Planting Trees for scalable and efficient Canonical Hub Labeling

Kartik Lakhotia[†], Rajgopal Kannan[‡], Qing Dong[†], Viktor Prasanna[†]
[†]Ming Hsieh Department of Electrical Engineering, University of Southern California
[‡]U.S Army Research Lab, Los Angeles, CA 90094
[†]{klakhoti, dongqing, prasanna}@usc.edu; [‡]rajgopal.kannan.civ@mail.mil

ABSTRACT

Hub labeling is widely used to improve the latency and throughput of Point-to-Point Shortest Distance (PPSD) queries in graph databases. However, constructing hub labeling, even via the state-of-the-art Pruned Landmark Labeling (PLL) algorithm is computationally intensive. PLL further has a sequential root order label dependency that makes it challenging to parallelize. Hence, the existing parallel approaches are often plagued by label size increase, poor scalability and inability to process large weighted graphs.

In this paper, we develop novel algorithms that construct the minimal (guaranteed) Canonical Hub Labeling on shared and distributed-memory parallel systems in a scalable and efficient manner. Our key contribution, the PLaNT algorithm, provides an *embarrassingly parallel* approach for label construction that scales well beyond the limits of current practice. Our approach *is the first* to employ a collaborative label partitioning scheme across multiple nodes of a cluster, for completely in-memory labeling and parallel querying on massive graphs whose labels cannot fit on a single node.

On a single node with 72-threads, our shared-memory algorithm is up to 47.4× faster than sequential PLL. While our labeling time is comparable to the state-of-the-art shared-memory paraPLL, our label size is 17% smaller on average.

PLaNT demonstrates superior parallel scalability. It can process significantly larger graphs and construct labeling orders of magnitude faster than the state-of-the-art distributed paraPLL. Compared to the best shared-memory parallel algorithm, it achieves up to 9.5× speedup on a 64 node cluster.

PVLDB Reference Format:
Kartik Lakhotia, Rajgopal Kannnan, Qing Dong, Viktor Prasanna. Planting Trees for scalable and efficient Canonical Hub Labeling. *PVLDB*, 13(4): 492-505, 2019.
DOI: https://doi.org/10.14778/3372716.3372722

1. INTRODUCTION

Point-to-Point Shortest Distance (PPSD) computations are one of the most important primitives encountered in graph databases. PPSD computations on *large weighted graphs* forms an important part of many applications such as phylogenetics [46, 35, 33], analysis of protein interaction networks [53, 49] in bioinformatics, context-aware search on knowledge graphs [47, 54], route navigation on roads [55], social network analysis [31, 43] etc. These applications generate a large number of PPSD queries and fast online query response is crucial to their performance. While graph traversal can be used to answer PPSD queries,, even state-of-the-art traversal methods [32, 14, 48, 44, 21, 56, 25] take hundreds of milliseconds to answer a single query on large graphs. On the other hand, pre-computing all pairs shortest paths enables constant time query response, but is impractical due to quadratic storage and time complexity.

Hub labeling offers a practical solution to this problem. By pre-computing for each vertex, the distance to a 'small' subset of vertices known as *hubs*, hub labeling enables fast response to PPSD queries, without incurring quadratic time and storage costs. The set of (hub, distance)-tuples for a vertex v are known as the *hub labels* of v. There are several labeling approaches that differ in terms of the number of hops (two [17, 4] or more [7]), completeness of labels (partial [26] vs complete cover [17, 4]), target graphs (complex [26, 10] vs road [7] networks) etc. Among these, 2-hop Hierarchical Hub Labeling (HHL) [17, 4] is a particularly comprehensive complete cover approach that uses a given vertex ranking R (also called network hierarchy) to define the usefulness of labels It only allows the appearance of the highest ranked vertex as a hub on any shortest path, thus enabling (1) reasonable labeling size for graphs with varied topology, dimensionality and edge weights; and (2) query response in microseconds on large graphs.

Pruned Landmark Labeling (PLL) [8] is arguably the most efficient *sequential* algorithm for constructing HHL. PLL iteratively selects vertices in rank order and constructs Shortest Path Trees (SPTs) rooted at the selected vertices. The root along with the distance is added as a new hub label for each vertex explored in an SPT. Significantly, PLL exploits the existing labels from previous SPTs to prune those paths in the current (under-construction) SPT, that are already covered by other hubs. This pruning forms the basis of PLL's efficiency and results in the minimal labeling (for a given R), also known as Canonical Hub Labeling (CHL) [4] (see section 2 for details). However, it also introduces a sequential *root order label dependency* between the SPTs, wherein pruning in an SPT depends on the knowledge of previously generated labels with higher-ranked hubs.

Despite aggressive pruning, PLL is computationally very demanding. We attempted labeling several real-world graphs using PLL and its parallel variants [45, 22], and observed that large weighted graphs such as *Pokec* (1.5M nodes, 30M

Proceedings of the VLDB Endowment, Vol. 13, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3372716.3372722

edges) and *LiveJournal* (4.8M nodes, 69M edges) can take several hours to days to process[1]. We also note that the size of CHL for weighted graphs can be significantly larger than the graph itself, stressing the available main memory on a single machine[2]. While disk-based labeling can extend system capabilities beyond DRAM, disk access and resulting algorithms [34, 26] are substantially slower than PLL.

These examples illustrate the fundamental challenges in scaling CHL construction to large weighted graphs - high computational requirements and memory space demand (due to large label sizes). This motivates the use of massively parallel systems for hub labeling, such as distributed-memory multi-node clusters that offer large amounts of extendable computational resources and main memory capacity. However, parallelizing CHL construction on such systems comes with its own set of challenges. Most existing parallel approaches [45, 24] concurrently construct multiple SPTs in PLL using parallel threads on a single multicore server. Such simple parallelization breaks the root order label dependency of PLL, violating the network hierarchy and resulting in larger label sizes. Many mission critical applications require the CHL for a *specific network hierarchy* and larger label sizes will directly impact query performance. Parallelizing label construction on a cluster of nodes exacerbates these problems, as the labels generated on a node are not immediately visible to other nodes for pruning. To the best of our knowledge, none of the existing parallel labeling approaches efficiently utilize the available main memory and parallelism in a cluster.

Motivated by these drawbacks in existing approaches to hub labeling, in this paper, we design novel parallel algorithms that address the multiple challenging facets of the parallel CHL construction problem. Two key perspectives drive the development of our algorithmic innovations and optimizations. First, we approach simultaneous construction of multiple SPTs in PLL as an *optimistic parallelization* that can result in mistakes. We develop PLL-inspired shared-memory parallel Global Local Labeling (GLL) and Distributed-memory Global Local Labeling (DGLL) algorithms that

1. only make mistakes from which they can recover, and
2. efficiently correct those mistakes.

Second, we note that mistake correction in DGLL generates huge amount of label traffic, thus limiting its parallel scalability. Therefore, we shift our focus from parallelizing PLL to the *primary problem* of parallel CHL construction. Drawing insights from the fundamental concepts behind CHL, we develop an *embarrassingly parallel* and *communication avoiding* algorithm called **PLaNT** (Prune Labels and (do) Not (Prune) Trees). Unlike PLL which prunes SPTs but inserts labels for all explored vertices, PLaNT does not prune SPTs but inserts labels selectively. PLaNT ensures correctness and minimality of output hub labels (for a given R) generated from an SPT without consulting previously discovered labels, as shown in fig. 1. This allows labeling to be partitioned across multiple nodes without increase in communication traffic and enables us to simultaneously scale effective parallelism and memory capacity using a cluster of nodes. By seamlessly transitioning between PLaNT and DGLL, we achieve both computational efficiency and high scalability.

[1]While CHL construction is a one-off task for static graphs, several real-world applications encounter dynamic graphs [16, 15, 13, 50] that require periodic relabeling [9], entailing fast CHL construction.

[2]For example, it requires $<$ 1GB to store the LiveJournal graph but $>$ 100GB to store its hub labels.

Table 1: Frequently used notations

$G(V, E, W)$	a weighted undirected graph with vertex set V and edges E
n, m	number of vertices and edges; $n = \|V\|, m = \|E\|$
N_v	neighboring vertices of v
$w_{u,v}$	weight of edge $e = (u, v) \in E$
$SP_{u,v}$	(vertices in) shortest path(s) between u, v (inclusive)
SPT_v	shortest path tree rooted at vertex v
$d(u, v)$	shortest path distance between vertices u and v
$(h, d(v, h))$	a hub label for vertex v with h as the hub
L_v	set of hub labels for vertex v
q	number of nodes in the cluster

Overall, our contributions can be summarized as follows:

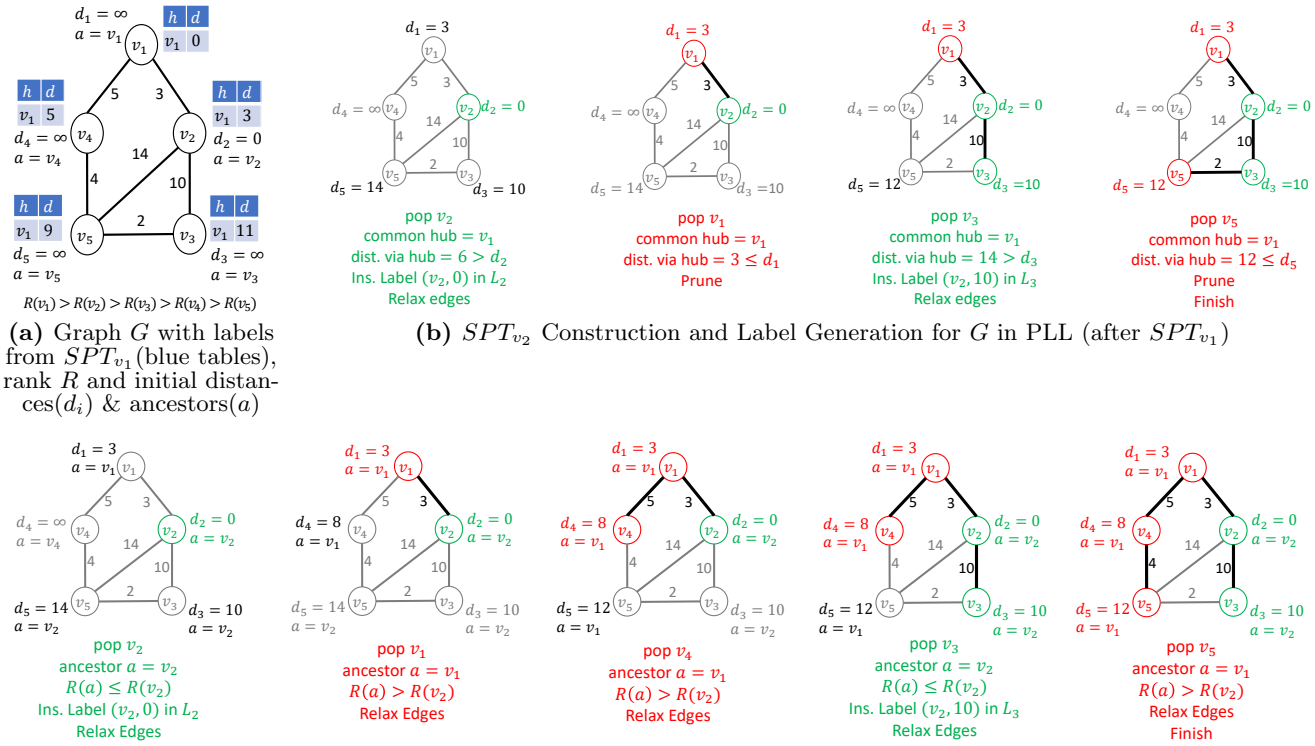
- We develop the first shared and distributed-memory parallel algorithms that output the minimal hub labeling (CHL) for a given weighted graph and network hierarchy.
- We develop PLaNT - a new embarrassingly parallel algorithm for distributed-memory CHL construction, that eschews all label order dependencies to completely avoid inter-node communication and achieve high *scalability* (at the cost of some extra computation). We then extend it to a Hybrid algorithm that combines the scalability of PLaNT with the *efficiency* of label-based pruning. The Hybrid algorithm can label the aforementioned Pokec and LiveJournal graphs in just 13 and 37 minutes, respectively.
- Our distributed approach *is the first* to use the memory of multiple cluster nodes in a collaborative fashion to enable completely in-memory processing of large graphs whose labels do not fit on the main memory of a single node. For instance, we can process the Livejournal graph with $>$ 100 GB labeling size on a cluster with only 64 GB DRAM per node, in contrast to existing main memory based approaches.
- We develop different schemes for label data distribution in a cluster to *increase query throughput* by utilizing parallel processing power of multiple nodes. None of the existing works apportion labels and parallelize query response computation on multiple nodes.

We use 12 real-world datasets to evaluate our algorithms. On a 64 node cluster, our distributed-memory approach achieves 42× self-relative speedup. It exhibits up to 9.5× speedup over the state-of-the-art shared-memory algorithm and orders of magnitude improvement over the state-of-the-art distributed-memory algorithm.

2. BACKGROUND

Table 1 lists some frequently used notations in this paper. For clarity of description, we consider $G(V, E, W)$ to be weighted and undirected. However, the techniques described here can be easily extended to directed graphs by using *forward* and *backward* labels for each vertex [4].

2-hop Hub Labeling [17]: A 2-hop hub labeling directly connects each vertex to its respective hubs such that the shortest distance between vertices u and v can be computed by hopping from u to a common hub h and from h to v. Such a labeling can correctly answer any PPSD query if it satisfies the following *cover property*: Every connected pair



(a) Graph G with labels from SPT_{v_1} (blue tables), rank R and initial distances(d_i) & ancestors(a)

(b) SPT_{v_2} Construction and Label Generation for G in PLL (after SPT_{v_1})

(c) SPT_{v_2} Construction and Label Generation for G in PLaNT (after SPT_{v_1})

Figure 1: Steps of PLL Dijkstra and PLanT Dijkstra for constructing SPT_{v_2}, along with the corresponding actions taken at each step (Red = label pruned; Green = label generated). For any vertex v_i visited, PLL confirms if SP_{v_2,v_i} is already covered by a higher ranked hub (v_1 in this case), by performing a set intersection on the previously generated labels(shown in fig1a) of v_2 and v_i. Contrarily, PLaNT only uses information intrinsic to SPT_{v_2} by tracking the most important vertex (ancestor a) in the shortest path(s) SP_{v_2,v_i}. PLaNT generates the same (non-redundant) labels as PLL, albeit at the cost of additional exploration in SPT_{v_2}.

of vertices u, v is covered by a hub vertex h from their shortest path i.e. there exists an $h \in SP_{u,v}$ such that $(h, d(u,h))$ and $(h, d(v,h))$ are in the label sets of u and v, respectively.

Canonical Hub Labeling [4]: Querying on a 2-hop labeling requires intersecting the labels of source and destination vertices. Clearly, the query response time is dependent on the average label size. However, finding the optimum labeling (with minimum label size) is NP-hard [17]. Let $R_{V \to \mathbb{N}}$ denote a total order on all vertices i.e. a *ranking function*, also known as *network hierarchy*. Rather than find the optimum labeling, Abraham et al.[4] conceptualize CHL in which *only* the highest-ranked hub on $SP_{u,v}$ is added to the labels of u and v. The CHL satisfies the cover property. Importantly, the CHL is *minimal* for a given R since removing any label from it results in a violation of the cover property. Intuitively, a good ranking function R will prioritize highly central vertices. Such vertices are good candidates for being hubs - a large number of shortest paths in the graph can be covered with a few labels. Therefore, a labeling which is minimal for a good R will be quite efficient overall.

Pruned Landmark Labeling (PLL) [8]: Distance between a hub h and other vertices in the graph can be found by constructing an SPT rooted at h. However, within SPT_h, further exploration from a vertex v is futile if $SP_{h,v}$ is covered by a higher-ranked hub. This motivates the PLL algorithm which constructs SPTs in decreasing rank order of root vertices, pruning them on-the-fly by querying on previously discovered labels. Fig.1 gives an example of SPT construction and pruning in PLL with v_2 as root. For every visited vertex v, we determine if there exists a higher-ranked hub h (v_1 in this case) in both L_{v_2} and L_v such that $d(h, v_2) + d(h, v) \leq d_v$, where d_v is the distance to v in SPT_{v_2}. In fig.1b, this condition holds for v_1 and v_5, because of which their adjacent edges are not explored and v_4 is never visited. We denote this as *distance query pruning*. Thus, pruning not only restricts label size, but also limits graph traversal resulting in an efficient labeling algorithm.

3. RELATED WORK

Hub labeling for shortest distance computation has been heavily studied in the last decade. Since its inception (credited to Cohen et al.[17]), several labelings and corresponding algorithms have been developed [17, 4, 5, 8, 10, 7, 26, 34, 3, 38]. In particular, CHL [4] is a popular labeling in which the labels cover all shortest paths in a graph (complete cover), typically allowing query responses within microseconds. With the use of network hierarchies based on betweenness [3, 4, 12] and degree [8, 34], the applicability of CHL has been extended to road networks as well as complex graphs such as social networks. Delling et al.[18] propose an optimized greedy hub labeling algorithm which produces smaller labels than CHL, but is limited to small graphs due to high complexity of the algorithm.

Abraham et al. develop a series of CHL construction algorithms primarily intended for road networks [3, 5, 2, 4, 19]. Their algorithms are based on the notion of Contraction Hierarchies built by shortcutting vertices in rank order. A vertex's hub labels are generated by merging the label sets of neighbors [4] (or from reachable vertices in a CH search [5]) in the augmented graph. However, Akiba et al. show that

this approach can be prohibitively expensive due to the cost of building (and searching) augmented graphs and merging neighbors' labels therein [8, 39]. They propose the state-of-the-art sequential algorithm PLL [8], that pushes a vertex as a hub into the labels of vertices reachable in a pruned search. Yet, even with PLL, labeling *large weighted graphs* using a single thread is often infeasible due to the high execution time [22, 45]. Moreover, the label size of a complete cover such as CHL, can be much larger than the graph size.

To enable better scalability in terms of graph size, several partial labeling approaches have been developed [10, 26]. IS-label [26] is a well-known labeling technique that builds an increasing hierarchy of independent sets and augmented graphs obtained by removing those sets. Hub labels are constructed in top-down order of the hierarchy (of augmented graphs). The total label size can be controlled by stopping the hierarchy at any desired level. PPSD computation requires querying on these labels along with traversal on the highest augmented graph in the hierarchy. Similarly, [10] also generates labels only from the dense core structure of a graph, to assist graph exploration. Querying on partial labels requires additional graph traversal and hence, their response time can be orders of magnitude larger than a complete labeling [34, 39]. Moreover, IS-label does not guarantee small label sizes and is not applicable to road networks [39]. As shown in [34], IS-label execution time can be higher than PLL or the disk-based 2-hop doubling algorithm of [34].

Parallel Hub Labeling: Parallelization offers great avenues to scale the performance and input size of an application. It is the backbone of big graph analytics in the modern era [41, 28, 48, 44, 56, 29]. Specifically, distributed-memory parallelism allows modular expansion of the DRAM capacity and the computing resources in a system, by simply adding more nodes in the cluster. Even though several parallel hub labeling approaches have been proposed [24, 45, 22, 38], these are only suitable for multi-threaded execution on a single server. This is because all of these methods require (complete) prior label information for pruning. Inter-node communication and synchronization needed for label exchange can quickly bottleneck their performance, rendering a large cluster slower than even a single node [42, 40].

Many of the existing techniques [24, 22, 45] parallelize PLL [8] using multiple threads on a single machine. Ferizovic et al.[24] construct a task queue of all vertices. Every thread then atomically pops the highest ranked vertex v still in the queue and constructs a pruned BFS tree rooted at v. Dong et al.[22] propose a hybrid scheme for weighted graphs, utilizing parallel Bellman ford for the large SPTs, and concurrent dijkstra instances for small SPTs in the later half of the execution. While this approach ensures close to minimal label size, its parallel performance is dependent on graph topology, and could be even worse than sequential PLL due to high complexity of Bellman Ford. In a very recent work, Li et al.[38] remark on the sequential dependencies of PLL and its unsuitability for parallelization. They propose the highly scalable Parallel Shortest distance Labeling (PSL) algorithm which replaces the root order label dependency of PLL with a distance label dependency. In a given round i, PSL generates all hub labels with distance i in parallel, pruning redundant labels using labels from previous rounds. PSL is very efficient on *unweighted* small-world networks which exhibit small diameter and hence, require less number of rounds. Contrarily, we target a generalized problem of labeling weighted graphs with arbitrary diameters, where these approaches are either not applicable or not effective.

paraPLL: Qiu et al.[45] propose the paraPLL framework that uses Dijkstra's algorithm to process *weighted* graphs. Similar to [24], threads in paraPLL select the highest-ranked unselected vertex and construct a pruned SPT rooted at that vertex. While this approach benefits from the dynamic rank order task scheduling, it may not respect R as parallel Dijkstra instances can finish at varying times, allowing low-ranked SPTs to generate labels that prune high-ranked SPTs. The label size can also be significantly larger than CHL if the number of threads is large.

paraPLL also provides a distributed-memory implementation (DparaPLL) that statically divides the tasks (root IDs) to multiple nodes in a circular manner. Each node then runs the multi-threaded version on the assigned vertices, periodically synchronizing with other nodes to exchange generated labels for future pruning. DparaPLL generates large amount of label traffic and requires storing *all* hub labels on every node. Hence, it does not scale well and its capability is limited by the memory of a single node instead of the cluster. Further, DparaPLL generates substantial amount of redundant labels due to (1) massive parallelism in clusters, and (2) absence of labels generated on other nodes (required for pruning), in-between the synchronization points.

[8, 38] propose label size reduction methods, some of which are specific to unweighted graphs, such as bit-parallel labeling. This study focuses on scalable parallel labeling algorithms which is a complementary approach and thus, only a friend and not a foe. Our ideas will not impact the efficacy of such techniques when used in conjunction with them.[3]

Fundamental Differences with existing work: Unlike PLL (and variants) [8, 45, 24, 22] and PSL that exhibit root order and distance label dependency, respectively, our main algorithm PLaNT eliminates all dependencies while guaranteeing the minimal CHL as the output. Thus, it is embarrassingly parallel and applicable to a variety of graphs: weighted or unweighted, complex or road networks. PLaNT achieves this by avoiding label based pruning, albeit with extra exploration overhead. This makes PLaNT a good fit for massively parallel distributed systems, where most labels are not locally accessible to the compute nodes.

4. SHARED-MEMORY LABELING

4.1 Label Construction and Cleaning

In this section, we discuss LCC - a two-step Label Construction and Cleaning (LCC) algorithm that generates the CHL for a given graph $G(V, E, W)$ and ranking R. LCC utilizes shared-memory parallelism and forms the basis for the other algorithms discussed in this paper.

We first define some labeling properties. Recall that a labeling can correctly answer any PPSD query if it satisfies the cover property.

DEFINITION 1. *A hub label $(h, d(v, h)) \in L_v$ is said to be redundant if it can be removed from L_v without violating the cover property (section 2).*

[3]For example, the Local Minimum Set Elimination (LMSE) technique [38] can be incorporated in our algorithms by simply not inserting labels for locally minimal ranked vertices. Our rank query optimization (section 4) ensures that labels for such vertices are never used for pruning. However, LMSE inherently increases final query response time.

DEFINITION 2. *A labeling L satisfies the minimality property if it has no redundant labels.*

Let R be any network hierarchy. For any pair of connected vertices u and v, let $h_m = \arg\max_{w \in SP_{u,v}}\{R(w)\}$.

DEFINITION 3. *A labeling respects R if $(h_m, d(u,h_m)) \in L_u$ and $(h_m, d(v,h_m)) \in L_v$, for all connected vertices u, v.*

LEMMA 1. *A hub label $(h, d(v,h)) \in L_v$ in a labeling that respects R is redundant if h is not the highest ranked vertex in $SP_{v,h}$.*

PROOF. WLOG, let $w = \arg\max_{u \in SP_{v,h}}\{R(u)\}$. By assumption, $w \neq h$. By definition, $w \in SP_{v,v'}$, $\forall v' \in S_h^v$, where $S_h^v = \{v' | h \in SP_{v',v}\}$. Since the labeling respects R, for any v', we must have $(w', d(v,w)) \in L_v$ and also $(w', d(v',w)) \in L_{v'}$, where $w' = \arg\max_{u \in SP_{v,v'}}\{R(u)\}$. Clearly, $R(w') \geq R(w) > R(h)$ which implies that $w' \neq h$. Thus, for every $v' \in S_h^v$, there exists a hub $w' \neq h$ that covers v and v' and $(h, d(v,h))$ can be removed without affecting the cover property. □

LEMMA 2. *Given a ranking R and a labeling that respects R, a redundant label $(h, d(v,h)) \in L_v$ can be detected by a PPSD query between the vertex v and the hub h.*

PROOF. Let $w' = \arg\max_{u \in SP_{v,h}}\{R(u)\}$. By Lemma 1, $w' \neq h$. Further, since the labeling respects R, w' must be a hub for v and h. Thus a PPSD query between v and h (with hub rank priority) will return hub w' and distance $d(v,w') + d(h,w') \leq d(v,h)$, allowing us to detect redundant label $(h, d(v,h))$ in L_v. □

Lemmas 1 and 2 show that redundant labels (if any) in a labeling can be detected if it respects R. Next, we describe our parallel LCC algorithm and show how it outputs the CHL. Note that the CHL [4] respects R and is minimal.

The main insight underlying LCC is that simultaneous construction of multiple SPTs can be viewed as an *optimistic parallelization* of sequential PLL - that allows some 'mistakes' (generate redundant labels not in CHL) in the hub labeling. However, only those mistakes shall be allowed that can be corrected to obtain the CHL. LCC addresses two major parallelization challenges:

- Label Construction → Construct in parallel, a labeling that *respects R.*
- Label Cleaning → Remove all redundant labels in parallel.

Label Construction: In addition to the distance query pruning, LCC also incorporates a crucial element – *Rank Query pruning* (algorithm 1–Line 5). Specifically, during construction of SPT_h, if a higher ranked vertex v is visited, we 1) prune SPT_h at v and 2) do not insert h as a hub into L_v *even if the corresponding distance query is unable to prune.* Since LCC constructs multiple SPTs in parallel, it is possible that the SPT of a higher ranked vertex which should be a hub for h (for example v above) is still under construction and thus the hub list of h is incomplete. Step (2) above guarantees that for any pair of connected vertices (h, v) with $R(v) > R(h)$, either v is labeled a hub of h or they both share a higher ranked hub. This fact will be crucial in proving the minimal covering property of LCC after its *label cleaning* phase. Note that h might get unnecessarily inserted as a hub for some other lower-ranked vertex u even if $SP_{h,u}$ is covered by v. However, as we will show subsequently, such 'optimistic' labels can be cleaned (deleted).

The parallel label construction step in LCC is shown in algorithm 2. Similar to [45, 24], each concurrent thread selects a unique vertex in rank order R (by atomic updates to a global counter), and constructs the corresponding SPT using pruned Dijkstra. This parallelization strategy exhibits good load balance as all threads are working until the very last SPT and there is no global synchronization barrier where threads may stall. However, unlike previous works, LCC's pruned Dijkstra is also enabled with Rank Queries.

Algorithm 1 Pruned Dijkstra in LCC (`pruneDijRQ`)

Input: $G(V,E,W)$, R, root h, current labels $L = \cup_{v \in V} L_v$; **Output:** hub labels with hub h
$\delta_v \rightarrow$ distance to v, $Q \rightarrow$ priority queue
1: $LR = \texttt{hash}(L_h)$, $\delta_h = 0, \delta_v = \infty\ \forall v \in V \setminus \{h\}$ ▷ initialize
2: add $(h, 0)$ to Q
3: **while** Q is not empty **do**
4: pop (v, δ_v) from Q
5: **if** $R(v) > R(h)$ **then** continue ▷ Rank-Query
6: **if** $\texttt{DQ}(v, h, \delta_v, LR, L_v)$ **then** continue ▷ Dist. Query
7: $L_v = L_v \cup \{(h, \delta_v)\}$
8: **for each** $u \in N_v$
9: **if** $\delta_v + w_{v,u} < \delta_u$ **then**
10: $\delta_u = \delta_v + w_{v,u}$; update Q
11: **function** $\texttt{DQ}(v, h, \delta, LR, L_v)$
12: **for each** $(h', d(v,h')) \in L_v$
13: **if** $(h', d(h,h')) \in LR$ **then**
14: **if** $d(v,h') + d(h,h') \leq \delta$ **then** return `true`
15: return `false`

Algorithm 2 LCC: Label Construction and Cleaning

Input: $G(V,E,W)$, R; **Output:** $L = \cup_{v \in V} L_v$
$p \rightarrow \#$ parallel threads, $t_c \rightarrow$ tree count
$Q \rightarrow$ queue containing vertices ordered by rank
1: $L_v = \phi\ \forall\ v \in V$ ▷ initialization
2: **for** $t_{id} = 1, 2...p$ **do in parallel** ▷ LCC-I: Label Construction
3: **while** $Q \neq$ empty **do**
4: atomically pop highest ranked vertex h from Q
5: $\texttt{pruneDijRQ}(G, R, h, L)$
6: **for** $v \in V$ **do in parallel**
7: sort labels in L_v using hub rank
8: **for** $v \in V$ **do in parallel** ▷ LCC-II: Label Cleaning
9: **for each** $(h, \delta_{v,h}) \in L_v$
10: **if** $\texttt{DQ\_Clean}(v, h, \delta_{v,h}, L_h, L_v, R)$ **then**
11: delete $(h, \delta_{v,h})$ from L_v
12: **function** $\texttt{DQ\_Clean}(v, h, \delta, L_h, L_v, R)$ ▷ Cleaning Query
13: compute the set W of common hubs in L_h and L_v such that $d(w,v) + d(w,h) \leq \delta\ \forall\ w \in W$
14: find the highest ranked vertex u in W
15: **if** $(W = empty)$ or $R(u) \leq R(h)$ **then** return `false`
16: **else** return `true`

CLAIM 1. *The labeling generated by LCC's label construction step (LCC-I) satisfies the cover property and respects R.*

PROOF. Let H_v^P (H_v^S, respectively) denote the set of hub vertices of a vertex v after LCC-I (sequential PLL, respectively). We will show that $H_v^S \subseteq H_v^P$. Suppose $h \notin H_v^P$ for some vertex h. Consider three cases:
Case 1: $h \notin H_v^P$ because a Rank-Query pruned SPT_h at v in LCC-I. Thus we must have $R(v) > R(h)$. Since sequential PLL is also the CHL, $h \notin H_v^S$ also.
Case 2: $h \notin H_v^P$ because a Distance-Query pruned SPT_h at v in LCC-I. This can only happen if LCC found a shorter distance $d(h, v)$ through a hub vertex $h' \in SP_{h,v}$ (alg. 1 : lines 13-14). Since LCC with Rank-Querying identified h' as a hub for both h and v, we must have $R(h') > R(h) > R(v)$ and thus $h \notin H_v^S$.
Case 3: $h \notin H_v^P$ because v was not discovered by SPT_h due to pruning. Similar to Case 2 above, this implies $\exists h' \in SP_{v,h}$ with $R(h') > R(h)$ and therefore $h \notin H_v^S$.
Combining these cases, we can say that $H_v^S \subseteq H_v^P$. Since sequential PLL generates the CHL, the claim follows. □

Label Cleaning: The extra labels created due to concurrent construction of multiple SPTs in LCC-I are redundant, since there exists a canonical subset of H_V^P (i.e. H_V^S) satisfying the cover property. LCC eliminates redundant labels using the `DQ_Clean` function (alg 2, lines 12-16)[4] - for a vertex v, a label $(h, d(v, h))$ is redundant if a distance query on (v, h) finds a common hub u such that $R(u) > R(h)$ and the distance between v and h via u, is not greater than $d(v, h)$.

CLAIM 2. *The final labeling generated by LCC after the Label Cleaning step (LCC-II) is the CHL.*

PROOF. Claim 1 implies that the labeling after LCC-I respects R and satisfies cover property. Lemma 2 implies that LCC-II removes all redundant labels, resulting in a minimal labeling which by definition, is the CHL. □

LEMMA 3. *LCC is work-efficient. It performs $\mathcal{O}(wm \log^2 n + w^2 n \log^2 n)$ work, generates $O(wn \log n)$ hub labels and answers each query in $\mathcal{O}(w \log n)$ time, where w is the tree-width of G.*

PROOF. Consider the centroid decomposition $(\chi, T(V_T, E_T))$ of minimum-width tree decomposition of the input graph G, where $\chi = \{X_t \subseteq V \; \forall \; t \in V_T\}$ maps vertices in T (bags) to subset of vertices in G [8]. Let $R(v)$ be determined by the minimum depth bag $\{b_v \in V_T \mid v \in X_{b_v}\}$ i.e. vertices in root bag are ranked highest followed by vertices in children of root and so on. Since we prune using *Rank-Query*, SPT_v will never visit vertices beyond the parent of b_v. A bag is mapped to at most w vertices and the depth of T is $\mathcal{O}(\log n)$. Since the only labels inserted at a vertex are its ancestors in the centroid tree, there are $\mathcal{O}(w \log n)$ labels per vertex.

Each time a label is inserted at a vertex, we evaluate all its neighbors in the distance queue. Thus the total number of distance queue operations is $\mathcal{O}(wm \log n)$. Further, distance queries are performed on vertices that cannot be pruned by rank queries. This results in $\mathcal{O}(n \cdot w \log n \cdot w \log n) = \mathcal{O}(w^2 n \log^2 n)$ work.

Label Cleaning step sorts the label sets and executes PPSD queries performing $\mathcal{O}(nw \log n \log w \log \log n + w^2 n \log^2 n) = \mathcal{O}(w^2 n \log^2 n)$ work. Thus, overall work complexity of LCC is $\mathcal{O}(wm \log^2 n + w^2 n \log^2 n)$ which is the same as the sequential algorithm [45], making LCC work-efficient. □

[4]Actual implementation of `DQ_Clean` stops at the first common hub (also the highest ranked) in sorted L_h and L_v which satisfies the condition in line 13 of alg. 2.

Note that labeling generated by paraPLL[45] may not respect R and hence, Label Cleaning after paraPLL may result in a labeling that violates the cover property.

Although LCC is theoretically efficient, in practice, the Label cleaning step adds non-trivial overhead to the execution time. In the next subsection, we describe an algorithm that drastically reduces the overhead of cleaning.

4.2 Global Local Labeling (GLL)

The main goal of GLL is to severely restrict the size of label sets used for label cleaning queries. A natural way to accelerate label cleaning is by avoiding futile computations (in `DQ_Clean`) over hub labels that were already consulted during label construction. However, to achieve notable speedup, these pre-consulted labels must be skipped in constant time without actually iterating over all of them.

GLL overcomes this challenge by using a novel Global Local Label Table data structure and interleaved cleaning strategy. Unlike LCC, GLL utilizes multiple synchronizations where the threads switch between label construction and cleaning. We denote the combination of a label construction and the subsequent label cleaning step as a *superstep*. During label construction, the newly generated labels are pushed to a *Local Label Table* and their volume is tracked. Once the number of labels in the local table becomes greater than αn, where $\alpha > 1$ is the synchronization threshold, the threads synchronize, sort, clean the labels in local table and commit them to the *Global Label Table*.

In any superstep, it is known that *all* labels in the global table were consulted during the label construction. Therefore, label cleaning only needs to query on the local table for redundancy check, thus dramatically reducing the number of repeated computations. After a label construction step, the local table holds αn labels. Assuming $\mathcal{O}(\alpha)$ average labels per vertex (we empirically observe that labels are almost uniformly distributed except few highest ranked vertices), each cleaning step should perform on average $\mathcal{O}(n\alpha^2)$ work. The number of cleaning steps is $\mathcal{O}\left(\frac{wn \log n}{\alpha n}\right)$ and thus we expect the total complexity of cleaning in GLL to be $\mathcal{O}(n\alpha w \log n)$ as opposed to $\mathcal{O}(nw^2 \log^2 n)$ in LCC. If $\alpha \ll w \log n$, cleaning in GLL is more efficient than LCC.

GLL also reduces locking during distance queries. LCC locks label sets before reading because label sets are dynamic arrays that can undergo memory (de)allocation when a label is appended. Contrarily, in GLL, most pruning distance queries are answered by label sets in the global table whereas labels are only appended to the local table.

5. DISTRIBUTED-MEMORY LABELING

Distributed-memory systems present strikingly different challenges than shared-memory systems, in general as well as in the specific context of hub labeling. Therefore, a trivial extension of GLL algorithm is unsuitable for a multi-node cluster. Particularly, the labels generated on a node are not readily available to other nodes until they synchronize and exchange labels. Further, unlike paraPLL, our aim is to harness not just the compute but also the collective memory capability of multiple nodes to construct CHL for large

graphs. This mandates that labels be partitioned and distributed across multiple nodes at all times, and severely limits the knowledge of labels created on other nodes even after synchronization. This absence of labels dramatically affects the pruning efficiency during label construction, resulting in large number of redundant labels and consequently, huge communication volume that bottlenecks the performance.

In this section, we present novel algorithms and optimizations that systematically conquer these challenges. We begin by discussing a distributed extension of GLL that highlights the basic data distribution and parallelization approach.

5.1 Distributed GLL (DGLL)

DGLL divides the task queue for SPT creation uniformly among q nodes in a rank circular manner. The set of root vertices assigned to node i is $TQ_i = \{v \mid R(v) \bmod q = i\}$. Every node loads the graph (and ranking R for rank queries) and executes GLL on its allotted task queue. DGLL has two key optimizations tailored for distributed implementation:

1. Label Set Partitioning: In DGLL, nodes only store labels generated *locally* i.e. all labels at node i are of the form $(h, d(v,h))$, where $h \in TQ_i$. Equivalently, the labels of a vertex v are disjoint and distributed across nodes i.e. $L_v = \cup_i L_{i,v}$. Thus, all nodes collaborate to provide main memory space to store the labels and the *effective memory scales in proportion to the number of nodes*. This is in stark contrast with paraPLL that stores $\{\cup_{v \in V} L_v\}$ on every node, rendering effective memory same as that of a single node.

2. Synchronization and Label Cleaning: For every superstep in DGLL, we decide the synchronization point apriori in terms of the number of SPTs to be created. The synchronization point computation is motivated by the label generation behavior of the algorithm. Fig.2 shows that the number of labels generated by initial SPTs rooted at high-ranked vertices is very large and it drops exponentially as the rank decreases. To maintain cleaning efficiency with few synchronizations, we increase the number of SPTs constructed in each superstep by a multiplicative factor of β i.e. if superstep i constructs x SPTs, superstep $i+1$ will construct $\beta \cdot x$ SPTs. Contrarily, distributed paraPLL [45] constructs the same number of trees in every superstep.

After synchronization, all labels generated in the superstep are broadcasted to all nodes for redundancy check. Each node creates a bitvector containing response of all cleaning queries. The bitvectors are then combined using an all reduce operation to obtain final redundancy information.

Note that DGLL uses both global and local tables to answer cleaning queries. Yet, interleaved cleaning is beneficial as it removes redundant labels, thereby reducing query response time for subsequent cleaning steps. For some datasets, we empirically observe $> 90\%$ redundancy in labels generated in some supersteps. Presence of such large number of redundant labels can radically slow down future queries.

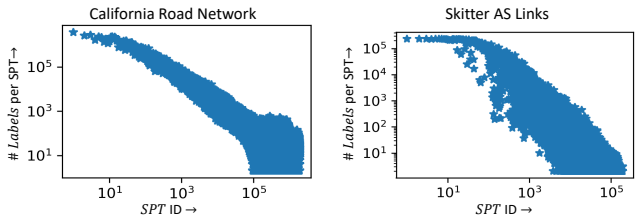


Figure 2: Labels generated by SPTs. ID of SPT_v is $n - R(v)$.

5.2 Prune Labels and (do) Not (prune) Trees (PLaNT)

The redundancy check in DGLL can severely restrict scalability of the algorithm due to huge label broadcast traffic (redundant + non-redundant labels), motivating the need for an algorithm that can avoid redundancy without communicating labels with other nodes.

To this purpose, we propose the Prune Labels and (do) Not (prune) Trees (PLaNT) algorithm that accepts some loss in pruning efficiency to achieve a dramatic decrease in communication across cluster nodes, by outputting completely non-redundant labels without additional label cleaning. We note that the redundancy of a label $(h, d(v,h)) \in L_v$ is only determined by whether or not h is the highest ranked vertex in $SP_{v,h}$. This is the key idea behind PLaNT: When constructing SPT_h, if, when resolving distance queries, embedded information about high-ranked vertices on paths can be retrieved, SPT_h will *intrinsically* have the requisite information to detect redundancy of h as a hub.

Algorithm 3 (PLaNTDijkstra) depicts the construction of a shortest path tree using PLaNT, which we call PLaNTing trees. Instead of pruning using distance or rank queries, PLaNTDijkstra tracks the *most important ancestor* $a[v]$ encountered on the path from h to v by allowing ancestor values to propagate along with distance values. When v is popped from the distance queue, a label is added to L_v if neither v nor $a[v]$ are ranked higher than the root. Thus, for any shortest path $SP_{h,v}$, only $h_m = \text{argmax}_{w \in SP_{h,v}} \{R(w)\}$ succeeds in adding itself to the labels of u and v, *guaranteeing minimality of the labeling while simultaneously respecting R and satisfying cover property.* Fig.1c provides a detailed graphical illustration of label generation using PLaNT and shows that it generates the same labeling (CHL) as the PLL.

If there are multiple shortest paths from h to v, the path with the highest-ranked ancestor is selected. This is achieved in the following manner: when a vertex v is popped from the dijkstra queue and its edges are relaxed, the ancestor of a neighbor $u \in N_v$ is allowed to update even if the newly calculated tentative distance to u is *equal* to the currently assigned distance to u (line 12 of algorithm 3). For example, in fig.1c, the shortest paths to v_5, $P_1 = \{v_2, v_1, v_4, v_5\}$ and $P_2 = \{v_2, v_3, v_5\}$ have the same length and P_1 is selected by setting $a[v_5] = v_1$ because $R(v_1) > R(v_2)$.

Note that PLaNT not only avoids dependency on the labels on remote nodes, it rids SPT construction of *any dependency* on the output of other SPTs, effectively providing an **embarassingly parallel** solution for CHL construction with $\mathcal{O}(m + n \log n)$ depth (complexity of a single instance of dijkstra) and $\mathcal{O}(mn + n^2 \log n)$ work. Due to its embarassingly parallel nature, PLaNT does not require SPTs to be constructed in a specific order. However, to enable optimizations discussed later, we follow the same rank determined order in PLaNT as used in DGLL (section 5.1).

Early Termination: To improve the computational efficiency of PLaNT and prevent exploring the full graph for every SPT, we propose the following *early termination* strategy: stop further exploration when the rank of either the ancestor or the vertex itself is higher than the root, for *all* vertices in dijkstra's queue[5]. Early termination can dramatically cut down traversal in SPTs with low-ranked roots.

[5]Further exploration from such vertices only creates shortest paths with at least one vertex ranked higher than the root.

Algorithm 3 `PLaNTDijkstra` algorithm to plant SPTs

Input: $G(V, E, W)$, R, root h
$\delta_v \rightarrow$ distance to v, $a[] \rightarrow$ ancestor array, $Q \rightarrow$ priority queue, $cnt \rightarrow$ number of vertices v with $a[v] = h$

$\delta_h = 0, a[h] = h$ and $a[v] = v, \delta_v = \infty \; \forall \; v \in V \setminus h$
add h to Q; $cnt = 1$
while Q is not empty **do**
  if $cnt = 0$ **then** exit ▷ Early Termination
  pop (v, δ_v) from Q; compute $nA = \underset{x \in \{v, a[v]\}}{\text{argmax}} \; R(x)$
  if $a[v] = h$ **then** $cnt = cnt - 1$
  if $R[nA] > R[h]$ **then** continue
  $L_v = L_v \cup \{(h, \delta_v)\}$
  for each $u \in N_v$
    $pA = a[u]$
    if $\delta_v + w_{v,u} < \delta_u$ **then** $a[u] = \underset{x \in \{nA, u\}}{\text{argmax}} \; R(x)$
    else if $\delta_v + w_{v,u} = \delta_u$ **then** $a[u] = \underset{x \in \{nA, pA\}}{\text{argmax}} \; R(x)$
    if $a[u] = h$ and $pA \neq h$ **then** $cnt = cnt + 1$
    else if $a[u] \neq h$ and $pA = h$ **then** $cnt = cnt - 1$
    $\delta_u = \min(\delta_u, \delta_v + w_{v,u})$; update Q

Despite early termination, PLaNTed trees can possibly explore a large part of the graph which PLL would have avoided by pruning. For each SPT, let Ψ denote the average # vertices explored per label generated. Fig.3 shows that in PLaNT, Ψ for many SPTs can be even higher than 10000.

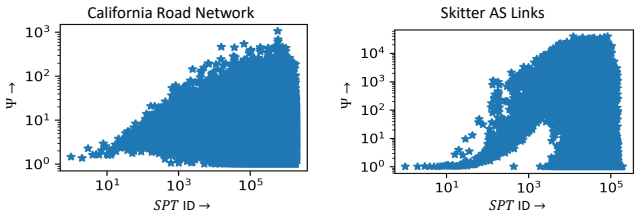


Figure 3: Ψ for SPTs in PLaNT.

5.2.1 Hybrid PLaNT + DGLL

Apart from its embarrassingly parallel nature, an important virtue of PLaNT is its *compatibility* with DGLL. Since PLaNT also constructs SPTs in rank order and generates labels with root as the hub, we can seamlessly transition between PLaNT and DGLL to enjoy the best of both worlds. We propose a *Hybrid* algorithm that initially uses PLaNT and later, switches to DGLL. The initial SPTs rooted at high ranked vertices generate most of the labels in CHL (fig.2) and exhibit low Ψ value (fig.3).

By PLaNTing these SPTs, we (1) efficiently parallelize bulk of the computation and avoid communicating a large fraction of the overall labeling at at the cost of little extra exploration in the trees, and (2) avoid a large number of distance queries that PLL or DGLL would have done on all the visited vertices in these SPTs. In the later half of execution, when Ψ becomes high and few labels are generated per SPT, the Hybrid algorithm uses DGLL to exploit the heavy pruning and avoid the inefficiencies associated with PLaNT.

The Hybrid algorithm is a natural fit for scale-free networks. These graphs have a large tree-width w but they exhibit a *core-fringe* structure where removal of a small dense core leaves a fringe like structure with very low tree-width [51, 10]. Typical degree and betweenness based hierarchies also prioritize vertices in the dense core. In such graphs, the Hybrid algorithm PLaNTs SPTs rooted at core vertices which generate a large number of labels. SPTs rooted on fringe vertices generate few labels and are constructed using DGLL which exploits heavy pruning to limit computation.

For graphs with a *core-fringe* structure, a relaxed tree decomposition $(\chi, T(V_T, E_T))$ parameterized by an integer c can be computed such that $|X_{t_r}| = w_m \wedge |X_t| \leq c \; \forall \; t \in V_T \setminus t_r$, where t_r is the root of T and $\chi = \{X_t \subseteq V \; \forall \; t \in V_T\}$ maps vertices in T (bags) to subset of vertices in G [10]. In other words, except root bag, $|X_t|$ is bounded by a parameter $\{c | c \ll w \leq w_m\}$.

LEMMA 4. *The hybrid algorithm performs* $\mathcal{O}(m \cdot (w_m + c \log^2 n) + nc \log n \cdot (w_m + c \log n))$ *work, broadcasts only* $\mathcal{O}(cn \log n)$ *data, generates* $\mathcal{O}(n \cdot (w_m + c \log n))$ *hub labels and answers each query in* $\mathcal{O}(w_m + c \log n)$ *time.*

PROOF. Consider the relaxed tree decomposition $(\chi, T(V_T, E_T))$ with root t_r and perform centroid decomposition on all subtrees rooted at the children of t_r to obtain tree T'. The height of any tree in the forest generated by removing t_r from T' is $\mathcal{O}(\log n)$. Hence, the height of $T' = \mathcal{O}(\log n + 1) = \mathcal{O}(\log n)$.

Consider a ranking R where $R(v)$ is determined by the minimum depth bag $\{b \in V_{T'} | v \in X_b\}$. For GLL, the number of labels generated by SPTs from vertices in root bag is $O(w_m n)$. Combining this with lemma 3, we can say that total labels generated by GLL is $\mathcal{O}(n \cdot (w_m + c \log n))$ and query complexity is $\mathcal{O}(w_m + c \log n)$. The same also holds for the Hybrid algorithm since it outputs the same CHL as GLL.

If Hybrid algorithm constructs w_m SPTs using PLaNT and rest using DGLL, the overall work-complexity is $\mathcal{O}(w_m \cdot (m + n \log n)) + \mathcal{O}(mc \log^2 n + nc \log n \cdot (w_m + c \log n)) = \mathcal{O}((m \cdot (w_m + c \log^2 n) + nc \log n \cdot (w_m + c \log n)))$.

The Hybrid algorithm only communicates the labels generated after switching to DGLL, resulting in $\mathcal{O}(cn \log n)$ data broadcast. In comparison, doing only DGLL for the same ordering will broadcast $\mathcal{O}(w_m n + cn \log n)$ data. □

In reality, we use the ratio Ψ as a heuristic, dynamically switching from PLaNT to DGLL when Ψ becomes greater than a threshold Ψ_{th}.

LEMMA 5. *The Hybrid algorithm consumes* $\mathcal{O}\left(n \cdot (w_m + c \log n)/q + n + m\right)$ *main memory per node, where* q *is the number of nodes used.*

PROOF. Storing labels requires $\mathcal{O}\left(n \cdot (w_m + c \log n)/q\right)$ space per node and graph requires $O(n + m)$ space. □

5.3 Enabling efficient multi-node pruning

We propose an optimization that *simultaneously* solves the following two problems:

1. Pruning traversal in PLaNT → The reason why PLaNT cannot prune using rank or distance queries is that if we prune using partial labels, an SPT can still visit those vertices which would be pruned if *all* prior labels were available and possibly, through non shortest paths with the wrong ancestor information. This can lead to redundant label generation and defeat the purpose of PLaNT. In general, if a node prunes using H_u, it must have $\{H_v \forall v \in V | R(v) \geq R(u)\}$ to guarantee non-redundant labels. In this situation, a vertex is either visited through the shortest path with correct ancestor or is pruned. We skip the proof details for brevity.

2. Redundant labels in DGLL → Fig.4 shows the label count generated by PLL if pruning queries are restricted to use hub labels from few top-ranked hubs only. We observe that label count decreases dramatically even if pruning utilizes only few highest-ranked hubs.

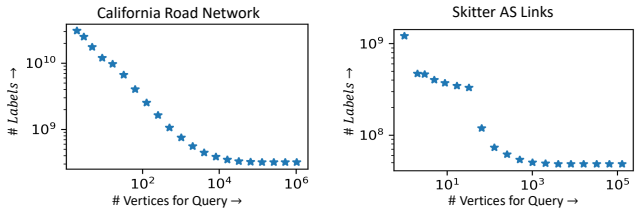


Figure 4: # Labels generated if pruning queries in PLL use few (x-axis) highest rank hubs. X-axis= 0 means rank queries only. When pruning is completely absent, # labels = $|V|^2$

Thus, for a given integer η, if we store *all* labels from η most important hubs on every compute node i.e. $HC = \cup_{v \in V | R(v) \geq n-\eta}\{H_v\}$, we can

- use distance queries on HC to prune PLaNTed trees, and
- drastically increase pruning efficiency of DGLL.

To this purpose, we allocate a *Common Label table* on every node that stores common labels HC. These labels are broadcasted even if they are generated by PLaNT. For $\eta = \mathcal{O}(1)$, using common labels incurs additional $\mathcal{O}(n)$ broadcast traffic, $\mathcal{O}(w_m n)$ queries of $\mathcal{O}(1)$ complexity each , and consumes $O(n)$ more memory per node. Thus, it does not alter the theoretical bounds on work-complexity, communication volume and space requirements of the Hybrid algorithm. In our implementation, we store labels from $\eta = 16$ highest ranked hubs in the Common Label Table.

5.4 Extensions

Our algorithms are not restricted to clusters and can be used for any massively parallel system, such as GPU. On GPUs, simply parallelizing PLL can blow up the label size, as thousands of concurrent threads when working on their very first tree, will not have any label information for pruning. This can make Label Cleaning infeasible or even make the system run out of memory. Instead, we can use PLaNT to construct first few SPTs for every thread and switch to GLL afterwards. Our approach can also be extended to disk-based processing where access cost to labels is very high. The Common Label Table can be mapped to faster memory in the hierarchy (DRAM) to accelerate labeling.

6. QUERYING

We provide three modes to the user for distance queries:

- *Querying with Labels on Single Node (QLSN)* → All labels are stored on every node and a query response is computed only by the node where the query emerges. Existing hub labeling frameworks [45, 8, 22, 24] only support this mode.
- *Querying with Fully Distributed Labels (QFDL)* → The label set of every vertex is partitioned between all nodes. Queries are broadcasted to all nodes and individual responses of the nodes are reduced using MPI_MIN to obtain the shortest distance. It utilizes parallelism of multiple nodes and consumes only $\mathcal{O}\left(n \cdot (w_m + c \log n)/q\right)$ memory per node, but incurs high communication costs.
- *Querying with Distributed Overlapping Labels (QDOL)* → In this mode, we divide the vertex set V into ζ partitions. For every possible partition pair, a node is assigned to store entire label set of vertices in that pair. Thus, a given query is answered only by a single node but not by every node. Unlike QFDL, this mode utilizes the more efficient P2P communication instead of broadcasting. Each query (u, v) is mapped to the node that has labels for vertex partitions containing u and v and then communicated to this node which single-handedly computes and sends back the response. In QDOL, multi-node parallelism is exploited in a batch of queries where different nodes concurrently compute responses to the respective queries mapped to them. For a cluster of q nodes, ζ can be computed as follows:

$$\binom{\zeta}{2} = q \implies \zeta = \frac{1 + \sqrt{1 + 8q}}{2}$$

Storing labels of two vertex partitions consumes $2n \cdot (w_m + c \log n)/\zeta = \mathcal{O}\left(n \cdot (w_m + c \log n)\sqrt{q}\right)$ memory per node (much larger than QFDL).

7. EXPERIMENTS

7.1 Setup

We conduct shared-memory experiments on a 36 core, 2-way hyperthreaded, dual-socket linux server with two Intel Xeon E5-2695 v4 processors@ 2.1GHz and 1TB DRAM; all 72 threads are used for labeling. For distributed-memory experiments, we use a 64-node cluster. Each node has an 8 core, 2-way hyperthreaded, Intel Xeon E5-2665@ 2.4GHz processor and 64GB DRAM; all 16 threads on each node are used for labeling. Programs are compiled using G++ 9.1.0 with the highest optimization -O3 flag. We use OpenMP v4.5 for multithreading within a node and OpenMPI v3.1.2 for parallelization across multiple nodes.

Baselines: We use sequential PLL (seqPLL), paraPLL shared-memory (SparaPLL) and distributed-memory (DparaPLL) versions [45] for labeling time comparison. As given in [45], we use[6] dynamic task assignment policy in SparaPLL and static circular task division among multiple nodes in DparaPLL. We also report the performance of DGLL for effective comparison. Both DGLL and DparaPLL synchronize $\log_8 n$ times to exchange labels.

Implementation Details: We vary the synchronization threshold α in GLL and switching threshold Ψ_{th} in the Hybrid algorithm to empirically assess their impact on the performance. Figure 5 shows the impact of α on GLL. Execution time of GLL is robust to significant variations in α within a range of 2 to 32. Intuitively, a small value of α reduces cleaning time (section 4.2) but making it too small can lead to frequent synchronizations that hurt parallel performance. Based on our observations, we set $\alpha = 4$.

Figure 6 shows the effect of Ψ_{th} on the hybrid algorithm. Intuitively, keeping Ψ_{th} too large increases the computation overhead (seen in scale-free networks) because even low-ranked SPTs that generate few labels, are PLaNTed. On the other hand, keeping Ψ_{th} too small results in poor scalability (seen in road networks) as the algorithm switches to DGLL quickly and communication avoidance of PLaNT remain underutilized. Based on these findings, we set $\Psi_{th} = 100$ for scale-free networks and $\Psi_{th} = 500$ for road networks.

[6]paraPLL code is not publicly available.

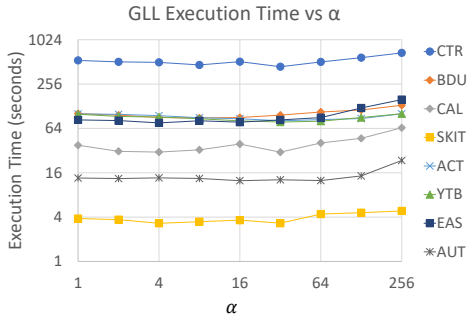


Figure 5: GLL execution time vs synchronization threhsold α

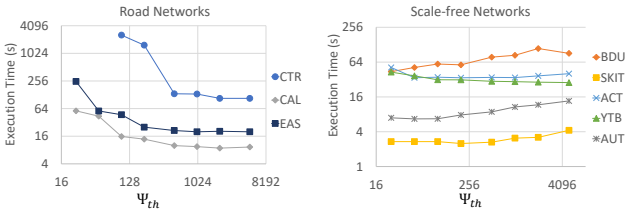


Figure 6: Execution time of Hybrid algorithm on 16 compute nodes vs switching threshold Ψ_{th}

7.1.1 Datasets

We evaluate our algorithms on 12 real-world graphs with varied topologies (high-dimensional (complex) graphs vs low-dimensional road networks, uniform-degree vs power-law social networks), as listed in table 2[7]. The ranking R is determined by degree for scale-free networks [8] and betweenness for road networks [39][8]. As opposed to complex graphs, road networks typically exhibit smaller label size and efficient early termination in PLaNT due to the high betweenness of highways that cover most of the shortest paths.

Table 2: Datasets for Evaluation

Dataset	n	m	Description	Type
CAL[20]	1,890,815	4,657,742	California Road network	Undirected
EAS[20]	3,598,623	8,778,114	East USA Road network	Undirected
CTR[20]	14,081,816	34,292,496	Center USA Road network	Undirected
USA[20]	23,947,347	58,333,344	Full USA Road network	Undirected
SKIT[1]	192,244	636,643	Skitter Autonomous Systems	Undirected
WND[11]	325,729	1,497,134	Univ. Notre Dame webpages	Directed
AUT[27]	227,320	814,134	Citeseer Collaboration	Undirected
YTB[36]	1,134,890	2,987,624	Youtube Social network	Undirected
ACT[36]	382,219	33,115,812	Actor Collaboration Network	Undirected
BDU[36]	2,141,300	17,794,839	Baidu HyperLink Network	Directed
POK[36]	1,632,803	30,622,564	Social network Pokec	Directed
LIJ[36]	4,847,571	68,993,773	LiveJournal Social network	Directed

7.2 Shared-memory Algorithms

Table 3 compares the performance of GLL, LCC, SparaPLL and seqPLL. It also shows the Average Label Size per vertex (ALS) for CHL (GLL, LCC and seqPLL) and the labeling generated by SparaPLL. Query response time is directly proportional to ALS and hence, it is a crucial metric for evaluating any hub labeling algorithm[9].

[7]Scale-free networks did not have edge weights from the download sources. For each edge, we choose an integral weight value between $[1, \sqrt{n})$ uniformly at random.
[8]Most vertices in road networks have a small degree and their importance cannot be identified by their degree.
[9]For LIJ, the ALS of CHL was obtained from distributed algorithms since none of the shared-memory algorithms finished execution.

Table 3: Performance comparison of GLL, LCC and baselines. Time=∞ implies that execution did not finish in 4 hours.

	SparaPLL		CHL	seqPLL	LCC Time(s)		GLL
Dataset	ALS	Time(s)	ALS	Time(s)	LCC-I	Total	Time(s)
CAL	108.3	51.2	83.4	215	26	41.4	35.4
EAS	138.1	116.3	116.8	680.6	73.8	108.7	88
CTR	178.7	424.2	160.9	5045	415	664.1	567.7
USA	185.6	816.9	166.1	∞	715	1149	834
SKIT	88.3	2.5	85.1	95.8	3	4.85	3.9
WND	39.6	2.4	23.5	21.98	1.9	2.94	2.1
AUT	240.2	10.4	229.6	670	10	18.4	14.6
YTB	208.9	69.6	207.5	2693	73.8	126.7	104.6
ACT	376.1	112.4	366.3	2173	114.9	151.3	141.9
BDU	100.1	103.1	90.7	4736	108	133.9	99.9
POK	2243	4159	2231	∞	4213	8748	3917
LIJ	–	∞	1223	∞	∞	∞	∞

To understand the individual benefits of parallelism and rank queries, we compare the execution times of seqPLL, SparaPLL (which is essentially parallel PLL without rank queries) and Label Construction in LCC (LCC-I). The advantage of pure parallelism is evident in the substantial speedup achieved by SparaPLL over seqPLL. However, ALS of SparaPLL is noticeably larger than CHL (see CAL, EAS, WND). Rank querying improves pruning during parallel execution as can be seen by comparing the execution times of LCC-I and SparaPLL for these datasets. Note that the (potential) performance improvement is a side-effect of rank queries. The primary motivation for rank querying was to ensure that the labeling respects R so that LCC and GLL can clean redundant labels.

We observe that on average, GLL generates 17% less labels than SparaPLL. Its execution time is comparable to SparaPLL, even though it re-checks every label using the cleaning queries. For some graphs such as CAL, GLL is even 1.3× faster than SparaPLL. This is primarily because rank queries and interleaved cleaning limit graph exploration overhead and label size, resulting in faster distance-pruning queries. GLL also reduces label table locking (section 4.2).

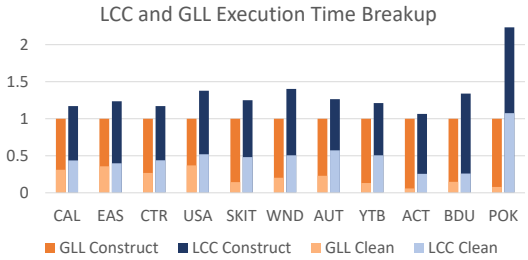


Figure 7: Time taken for label construction and cleaning in LCC and GLL, normalized by the total execution time of GLL.

Fig. 7 shows execution time breakup for LCC and GLL. GLL cleaning is significantly faster than LCC because of the reduced cleaning complexity (section 4.2). Overall, GLL is 1.3× faster than LCC on average. However, for some graphs such as CAL, fraction of cleaning time is $> 30\%$ even for GLL. This is because in the first superstep of GLL, more than αn labels get generated as there are no labels for distance query pruning and at least p ($p > \alpha$ is # threads) SPTs are simultaneously constructed.

7.3 Distributed-memory Algorithms

To assess the scalability of distributed hub labeling algorithms, we vary q from 1 to 64 (# compute cores from 8 to

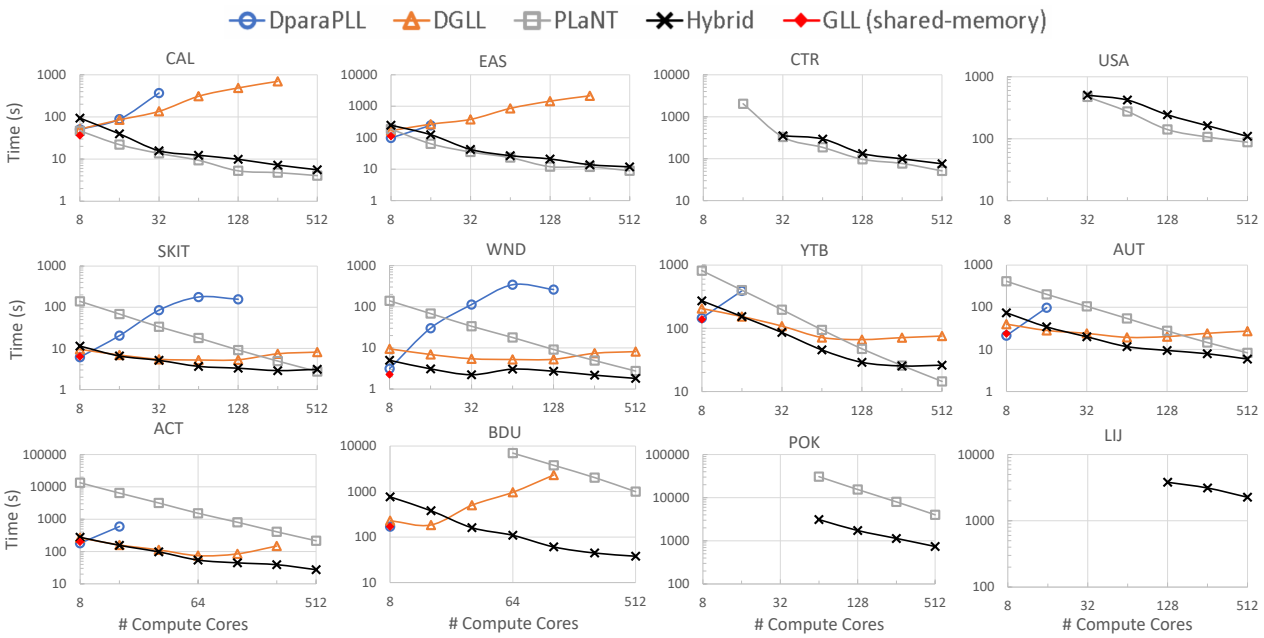


Figure 8: Strong scaling results of DparaPLL, DGLL, PLaNT and Hybrid algorithms. Missing curves or points mean that the algorithm incurred OOM error or did not finish within 4 hours. Also, # compute cores = 8∗(# compute nodes). For effective comparison, we also show single node execution time for shared-memory parallel GLL algorithm.

512). Fig. 8 shows the strong scaling of different algorithms in terms of labeling construction time.

We note that both DparaPLL and DGLL do not scale well with q. DparaPLL often runs out-of-memory when q is large. This is because in the first superstep itself, a large number of hub labels are generated that when exchanged, overwhelm the memory of the nodes. DGLL, on the other hand, limits the amount of labels per superstep by synchronizing relatively frequently in the early stage of execution. Also, it does not store all the labels on every node and hence, requires less main memory than DparaPLL.

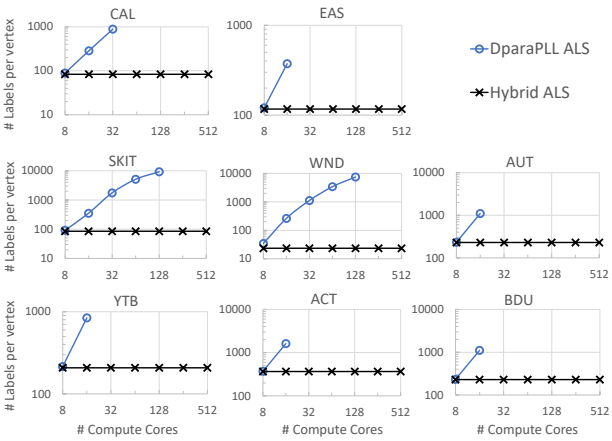


Figure 9: Label size of DparaPLL and Hybrid algorithms.

Moreover, label size of DparaPLL explodes as q increases (fig.9), partly due to the absence of rank queries. Contrarily, our algorithms (DGLL, PLaNT and Hybrid) generate the same labeling - CHL and have the same ALS, irrespective of q. Efficacy of distance query based pruning in DparaPLL suffers because on every node, labels from several high-ranked hubs are missing in between the synchronization points. Increase in label size further slows down the distance querying and the labeling process. On the other hand, *rank queries* in DGLL allow pruning even at those hubs whose SPTs were not created on the querying node. Further, it periodically cleans redundant labels to retain the performance of distance queries. Yet, DGLL incurs significant communication and does not scale well. Neither DparaPLL nor DGLL can process the large CTR, USA, POK and LIJ datasets, either running out-of-memory or time limit.

PLaNT on the other hand, paints a completely different picture. Owing to its embarrassingly parallel nature, PLaNT exhibits excellent near-linear speedup up to $q = 64$ for almost all datasets. On 64 nodes, PLaNT achieves an average 42× speedup over single node execution. However, for scale-free graphs, PLaNT is not efficient. It cannot process LIJ and takes > 1 hr to process POK even on 64 nodes.

The Hybrid algorithm combines the scalability of PLaNT with the pruning efficiency of DGLL (powered by Common Labels). It scales well up to $q = 64$ and for most datasets, achieves $> 10\times$ speedup over single node execution. For scale-free datasets ACT, BDU and POK, it is able to construct CHL 7.8×, 26.2× and 5.4× faster than PLaNT, respectively, on 64 nodes. Compared to DparaPLL, the Hybrid algorithm is 3.8× faster on average when run on 2 compute nodes. For SKIT and WND, the Hybrid algorithm is 47× and 96.8× faster, respectively, than DparaPLL on 16 nodes. When processing scale-free datasets on small number of nodes, Hybrid beats PLaNT by more than an order of magnitude difference in execution time.

On a single node, GLL is faster than Hybrid and DGLL, as it does not incur the overheads associated with distributed computation, extra graph traversal in PLaNTed Trees and global table search for cleaning queries. However, Hybrid algorithm (1) outperforms GLL for all graphs after just 4 nodes, and (2) can process large graphs, such as LIJ, using

Table 4: Query Processing Throughput, Latency and Total Memory Consumption for different modes on 16 compute nodes. "-" = unsupported due to main memory constraints. DparaPLL only supports single node querying, hence we use QLSN mode as a proxy.

Dataset	Throughput (million queries/s)			Latency (μs per query)			Memory Usage (GB)		
	QLSN	**QFDL**	**QDOL**	**QLSN**	**QFDL**	**QDOL**	**QLSN**	**QFDL**	**QDOL**
CAL	10.1	12.1	**29.6**	**2.8**	22.3	8.4	43.8	**2.4**	13.7
EAS	7.1	8.9	**14.6**	**3.6**	24	11.4	125.4	**7.4**	39.2
CTR	-	6.5	**9**	-	26.6	**14.7**	-	**45**	242.1
USA	-	5.4	**10**	-	29.5	**20**	-	**80**	413.3
SKIT	15.8	18.5	**29.8**	**1**	20.7	7.9	4.5	**0.3**	1.4
WND	37.5	19.6	**42.7**	**0.3**	22.7	7.1	0.6	**0.1**	0.6
AUT	4.9	9.9	**27.5**	**3.7**	21.7	12.9	16.6	**1**	5.2
YTB	10.4	23.3	**30.3**	**2.2**	23.9	13.6	74.9	**4.6**	23.4
ACT	3.2	10.4	**21.3**	**4.8**	22.8	18.1	46.1	**2.8**	14.4
BDU	13.2	16.4	**21.5**	**1.5**	22.1	11.1	54.7	**3.2**	17.1
POK	-	5.1	**7.5**	-	**32**	34.5	-	**77.6**	388.9
LIJ	-	**6**	-	-	**31.6**	-	-	**125.8**	-

multiple nodes. SparaPLL performance on single cluster node was similar to GLL and is not shown in fig.8 for clarity.

We also observe superlinear speedup in some cases (for eg. Hybrid on CAL and EAS – 1 node vs 4 nodes). This is because running on few nodes poses high DRAM utilization and stress on memory manager due to frequent memory (re)allocations for several label arrays. In such cases, increasing number of nodes releases the pressure on memory manager, resulting in a super linear speedup.

Graph Topologies: We observe that PLaNT alone not only scales well but is also extremely efficient for *road networks*. On the other hand, in scale-free networks, PLaNT although scalable is not efficient as it incurs large overhead of additional exploration. This is consistent with our observations in figure 3 where the maximum value of Ψ for SKIT was $> 10\times$ that of maximum Ψ in CAL dataset. The Hybrid algorithm that cleverly manages the trade-off between additional exploration and communication avoidance, is significantly faster than PLaNT for most scale-free networks. However, it does not scale equally well for small datasets. This is because even few synchronizations of large number of nodes completely dominate their small labeling time.

7.4 Evaluating Query Modes

In this section, we assess the different query modes on the basis of their memory consumption, query response latency and query processing throughput.

Table 4 shows the memory consumed by label storage under different modes. QLSN requires all labels to be stored on every node and is the most memory hungry mode. Both QDOL and QFDL distribute the labels across multiple nodes enabling queries on large graphs where QLSN fails. Our experiments also confirm the theoretical insights into the memory usage of QFDL and QDOL presented in section 6. On average, QDOL requires $5.3\times$ more main memory for label storage than QFDL. This is because the label size per partition in QDOL scales with $\mathcal{O}\left(1/\sqrt{q}\right)$ and every compute node has to further store label set of 2 such partitions.

To evaluate the latency of various query modes, we generate 1 million random PPSD queries and compute their response one at a time. In QFDL (QDOL) mode, one query is transmitted per MPI_Broadcast (MPI_Send, respectively) and inter-node communication latency becomes a major contributor to the overall query response latency. This is evident from the results (table 4) where latency of QFDL shows little variation across different datasets. Contrarily, QLSN does not incur inter-node communication and compared to QDOL and QFDL, has significantly lower latency although it increases proportionally with ALS. For most datasets, QDOL latency is $< 2\times$ compared to QFDL, because of the cheaper point-to-point communication as opposed to more expensive broadcasts (section 6). An exception is POK, where average label size is huge (table 3) and QFDL takes advantage of multi-node parallelism to reduce latency.

To evaluate the query throughput, we create a batch of 100 million random PPSD queries and compute their responses in parallel. For most datasets, the added multi-node parallelism of QFDL and QDOL[10] overcomes the query communication overhead and results in higher throughput than QLSN. QDOL is further $1.8\times$ faster than QFDL because of reduced communication overhead[11].

8. CONCLUSION AND FUTURE WORK

In this paper, we propose novel algorithmic innovations and optimizations that effectively utilize the massive parallelism in shared and distributed memory systems. Our embarrassingly parallel algorithm PLaNT is the first distributed memory approach that enables collaborative memory usage by partitioning the labels across multiple nodes, while simultaneously ensuring high parallel scalability.

There are several interesting directions to pursue in the context of this work. System level aspects, such as branch mispredictions [30], cache misses [52, 6] and memory stalls [37, 23] are known to exert great influence on the performance of graph algorithms. Exploring optimizations for these aspects of hub labeling could be a great avenue for further acceleration. We will also explore the use of distributed atomics and RMA calls to dynamically allocate tasks in a cluster for better load balancing.

Acknowledgements. This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086 and National Science Foundation (NSF) under Contract Number CNS-1643351. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or NSF. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

[10]Results for QDOL mode also include the time taken to sort the queries and reorder the responses.

[11]QDOL has better memory access locality as every node scans all labels of vertices in the assigned queries. Contrarily, each node in QFDL scans a part of labels for all queries, frequently jumping from one vertex's labels to another.

9. REFERENCES

[1] The skitter as links dataset, 2019. [Online; accessed 8-April-2019].

[2] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. Vc-dimension and shortest path algorithms. In *International Colloquium on Automata, Languages, and Programming*, pages 690–699. Springer, 2011.

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, pages 230–241. Springer, 2011.

[4] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*, pages 24–35. Springer, 2012.

[5] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 782–793. Society for Industrial and Applied Mathematics, 2010.

[6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number CONF, pages 266–277, 1999.

[7] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 147–154. SIAM, 2014.

[8] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013.

[9] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceedings of the 23rd international conference on World wide web*, pages 237–248. ACM, 2014.

[10] T. Akiba, C. Sommer, and K.-i. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 144–155. ACM, 2012.

[11] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the world-wide web. *nature*, 401(6749):130, 1999.

[12] M. Babenko, A. V. Goldberg, H. Kaplan, R. Savchenko, and M. Weller. On the complexity of hub labeling. In *International Symposium on Mathematical Foundations of Computer Science*, pages 62–74. Springer, 2015.

[13] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM, 2006.

[14] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3), 2010.

[15] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[16] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98. ACM, 2012.

[17] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[18] D. Delling, A. V. Goldberg, R. Savchenko, and R. F. Werneck. Hub labels: Theory and practice. In *International Symposium on Experimental Algorithms*, pages 259–270. Springer, 2014.

[19] D. Delling, A. V. Goldberg, and R. F. Werneck. Hub label compression. In *International Symposium on Experimental Algorithms*, pages 18–29. Springer, 2013.

[20] C. Demetrescu, A. Goldberg, and D. Johnson. 9th dimacs implementation challenge–shortest paths. *American Mathematical Society*, 2006.

[21] L. Dhulipala, G. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 293–304. ACM, 2017.

[22] Q. Dong, K. Lakhotia, H. Zeng, R. Karman, V. Prasanna, and G. Seetharaman. A fast and efficient parallel algorithm for pruned landmark labeling. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[23] S. Eyerman, W. Heirman, K. D. Bois, J. B. Fryman, and I. Hur. Many-core graph workload analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 22:1–22:11, Piscataway, NJ, USA, 2018. IEEE Press.

[24] D. Ferizovic and G. E. Blelloch. Parallel pruned landmark labeling for shortest path queries on unit-weight networks. 2015.

[25] J. S. Firoz, M. Zalewski, T. Kanewala, and A. Lumsdaine. Synchronization-avoiding graph algorithms. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 52–61. IEEE, 2018.

[26] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *Proceedings of the VLDB Endowment*, 6(6):457–468, 2013.

[27] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering*

& Expermiments, pages 90–100. Society for Industrial and Applied Mathematics, 2008.

[28] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[29] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[30] O. Green, M. Dukhan, and R. Vuduc. Branch-avoiding graph algorithms. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 212–223. ACM, 2015.

[31] S. Hangal, D. MacLean, M. S. Lam, and J. Heer. All friends are not equal: Using weights in social graphs to improve search. In *Workshop on Social Network Mining & Analysis, ACM KDD*, 2010.

[32] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[33] S. Horvath. *Weighted network analysis: applications in genomics and systems biology*. Springer Science & Business Media, 2011.

[34] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *Proceedings of the VLDB Endowment*, 7(12):1203–1214, 2014.

[35] B. H. Junker and F. Schreiber. *Analysis of biological networks*, volume 2. Wiley Online Library, 2008.

[36] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[37] K. Lakhotia, R. Kannan, and V. Prasanna. Accelerating pagerank using partition-centric processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 427–440, 2018.

[38] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin. Scaling distance labeling on small-world networks. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1060–1077, New York, NY, USA, 2019. ACM.

[39] Y. Li, M. L. Yiu, N. M. Kou, et al. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11(4):445–457, 2017.

[40] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[41] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[42] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[43] M. E. Newman. Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality. *Physical review E*, 64(1):016132, 2001.

[44] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[45] K. Qiu, Y. Zhu, J. Yuan, J. Zhao, X. Wang, and T. Wolf. Parapll: Fast parallel shortest-path distance query on large-scale weighted graphs. In *Proceedings of the 47th International Conference on Parallel Processing*, page 2. ACM, 2018.

[46] S. A. Rahman, P. Advani, R. Schunk, R. Schrader, and D. Schomburg. Metabolic pathway analysis web service (pathway hunter tool at cubic). *Bioinformatics*, 21(7):1189–1193, 2004.

[47] P. Shiralkar, A. Flammini, F. Menczer, and G. L. Ciampaglia. Finding streams in knowledge graphs to support fact checking. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2017.

[48] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[49] Y. Tang, M. Li, J. Wang, Y. Pan, and F.-X. Wu. Cytonca: a cytoscape plugin for centrality analysis and evaluation of protein interaction networks. *Biosystems*, 127:67–72, 2015.

[50] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 37–42. ACM, 2009.

[51] F. Wei. Tedi: efficient shortest path query answering on graphs. In *Graph Data Management: Techniques and Applications*, pages 214–238. IGI Global, 2012.

[52] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.

[53] J. Xu and Y. Li. Discovering disease-genes by topological features in human protein–protein interaction network. *Bioinformatics*, 22(22):2800–2805, 2006.

[54] S. A. Yahia, M. Benedikt, L. V. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *Proceedings of the VLDB Endowment*, 1(1):710–721, 2008.

[55] D. Zhang, C.-Y. Chow, A. Liu, X. Zhang, Q. Ding, and Q. Li. Efficient evaluation of shortest travel-time path queries through spatial mashups. *GeoInformatica*, 22(1):3–28, 2018.

[56] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.