A Fast and Efficient Parallel Algorithm for Pruned Landmark Labeling

Qing Dong,[1] Kartik Lakhotia,[2] Hanqing Zeng,[2] Rajgopal Kannan,[3] Viktor Prasanna,[1,2] Guna Seetharaman[4]
Department of Computer Science,[1] Ming Hsieh Department of Electrical Engineering,[2] University of Southern California
US Army Research Lab-West[3]
US Naval Research Laboratory[4]
{qingdong, klakhoti, zengh, rajgopak, prasanna}@usc.edu, guna@nrl.navy.mil

***Abstract*—Hub labeling based shortest distance querying plays a key role in many important networked graph applications, such as route planning, socially-sensitive search and web page ranking. Over the last few years, Pruned Landmark Labeling (PLL) has emerged as the state-of-the-art technique for hub labeling. PLL drastically reduces the complexity of label construction by pruning Shortest-Path Trees (SPTs). However, PLL is inherently sequential, as different SPTs must be constructed in a specific order of source vertices to ensure small label size. Particularly, for large graphs, it takes significant processing time to construct even pruned SPTs from all vertices in the graph. While there are many works on parallelizing single source shortest path, these solutions cannot be directly used for PLL, as pruning and label querying introduce significant additional complexity while restricting parallelism within an SPT.**

In this paper, we propose a novel, fast and efficient algorithm to significantly accelerate PLL on large graphs based on a two-level parallelization of SPTs: intra- and inter-tree. For intra-tree, we generate pruned SPTs based on a modification of the Bellman-Ford (BF) algorithm. We further optimize BF to reduce SPT label querying and initialization costs. We implement our algorithm using the recently proposed Graph Processing Over Partitions (GPOP) which dramatically improves cache-efficiency and DRAM communication-bandwidth. When pruned SPTs become very small and parallelizing individual SPTs is not advantageous, we switch to inter-tree parallelization and construct multiple trees concurrently in a batch. Experiments conducted on a 36 core (2-way hyperthreaded) Intel Broadwell server show that on some datasets, our proposed parallel algorithm can achieve greater than $35.1\times$ speedup over state-of-the-art sequential algorithm.

I. Introduction

Computing the shortest distance between nodes in a graph or network is a key task in many applications, for example, (a) in social and web networks - for recommendations and node ranking [1], (b) in knowledge graphs - for detecting concept similarities [2], and (c) in transport networks - for distance-sensitive navigation [3].

Hub labeling [4][5][6] has become the state-of-the-art preprocessing technique for low-response time shortest distance querying. Pruned Landmark Labeling (PLL) is a key step in many hub labeling algorithms and is also an application of interest in DARPA's HIVE program for accelerating graph analytics. However PLL, which is based on Dijkstra's algorithm is inherently sequential. As graphs grow ever larger, hub label preprocessing becomes a major bottleneck for shortest path query processing. For example, we saw that the runtime of sequential PLL exceeded 24 hours for a 1.6 million node social network graph (actor, see Section V). To the best of our knowledge, there have been no significant efforts towards accelerating PLL by taking advantage of modern multi-core processing architectures. In this paper, we present a novel, fast and efficient algorithm for accelerating PLL using a two-fold parallelization approach: intra-tree parallelization (parallelize computations within a Shortest Path Tree (SPT)) and inter-tree parallelization (parallelize computations across multiple SPTs).

For intra-tree parallelization, we first provide an algorithm to construct SPTs based on a modification of the Bellman-Ford (BF) algorithm. Unfortunately, a straightforward replacement of Dijkstra's by BF can radically increase the number of label queries and heavily deteriorate performance. Thus we propose novel optimizations to ensure that the number of label queries remains the same as in sequential Dijkstra-based PLL. Further, we parallelize the optimized algorithm using the recently proposed Graph Processing Over Partitions (GPOP) [7][8]. GPOP provides improved cache performance and enables lock-free access to each vertex. This becomes especially crucial for accelerating PLL as we can query and update labels without worrying about the conflict between multiple threads. We also note that most labels tend to be added during the first few SPTs. Consequently, the size of the graph explored is substantially reduced in subsequent SPTs and there is little intra-tree parallelism to exploit.

We therefore propose inter-tree parallelization to *batch* process multiple later stage SPTs and prove theoretically that the resulting labeling is correct (can answer any shortest path query correctly). To better exploit the benefits of intra- versus inter-tree parallelization, we explore the optimum switch points between the two modes through empirical analysis on various real-life datasets. Finally, we demonstrate the efficiency of the proposed methods through experiments on various large-scale real-world networks. We summarize the major contributions of our work below:

- We develop a novel algorithm for construction of individual SPTs (intra-tree parallelization) through a modification of Bellman-Ford coupled with a novel GPOP-based implementation that includes specific optimizations for efficient label querying during tree construction.
- We develop an inter-tree parallelization method to batch

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

process multiple SPTs in parallel when tree size becomes small.
- We show experimentally that pure intra-tree parallelization can achieve up to $32.1\times$ speedup for constructing *all* SPTs and up to $103\times$ speedup for the *first 0.1%* of SPTs, using 64 threads.
- We also show that employing hybrid intra- and inter-tree parallelization can provide up to $35.1\times$ speedup with a negligible increase in label size.

The rest of the paper is organized as follows. Section II introduces some related work. Section III provides definitions and notations used in the paper. In Section IV, we explain our design of the parallel algorithm. We show experimental results in section V with conclusions in Section VI.

II. Related Work

A. Shortest Distance Queries

Dijkstra's algorithm [9] answers shortest distance queries with a greedy strategy by expanding to neighbors repeatedly. With an efficient implementation of min-heap (e.g. Fibonacci heap), Dijkstra's algorithm takes $O(|E|+|V|\log|V|)$ time [10] in the worst case (where $|V|$ and $|E|$ are the number of vertices and edges). It is a near-linear time algorithm but cannot be parallelized. Bellman-Ford [11][12] is another algorithm to compute shortest distance and able to process graphs with negative weights. The time complexity for Bellman-Ford is $O(|V|\cdot|E|)$ in the worst case which is more than Dijkstra's. However, unlike Dijkstra's algorithm, Bellman-Ford algorithm is efficiently parallelizable.

Following the idea of network traversal, many algorithms have been proposed to answer shortest distance queries, e.g. A* [13], CH [14], TNR [15] and REAL [16]. Most of them abstract distance information to build various indices. These auxiliary indices are then used to accelerate shortest distance queries. One extreme solution is to precompute the shortest distances for all pairs of vertices. The queries can be answered very fast but the index size is $O(|V|^2)$ which is infeasible to store for large graphs [5]. Another line of research is to answer the shortest distance without network traversal where hub labeling techniques are the commonly used.

B. Hub Labeling Algorithms

By constructing a labeling for each vertex (a labeling usually contains certain hubs and the distance between the vertex and the hub), the shortest distance can be easily retrieved by checking the labels of source and destination vertices. Cohen et al. proposed the first hub labeling algorithm [17]. Abraham et al. then proposed hierarchical hub labeling to speedup preprocessing and find smaller labels [4]. Akiba et al. proposed Pruned Landmark Labeling (PLL) [18][5] and Delling et al. proposed labeling compression techniques [6].

III. Preliminaries

A. Notations

Table I lists some important notations used in this paper.

TABLE I: Frequently used notations

symbol	description
$G=(V,E,W)$	a graph with vertices V, edges E and edge weights W
$SP(s,t)$	the shortest path from s to t
$d(s,t)$	the shortest distance from s to t
O	an ordering of all vertices in V
$O(i)$	the i-th vertex in the ordering O
$L_{f/b}(v)$	the forward/backward labels of v
$L^i_{f/b}(v)$	the forward/backward labels of v after iteration i

B. Problem Definition

This paper studies the following problem: given a graph $G=(V,E,W)$ and an ordering O of all vertices indicating their *importance* based on some heuristic, design a parallel algorithm to efficiently construct an index (2-hop cover labeling) to answer shortest distance queries between an arbitrary pair of vertices.

C. Pruned Landmark Labeling

The hub labeling algorithm and most of its variants follow a two-step procedure (see Fig.1). The first step ranks all vertices in the order of their *importance* using heuristics (e.g. degree [18][19], betweenness [4] or closeness [18]). The second step generates labeling from pruned SPTs constructed in the order determined by the first step. The PLL [18] generates correct and minimal sized labeling for a given ordering. It has been generally accepted as the most efficient method to conduct step 2 [20]. Although optimal ordering would result in minimum average label size, it is NP-hard [17] to find such ordering resulting in the use of heuristics for step 1.

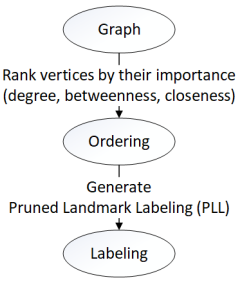

Fig. 1: Two-step procedure of hub labeling algorithm

From the most important vertex to the least important vertex, PLL generates a pruned SPT. The pruning happens when the shortest distance to a node is obtained by *label query* (there exists a more important vertex serving as a hub between the node currently visited and the root node of the SPT). The neighbors of pruned nodes are not explored, reducing the number of vertices touched. In undirected graphs, only one SPT will be generated from each root node. In directed graphs, forward SPTs will be generated based on the original graph to construct backward labels while backward SPTs will be generated based on the reverse graph to construct forward labels. The shortest distance between an arbitrary pair of vertices s and t can be found by searching the forward labels of s and backward labels of t. Akiba et al. [18] proved that PLL generates *correct* (can answer any shortest path query

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

correctly) and *minimal* (deleting any hub violates the *cover property*).

Cover property: for any s and t, there must be some common hub in $L_f(s) \cap L_b(t)$ which is included in one shortest path from s to t, which means we can always get correct shortest distance by finding the minimum distance by the merge-join of $L_f(s)$ and $L_b(t)$.

Akiba et al. [18] employ Dijkstra's algorithm to construct SPTs which is completely sequential. Step 1 can be implemented using various heuristics such as degree-based ordering, PageRank, etc. For most of these heuristics, the total cost of step 1 is much smaller than step 2. Hence, accelerating the second step (i.e. PLL) is the key to reduce the preprocessing cost of hub labeling algorithms.

D. Graph Processing Over Partitions (GPOP)

Recently, Lakhotia et al. proposed a cache- and work-efficient framework for graph processing over partitions [7] [8]. Different from current vertex-centric and edge-centric methods, GPOP divides the vertex set V into k disjoint partitions and implements each iteration of a graph algorithm in a two-phase (Gather-Apply-Scatter, GAS) model.

In the scatter phase, the out edges of active vertices in a partition are streamed in and vertex data is communicated to other partitions in the form of messages. In the gather phase, incoming messages of a partition are streamed in and the vertex data is updated as per a user-defined function. GPOP processes multiple partitions in parallel during both of the phases. Moreover, each partition is processed by a single thread and has its own distinct memory space to write outgoing messages.

GPOP improves the cache performance by exploiting the locality of partitioning. For PLL, GPOP can achieve good scalability by enabling label query and insertion without locks and atomics. It also uses a hybrid of source and destination centric communication modes in a way that ensures work-efficiency of each iteration and simultaneously boosts high bandwidth sequential memory accesses.

IV. Algorithm

A. Overall design

PPL constructs $O(|V|)$ SPTs. Due to pruning, the workload of each SPT's construction varies. In general, most of the labels are added in the first few SPTs (see Fig.3). Therefore, we exploit intra-tree parallelization for the construction of large initial SPTs. Subsequently, as SPTs become small, the available parallelism within each tree becomes limited. To construct such SPTs, we exploit inter-tree parallelization and build each tree using sequential Dijkstra's algorithm. The overall flow of the algorithm is shown in Fig. 2.

B. Intra-tree parallelization

In this section, we introduce the proposed intra-tree parallel algorithm for PLL which is based on the Bellman-Ford algorithm. Since pruning is effectively used in the original Dijkstra-based algorithm to reduce the running time, we first modify

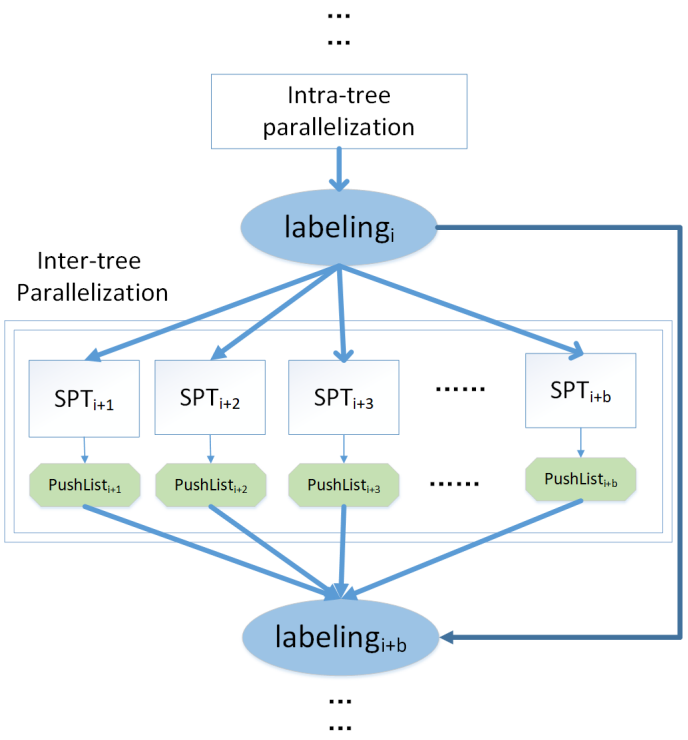

Fig. 2: Overall design of the hybrid intra- and inter-tree parallel PLL algorithm

Bellman-Ford to enable pruning. As opposed to conventional Bellman-Ford, we update the shortest distance of a node by the minimum of distance obtained by label query and by relaxing the edge with an active neighbor. Further, if the former is selected, the node under consideration is not activated to avoid exploring its neighbors.

We use the GPOP framework [8] to parallelize the proposed algorithm. In each iteration of Bellman-Ford, GPOP does work proportional to active edges. Thus, parallel Bellman-Ford using GPOP efficiently represents the pruning process. Algorithm 1 shows PLL implementation using GPOP APIs.

For the proposed algorithm, we implement the following optimizations to improve the efficiency.

1) Avoid repeated queries: When the program is constructing a single SPT, one vertex can become active multiple times. Every time it becomes active, the updated distance will be compared with the queried distance from the current labeling. To avoid repeated queries, we store the queried distance to the distance array. When a vertex becomes active again, new distance is simply compared with the value in distance array. Therefore, even though a vertex may become active several times, their labels are only queried once. Hence, the total work done in label queries is the same as the original sequential algorithm.

2) Asynchronous update: GPOP adopts a 2-phase Scatter-Gather model. The default Bellman-Ford algorithm in GPOP is synchronous in which all active nodes first send their distance

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

Algorithm 1 Pruned Bellman-Ford algorithm using GPOP APIs

```
    Global variables: srcId, distance
 1: procedure SCATTERFUNC(node)
 2:     return distance[node]
 3: procedure REINIT(node)
 4:     return false
 5: procedure GATHERFUNC(updateVal, destId)
 6:     queriedDis = label.query(srcId, destId)
 7:     if updateVal < distance[destId] then
 8:         if updateVal < QueriedDis then
 9:             distance[destId] = updateVal
10:             label.add(destId, srcId, updateVal)
11:             Return true
12:         else
13:             distance[destId] = queriedDis
14:             Return false
15:     else
16:         Return false
17: procedure APPLY(node)
18:     return true
19: procedure APPLYWEIGHT(updateVal, weight)
20:     return updateVal + weight
21: function MAIN
22:     Graph G
23:     initialize(G, filename)
24:     initBin(G)
25:     for i from 0 to |V| do
26:         srcId = order[i]
27:         distance={∞, ∞, ∞, ...∞}
28:         Frontier{srcId}
29:         loadFrontier()
30:         while G → FrontierSize > 0 do
31:             GPOP(G)
```

to neighbors and in the second phase, the neighbors compute and update their own distance. Consequently, any update to a node's distance value is only visible in the next iteration. We replace the default mode by asynchronous update where active nodes scatter address of the distance values. This optimization increases the rate of convergence of Bellman-Ford algorithm to reduce the running time.

3) Avoid unnecessary re-initialization: In the proposed algorithm, all the values in the distance array are set to be infinity before each SPT construction. However, since pruning techniques are used, a lot of the values in the array keep untouched. In this case, re-initializing the whole array before each SPT construction is not efficient. Instead, we record the vertices whose value has been changed and only re-initialize these vertices.

C. Inter-tree parallelization

Due to pruning, the size of SPTs reduces as the algorithm proceeds. For example, Fig.3 shows that most of the labels for coAuthor (see in V-A2) dataset are added in the first few SPTs. The number of labels added also reflects the size of SPT and the number of nodes explored during the SPT construction. If the size of SPT is very small, there is little parallelism to exploit and the overheads kill the benefits of parallelizing an individual SPT. Therefore, to accelerate construction of subsequent SPTs in PLL, we propose the inter-tree parallelization method (see in Algorithm 2).

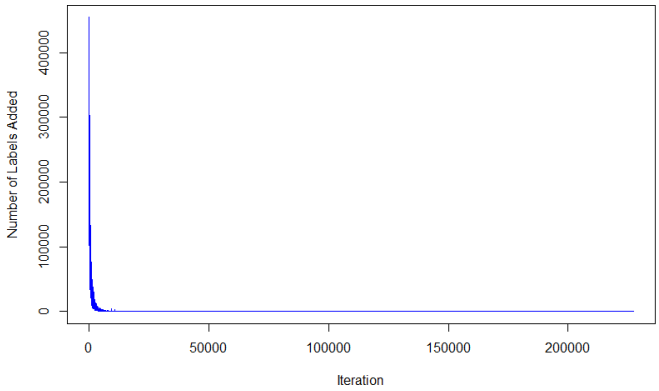

Fig. 3: Number of labels added in each SPT (coAuthor)

We construct multiple trees concurrently in a batch of size b. In other words, when we have finished i SPTs and obtained $L^i_{f/b}$, we start constructing trees $i+1$ to $i+b$ using Dijkstra's algorithm, on concurrent parallel threads. As a result, tree $i+j$ may not have the knowledge of trees $i+1$ to $i+j-1$ and redundant labels could be inserted. To avoid large label size due to this redundancy, we switch to inter-tree parallelization only when SPTs become very small. Consequently, the addition of redundant labels results in negligible increase in overall average label size. To balance the load among threads, we keep the batch size to be at least 8 times of the number of threads.

Algorithm 2 Inter-tree parallelization

```
    labeling_i: labeling after i-th SPT construction
    PushList_i: result data structure of i-th SPT
 1: for k ∈ [i+1, i+b) do in parallel          ▷ Inter-tree
 2:     PushList_k = PPL(Labeling_i, k)
 3:
 4: for all k ∈ [i+1, i+b) do
 5:     labeling_k ← push(labeling_{k-1}, PushList_k)
```

We will prove below that the algorithm can still generate *correct* labeling (can answer any shortest distance correctly) if not *minimum* (with no redundant labels).

1) Proof of correctness: In this part, L represents the labeling of the proposed algorithm while Lo represents the labeling of the original algorithm. Let's say we start inter-tree parallelization from SPT$_{i+1}$. Since $L^{i+1}_{f/b}$ takes $L^i_{f/b}$ as input, $L^{i+1}_{f/b}$ keeps the same with $Lo^{i+1}_{f/b}$. For SPT$_{i+j}$ ($2 \leq j \leq b$),

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

$L^{i+j}_{f/b}$ takes $L^{i}_{f/b}$as input while $Lo^{i+j}_{f/b}$ takes $Lo^{i+j-1}_{f/b}$ as input. In the original algorithm, $O(i+j)$ (the $i+j$-th vertex in ordering O) is pushed to the labeling of vertex t if $d(O(i+j),t)$ has not been covered by $L^{i+j-1}_{f/b}$. Since $L^{i}_{f/b}$ contains fewer labels than $Lo^{i+j-1}_{f/b}$, $d(O(i+j),t)$ has not covered by $L^{i}_{f/b}$ either. Therefore, the proposed algorithm will also push $O(i+j)$ to the labeling of vertex t. As we collect all the labels added in the batch to $L^{i+b}_{f/b}$, $L^{i+b}_{f/b}$ contains all of the labels in $Lo^{i+b}_{f/b}$ and any query to $L^{i+b}_{f/b}$ and $Lo^{i+b}_{f/b}$ give same result (which means the proof can be easily extended to later batches as well). Thus, the proposed algorithm generates *correct* labeling.

V. Experimental Evaluation

A. Experimental Setup

1) Platform and Environment: We implement our parallel algorithm using `C++` and `OpenMP` (version 4.5), which are consistent with the implementation of GPOP [8]. The codes are compiled by `g++ 4.8` with `-O3` flag. We use a dual-socket Broadwell server equipped with two 18-core Intel Xeon® E5-2695 v4 processors@ 2.1 GHz as our experimental platform which has 1 TB main memory with 55.1GBps copy bandwidth and 64.2 GBps add bandwidth.

2) Datasets: We use various types of graphs for evaluation. Table II summarizes the characteristics of the graphs. They are categorized into *complex* networks (e.g. social networks and hyperlink networks) and road networks. The ordering step uses vertex-degree as a heuristic for complex networks [21], and betweenness-centrality as a heuristic for road networks [20].

TABLE II: Input graph datasets

Type	Name	$\|V\|$	$\|E\|$	Avg. Deg
Complex	coAuthor [22]	227,320	814,134	3.58
	Actor [23][24]	382,219	33,115,812	86.64
	coPaper [22]	540,486	15,245,729	28.21
Road	FLA [25]	1,070,376	2,712,798	2.53

3) Baseline Algorithm: We use the sequential Dijkstra-based PLL [21] as the baseline since our work is the first study on parallelizing PLL to the best of our knowledge.

B. Evaluation of Intra-tree Parallelization

In this section, we evaluate the performance of our algorithm by exploiting intra-tree parallelization only. In other words, we construct all SPTs one by one in serial, and parallelize the construction of each tree by the pruned Bellman-Ford algorithm described in Algorithm 1.

1) Runtime for the first few SPTs: According to Figure 3, most of the workload is due to the construction of the very first few SPTs. Therefore, in Table III, considering the running time of the first 0.1% of SPTs' construction, we compare our pure intra-tree parallel algorithm (using 1 to 64 threads) with the sequential PLL implementation (denoted as "Dij").

Compared with the Dijkstra-based sequential algorithm, the inter-tree parallel program achieves up to $103\times$ speedup for the constructions of first few SPTs (using 64 threads). Comparing the runtime of the intra-tree parallel program using 1 to 64 threads, the proposed program shows good scalability.

In Table III, for many of the datasets, even with one thread, our parallel algorithm is faster than the baseline sequential algorithm. For each SPT construction, the baseline implementation based on Dijkstra's algorithm is of computation complexity $O(|V|\log|V|+|E|)$, whereas our implementation based on Bellman-Ford algorithm is of $O(|V|\cdot|E|)$. From the total workload perspective, our proposed algorithm is more efficient than the baseline for dense and small-diameter graphs.

TABLE III: Runtime for the first 0.1% SPTs of the sequential Dijkstra-based algorithm and intra-tree parallel algorithm (T_i indicates the runtime (s) using i threads)

Dataset	Dij	T_1	T_2	T_4	T_8	T_{16}	T_{32}	T_{64}
coAuthor	80	23	17	11	8	6	5	5
actor	2684	328	239	140	76	45	28	26
coPaper	2204.28	582	557	278	149	84	50	47
FLA	52	269	201	179	148	127	122	145

2) Effect of Various Optimizations: We proposed three optimizations to improve the efficiency in Section IV-B. Here we test the effect of various optimization using the coAuthor and FLA datasets as representatives of complex and road networks, respectively. Table IV summarizes the runtime comparison under various settings. Each of the experiments executes all SPTs construction using 16 threads on the parallel platform. Below we summarize the notations used: (i). Query methods "r"/"nr": **r**epeated/**n**on-**r**epeated queries (Section IV-B1). (ii). Update methods "asy"/"sy": **asy**nchronous/**sy**nchronous updates (Section IV-B2). (iii). Re-initialization methods "all"/"touched": "all" means to re-initialize the entire distance array with infinity. "touched" means to re-initialize the touched vertices only (Section IV-B3).

TABLE IV: Effect of various optimizations on the runtime of intra-tree parallel PLL

Dataset	update	re-init	query	runtime (s)
coAuthor	sy	touched	nr	70
coAuthor	sy	all	nr	108
coAuthor	sy	touched	r	79
coAuthor	asy	touched	nr	70
FLA	sy	touched	nr	752
FLA	sy	all	nr	1627
FLA	sy	touched	r	803
FLA	asy	touched	nr	741

The experimental results in Table IV show the effectiveness of our various optimization techniques. Specifically, the execution time is reduced by $1.12\times$ and $1.06\times$ due to the avoidance of repeated queries. The execution time is reduced by $1.54\times$ and $2.16\times$ due to the re-initialization on touched vertices only. The execution time is reduced by $1.04\times$ on FLA due to the asynchronous updates. The execution time does not benefit much from asynchronous updates for the coAuthor dataset since complex networks are of smaller diameters.

C. Evaluation of Hybrid Inter- and Intra-Tree Parallelization

Since most of the labels are added during the construction of the first few SPTs, SPT construction afterward only need to traverse a small fraction of the entire graph. Hence, the

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

TABLE V: Execution time (s) and average label size of PLL using different algorithms

Dataset	Dijkstra-based algorithm		intra-tree parallelization			inter- & intra-tree parallelization			
	runtime	average label size	runtime (1 thread)	runtime (64 threads)	average label size (64 threads)	switch point	runtime (1 thread)	runtime (64 threads)	average label size (64 threads)
coAuthor	537	273.106	424	193	273.106	0.005	496	44	277.482
actor	>24h	x	51669	2695	1135.1	0.1	55084	2464	1136.41
coPaper	>24h	x	>24h	14865	3629.43	0.2	>24h	13001	3636.18
FLA	99	67.875	377	1968	67.875	0.005	345	273	74.8023

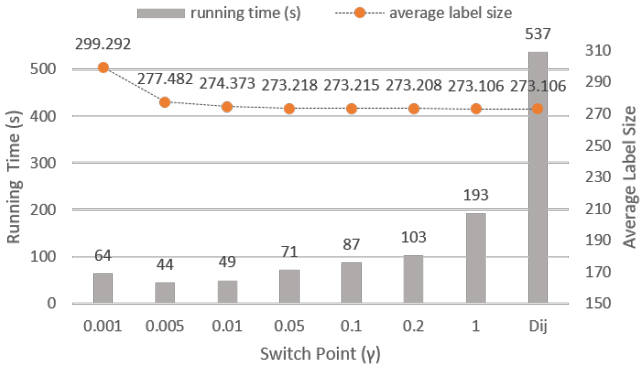

Fig. 4: The runtime and labeling size of hybrid intra- and inter-tree parallel PLL algorithm with different switch points

intra-tree parallelism becomes limited and the overheads kill the benefits of parallelizing an individual SPT. Therefore, we propose a hybrid parallelization scheme combining intra- and inter- SPT parallelization. Specifically, for the construction of the first $\gamma \cdot |V|$ number of SPTs (γ is the *switch point* of intra- and inter- tree parallelization), we exploit the intra-tree parallelization by our pruned Bellman-Ford algorithm. And for the remaining $(1-\gamma) \cdot |V|$ number of SPTs, we construct multiple trees in parallel, where each tree is built by the sequential Dijkstra's algorithm.

1) Effect of switch point: From the running time perspective, there exists an optimal switching point determined by γ_{opt}. The value of γ_{opt} depends on the characteristic of the input graph. Below we use the coAuthor dataset to explore the effect of γ. As shown in Figure 4, when using 64 threads, we can achieve up to $4.38\times$ speedup compared with the implementation using intra-tree parallelization only. As for label size, there is only a minor increase on label size, as expected. As an example, even when γ is as small as 0.5%, there is only 1.6% of increase on average label size.

2) Performance: We measure the runtime of sequential Dijkstra-based algorithm (using a single core of experiment platform), pure intra-tree parallel algorithm (using 1 and 64 threads) and hybrid intra- & inter-tree parallel algorithm (using 1 and 64 threads). Detailed result is shown in Table V. For complex networks, pure intra-tree parallelization provides $2.78\times$ to $> 32.1\times$ speedup using 64 threads (compared with Dijkstra-based algorithm). By combining inter-tree parallelization, the algorithm achieves $1.1\times$ to $4.38\times$ *additional* speedup.

Both of the intra-tree parallel algorithm and the hybrid algorithm show good scalability for large graphs like actor and coPaper. By increasing number of threads from 1 to 64, the pure intra-tree parallel algorithm achieves up to $19.17\times$ speedup and the hybrid parallel algorithm achieves up to $22.3\times$ speedup.

For road networks, the running time of parallel algorithm is larger than the Dijkstra-based algorithm. The reason is that the road networks usually have a large diameter and small average degree which provides little intra-tree parallelism. In such cases, small γ (switch to inter-tree parallelization earlier) usually helps to reduce the runtime.

VI. Conclusion & Future Work

In this paper, we developed one of the first parallel Pruned Landmark Labeling (PLL) algorithm that combines the benefits of intra-tree and inter-tree parallelization. For intra-tree parallelization, we developed a modified Bellman-Ford based algorithm with optimizations and parallelized it using the recently proposed Graph Processing Over Partitions (GPOP). For inter-tree parallelization, we set aside cross-loop dependency and processed multiple SPTs as a batch. Our hybrid PLL algorithm adaptively transitions from intra- to inter-tree parallelism in a way that ensures high scalability with negligible increase in label size.

We observe that intra-tree parallelization can provide up to $32.1\times$ speedup over the state-of-the-art (sota) sequential algorithm [5] and a speedup of more than $35.1\times$ can be achieved by combining inter-tree parallelism. For some large graphs, while the state-of-the-art does not finish in a day, our parallel algorithm computes labeling in less than an hour. However, we see that in FLA road network (which is characterized by large diameter), our parallel algorithm is slower than the sequential algorithm. This is because Bellman-Ford does more work ($O(|E| \cdot |V|)$) than the Dijkstra's algorithm ($O(|V| \log |V| + |E|)$) for constructing an SPT. In the future, we would like to explore pruning in work-efficient shortest path algorithms, such as delta-stepping [26].

Further, we studied only weighted graphs in this paper. For unweighted graphs, pruned Breadth-First Search (BFS) is used to compute PLL [18]. We believe that for BFS, more efficient techniques can be explored to employ inter-tree parallelization. Due to inter-tree parallelization, our current parallel algorithm does not guarantee minimum label size for a given ordering. Reducing label size to imitate the output of the sequential algorithm is another direction that can be pursued in future research.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

ACKNOWLEDGMENT

This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086, National Science Foundation (NSF) under Contract Numbers CNS-1643351 and ACI-1339756 and Air Force Research Laboratory under Grant Number FA8750-15- 1-0185. 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, NSF or AFRL. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] T. Carnes, C. Nagarajan, S. M. Wild, and A. Van Zuylen, "Maximizing influence in a competitive social network: a follower's perspective," in *Proceedings of the ninth international conference on Electronic commerce*. ACM, 2007, pp. 351–360.

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

[3] T. Abeywickrama and M. A. Cheema, "Efficient landmark-based candidate generation for knn queries on road networks," in *International Conference on Database Systems for Advanced Applications*. Springer, 2017, pp. 425–440.

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

[5] 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)*. SIAM, 2014, pp. 147–154.

[6] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Robust distance queries on massive networks," in *European Symposium on Algorithms*. Springer, 2014, pp. 321–333.

[7] K. Lakhotia, R. Kannan, and V. K. Prasanna, "Accelerating pagerank using partition-centric processing," in *2018 USENIX Annual Technical Conference (Usenix ATC)*. USENIX, 2018.

[8] K. Lakhotia, S. Pati, R. Kannan, and V. Prasanna, "GPOP: A cache-and work-efficient framework for graph processing over partitions," *arXiv preprint arXiv:1806.08092*, 2018.

[9] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[10] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/28869.28874

[11] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[12] L. R. Ford Jr, "Network flow theory," RAND CORP SANTA MONICA CA, Tech. Rep., 1956.

[13] 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*, vol. 4, no. 2, pp. 100–107, 1968.

[14] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2005, pp. 156–165.

[15] H. Bast, S. Funke, P. Sanders, and D. Schultes, "Fast routing in road networks with transit nodes," *Science*, vol. 316, no. 5824, pp. 566–566, 2007.

[16] A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Reach for a*: Efficient point-to-point shortest path algorithms," in *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2006, pp. 129–143.

[17] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1338–1355, 2003.

[18] 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*. ACM, 2013, pp. 349–360.

[19] 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*, vol. 7, no. 12, pp. 1203–1214, 2014.

[20] 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*, vol. 11, no. 4, pp. 445–457, 2017.

[21] Savrus, "savrus/hl," Feb 2015. [Online]. Available: https://github.com/savrus/hl

[22] R. Geisberger, P. Sanders, and D. Schultes, "Better approximation of betweenness centrality," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 2008, pp. 90–100.

[23] "Actor collaborations network dataset – KONECT," Apr. 2017. [Online]. Available: http://konect.uni-koblenz.de/networks/actor-collaboration

[24] A.-L. Barabsi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

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

[26] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE