0 < \rho < 1$, independent from every other edge. The graphs we use in the experiments are generated setting $\rho = 0.1$. The weight of each edge is also chosen independently from the weight of every other edge at random from 1 to 10. The results are presented and analyzed only for *move* operations on a single shared object. The *move* operations are generated uniformly at random among the available nodes of the graph every time a request is issued. We implement LB-Spiral in sequential executions ranging from 10 to 10,000 *move* operations. The main goal is to see how LB-Spiral does in terms of processing load and communication overhead since communication cost has already been evaluated for Spiral in [13]. The directory $\mathcal{Z}$ built is initialized by creating a downward path from the root to a bottom level node that currently owns the object through a *publish* operation. The results are compared with the state-of-the-art algorithm Spiral that does not balance the processing load (only minimizes the stretch). The results presented are the average of 10 experimental runs. The data points plotted also show the deviation (from average) on the measurements.

[2] The implementation is available in Github through the following link: https://github.com/shishirrraic/LB-Spiral.

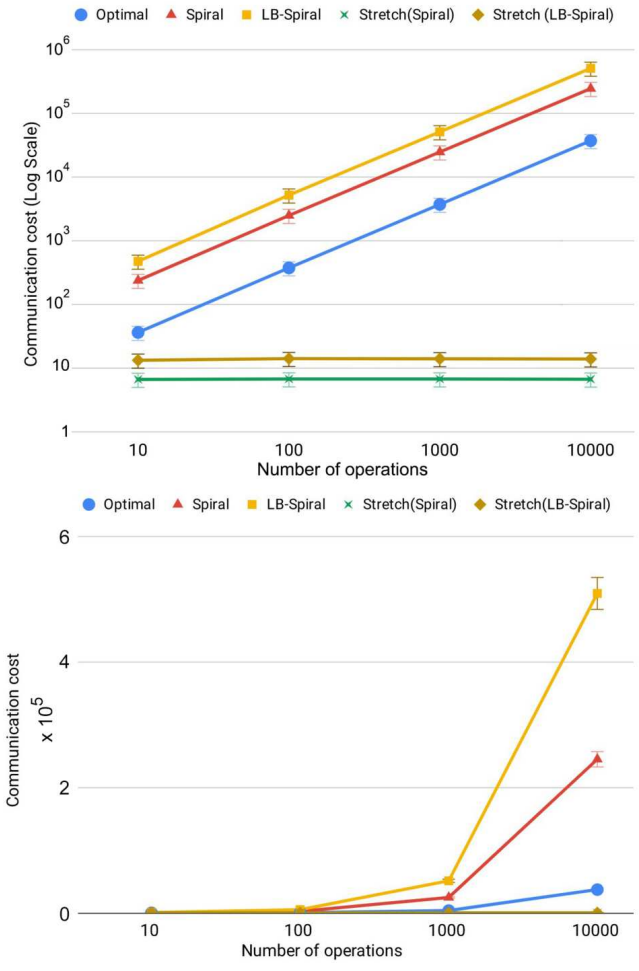

Fig. 9. The communication cost results of LB-Spiral and Spiral in executing 10–10,000 *move* operations in random networks of 512 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

The performance of the protocols is measured with respect to (a) the communication cost, (b) the communication overhead, and (c) the processing load. The communication cost is measured through the sum of the weights of the edges the requests traverse following the protocol. The processing load of a node is measured through the number of times that node is used while serving the requests following the protocol. The communication overhead is measured summing the extra distance the requests need to traverse in the leader change process in LB-Spiral; Spiral does not incur this cost as leaders are pre-selected and they never change. We then compare the total communication cost (the total processing load) with the optimal communication cost (the optimal processing load) and present the results in terms of the ratio (stretch and processing load ratio). We assume that the execution proceeds in steps such that every node can receive, process, and send a message in each step.

7.1. Communication cost results

We are now ready to present the communication cost (stretch) and overhead results. We start with the communication cost results varying the number of *move* operations in a network of fixed size. Figs. 8–10 show the communication cost results of LB-Spiral and Spiral in executing 10–10,000 *move* operations in random networks of 128, 512, and 1024 nodes, respectively. The results show that Spiral performs better in terms of communication cost which is in line of the theoretical results since the stretch of LB-Spiral is $O(\log n)$ factor worse compared to Spiral. However, the performance of LB-Spiral is within a factor of 4 compared to Spiral in all our experiments. The observation here is that the $O(\log n)$ does not always appear in the communication cost.

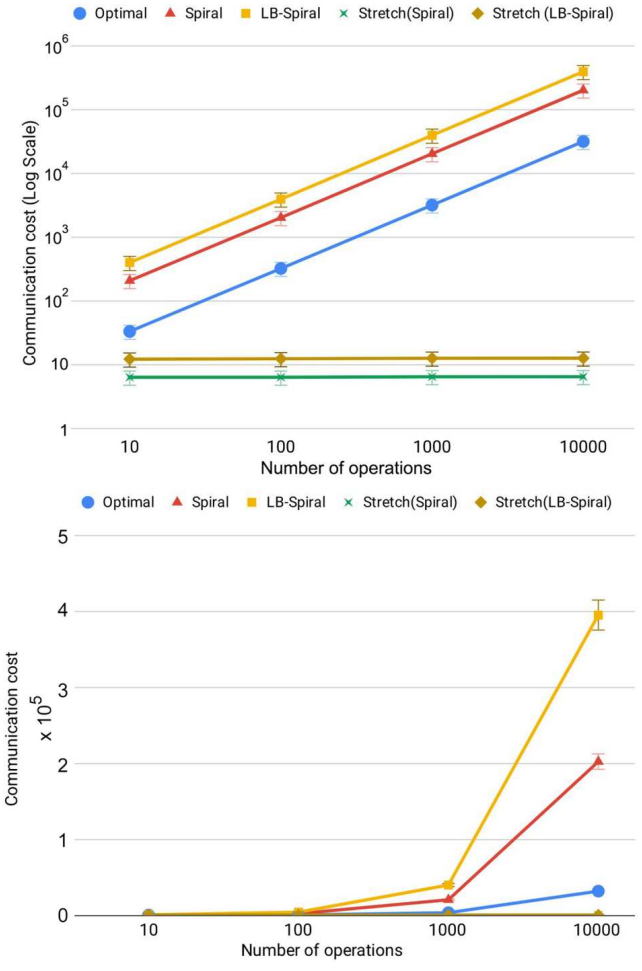

Fig. 10. The communication cost results of LB-Spiral and Spiral in executing 10–10,000 *move* operations in random networks of 1024 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

We are now interested to see how the performance of LB-Spiral compares with the performance of Spiral in executing the fixed set of *move* operations while varying the network size. Figs. 11 and 12 show the communication cost results of LB-Spiral and Spiral in executing 1,000 and 10,000 *move* operations in random networks consisting of 64, 128, 256, 512, and 1,024 nodes, respectively. The results show the consistent difference on performance between LB-Spiral and Spiral. This consistency is due to the denseness of the graphs with more nodes and substantial decrease in diameter.

Figs. 13 and 14 show the overhead cost results of LB-Spiral in executing 1,000 and 10,000 *move* operations in random networks consisting of 64, 128, 256, 512, and 1,024 nodes, respectively. Note that since the leaders in each cluster is pre-selected at the directory $\mathcal{Z}$ construction time in Spiral, there is no overhead communication cost for Spiral. The overhead cost in LB-Spiral is the cost involved in informing the leaders of the parent and child clusters about the leader change in the current cluster in the spiral path. The results show the overhead in LB-Spiral is within a small constant factor of the communication cost of Spiral which has no overhead.

7.2. Processing load results

We now present the processing load results of LB-Spiral and Spiral in executing 1,000 and 10,000 *move* operations in random networks consisting of 128, 512, ans 1,024 nodes, respectively. The results are in Fig. 15–20. The results show that LB-Spiral balances the processing load better compared to Spiral. Results also show that the processing load increases proportionally with network size.

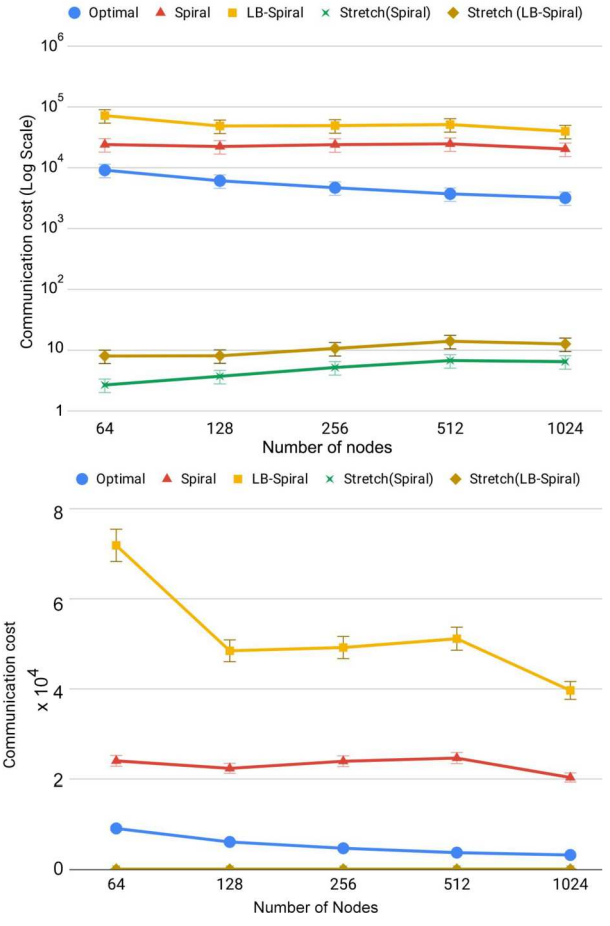

Fig. 11. The communication cost results of LB-Spiral and Spiral in executing 1,000 *move* operations in random networks consisting of 64, 128, 256, 512, and 1024 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

8. Concluding remarks

We have presented a DDP LB-Spiral for shared objects for supporting shared memory implementations in distributed systems working following general network topologies. The distinctive feature of LB-Spiral is that it is the first DDP for general network topologies that is (processing) load balanced simultaneously with low stretch. Previous DDPs for general network topologies only minimized stretch. The *move* stretch of LB-Spiral can be improved to $O(\log^3 n \cdot \min\{\log n, \log D\})$ (to $O(\min\{\log n, \log D\})$ for constant doubling graphs) using the analysis technique of [19]. The simulation results showed the usefulness of LB-Spiral in practical scenarios in balancing the processing load compared to the state-of-the-art protocol Spiral [13], without much increasing the stretch. For future work, it would be interesting to extend LB-Spiral for dynamic networks where nodes enter and leave at any time and make it fault-tolerant.

Declaration of competing interest

None declared.

Acknowledgments

This work is supported by The National Science Foundation grant CCF-1936450.

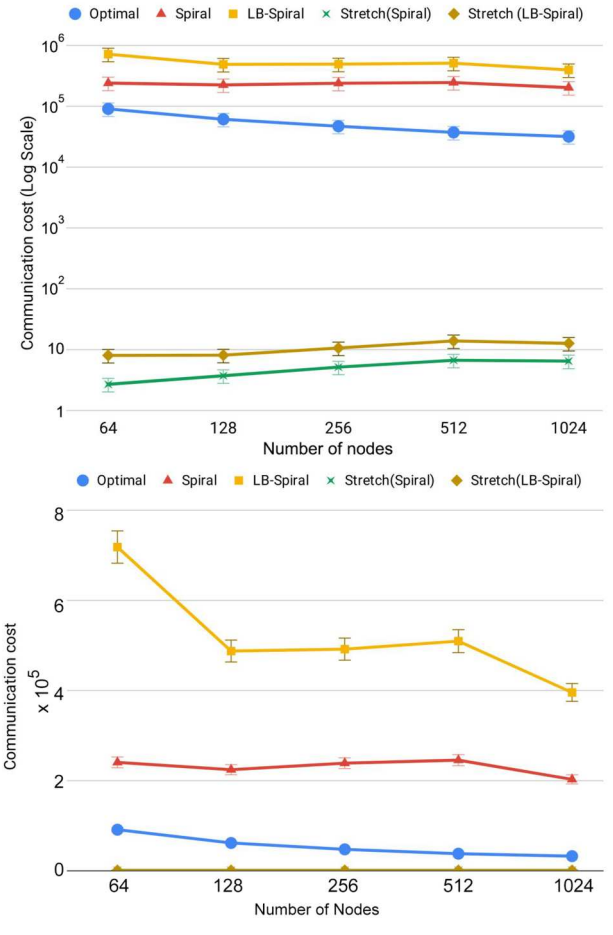

Fig. 12. The communication cost results of LB-Spiral and Spiral in executing 10,000 *move* operations in random networks consisting of 64, 128, 256, 512, and 1,024 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

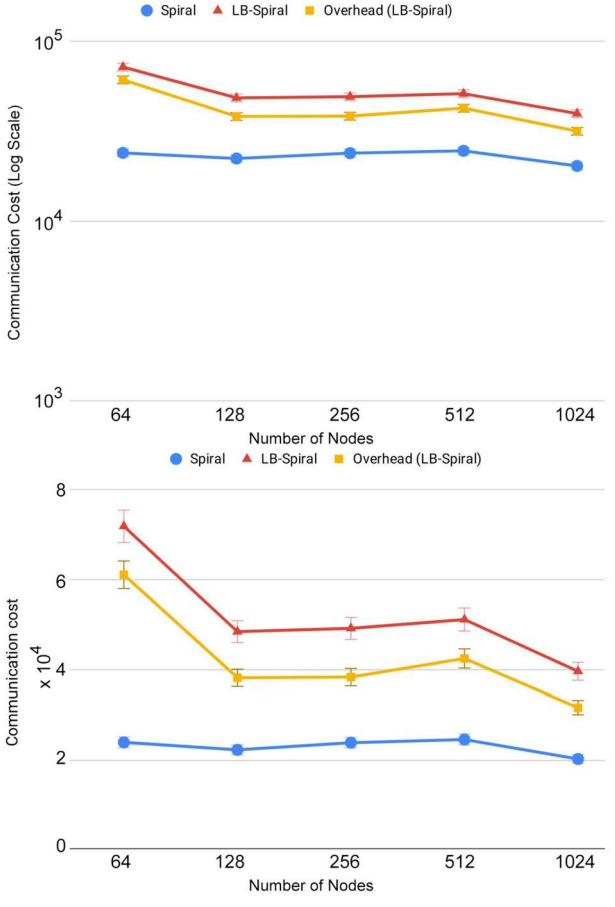

Fig. 13. The overhead cost results of LB-SPIRAL in executing 1,000 *move* operations in random networks consisting of 64, 128, 256, 512, and 1024 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

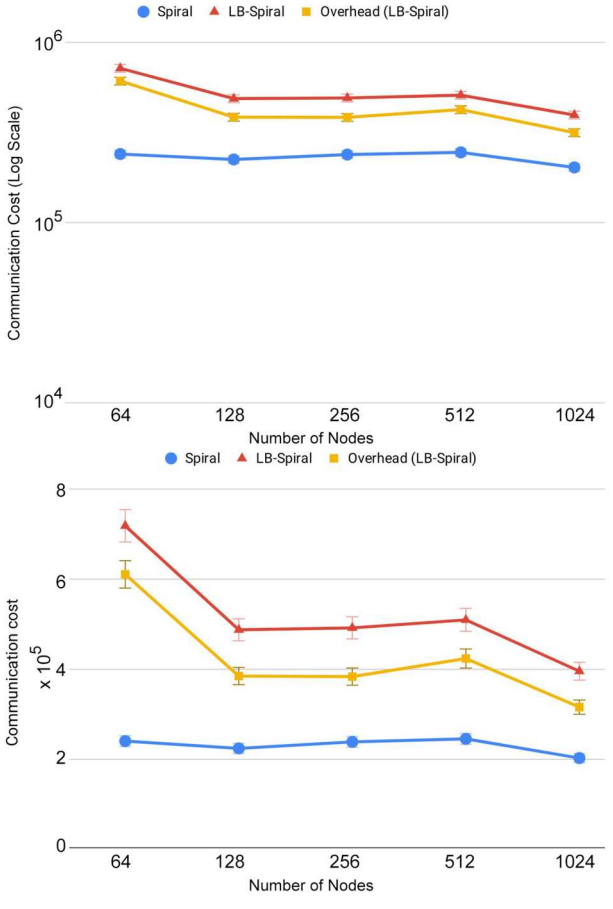

Fig. 14. The overhead cost results of LB-SPIRAL in executing 10,000 *move* operations in random networks consisting of 64, 128, 256, 512, and 1,024 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

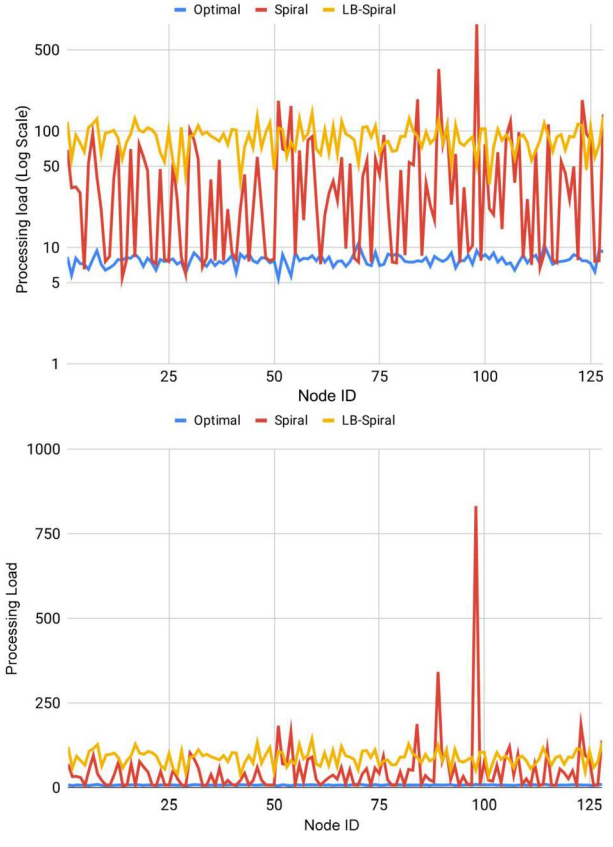

Fig. 15. The processing load results of LB-Spiral and Spiral in executing 1,000 *move* operations in random networks of 128 nodes: (top) log scale and (bottom) non-log scale. Lower is better. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

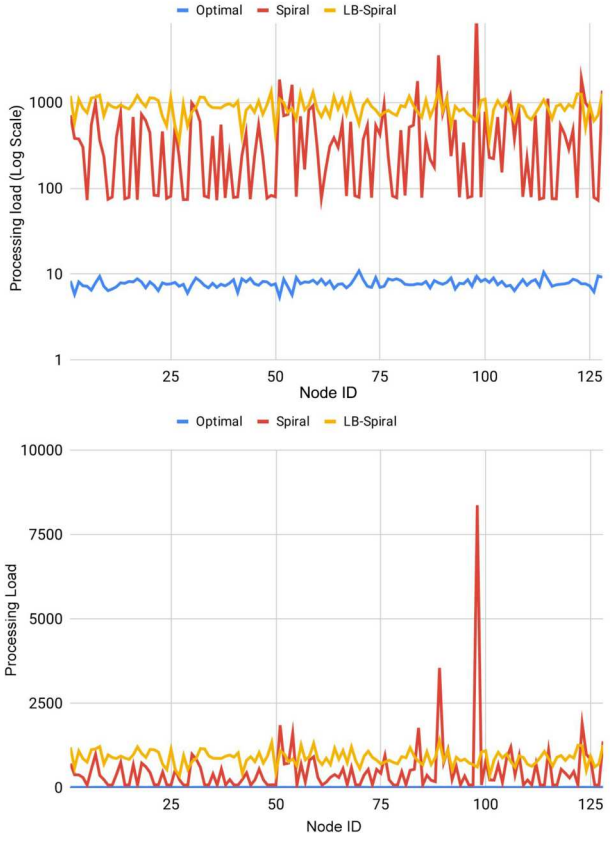

Fig. 16. The processing load results of LB-Spiral and Spiral in executing 10,000 *move* operations in random networks of 128 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

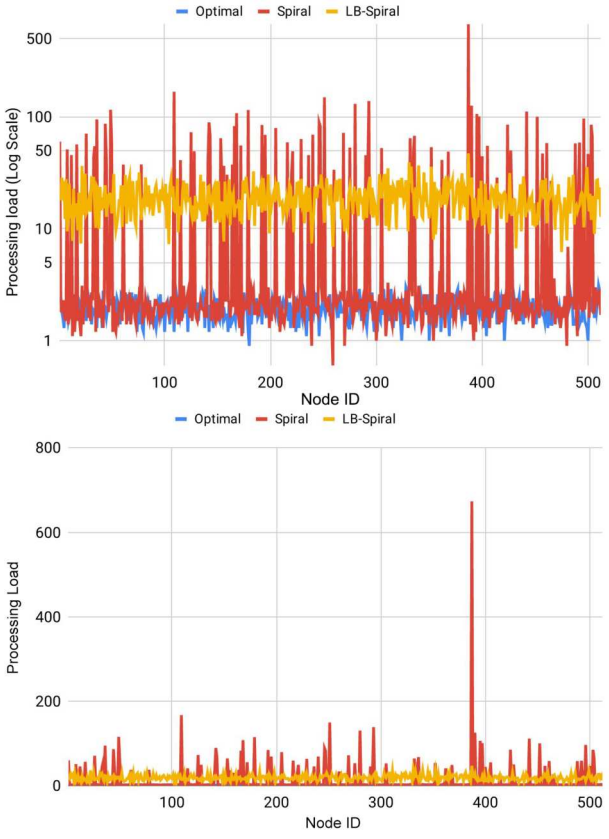

Fig. 17. The processing load results of LB-Spiral and Spiral in executing 1,000 *move* operations in random networks of 512 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

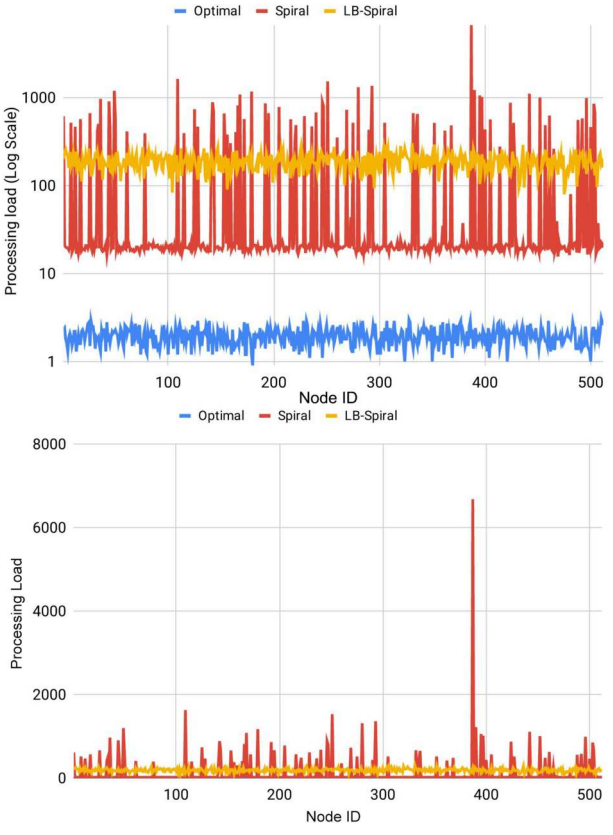

Fig. 18. The processing load results of LB-Spiral and Spiral in executing 10,000 *move* operations in random networks of 512 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

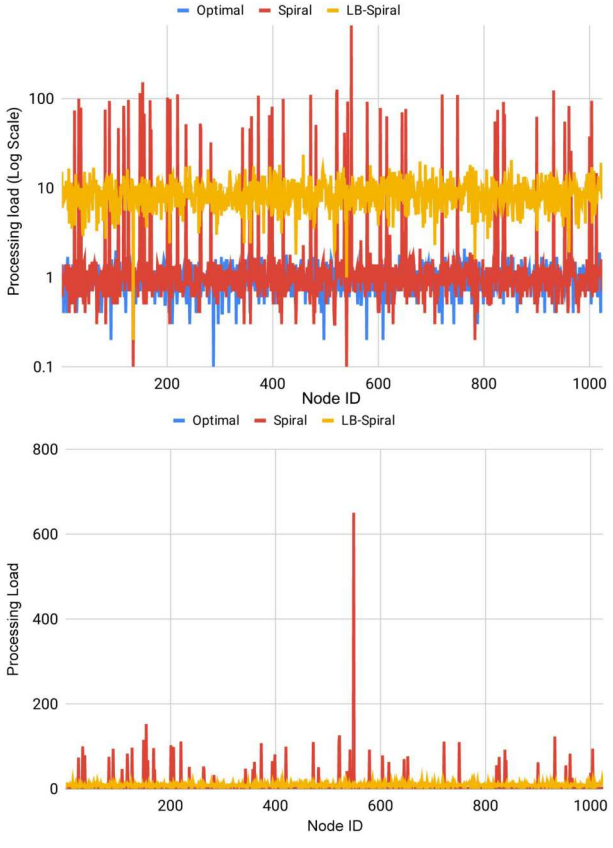

Fig. 19. The processing load results of LB-Spiral and Spiral in executing 1,000 *move* operations in random networks of 1024 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

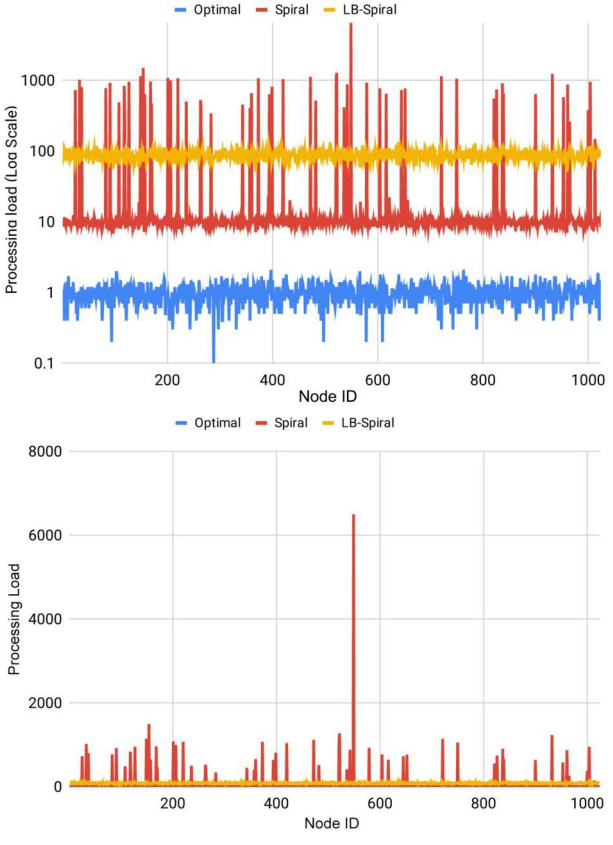

Fig. 20. The processing load results of LB-Spiral and Spiral in executing 10,000 *move* operations in random networks of 1024 nodes: (top) log scale and (bottom) non-log scale. Lower is better.

References

[1] S. Rai, G. Sharma, C. Busch, M. Herlihy, Load balanced distributed directories, in: SSS, 2018, pp. 221–238.
[2] M. Herlihy, Y. Sun, Distributed transactional memory for metric-space networks, Distrib. Comput. 20 (3) (2007) 195–208.
[3] D. Chaiken, C. Fields, K. Kurihara, A. Agarwal, Directory-based cache coherence in large-scale multiprocessors, Computer 23 (6) (1990) 49–58.
[4] L.M. Censier, P. Feautrier, A new solution to coherence problems in multicache systems, IEEE Trans. Comput. 27 (12) (1978) 1112–1118.
[5] A. Agarwal, D. Chaiken, D. Kranz, J. Kubiatowicz, K. Kurihara, G. Maa, D. Nussbaum, M. Parkin, D. Yeung, The mit alewife machine: a large-scale distributed-memory multiprocessor, in: Workshop on Scalable Shared Memory Multiprocessors, 1991, pp. 239–261.
[6] M.J. Demmer, M. Herlihy, The arrow distributed directory protocol, in: DISC, 1998, pp. 119–133.
[7] B. Awerbuch, D. Peleg, Concurrent online tracking of mobile users, SIGCOMM Comput. Commun. Rev. 21 (4) (1991) 221–233.
[8] K. Raymond, A tree-based algorithm for distributed mutual exclusion, ACM Trans. Comput. Syst. 7 (1) (1989) 61–77.
[9] N. Shavit, D. Touitou, Software transactional memory, Distrib. Comput. 10 (2) (1997) 99–116.
[10] M. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lock-free data structures, in: ISCA, 1993, pp. 289–300.
[11] H. Attiya, V. Gramoli, A. Milani, Directory protocols for distributed transactional memory, in: Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001, Springer, 2015, pp. 367–391.
[12] B. Zhang, B. Ravindran, Brief announcement: Relay: a cache-coherence protocol for distributed transactional memory, in: OPODIS, 2009, pp. 48–53.
[13] G. Sharma, C. Busch, Distributed transactional memory for general networks, Distrib. Comput. 27 (5) (2014) 329–362.
[14] G. Sharma, C. Busch, A load balanced directory for distributed shared memory objects, J. Parallel Distrib. Comput. 78 (2015) 6–24.
[15] N. Alon, G. Kalai, M. Ricklin, L.J. Stockmeyer, Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling, Theor. Comput. Sci. 130 (1) (1994) 175–201.
[16] L. Jia, G. Lin, G. Noubir, R. Rajaraman, R. Sundaram, Universal approximations for tsp, steiner tree, and set cover, in: STOC, 2005, pp. 386–395.
[17] M.T. Hajiaghayi, R. Kleinberg, T. Leighton, Improved lower and upper bounds for universal tsp in planar metrics, in: SODA, 2006, pp. 649–658.
[18] I. Gorodezky, R.D. Kleinberg, D.B. Shmoys, G. Spencer, Improved lower bounds for the universal and a priori tsp, in: APPROX/RANDOM, 2010, pp. 178–191.
[19] G. Sharma, C. Busch, Optimal nearest neighbor queries in sensor networks, Theor. Comput. Sci. 608 (2015) 146–165.
[20] P. Erdös, A. Rényi, On random graphs i, Publ. Math. (Debr.) 6 (1959) 290.
[21] C. Busch, M. Magdon-Ismail, J. Xi, Optimal oblivious path selection on the mesh, IEEE Trans. Comput. 57 (5) (2008) 660–671.
[22] H. Räcke, Minimizing congestion in general networks, in: FOCS, 2002, pp. 43–52.
[23] G. Sharma, C. Busch, An analysis framework for distributed hierarchical directories, Algorithmica 71 (2) (2015) 377–408.
[24] A. Ghodselahi, F. Kuhn, Dynamic analysis of the arrow distributed directory protocol in general networks, in: DISC, 2017, pp. 22:1–22:16.
[25] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan Chord, A scalable peer-to-peer lookup service for internet applications, SIGCOMM Comput. Commun. Rev. 31 (4) (2001) 149–160.
[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, SIGCOMM Comput. Commun. Rev. 31 (4) (2001) 161–172.
[27] A.I.T. Rowstron, P. Druschel, Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems, in: Middleware, 2001, pp. 329–350.
[28] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment, IEEE J. Sel. Areas Commun. 22 (1) (2006) 41–53.
[29] R. Rajaraman, A.W. Richa, B. Vöcking, G. Vuppuluri, A data tracking scheme for general networks, in: SPAA, 2001, pp. 247–254.
[30] C.G. Plaxton, R. Rajaraman, A.W. Richa, Accessing nearby copies of replicated objects in a distributed environment, in: SPAA, 1997, pp. 311–320.
[31] K. Talwar, Bypassing the embedding: algorithms for low dimensional metrics, in: STOC, 2004, pp. 281–290.
[32] R. Krauthgamer, J.R. Lee, Navigating nets: simple algorithms for proximity search, in: SODA, 2004, pp. 798–807.
[33] G. Sharma, C. Busch, Distributed transactional memory for general networks, Distrib. Comput. 27 (5) (2014) 329–362.
[34] A. Gupta, M.T. Hajiaghayi, H. Räcke, Oblivious network design, in: SODA, 2006, pp. 970–979.
[35] G. Robins, A. Zelikovsky, Improved steiner tree approximation in graphs, in: SODA, 2000, pp. 770–779.
[36] M. Luby, A simple parallel algorithm for the maximal independent set problem, in: STOC, 1985, pp. 1–10.

Sponsor names

Do not correct this page. Please mark corrections to sponsor names and grant numbers in the main text.

The National Science Foundation, *country*=United States, *grants*=CCF-1936450