9 SUPPLEMENTARY MATERIALS

9.1 Complete Definition of Hierarchical Hub Labeling (HHL) and Canonical Hierarchical Hub Labeling (CHHL)

An important direction to make 2-hop labeling feasible and scalable for large graph is to restrict the choices of labeling (by imposing some special properties on what can be added into the labels).

Definition 1. *(Hierarchical Hub Labeling) Given two distinct vertices u and v, we say $u \geq v$ if $u \in L(v)$ (u is a hub of v). A hub (2-hop) labeling is* hierarchical *if $\geq$ forms a partial order.*

In fact, any partial order can be extended to a total order (the order-extension principle) and for a set of vertices V, the total order is defined as a bijection $\pi : V \longrightarrow 1, \cdots, |V|$ ($\pi(v)$ is the rank of v). Given this, we can say that a label is hierarchical if there is a total order π which satisfies: $u \in L(v)$ then $\pi(u) < \pi(v)$ (u ranks higher than v).

Definition 2. *(Canonical Hierarchical Hub Labeling) Let the shortest path vertex set P_{uv} consist of all vertices on shortest paths between u and v (including u and v). Given a total order π on V, its canonical hub labeling is defined as follows: $u \in L(v)$ if u has the highest order in P_{uv}, i.e., no other vertex w in P_{uv} such that $\pi(w) < \pi(u)$.*

An important implication of canonical hierarchical hub labeling is that *it produces the minimal hierarchical hub labeling* for a given order [3]. Thus, the optimal HHL problem can be transformed into two sub-problems: 1) finding the optimal order that minimizes the label size; 2) computing the canonical HHL with respect to a given vertex order.

A main breakthrough enabling efficient 2-hop labeling is the discovery of a simple, yet elegant algorithm called pruned landmark labeling (PLL) [2]. It computes the canonical HHL (the second subproblem) for a given vertex order efficiently. Independently, essentially the same style algorithm was discovered for 2-hop reachability labeling, and is called *distribution labeling* [34]. In the past few years, a number of studies [16, 40] have further validated and confirmed the efficiency and effectiveness of PLL style algorithms for distance labeling.

Theoretically, the optimal hierarchical hub labeling (HHL) as well as the original 2-hop labeling have recently been proved to be NP-hard [3], which implies that the optimal order sub-problem (the first sub-problem listed above) is NP-hard as well. A few heuristics, such as the ranking by degree and betweenness, have been developed for addressing this sub-problem [40]. The second sub-problem (labeling generation) typically dominates the overall labeling computation and is thus the focus of this study.

9.2 Proof of VC-PLL Theoretical Properties

Theorem 3. *VC-PLL (Algorithm 3) produces the canonical hierarchical hub labeling given a vertex order π.*

Proof Sketch: Recall the shortest path vertex set P_{uv} consists of all vertices on shortest paths between u and v (including u and v). Then, we need to prove $u \in L(v)$ *iff u has the highest order in P_{uv} (Definition 2).*

First ($\rightarrow$), we can see that if $u \in L(v)$, then we cannot find another vertex w with rank higher than u, such that $d(u,v) \geq d(u,w) + d(w,v)$. Thus, u must have highest order in P_{uv}. If not, assume we have another vertex $w \neq u$ that has the highest rank in P_{uv}. Then, based on our algorithm, w will be the highest ranked in P_{wu} and P_{wv}. Thus, w can always reach u and v before u reaches v (Figure 10) and it is in L_u and L_v when u reaches v.

Second ($\leftarrow$), assuming u has the highest order P_{uv}, then, based on the same argument, it can definitely go through a shortest path from u to v using Algorithm 3 and if it reaches v, no other vertices in L_v (and L_u) can prune it. □

The following corollary can be immediately obtained.

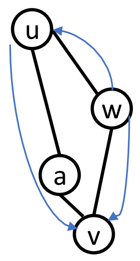

Figure 10:

Corollary 1. *In VC-PLL, when a distance label $(u, d(u,v))$ is added into $\delta L(v)$, $d(u,v)$ is the exact shortest path distance between u and v, and u has the highest rank in P_{uv}. Further, at any time $L(v) \subseteq \mathcal{L}(v)$, where $\mathcal{L}(v)$ is the final complete label of v.*

9.3 Detailed Explanation of VC-PLL Limitations

Additional Cost of Distance Label Generation: For a given vertex u, PLL will send it to a vertex v only once. In BFS, PLL will flag v after one distance label $(u, d(u,v))$ is passed through (Line 7 in Algorithm 1 is sequentially executed). But VC-PLL can send multiple $(u, d(u,v))$ messages to the same v at two consecutive iterations.

Lemma 1. *Given vertex u and vertex v, a distance label $(u : d(u,v))$ may reach v at exactly two possible and consecutive iterations: Let a be a neighbor of v, and $u \in L(a)$ (u is the highest rank vertex in P_{ua}), then it reaches v at $d(u,a) + 1$ iteration, which is either: 1) equal to the shortest path distance between u and v, and u may or may not be added to $L(v)$; or 2) equal to $d(u,v) + 1$, i.e., the path from u to a to v is one step longer than the shortest path between u and v, and u will be pruned.*

Proof Sketch: To see this, we first need to prove that the shortest path distance $d(u,a)$ is smaller than or equal to $d(u,v) + 1$, where a and v is the direct neighbor of one another. By way of contradiction, let us assume $d(u,a) \geq d(u,v) + 2$. Then, let w be the highest rank vertex in P_{ua}, then, we can find a path from u to w to v to a, which is $d(u,v) + 1$. This suggests $d(u,a) \leq d(u,v) + 1$. Next, we show u indeed can reach v in two consecutive iterations. This happens when u reach v via a being the shortest path between u to v: $d(u,v) = d(u,a) + 1$; but u is not the highest rank one in P_{uv}. Thus, u is not added to $L(v)$ in $d(u,v)$ iteration. Now, assume u reach a' with $d(u,a') = d(u,v)$ and $u \in L(v')$ (added in $d(u,v)$ iteration. If a' is the neighbor of v, then u will be sent to v in $d(u,v) + 1$ iteration as well. □

In addition, at each of these two iterations, if u has not been or is not in the label of v, then different neighbors of v may send the same $(u, d(u,v))$ messages to v.

Additional Cost of Distance Check:

Lemma 2. *The set consisting of all pairs (u,v) for distance checks is the same in PLL and VC-PLL.*

Proof Sketch: Let $reach(u)$ be the subset of vertices u reaches. In PLL, it corresponds to all the $(u, d(u, v))$ messages added into the Q (Line 7 in Algorithm 1). In VC-PLL, it corresponds to all the $(u, d(u, v))$ messages being sent to vertex v (Line 5 in Algorithm 3). Thus, $\bigcup_{u \in V} \{u\} \times reach(u)$ is the set consisting of all pairs (u, v) for distance check. In PLL and VC-PLL, for a vertex u, it is assigned to the same subset of vertices (Corollary 1). Also, it will also be sent to the same set of vertices which do not use u as label. Thus, the set $\bigcup_{u \in V} \{u\} \times reach(u)$ is the same for both. □

However, *the number of distance checks in VC-PLL can be higher than PLL, as a vertex u can be sent to v in two consecutive iterations in VC-PLL.*

The computational cost of distance check

$$d(u, v) < \min_{h \in L(u) \cap L(v)} \{d(u, h) + d(h, v)\}$$

in VC-PLL is also higher than that in *PLL*. In VC-PLL, the cost is $O(|L(u)| + |L(v)|)$, where $L(u)$ and $L(v)$ are (partial) labels of u and v at the time of distance check for $d(u, v)$. Assuming $L(u)$ and $L(v)$ are not sorted, we can first map $L(u)$ into an array or hash-table, and then check all the vertices in $L(v)$ against the above data structure. In PLL [2], since we process vertex u one at a time, and when we try to process u, its label $L(u)$ is already computed. Thus, we can first map $L(u)$ to an array only once at the beginning of the BFS iteration. Thus, the cost of $O(|L(u)|)$ can be practically saved for each distance check; thus the distance check for PLL is only $O(|L(v)|)$. For VC-PLL, we cannot do this directly as it is prohibitively expensive to map every $L(u)$ to an array or hash-table at the same time.

9.4 Complete Computation Cost Comparison between BVC-PLL and PLL

In the following, we provide an apple-to-apple computational cost analysis between BVC-PLL and PLL. We will focus on the cost of generating (sending) distance labels and distance checks.

Cost of Distance Label Generation: Since in BVC-PLL, each vertex u can be sent to v exactly once, together with Lemma 2 (the same set of u reaches v), we thus observe:

Lemma 3. *The time complexity of sending vertex label messages $(u, d(u, v))$ along the edges in graph G given an order π, is the same for PLL and BVC-PLL.*

Following Lemma 3, we obtain the following corollary.

Corollary 2. *The total number of distance checks (applying canonical labeling criterion) being invoked in PLL (Line 5 in Algorithm 1) is the same as those being invoked in BVC-PLL (Line 14 in Algorithm 4).*

This is because the number of distance checks is equivalent to the total number of generated distance label message: $\sum_{u \in V} |reach(u)|$ (following the algorithm logic).

Cost of Distance Check: Now, the cost of the same distance check on $d(u, v)$: $d(u, v) < \min_{h \in L(u) \cap L(v)} d(u, h) + d(h, v)$, in PLL and BVC-PLL, is $O(|L(v)|)$. However, $L(v)$ are different for PLL and BVC-PLL: In PLL, when u reaches v, $L(v)$ consists of all vertex labels which have higher rank than u; In BVC-PLL, assuming u in batch B_i, $L(v)$ consists of all the vertex labels in all the batches before B_i (those are the same as those in PLL) and the vertices in the current batch which are within the distance of $d(u, v)$.

Given this, let us focus on only those vertices being added at batch B_i for $L(v)$, and denote it as $L^i(v)$. Next, we break the distance check cost on $|L^i(v)|$ into two categories: 1) the *positive* distance check which will confirm the vertex u and can add it into the corresponding label of v; 2) the *negative* distance check will return false on the distance check and thus prune the vertex u.

Theorem 5. *(Positive Distance Check) The time complexity of all positive distance checks in BVC-PLL is lower than or equal to that of PLL.*

Proof Sketch: Let us consider any batch B_i. For the positive cases of distance check $d(u, v)$ here, given a vertex v and u, u will always be added to the label of v. For PLL, for a vertex v, let its complete $\mathcal{L}^i(v)$ consists of $u_1, u_2, \cdots, u_n \in B_i$, where $n = |\mathcal{L}^i(v)|$ and $\pi(u_1) < \pi(u_2) \cdots < \pi(u_n)$. Then the total cost of distance check with respect to $|L^i(v)|$ is simply $0 + 1 + \cdots + n - 1 = n(n-1)/2$, because in PLL, when u_i arrives, $L^i(v)$ already consists of partial labels $\{u_1, \cdots, u_{i-1}\}$. For BVC-PLL, for a vertex v, we note that its distance label in B_i is arriving in group according to their distances. Let $g_1, g_2, \cdots g_k$ be the groups ordered by arriving (as well as distance), i.e., given any two vertices $x, y \in g_i$, $d(x, v) = d(y, v)$, and their distance is smaller than those in g_{i+1}. Note that *for any vertex $u \in g_i$, we only utilize $L^i(v) = g_1 \cup \cdots \cup g_{i-1}$ for distance check* (See Lines 11 – 15 in Algorithm 3, $L^i(v)$ will be updated until all the distance checks in a batch g_i are done). Let $n_i = |g_i|$ and $n = \sum_{i=1}^{k} n_i$, making the total cost of distance checks of vertex v with respect to $|L^i(v)|$ in BVC-PLL to be

$$0 + n_1 \times n_2 + (n_1 + n_2) \times n_3 + \cdots + (\sum_{i=1}^{k-1} n_i) \times n_k$$

$$= (n - n_1)n_1 + (n - (n_1 + n_2))n_2 + \cdots + (n - \sum_{i=1}^{k-1} n_{k-1})$$

$$= n(n-1)/2 - \sum_{i=1}^{k} n_i(n_i - 1)/2. \square$$

Figure 11 illustrates the key idea in the proof of Theorem 5. Assuming 9 vertices $a, b, \cdots, i$ in one batch being added into $L(v)$ in PLL labeling, its total distance check cost is 36 no matter which order they are received in (visualized as the area under the diagonal stairs). Now assuming they arrive in three groups as shown in Figure 11(a), then in BVC-PLL, their total distance check cost is $3 + 3 \times 6 = 27$, a 25% reduction compared to PLL.

Theorem 5 essentially shows that BVC-PLL is able to save the intro-group cross-vertex comparison in each batch. Basically, if vertices arrive at the same time, they have the same distance to vertex v and cannot prune one another.

To compare the time complexity difference between PLL and BVC-PLL for the negative distance check, we introduce the following notation: for any vertex x, and one of its vertex label u ($u \in L^i(x)$), we denote $< x, u >$ to be a subset: $\{x \in$

$$\bigcup_{y \in N(x)} L^i(y) \setminus L^i(x) : \pi(u) < \pi(v) < \pi(x), d(x, u) > d(v, y) + 1\}$$

Similarly, we define $< y, v >$ for vertex y with its label v, $v \in L^i(y)$:

$$\{u \in \bigcup_{x \in N(y)} L^i(x) \setminus L^i(y) : \pi(u) < \pi(v), d(x, u) > d(v, y) + 1\}$$

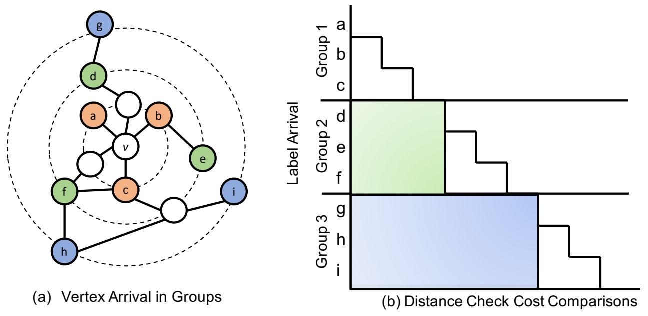
(a) Vertex Arrival in Groups
(b) Distance Check Cost Comparisons
Label Arrival
Group 1
Group 2
Group 3
a
b
c
d
e
f
g
h
i
v

Figure 11: Theorem 5

Theorem 6. *(Negative Distance Check) In batch B_i, and on negative distance check, the time complexity saved by BVC-PLL compared with PLL is no higher than*

$$O(\sum_{x \in V} \sum_{u \in L^i(x)} |\langle x, u \rangle| - \sum_{y \in V} \sum_{v \in L^i(y)} |\langle y, v \rangle|).$$

The time complexity saved by PLL compared with BVC-PLL is no higher than

$$O(\sum_{y \in V} \sum_{v \in L^i(y)} |\langle y, v \rangle| - \sum_{x \in V} \sum_{u \in L^i(x)} |\langle x, u \rangle|).$$

Proof Sketch: To quantify the difference of the time complexities between two algorithms, we focus on the cases where one algorithm can save computational cost when the $L^i(v)$ will be different for distance check $d(u, v)$.

For the first case, let us consider vertex x, it has a vertex $u \in L^i(x)$. Now, consider any vertex $v \in B_i$ reaches vertex x for distance check and returns negative result. If v can reach x, it must be a label of neighbor y of x, i.e., $v \in L^i(y), y \in N(x)$, and $v \notin L^i(x)$ (false distance check). When v reaches x, it has also lower rank than u but higher than x: $\pi(u) < \pi(v) < \pi(x)$. Given this, for PLL, u is already in $L(x)$; however, for BVC-PLL, v can reach x before u reaches x. Thus, this case will introduce a gain for BVC-PLL; and such v is characterized and recorded in set $\langle x, u \rangle$.

For the second case, let us consider vertex y, and it has a vertex $v \in L^i(y)$. Now, consider any vertex $u \in B_i$ reaches vertex y for distance check and returns negative result. If u can reach y, it must be a label of neighbor x of y, i.e., $u \in L^i(x), x \in N(y)$, and $u \notin L^i(y)$ (false distance check). When u reaches y, it has also higher rank than v: $\pi(u) < \pi(v)$. Given this, for BVC-PLL, v is already in $L(y)$; however, for PLL, u can reach x before v is added into $L(y)$. Thus, this case will introduce a gain for PLL; and such u is characterized and recorded in set $\langle y, v \rangle$. □

Theorem 6 does not provide a clear winner on the cost of negative check. However, from the symmetric expression of these two qualities, we conjecture they should be close to one another. In Section 6, we will experimentally confirm this. In addition, for negative distance check, we typically do not need to traverse through the entire $L(v)$ set. Indeed, the bit-parallel mechanism proposed in the original PLL paper [2] can help provide almost $O(1)$ pruning. Since the number of negative checks is the same for PLL and BVC-PLL, we expect their overall cost will be fairly close to each other.

Putting It Together: Assuming that PLL and BVC-PLL have a similar cost for negative distance checks, theoretically, *BVC-PLL may have smaller computational cost than that of PLL (due to positive distance check) since they have the same cost of generating/sending distance labeling!* Furthermore, BVC-PLL is guaranteed to have a smaller memory access cost for graph topology than PLL as it groups messages together for each edge access. Overall, it seems BVC-PLL, an unexpected marriage between PLL and VC computation, can run faster than the original PLL sequentially and can also enjoy the scalability of the VC model! Indeed, Section 6 shows that it can be more than two times faster than PLL (both using one thread) on real-world graphs.

9.5 Some Implementation Details

Integrated Bitmap and Queue: Much temporary data is generated for both labeling vertices and active vertices during each batch processing. These steps require a *clearance* (e.g., Algorithm 4, line 22). The cost of this clearance is significant as this operation occurs for each batch. Traditionally, we often use either a bitmap or a queue to handle the set of active vertices. However, they become inefficient or insufficient for supporting BVC-PLL. For a bitmap, each of its cleanings can take $O(|V|)$ where $|V|$ is the total number of vertices; for a queue, it cannot support efficient checks for whether a given vertex is active or not. Given this, we propose a new traversal control data structure by combining both the bitmap and the queue. The basic idea is that a bitmap supports fast recording and checking visited vertices and a queue supports fast finding and clearing the visited vertices. Each time a vertex is processed, we add it to both the bitmap and the queue. This approach is different from the bitmap and queue used in the push and pull strategy presented in [5, 51, 63] because we use both the bitmap and queue simultaneously rather than in different stages of processing.

Bit-parallel Adoption: Similar to PLL [2], *bit-parallel* is also adopted to accelerate the *distance checking* in the implementation of BVC-PLL for unweighted graphs. Its construction is similar to multi-source BFS traversals and can be easily expressed in the Vertex-Centric computing model.