Parallelizing Pruned Landmark Labeling: Dealing with Dependencies in Graph Algorithms

Ruoming Jin*†
Kent State University
Kent, Ohio
rjin1@kent.edu

Zhen Peng*
William & Mary
Williamsburg, Virginia
zpeng01@email.wm.edu

Wendell Wu
Kent State University
Kent, Ohio
wwu12@kent.edu

Feodor Dragan
Kent State University
Kent, Ohio
fdragan@kent.edu

Gagan Agrawal
Augusta University
Augusta, Georgia
gagrawal@augusta.edu

Bin Ren
William & Mary
Williamsburg, Virginia
bren@cs.wm.edu

ABSTRACT

To help compute shortest path distances over large graphs efficiently, 2-hop labeling has emerged as a major tool, with Pruned Landmark Labeling (PPL) as a popular algorithm. This paper demonstrates the first scalable parallel implementation of the PPL algorithm that produces the same results as the sequential algorithm. Based on theoretical analysis, we show how computations on each vertex can be performed in parallel while maintaining correctness, resulting in the Vertex-Centrix PLL (VC-PLL) algorithm. We also show a formulation of this algorithm based on linear algebra and argue why the use of a library based on linear algebra operations will not produce an efficient implementation. Next, we introduce a batched VC-PLL (BVC-PLL) algorithm to reduce the computational inefficiency in VC-PLL. We have carried out a parallel implementation of this method for modern clusters, combining shared memory and distributed memory parallelism, that can efficiently execute on graphs with more than a billion edges. We also demonstrate how BVC-PLL algorithm can be extended to handle directed graphs and weighted graphs and how the version for weighted graphs can benefit from SIMD parallelization.

CCS CONCEPTS

• **Theory of computation → Shortest paths**; • **Mathematics of computing → Paths and connectivity problems**; • **Computing methodologies → Parallel algorithms**;

KEYWORDS

Parallel Graph Algorithms, Dependency Resolving, Multi-level Parallelization

*Both authors contributed equally to this research.
†Corresponding author

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392745

ACM Reference Format:
Ruoming Jin, Zhen Peng, Wendell Wu, Feodor Dragan, Gagan Agrawal, and Bin Ren. 2020. Parallelizing Pruned Landmark Labeling: Dealing with Dependencies in Graph Algorithms. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3392717.3392745

1 INTRODUCTION

Computing the shortest path distance between any two vertices stands out as one of the most fundamental graph operations, with applications ranging from transportation systems (for distance related navigation) [18], social networks/WWW/semantic web (for recommendations and ranking) [9], to knowledge graphs (for concept detection) [67], among others. This operation also serves as the basis for more complex graph analytics and mining operations, such as graph pattern matching [14, 81], distance join processing [62], and centrality computation [7].

However, computing shortest path distances over scale-free complex networks (e.g., massive social and web graphs) remains a challenging problem [3, 35, 53]. To help answer shortest path distance queries on demand, the 2-*hop labeling approach* [15] has emerged as an effective tool. Given a graph, it aims to assign each vertex v a label set $L(v)$, which comprises a list of vertices and their distances to v. Subsequently, given any two vertices u and v, we only need to use their respective label information, $L(u)$ and $L(v)$, to rapidly compute their exact distance.

This approach was made scalable by the *Pruned Landmark Labeling (PLL)* method [3]. This labeling approach adopts a fast greedy process to iteratively consider one vertex at a time (according to certain vertex order) and potentially assigns it to the label sets of other vertices, using a *distance check criterion*. Once the labeling process is done, the results are guaranteed to be *minimum* (or *canonical*) with respect to the given vertex order. In the past few years, a number of studies [2, 17, 44, 57] have further validated and confirmed the scalability of this approach.

Parallel PLL: The original PLL algorithm is inherently sequential; i.e., the algorithm operates one vertex at a time to label the entire graph, and the labeling of a vertex depends on the partial labeling results from earlier processed vertices. In view of this, on the one hand, the original PLL [3] suggested to simply parallelize the BFS labeling of each vertex instead of dealing with inter-vertex labeling

dependency, thus severely limiting parallelism. On the other hand, two recent attempts [20, 58] allow multiple vertices to be simultaneously processed, but do not produce the same compact label as the original PLL.

This paper studies how PLL can be parallelized in a scalable and exact fashion. We make the following **contributions**:

- **Linear Algebra Formulation (Section 2):**. Motivated by developments like GraphBLAS(T) [37] and GraphMat [72] that support graph computations based on a set of linear algebra operators, we show how the original PLL algorithm can be expressed as a series of matrix (and vector) computations. However, we also observe that because of a *masking* operation, the more efficient implementation will be the one based on a vertex-centric approach.
- **Parallel PLL Algorithm (Section 3):** To solve the mismatch between the inherent sequential/dependence of PLL algorithms and the goal of allowing independent operations on each vertex, we present a theoretical result and use it to develop a new VC-PLL algorithm that utilizes VC to parallelize PLL and is guaranteed to produce the same labels as original PLL.
- **Batched Vertex-Centric PLL (Section 4):** To deal with the limitations of VC-PLL, we introduce a batched VC-PLL (BVC-PLL) algorithm that largely preserves the same vertex computation function while reducing the costs of message (label) broadcasting and remote memory access.
- **Efficient Parallelization and Generalization (Section 5):** We combine intra-node (shared memory) and inter-node (distributed memory) parallelism, resulting in an implementation that can handle graphs with more than 1 billion edges. In addition, we show how BVC-PLL can be extended to handle directed graphs and weighted graphs, and how the extension for weighted graphs can benefit from SIMD parallelism.

In our **experimental study (Section 6)**, we show that the sequential BVC-PLL can run more than 2× faster than the original PLL (both using one single thread). The parallel BVC-PLL also demonstrates good scalability and obtains an average speedup of 6.68× over sequential BVC-PLL on a 20-core shared memory machine and a speedup up to 11.85× on a 16-node distributed cluster over 1-node version. We also demonstrate that the shared memory and distributed memory combined BVC-PLL gains good scalability for large graphs with over 1 billion edges. We finally extend BVC-PLL to process weighted graph with the help of SIMD parallelism, achieving up to 1.92× speedup over PLL (both with a single thread), and achieving up to 15.68× speedup in going from 1 to 20 threads.

2 2-HOP LABELING AND PLL

This section further describes the 2-hop labeling problem and the PLL algorithm that forms the basis for our work on parallelization. We also describe how this algorithm can be viewed as a series of linear algebra operations.

2.1 2-Hop Labeling

The 2-hop labeling algorithm [15], which was pioneered by Cohen *et al.* [15], provides an efficient scheme to answer on-demand shortest distance queries. It assigns each vertex u in an (undirected) graph a label set $L(u)$ such that for any two vertices u and v, their distance can be computed using only their respective label sets. Formally, we compute $L(u)$ and for each $h \in L(u)$, the corresponding distance from u, i.e, $d(h,u)$. Table 1 illustrates a 2-hop labeling of the undirected graph G that is shown in Figure 1a.

Algorithm 1 PLL for $G = (V, E)$ with Order π

```
1:  for all u ∈ V {following order π from high to low} do
2:    Queue Q = {(u, 0)} {BFS process to use u for labeling}
3:    while Q is not empty do
4:      (v, d(u, v)) ← Q.pop()
5:      if d(u, v) < min_{h∈L(u)∩L(v)} {d(u, h) + d(h, v)} then
6:        Add (u, d(u, v)) into L(v)
7:        for all v′ ∈ v's neighbor do
8:          if v′ is unvisited by u and π(u) < π(v′) then
9:            Add (v′, d(u, v) + 1) to Q
10:         end if
11:       end for
12:     end if
13:   end while
14: end for
```

Formally, the shortest path distance query $Dis(\cdot,\cdot)$ between any two vertices u and v can be answered as:

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

Thus, 2-hop labeling can answer distance queries efficiently by traversing two lists of vertices, with an operation similar to merge sort. As an example, in Table 1, the distance between nodes A and B can be computed as 2 by first identifying that D and I are common vertices in their label sets, and subsequently that the distance through D is the shortest (1+1).

Over a decade, numerous efforts [1, 12, 13, 24, 34, 63, 64] largely failed in making 2-hop labeling practical on large real-world graphs. The pruned landmark labeling algorithms [3, 36] are considered a major breakthrough in solving this problem and are the focus of our work.

2.1.1 *Pruned Landmark Labeling (PLL).*

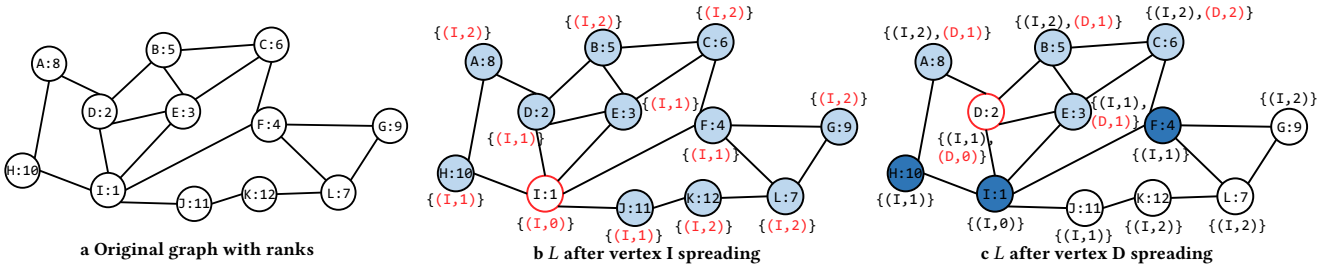
The main idea of this algorithm is to use a total ordering of all vertices (finding such an order optimally is NP-hard [75], but heuristics are feasible) for labeling. Given such a total order π of vertices, the pruned landmark labeling (PLL) [3] assigns each vertex, based on the order $(\pi(v_1) < \pi(v_2) < \cdots < \pi(v_n))$, to the labels of other vertices in the graph following a BFS process. Note that when $\pi(v_i) < \pi(v_j)$, we say that v_i has the higher rank. As PLL assigns the vertex u with the rank $\pi(u)$ as a label to a lower ranked vertex v, it needs to check if u is the highest ranked vertex in the shortest paths between u and v (P_{uv}). This can be done by checking

$$d(u,v) < d(v,h) + d(h,u), \text{ for all } h \in L(u) \cap L(v).$$

Intuitively, this is ensuring that the distance between u and v cannot be recovered by a certain higher ranked vertex. When the condition above does not hold, u will be *pruned* by v (i.e., is not added into the label of v and will not be further expanded from v) during the labeling process.

Figure 1 illustrates the processing associated with the two highest ranked vertices (I and E) for the graph in Figure 1a. Rank (or order) of each vertex is explicitly shown following the vertex ID.

Algorithm 1 sketches the labeling process for an undirected graph. Note that $d(u,v)$ in the algorithm is the distance computed



a Original graph with ranks **b** L after vertex I spreading **c** L after vertex D spreading

Figure 1: 2-Hop Labeling and PLL Example: spreading vertex is marked with red; light shadow vertices have already received the label spread (colored in red); pruning happens in dark shadow vertices; white vertices are not accessed by the vertex being spread because of pruning.

Table 1: 2-hop labeling for Graph G

Vertex	Labels
A	$\{(A,0),(D,1),(I,2)\}$
B	$\{(B,0),(D,1),(E,1),(F,2),(I,2)\}$
C	$\{(C,0),(B,1),(E,1),(F,1),(I,2),(D,2)\}$
D	$\{(D,0),(I,1)\}$
E	$\{(E,0),(D,1),(I,1)\}$
F	$\{(F,0),(I,1)\}$
G	$\{(G,0),(F,1),(L,1),(I,2)\}$
H	$\{(H,0),(A,1),(I,1)\}$
I	$\{(I,0)\}$
J	$\{(J,0),(I,1),(L,2)\}$
K	$\{(K,0),(J,1),(L,1),(F,2),(I,2)\}$
L	$\{(L,0),(F,1),(I,2)\}$

by the BFS process, which may not be the exact distance between u and v (due to the pruning effect). But the recorded distance in the label (Line 6) is always exact (since it can travel through all the shortest paths starting from u reaching to v). It can be proved that the results are guaranteed to be *minimum* (or *canonical*) with respect to the given vertex order.

2.2 A Linear Algebra View of PLL

It has been well known that a large number of graph algorithms can be stated as operations on (sparse) matrices [25, 38]. Multiplication of a sparse matrix with a vector (SpMV) and multiplication of two sparse matrices (SpGEMM) are the operations most commonly used. Multiple recent efforts have focused on using the work on optimizing (and parallelizing) SpMV and SpGEMM operations as the basis for graph processing [37, 72].

We have examined how PLL can also be viewed as a series of linear algebra operations. Our examination, however, shows that PLL requires more than SpMV and SpGEMM and therefore is better parallelized by taking a vertex-centric view of the computations.

Let A be the adjacency matrix for graph G, where $A[u,v] = Dis(u,v)$. Let I be the identity matrix and let I_i be the i-th column of identity matrix I.

First, it is well known (see, for example [38]) that the following equation computes the shortest distances from a given vertex i, denoted as a vector $\vec{y}_i$:

$$\vec{y}_i = (I_i^T(I + A + A^2 + \cdots + A^d))^T = (I + A + A^2 + \cdots + A^d)^T I_i$$

where d is the diameter of the graph. We also denote

$$A^* = (I + A + A^2 + \cdots + A^d)^T.$$

Note that for any matrix M, M_i is the i-th column of M ($M_i = MI_i$).

Now, we represent the original PLL algorithm as a series of linear algebra operations. Following the order used in the PLL algorithm, let $\vec{x}_i$ be the vector recording the distance of all vertices to the i-th ordered vertex in the distance labeling:

$$\vec{x}_i[j] = d(j,i), v_i \in L(v_j); \quad \vec{x}_i[j] = +\infty, v_i \notin L(v_j).$$

Let $\ominus$ be the *generalized element-wise masking*: $\vec{x}[i] \ominus \vec{y}[i] = \infty$ if $\vec{x}[i]$ is larger than or equal to $\vec{y}[i]$ and $\vec{x}[i] \ominus \vec{y}[i] = \vec{x}[i]$ if $\vec{x}[i]$ is smaller than $\vec{y}[i]$. Given this, for two column vectors $\vec{x} = \{x_1, \cdots, x_n\}^T$ and $\vec{y} = \{y_1, \cdots, y_n\}^T$, $\vec{x} \ominus \vec{y} = \{x_1 \ominus y_1, \cdots, x_n \ominus y_1\}^T$. Then the labeling PLL essentially utilizes the following equation to assign the labeling:

$$\begin{aligned}
\vec{x}_1 &= A^* I_1 \\
\vec{x}_2 &= A^* I_2 \ominus (\vec{x}_1 \vec{x}_1^T) I_2 \\
\vec{x}_3 &= A^* I_3 \ominus (\vec{x}_1 \vec{x}_1^T + \vec{x}_2 \vec{x}_2^T) I_3 \\
&\cdots \\
\vec{x}_n &= A^* I_n \ominus (\textstyle\sum_{i=1}^{n-1} \vec{x}_i \vec{x}_i^T) I_n
\end{aligned}$$

Here, $\vec{x}_i \vec{x}_i^T$ generates the matrix recording the shortest distance between any two vertices via vertex i. Thus, $(\vec{x}_i \vec{x}_i^T) I_j$ corresponds to the shortest distance from any vertex to vertex j via vertex i.

In Algorithm 1, line 2–13 basically provides an efficient procedure to generate the distance label $\vec{x}_i$. Especially, Line 5 can be considered as the on-demand implementation of the generalized masking operation (without explicitly producing the masking vector, and testing the condition as needed).

As we can see above, though we have been able to map PLL to a set of linear algebra operations, the formulation involves more than SpMV and SpGEMM algorithms. Particularly, the use of linear algebra libraries will require materialization of the expression $(\sum_{i=1}^{k-1} \vec{x}_i \vec{x}_i^T) I_k$ during the k-th step, which can be much more expensive as compared to performing the masking operation (or pruning) on demand in Line 5 of Algorithm 1. As a result, we examine other models for parallelizing PLL.

3 PARALLELIZATION OF PLL

This section first gives background on vertex-centric model. It then states the challenges in parallelizing PLL using this model. We state an important theoretical result and then proceed to develop an initial (basic) parallel algorithm for PLL.

3.1 Vertex-Centric (and Other) Models

Graph algorithms have been frequently parallelized by thinking of independent computations on each vertex. This was the basis

Algorithm 2 Vertex-Centric (Scatter-Gather) (G=(V,E))

```
Initialize ActiveVertices ⊆ V
while ActiveVertices is not empty do
    {Scatter Phase:}
    for all a ∈ ActiveVertices do
        a.Scatter(a.edges) : {for each edge e = (a, v) of a, send
        message(a, e, v) to v}
    end for
    ActiveVertices ← ∅
    {Gather Phase:}
    for all v Received Message do
        v.Gather(v.messages): {vertex compute using received messages
        and update its value}
        ActiveVertices ← {v : v.value is updated}
    end for
end while
```

for the seminal vertex-centric programming model proposed by the Pregel paper [48], and many other parallel graph processing system research efforts [16, 27, 28, 39, 41, 46, 47, 52, 54, 66, 68, 74, 77]. Though other models have been used, including the recent projects that use (sparse) linear algebra operations (for example, GraphBLAS(T) [37] and GraphMat [72]), we find vertex-centric model to be a good fit for approaching parallelization of PLL.

In the vertex-centric model, parallel graph processing is viewed as an iterative process, where each iteration processes the set of *active* vertices. For each vertex in this set, we perform computations based on the data from incoming and/or outgoing edges together with the local vertex data, and then update the values/state associated with the vertex. The vertices that record a change in their local state become the active vertices for the next iteration. The parallelization typically uses the Bulk Synchronous Parallel (BSP) execution [73] and requires a global synchronization at the end of each iteration. The entire process terminates once the set of active vertices becomes empty.

A high level abstraction of the vertex-centric computation based on a scatter-gather model [48, 61] is sketched in Algorithm 2. Each vertex computation is described through two functions: 1) the *Scatter* function, which describes how each vertex uses its vertex value and edge value to propagate a message to its neighbors; and 2) *Gather* function, which describes how each vertex computes a new value based on its original value and all the new messages it received.

Various more advanced parallel graph programming models are proposed to further refine the vertex-centric model. This includes the GAS (Gather-Apply-Scatter) [27] and the push and pull models [6, 55, 68], where the goal is to better fit the computational and communication patterns of graph processing. There is also work on generalizing the model to finer granularity, such as the edge-centric model [61], or to coarser granularity, such as path- or subgraph-centric [59], and k-step neighborhood [8, 40] models.

Finally, as we stated earlier, there has been recent interest in the use of linear algebra libraries (and thus using existing methods for parallelizing them). However, because of the masking (or pruning) operation in PLL, a linear algebra based implementation is unlikely to be efficient.

In this paper, we focus our efforts on the vertex-centric model, with resulting code implemented efficiently through the use of MPI and OpenMP. Exploration of the use of other models and whether there can be a programmability or performance benefit from them is a subject for future work.

3.2 Vertex-Centric Approach and PLL

Recall that the PLL algorithm (Alg. 1) iterates following the vertex rank (order): at the i^{th} round, the vertex u with rank $\pi(u) = i$ will be distributed to all other vertices in the graph using a BFS process. The key condition to add u into the label of v is that the distance between u and v cannot be recovered by earlier processed vertices. The main challenge in parallelizing PLL is that adding a vertex u of rank $\pi(u)$ to another vertex v in the BFS traversal seems to be *dependent* on the completion of labeling of all higher ranked vertices in order to apply the distance check. In comparison, for parallelization with the vertex-centric model, we would like to distribute all vertices to their neighbors simultaneously for vertex labeling. This requirement seems to be in conflict as there is no guarantee that the higher vertices can finish the distribution before lower rank ones. Indeed, as we mentioned earlier, all the existing attempts have all failed to parallelize inter-vertex labeling while preserving the canonical labeling criterion [3, 20, 58].

We address this problem through the following important result.

Theorem 1. *Assume we spread all vertices simultaneously into the graph (starting by sending each vertex to their neighbors), and we do the spreading iteration by iteration following the vertex-centric programming model. Let us consider a vertex u with the rank $\pi(u)$ that reaches the vertex v at the j-th iteration. Then if there is a vertex w with the following properties: 1) with a higher rank than u ($\pi(w) < \pi(u)$); 2) with a shorter distance to v and u ($d(w,v) < d(u,v)$ and $d(w,u) < d(u,v)$), and 3) being recorded as a label of v ($w \in L(v)$) and u ($w \in L(u)$), then, w must be able to reach both u and v before the j-th iteration ($d(u,v)$ steps).*

Proof Sketch: We first note that conditions 1 and 3 ensure w cannot be pruned by other vertices with higher ranks between w and u (and v). Then vertex w can reach u and v in less than j iterations as $d(w,u) < d(u,v) = j$ and $d(w,v) < d(u,v) = j$. (By way of contradiction, if we assume w cannot reach u (or v) in j iterations, it either has a distance longer than j or it is pruned, i.e, there is another vertex w' with a higher rank and located on the shortest path between w and u. In the latter case, w cannot be recorded as a label in u (or v).) □

The Theorem implies that even when the spreading process is parallelized across the node, we can correctly determine if u should be added to $L(v)$ by testing if there is any other vertex, say w, with a higher rank than u ($\pi(w) < \pi(u)$), which can produce an equal or shorter distance, i.e., $d(u,v) \geq d(u,w) + d(w,v)$.

Recall that the distance check condition for canonical labeling criterion requires not only the labeling of higher ranked vertices h to be completed before the distance check between u and v, but also their distances $d(u,h)$ and $d(h,v)$ to be smaller than $d(u,v)$. The latter condition is the key to the parallelization of PLL.

The main result above can also be stated (or derived) through the linear algebra formulation. Recall from the last section that the original PLL computes the distance labels one vertex at a time $(\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_n)$. To parallelize it, we will generate labels for all vertices with the same distance in the same batch. The details of the

process are as follows. At the iteration 0 (initialization): for any vertex i, let $\vec{z}_i = I_i$ and $\vec{y}_i = z_i$ ($\vec{z}_i$ is the newly generated distance vector recording all vertices' distance to vertex i if they record v_i at the latest iteration. $\vec{y}_i$ is the "accumulated" vector for all vertices recording their distance to vertex i if they record v_i up to the latest iteration).

Next, at iteration 1, we have

$$\begin{array}{rcl}\vec{z}_1 & = & A^T\vec{z}_1 \ominus \{\vec{y}_1\}\\ \vec{z}_2 & = & A^T\vec{z}_2 \ominus \{\vec{y}_2 + \vec{y}_1\vec{y}_1^T I_2\}\\ \vec{z}_3 & = & A^T\vec{z}_3 \ominus \{\vec{y}_3 + (\vec{y}_1\vec{y}_1^T + \vec{y}_2\vec{y}_2^T)I_3\}\\ \cdots & & \\ \vec{z}_n & = & A^T\vec{z}_n \ominus \{\vec{y}_n + (\sum_{i=1}^{n-1}\vec{y}_i\vec{y}_i^T)I_n\}\end{array}$$

and then for all i, $\vec{y}_i = \vec{y}_i + \vec{z}_i$. Here, $\vec{y}_i\vec{y}_i^T$ generates the matrix recording the distance between any two vertices via vertex i (as their distance label), and $(\vec{y}_i\vec{y}_i^T)I_j$ corresponds to the distance from any vertex to vertex j via vertex i (they all use vertex j in their distance label). Using these values, we will repeat the above computations until no new label is generated.

We can easily prove the following result:

Theorem 2. *When the above algorithm stops, we have*

$$\vec{y}_i = \vec{x}_i, \text{for all } i.$$

Proof Sketch: We will prove this by induction. First, let us define both $\vec{x}_i$ and $\vec{y}_i$ by iteration. Let $A^{*k} = (I + A + A^2 + \cdots + A^k)^T$ for k-th iteration ($k \le d$, d is the diameter of the graph). We first observe that

$$\begin{array}{rcl}\vec{x}_1^k & = & A^{*k}I_1\\ \vec{x}_2^k & = & A^{*k}I_2 \ominus (\vec{x}_1^k\vec{x}_1^{kT})I_2\\ & = & A^{*k}I_2 \ominus (\vec{x}_1^{k-1}\vec{x}_1^{k-1T})I_2\\ \vec{x}_3^k & = & A^{*k}I_3 \ominus (\vec{x}_1^k\vec{x}_1^{kT} + \vec{x}_2^k\vec{x}_2^{kT})I_3\\ & = & A^{*k}I_3 \ominus (\vec{x}_1^{k-1}\vec{x}_1^{k-1T} + \vec{x}_2^{k-1}\vec{x}_2^{k-1T})I_3\\ \cdots & & \\ \vec{x}_n^k & = & A^{*k}I_n \ominus (\sum_{i=1}^{n-1}\vec{x}_i^k\vec{x}_i^{kT})I_n\\ & = & A^{*k}I_n \ominus (\sum_{i=1}^{n-1}\vec{x}_i^{k-1}\vec{x}_i^{k-1T})I_n\end{array}$$

Here, the above equation can be observed as the set $\vec{x}_i^k \ominus \vec{x}_i^{k-1}$ records what the vertex i can reach in exactly k steps, and this set will not help prune $\vec{x}_j^k$ (what vertex j can reach within k steps) for $j > i$.

Next, let us look at $\vec{y}_i^k$:

$$\begin{array}{rcl}\vec{y}_1^k & = & \vec{y}_1^{k-1} + A^T\vec{z}_1^{k-1} \ominus \{\vec{y}_1^{k-1}\}\\ \vec{y}_2^k & = & \vec{y}_2^{k-1} + A^T\vec{z}_2^{k-1} \ominus \{\vec{y}_2^{k-1} + \vec{y}_1^{k-1}(\vec{y}_1^{k-1})^T I_2\}\\ \vec{y}_3^k & = & \vec{y}_3^{k-1} + A^T\vec{z}_3^{k-1} \ominus \{\vec{y}_3^{k-1} + (\vec{y}_1^{k-1}(\vec{y}_1^{k-1})^T + \vec{y}_2^{k-1}(\vec{y}_2^{k-1})^T)I_3\}\\ \cdots & & \\ \vec{y}_n^k & = & \vec{y}_n^{k-1} + A^T\vec{z}_n^{k-1} \ominus \{\vec{y}_n^{k-1} + (\sum_{i=1}^{n-1}\vec{y}_i^{k-1}(\vec{y}_i^{k-1})^T)I_n\}\end{array}$$

Now, when $k = 0$ (initialization), the $\vec{y}_i^0 = \vec{x}_i^0$, for all i trivially holds. Now, we consider when k holds to be true, we will derive $k+1$ to be true. To prove this, again, we can start with $i = 1$, and we can easily observe: $\vec{y}_1^{k+1} = \vec{x}_1^{k+1}$. Now, assume $j \le i$ are all true, then, let us consider $i + 1$:

Algorithm 3 VC-PLL for $G = (V, E)$ with Order π

{Init.: ($L(v)$: label; $\delta L(v)$: new label from each iteration)}
1: ActiveVertices$\leftarrow V$; $\forall v \in V, \delta L(v) \leftarrow \{(v, 0)\}, L(v) \leftarrow \delta L(v)$
2: **while** ActiveVertices $\neq \emptyset$ **do**
{Scatter Phase:}
3: **for all** $a \in$ ActiveVertices **do**
a.Scatter(a.edges):
4: **for all** $(a, v) \in a.edges$ **do**
5: for all $(u, d(u, a)) \in \delta L(a)$, when $\pi(u) < \pi(v) \wedge u \notin L(v)$: send $(u, d(u, a) + 1)$ to $v.messages$
6: **end for**
7: **end for**
ActiveVertices $\leftarrow \emptyset$
{Gather Phase:}
8: **for all** $v \in V$: $v.messages \neq \emptyset$ {Received Message} **do**
v.Gather(v.messages):
9: $\delta L(v) \leftarrow \emptyset$
10: **for all** unique $(u, d(u, v)) \in v.messages$ **do**
11: **if** $d(u, v) < \min_{h \in L(u) \cap L(v)}\{d(u, h) + d(h, v)\}$ **then**
12: Add $(u, d(u, v))$ to $\delta L(v)$
13: **end if**
14: **end for**
15: If $\delta L(v) \neq \emptyset$: $L(v) \leftarrow L(v) \cup \delta L(v)$; Add v to ActiveVertices
16: **end for**
17: **end while**

$$\begin{array}{rcl}\vec{y}_{i+1}^{k+1} & = & \vec{y}_{i+1}^k + A^T\vec{z}_{i+1}^k \ominus \{\vec{y}_{i+1}^k + (\sum_{l=1}^{i}\vec{y}_l^k(\vec{y}_l^k)^T)I_l\}\\ & = & \vec{x}_{i+1}^k + A^T(\vec{x}_{i+1}^k \ominus \vec{x}_{i+1}^{k-1}) \ominus \{\vec{x}_{i+1}^k + (\sum_{l=1}^{i}\vec{x}_l^k(\vec{x}_l^k)^T)I_l\}\\ & = & A^{*k}I_{i+1} \ominus \{(\sum_{l=1}^{i}\vec{x}_l^k(\vec{x}_l^k)^T)I_l\}\\ & = & \vec{x}_{i+1}^{k+1}.\end{array}$$

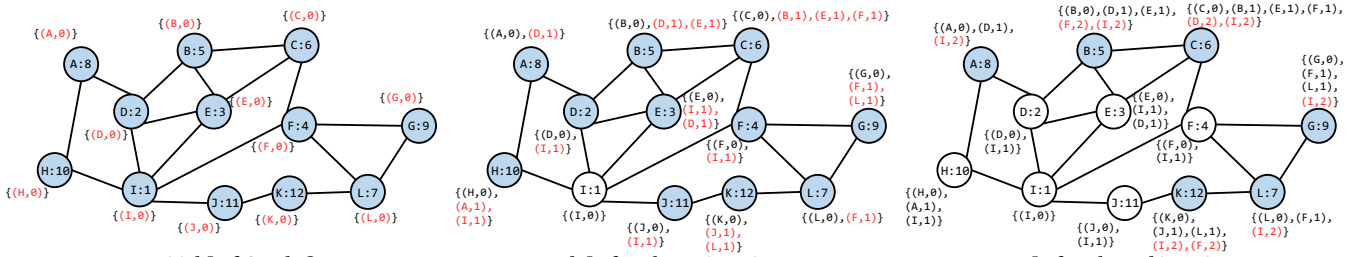
□

We note that in each iteration, all $\vec{z}_i$ can be computed simultaneously (as $\vec{z}_i$ now only depends on its own state and $\vec{y}_i$, which is computed from earlier iteration). Thus, we can parallelize PLL. However, we still do not have an efficient implementation based on linear algebra. More specifically, we have the requirement of materializing $(\sum_{i=1}^{k-1}\vec{y}_i\vec{y}_i^T)I_k$ in the k-th step, which will be very expensive.

3.3 Vertex-Centric Parallel Implementation

Algorithm Description: Algorithm 3 sketches the main process of performing PLL based on the vertex-centric computation model (Algorithm 2). In the *Initialization* phase, all vertices are active initially ($ActiveVertices = V$). For each vertex v, $L(v)$ records the partial label and $\delta L(v)$ records the new label being generated at each iteration. Initially, every vertex v records itself and distance 0 (any vertex reaches itself in zero steps). The main computation alternates between the *Scatter* phase and *Gather* phase and will continue until no new active vertices exist (Lines 2 to 17):

1) *Scatter* phase (Lines 3 to 7, also referred to as the push model): all active vertices with new labels perform a vertex *Scatter* function (Lines 4 to 6): each sends their new labels with the updated distance: $(u, d(u, a)) \in \delta L(a) \rightarrow (u, d(u, a) + 1)$ to all their neighbors (Line 5) with two conditions: the rank of vertex u needs to higher than v (otherwise, it will be pruned) and it has never been added to the label of v.



a Initial L of Graph G **b L after the 1st iteration** **c L after the 2nd iteration**

Figure 2: A VC-PLL Example: light shadow vertices got new labels (colored in red) in this iteration and will spread them in the next iteration.

2) *Gather* phase (Line 8-16): all vertices that receive a new message ($v.messages \neq \emptyset$) perform a vertex *Gather* function (Lines 9-15): For a vertex v, it traverses all its received messages (distance label from its neighbors), and for each *unique* vertex $(u, d(u,v))$ across the set of messages, it confirms the distance check for the canonical labeling criterion: for a distance label message $(u, d(u,v))$, $d(u,v)$ must be smaller than the distances via any existing labels (L), i.e., $d(u,v) < \min_{h \in L(u) \cap L(v)} d(u,h) + d(h,v)$ (Line 11). If this true, it will be added into $\delta L(v)$. Once $\delta L(v)$ is computed and it is not empty, we will add it into $L(v)$ and add v to ActiveVertices (Line 15). Note that we need to identify unique vertices in the step above, because two neighbors may send the same vertex u.

Running Example: Figures 2 illustrates the first 2 iterations of label spreading in VC-PLL, where the labels in red denote newly generated labels δL. At each iteration, $L(v)$ is simply the union of all $\delta L(v)$ from all earlier iterations.

3.4 Theoretical Properties

This section explains two key properties of the proposed VC-PLL algorithm.

Property I: Correctness. Theorem 3 proves that VC-PLL produces the same label as PLL.

Theorem 3. *VC-PLL (Algorithm 3) produces the minimum labeling size (or canonical hub labeling) [4] given a vertex order π.*

Due to the page limitation, please refer to our supplementary materials for the detailed proofs[1].

Property II: Time Complexity. Following the approach in PLL [3], we can obtain a theoretical upper-bound of VC-PLL's time complexity.

Theorem 4. *Assuming graph G with a tree-decomposition [60] of tree-width w, then there is a vertex order π, in which the VC-PLL takes $O(w|E| \log |V| + w^2|V|(\log |V|)^2)$ time (the same as that of PLL [3]).*

Proof Sketch: The main idea is to utilize a recursive centroid extraction procedure to reorganize the tree decomposition: the centroid bag of the tree can break the tree decomposition into disjoint components where each disjoint components has no more than half of the tree bags. The centroid bag will become of the root of the tree, and its children will be the centroid bag of the disjoint subtrees. Given this, the new tree's height is at most $\log |V|$. We then can *order the vertices in the graph based on their highest node in the tree, and if two vertices have the same height, they can break order arbitrarily.* Since each node in the tree corresponds to a graph separator, the maximal label size of any vertex is $w \log |V|$ (for a vertex in the leaf bag, all the vertex in its ancestors including its own bag of vertices can be added into the label). Then, the time complexity of generating distance labels is bounded by $O(w \log |V||E|)$. The time complexity of distance check is $O(w^2|V| \log^2 |V|)$. □

3.5 Limitations of VC-PLL

Sequential Performance Comparison: We implemented Alg. 3 (VC-PLL) and tested its performance on the DBLP graph (that is introduced in Section 6) against PLL using a single thread. We found that it has poor performance with a total execution time of 13,583 seconds compared to less than 100 seconds for PLL! It does not fare well against PLL in other graphs either.

Basic algorithm and performance analysis reveal that VC-PLL introduces additional computational costs due to extra labeling spreading and distance testing. 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 in two consecutive iterations. Therefore, VC-PLL introduces redundant distance labeling messages, which may also lead to redundant distance checks. Furthermore, individual distance checks in PLL can be much faster due to the reuse of $L(u)$ in an array or hash-table representation.

Thus, our question is: *can VC-PLL overcome its limitations and reduce those additional costs (message spreading and remote memory access)?*

4 BATCHED VERTEX-CENTRIC ALGORITHM

To deal with the performance inefficiency of VC-PLL discussed in the last section, we introduce a new algorithm. Specifically, here, the batches of the vertices are formed according to the rank of each vertex. For instance, the top $1K$ vertices form the first batch, and the next $1K$ vertices form the second batch, etc. Batches are processed in sequence with the vertices within each batch being processed in parallel. The reason to use batch (as we will discuss later in more details) is by using batch, we can 1) effectively and efficiently eliminate redundant message passing, and 2) significantly improve remote vertex memory access (as only the vertices in the batch need to be accessed remotely).

BVC-PLL Algorithm: Algorithm 4 sketches the batched Vertex-Centric algorithm for PLL, referred to as BVC-PLL. Specifically, here, the batches of the vertices are formed according to the rank

[1] https://drive.google.com/open?id=1VeNgaKqr4G2Flxov2v8gIUKdqJ8maB9z

Algorithm 4 BVC-PLL for $G = (V, E)$ with Order π

```
    {Init.: (L(v): label; δL(v): new label from each iteration)}
1:  ∀v ∈ V, L(v) ← ∅, C(v) ← ∅
2:  Split V into equal-size batches: B_1, B_2, ··· B_T where B_i include the
    vertices with rank (i − 1) × |V|/T + 1 to i × |V|/T
3:  for all B_i : i = 1 to T {Labeling in Batch} do
4:    ActiveVertices ← B_i; ∀u ∈ B_i, δL(u) ← {(u, 0)}, L(u) ← L(u) ∪
      δL(u), and map L(u) to Hashtable H(u)
5:    while ActiveVertices ≠ ∅ do
        {Scatter Phase:}
6:      for all a ∈ ActiveVertices do
          a.Scatter(a.edges):
7:        for all (a, v) ∈ a.edges do
8:          for all (u, d(u, a)) ∈ δL(a), when π(u) < π(v) ∧ u ∉ C(v):
            flag u in C(v) and send (u, d(u, a) + 1) to v.messages
9:        end for
10:     end for
        ActiveVertices ← ∅
        {Gather Phase:}
11:     for all v ∈ V : v.messages ≠ ∅ {Received Messages} do
          v.Gather(v.messages):
12:       δL(v) ← ∅
13:       for all (u, d(u, v)) ∈ v.messages do
14:         if d(u, v) < min_{h∈L(u)∩L(v)} {d(u, h) + d(h, v)} then
15:           Add (u, d(u, v)) to δL(v)
16:         end if
17:       end for
18:       If δL(v) ≠ ∅: L(v) ← L(v) ∪ δL(v); Add v to ActiveVertices
19:       If v ∈ B_i: Add δL(v) to H(v)
20:     end for
21:   end while
22:   ∀v ∈ V, C(v) ← ∅
23: end for
```

of each vertex (Line 2). The earlier processed batch consists of the vertices with higher ranks (Line 3). BVC-PLL labels vertices one batch at a time and for assigning the labels in each batch, the vertex-centric computation in VC-PLL is followed (Lines 5-21) – more specifically, the Scatter Phase and Scatter function, Gather Phase and Gather function is preserved with only minor revisions for dealing with message passing and remote memory access. Each vertex v is associated with a candidate-bit vector $C(v)$. Its length is equal to the batch size. It will be initialized for each batch (Lines 1 and 22). During the Scatter phase, for any vertex a to send a message $(u, d(u, a) + 1)$ to its neighbor v, it will check if u is sent to v before ($u \notin C(v)$, Line 8). This corresponds to the *unvisited* flag in the original PLL. Due to the atomic compare-and-swap operation, it can guarantee only one message from u is being sent to v and thus help resolve the redundant distance labeling generation problem (in Subsection 3.5).

Each vertex u in the batch B_i will map its existing label $L(u)$ to a hash-table (or array) $H(u)$ at the beginning of vertex-centric computation (Line 4). Since the new label of u may be generated during the labeling process, we will map the new label $\delta L(v)$ to $H(v)$ when the update is available (Line 19). Given this, the distance check (in Line 14) only needs to go through $L(v)$, and thus has the same distance check cost as the original PLL.

Next, we discuss two key optimization techniques which leverage the batch processing to reduce the additional computational costs from VC-PLL:

Using Bit Operation for Efficient Message Passing and Filtering: In each batch processing step, an active vertex only processes up to *batch_size* unique labels. Based on this important observation, we can use a compact bit-vector data structure called *candidate bit-vector* for efficient message filtering. The basic idea is as follows. Each active vertex maintains a candidate bit-vector with the length of *batch_size* bits, each bit corresponding to a vertex in the batch (e.g., if the *batch_size* is 1K, such candidate bit-vector is only 128 bytes). If a vertex u in the current batch is sent to a vertex v, then its corresponding bit in the candidate bit-vector of v is set. Note that the use of bit-vectors also allows atomic *compare-and-swap* operation in the shared memory setting. Note that without batch processing, we have to consider doing an expensive list merge for handling message passing and aggregation (as the scatter and gather functions in VC-PLL for distance label messaging and processing, respectively).

Improving Data Locality for Remote Vertex Memory Access: Simply speaking, only the vertices in the current processing batch can be accessed remotely during the vertex-centric computation. Because the number of vertices in each processing batch is limited, we can use a compact data structure such as an array or hash-table to store their labels for efficient $O(1)$ access (similar to what is done in PLL for each processed vertex in distance checks).

Correctness: It is easy to see that *BVC-PLL (Algorithm 4) produces the minimal labeling given a vertex order π*: the distance criterion ($u \in L(v)$ if u has the highest rank in P_{uv}) is maintained as BVC-PLL can assign u to $L(v)$ at u's batch correctly (Theorem 3) following the batch processing order. Another interesting property is that when the batch size reduces to one, i.e., when we process one vertex at a time, then BVC-PLL behaves exactly the same as the original PLL [3].

Complexity: We note that introducing and using bit-vector $C(v)$ for each vertex v and $H(u)$ for each processing batch vertex u does not introduce additional time complexity compared with PLL. PLL uses only one bit for each vertex v as the *visited* flag and one $H(u)$ for distance check, whereas BVC-PLL simply utilizes a group of them at the same time. Thus, the time complexity results of Theorem 4 hold for BVC-PLL as well.

5 VARIANTS AND PARALLEL IMPLEMENT.

5.1 Generalization

Directed Graphs: For a directed graph, each vertex v is assigned with two labels $L_{in}(v)$ and $L_{out}(v)$. VC-PLL and BVC-PLL can be easily extended to handle directed graphs by considering these as separate computations. Specifically, in the Scatter function, the new labels δL_{in} and δL_{out} will be sent out along the outgoing edges and incoming edges, respectively. In the Gather function, there will be two message queues: one for candidate vertices in L_{in}, and another for those in L_{out}. The labels generated by this algorithm will be canonical. The computational complexity analysis in Subsection 3.4 holds for directed graphs as well.

Weighted Graphs: The direct application of VC-PLL and BVC-PLL (by changing $d(u, v) + 1$ to $d(u, v) + w_e$ where w_e is the edge weight) on weighted graphs can produce a 2-hop labeling; but it may not be a canonical labeling. This is because unlike unweighted graphs, the iteration on the vertex-centric model will not be in sync with the

distance between two vertices. For instance, when vertex u reaches v in two iterations, their distance may be larger than a path via vertex w with a higher rank, but w may take more than 2 iterations to reach v and u. Given this, we cannot use the partial label $L(u)$ at an arbitrary iteration to fully determine if vertex v is a true or final label for u anymore. Thus, adding vertex v into u's partial label $L(u)$ or $\delta L(u)$ (using the partial labels in the weighted graph) may lead to unnecessary vertices being spread in the networks. To deal with this problem, at the end of each batch processing (Line 22 in BVC-PLL), we can perform a distance recheck using only the labels from the batch. Since the hash tables of the labeling vertices in the batch are still in the memory, this recheck can be quite efficient.

5.2 Parallel Implementation Issues

Shared Memory Implementation: The BVC-PLL computation (as shown in Algorithm 4) can be easily parallelized with coarse-grained threads. The computation of each batch uses vertex-centric processing (line 5 to line 21) that consists of two parallel phases: (*Scatter* and *Gather*), with an implicit synchronization between them. In each phase, each thread processes a chunk of active vertices with dynamic scheduling to achieve load balance. In shared memory implementation, we use OpenMP to parallelize each batch, manage the workload of each thread dynamically, and expand active vertices.

Extension to Distributed Platforms: BVC-PLL's shared memory parallel implementation was also extended to distributed platforms using MPI. Because BVC-PLL processes nodes based on an order, and this order is, in turn, based on degrees, we use the following process to maintain load balance. After sorting vertices according to their rank (i.e. degrees) from high to low, vertices are assigned to nodes in a *zigzag round-robin* way. For instance, assuming there are p nodes in total, p consecutive vertices are assigned to node 0 to node $p-1$, respectively; then the next p consecutive vertices are assigned to node $p-1$ to node 0, and so on.

In maintaining information for each node, we follow the *master-mirror* notion (as also used by PowerGraph [27]). Every edge is placed on the computer node that owns its destination vertex (as a master) [70]. As described above, a vertex is assigned to a specific computer node on which the vertex is a *master*. If the vertex is the source of an edge that is assigned to a different computer node, the same vertex is a *mirror* on that computer node. Only the master contains all labels of that vertex.

The parallel implementation is based on MPI. In the scatter phase, every master sends its newly added labels to all of its mirrors. Then a mirror scatters those labels to its neighbors on that computing node. In the gather phase, every node maintains a hash-table to store all labels of vertices in the current batch. When conducting distance check, a vertex only needs to look up its own labels and this hash-table. Thus, the distance check only requires local operations without any message passing to remote nodes. The hash-table is once established at the beginning of every batch and is synchronized at the end of the scatter phase. Note that the batch strategy is even more important for the distributed implementation than the shared memory one as it reduces the remote access overhead significantly.

SIMD Parallelization for Weighted Graphs: BVC-PLL is able to significantly increase the data locality for remote vertex memory access, thereby offering us extra opportunities to better exploit fine-grained data-level parallelism (i.e., SIMD parallelism or vectorization). Consider the *Gather Phase* in Algorithm 4 that involves an intensive label distance check kernel (line 13 to line 17). BVC-PLL can vectorize this kernel with the help of advanced SIMD gather/scatter and mask instructions in the latest AVX-512 intrinsic set[2]. Unfortunately, such vectorization requires a dynamic expansion of the compact label structure that offsets most benefits of SIMD parallelization for unweighted graphs. However, vectorization turns to be useful for weighted graphs because the distance recheck operation incurs extra computation overhead. Efficient SIMD parallelization can significantly reduce such overhead.

Table 2: Characterization of evaluation graphs.

Name	Graph	Category	$\|V\|$	$\|E\|$
GNUT	Gnutella	Social	63 K	148 K
DBLP	DBLP	Citation	317 K	1 M
WIKI	Wikipedia Talk	Comm.	2.4 M	5 M
YOUT	YouTube	Social	3.2 M	9 M
TREC	TREC WT10g	Hyperlink	1.6 M	8.0 M
SKIT	Skitter	Computer	1.7 M	11 M
CADO	Catster/Dogster	Social	62 K	15 M
FLIC	Flickr	Social	2.3 M	33 M
HOLY	hollywood-2009	Social	1.1 M	114 M
INDO	indochina-2004	Hyperlink	7.4 M	194 M
IT	it-2004	Hyperlink	41.3 M	1.1 B
GSH	gsh-2015-host	Computer	68.6 M	1.8 B

6 EVALUATION

In this section, we perform a detailed evaluation of BVC-PLL, focusing on answering the following questions: 1) How does BVC-PLL algorithm perform against the original PLL in a sequential setting (single thread; no parallelism)? 2) In shared memory setting, how does BVC-PLL scale as the number of threads increases? 3) How well does our distributed memory implementation scale (especially on large graphs with more than a billion edges)? 4) How does the weighted extension of BVC-PLL with SIMD perform and how does it fare against ParaPLL [58] (the state-of-the-art parallel weighted PLL algorithm)?

6.1 Experimental Setup

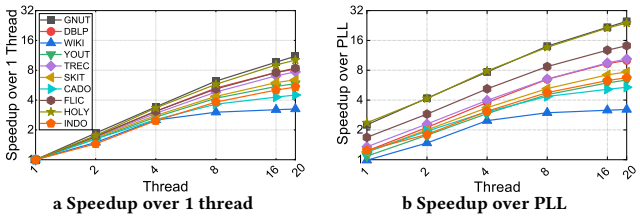
Platforms: Our *shared memory* scaling experiments were performed on an Intel Xeon Gold 6138 CPU. It is a Skylake processor with 20 cores running at 2.0 GHz supporting 512-bit AVX-512 intrinsics, with 27.5 MB L3 cache and 192 GB DDR4 memory. All code is compiled with an Intel icc compiler (version 19.0.2.187) with `-O3` optimization option. Hyper-threading is not used to simplify the analysis of experiment results. Our experiments for *distributed memory* scalability (while using one thread per node) were performed on a cluster with 16 nodes, each of which has an Intel Xeon E5-2680 CPU at 2.4 GHz with 35.8 MB L3 cache and 125 GB DDR4 memory. For evaluating the version that combines *shared and distributed memory* parallelism (and ability to process large graphs), we use another cluster with up to 224 cores (maximum of 28 cores per node and up to 8 nodes) and up to 512 GB memory per node.

Graph Datasets: The 12 graphs used in our evaluation are summarized in Table 2. They are from 5 categories (Social, Citation,

[2] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Table 3: Unweighted Performance (sec.): BVC-PLL vs. PLL. LT denotes labeling time (sec.). $|L|$ denotes the average label size for each vertex. SP denotes speedup. BVC-PLL and PLL have the same label size. The same short names are used for Table 4.

Name	$\|L\|$	PLL	BVC-PLL	
		LT	LT	SP
GNUT	477	33	**13**	2.46
DBLP	214	61	**47**	1.30
WIKI	12	40	**32**	1.24
YOUT	70	285	**249**	1.15
TREC	269	462	**323**	1.43
SKIT	138	317	**242**	1.31
CADO	96	117	**92**	1.28
FLIC	442	1,624	**909**	1.79
HOLY	2,199	10,743	**4,368**	2.46
INDO	442	4,755	**3,508**	1.36



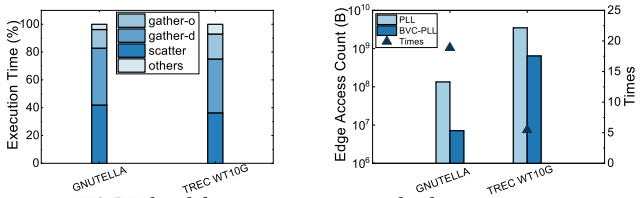
a Speedup over 1 thread **b Speedup over PLL**

Figure 3: The scalability of BVC-PLL (unweighted)

Communication, Hyperlink, and Computer) with varied numbers of vertices and edges – GNUT, and WIKI are from SNAP[3], DBLP, YOUT, TREC, SKIT, CADO, and FLIC are from KONET[4], HOLY and INDO are from SuiteSparse Matrix Collection[5], and IT and GSH are from WebGraph[6]. Particularly, IT and GSH are two large graphs with more than 1 billion edges. Because of large memory and computation time associated with these two massive datasets, experiments were limited to the version that combined shared and distributed memory parallelism, and used either 4, 6, or 8 nodes, with 4, 8, 16, or 28 threads on each node. For all other datasets, because of limited size (and thus parallelism), we either used only 1 node or 1 thread per node. These graphs are all unweighted. To test the performance of our BVC-PLL on weighted graphs, we randomly assign weights (from 1 to 7 with a uniform distribution) to their edges. Since we only evaluate algorithms for undirected graphs, we have transformed the edges in the directed graphs in Citation and Hyperlink as undirected edges.

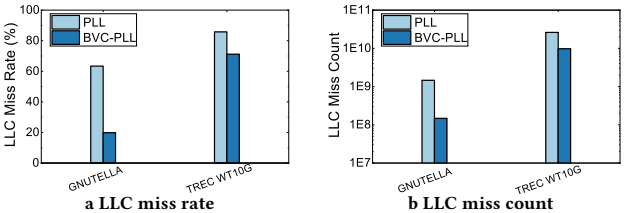
Baselines: For the sequential performance comparison on unweighted graphs, we compare BVC-PLL against the PLL implementations by the original authors [3] and by [44]. We found these two implementations provide comparable performance with the former being slightly faster. Given this, we only compare against this version. For performance comparison on weighted graphs, we compare the weighted BVC-PLL against the implementation of ParaPLL [58], applying SIMD parallelism to both. Note that existing published work on parallelizing PLL (across threads or nodes) either has limited parallelism or does not produce the same results as a sequential implementation, and thus no comparison was performed. In all of our experiments, we determine the vertex order through

[3] https://snap.stanford.edu/snap/
[4] http://konect.uni-koblenz.de/
[5] https://sparse.tamu.edu/
[6] http://law.di.unimi.it/datasets.php



a BVC-PLL breakdown **b Edge access count**

Figure 4: Performance analysis



a LLC miss rate **b LLC miss count**

Figure 5: Data locality: BVC-PLL vs. PLL

the original and the most popular method where the vertices are ordered by their vertex degree [3, 44].

Throughout our experiments, we use 1024 as the batch size for unweighted graphs and use 512 for weighted graphs. In our experimental platform, we found those two are the optimal batch size. In general, we observe the larger the batch size the better the performance, as long as there is available memory. Due to the space limitation, we will not report the results that explore the sensitivity to batch size.

6.2 BVC-PLL vs PLL and Shared Memory Scalability

Table 3 shows the performance comparison between BVC-PLL as a sequential algorithm and PLL (both using a single thread and no other parallelism, such as SIMD) on all graphs. Both algorithms use the same vertex order and produce the same label size, as expected. Interestingly, the BVC-PLL algorithm consistently outperforms PLL with the speedup ranging from 1.15× (YOUT) to 2.46× (GNUT and HOLY) with an average speedup 1.58× over PLL. Later in this section, we will report a more detailed cost breakdown and comparison.

Figure 3 shows the shared-memory scalability of BVC-PLL on all but the two largest graphs (1 node execution was not feasible for these graphs because of memory requirements). Figure 3a shows its speedup over 1-thread BVC-PLL, while Figure 3b shows its speedup over the original sequential PLL. With 20 threads, BVC-PLL can achieve up to 11.08× and 24.95× speedup over its 1-thread version and PLL, respectively (with geometric mean 6.68× and 9.33×, respectively), demonstrating good scalability.

In addition, by comparing Figure 3 and the average label sizes in Table 3, we found that generally, BVC-PLL scales better as the average label size increases. For example, GNUT and HOLY with the largest average label sizes result in the best scalability while WIKI with the smallest results in lower scalability. The labeling size provides a good indication of the total computational costs (message passing and distance checks) involved for each vertex.

Figure 4a shows the overall running time breakdown on two graphs: GNUT and TREC. Due to space limitation, we only report results for these two – trends are similar in other graphs. We can

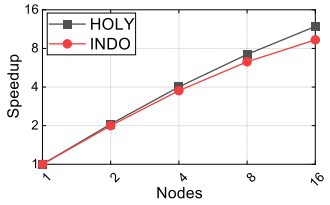
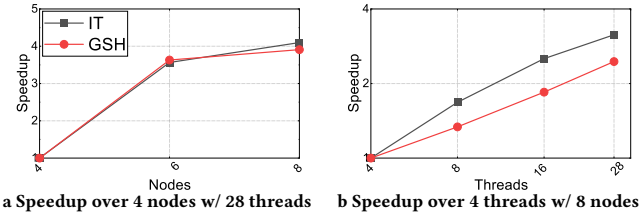


Figure 6: The scalability of distributed BVC-PLL (unweighted)



a Speedup over 4 nodes w/ 28 threads **b Speedup over 4 threads w/ 8 nodes**

Figure 7: Scalability of BVC-PLL (unweighted) on large graphs

see the Gather and the Scatter phases dominate the overall computational costs. In addition, within gather, the distance check time takes about 60%−80% and 30%−40% of the gather phase and overall time, respectively.

Figure 4b shows the total number of edge access for BVC-PLL and PLL on two graphs: GNUT and TREC. We can see that BVC-PLL has 5 and 18 times reduction for both graphs. Finally, Figure 5 shows the LLC (last level cache) miss rate and miss access count for the entire labeling process of BVC-PLL and PLL. We can see BVC-PLL has consistent lower LLC miss rate and access count than PLL.

6.3 Distributed Memory Results

To further test scalability, we experimented with distributed memory parallelism. For the third and fourth largest datasets in our collection, HOLY and INDO, we used 1 thread per node. (Experiments on the two largest datasets, IT and GSH, combine both distributed memory and shared memory parallelism and are reported in the next subsection.) Figure 6 shows the distributed memory scalability of unweighted BVC-PLL on HOLY and INDO. For HOLY, BVC-PLL achieves 2.05×, 4.00×, 7.18×, and 11.85× speedup over 1 node as the number of nodes is 2, 4, 8, and 16, respectively. For INDO, the speedups over 1 node are 1.99× (on 2 nodes), 3.74× (on 4 nodes), 6.29× (on 8 nodes), and 9.32× (on 16 nodes).

When the number of nodes is large (e.g., 16 nodes), BVC-PLL cannot achieve near linear speedups mainly because of the communication overhead caused by the large volume of label data. This is particularly obvious for INDO that generates large label data.

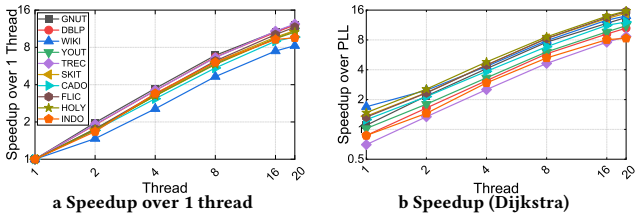
6.4 Test on Large Graphs w/ Billions of Edges

BVC-PLL works on large graphs as well. We report results from IT and GSH that have over 1 billion edges. Because of computation time and memory requirements, experiments on these graphs involve the combination of distributed memory and shared memory parallelism, with 4, 6, or 8 nodes and 4, 8, 16, or 28 threads per node.

Figure 7 shows the results – Figure 7a shows the scalability from 4 to 8 nodes (each with 28 threads). From 4 to 6 nodes, IT shows super-linear speedup mainly due to the increase of available memory and cache capacity; while from 6 to 8 nodes, its speedup is sublinear because of the intensive label data communication. GSH

Table 4: Weighted Performance (sec.): BVC-PLL vs. PLL (Dijkstra). "-S" denotes SIMD version. "-N" denotes non-SIMD version.

Name	$\|L\|$	PLL		BVC-PLL			
		LT-N	LT-S	LT-N	LT-S	SP-N	SP-S
GNUT	656	52	47	123	**37**	0.42	1.26
DBLP	387	152	**139**	293	146	0.52	0.95
WIKI	152	350	327	396	**171**	0.88	1.92
YOUT	147	652	625	763	**546**	0.85	1.14
TREC	304	632	**579**	1,140	662	0.55	0.87
SKIT	432	1,511	1,467	2,146	**921**	0.70	1.59
CADO	224	527	510	442	**347**	1.19	1.47
FLIC	653	3,879	3,826	4,189	**2,483**	0.93	1.54
HOLY	2,217	18,707	18,041	24,161	**10,399**	0.77	1.73
INDO	828	13,940	**13,105**	26,744	14,768	0.52	0.89



a Speedup over 1 thread **b Speedup (Dijkstra)**

Figure 8: Scalability of BVC-PLL (weighted) on shared memory

shows a similar trend. Moreover, Figure 7b reports BVC-PLL's good scalability on 8 nodes with changing the OpenMP thread count from 4 to 28.

6.5 Extension to Weighted Graphs and SIMD

A similar performance study is conducted between PLL and BVC-PLL for weighted graphs. To evaluate weighted BVC-PLL's sequential performance against PLL, we have modified the original PLL implementation as suggested in [3], changing its BFS traversal to Dijkstra. We also extended BVC-PLL as described in Subsection 5.1. Please notice: both PLL and BVC-PLL are optimized with SIMD for the weighted version (and for unweighted, we also implemented them with SIMD however without obvious speedup change).

Table 4 shows the comparison results for 1-thread SIMD and non-SIMD versions of PLL and BVC-PLL. For all non-SIMD tests, PLL consistently performs better than BVC-PLL; while for most SIMD tests, BVC-PLL outperforms PLL. This is because the weighted BVC-PLL introduces additional distance check (due to additional message passing) and rechecks, which significantly increases the number of instructions for BVC-PLL, resulting in degraded performance. However, SIMD parallelism is a good remedy that can significantly reduce the number of instructions. It should be noted that BVC-PLL is able to effectively exploit SIMD parallelism because the data locality has been improved. (See the performance analysis in the last Subsection). In particular, for SIMD version, BVC-PLL outperforms PLL for 7 out of 10 graphs, resulting in 1.14× to 1.92× speedup with an average of 1.34×. For the slowdown cases, BVC-PLL's performance is only degraded up to around 10%. Our BVC-PLL is able to continue exploring hierarchical parallelism to further extract the most out of the massive parallelism of modern processors.

Figure 8 shows the scalability of BVC-PLL on all weighted graphs, in which, Figure 8a shows its speedup over 1-thread BVC-PLL while Figure 8b shows its speedup over PLL. With 20 threads, BVC-PLL

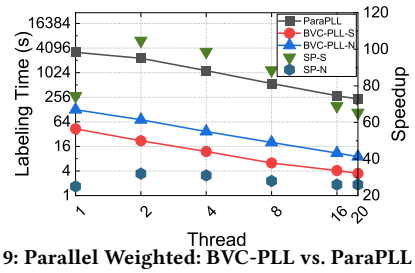


Figure 9: Parallel Weighted: BVC-PLL vs. ParaPLL on GNUT

can achieve up to 12.34× and 15.68× speedup over 1-thread and PLL, demonstrating good scalability.

Finally, we compare BVC-PLL with the state-of-the-art ParaPLL, which does weighted parallel PLL. Unfortunately, it can only run on small graphs (this is consistent on what being presented in their original paper [58]). In Figure 9 shows the performance comparison of BVC-PLL and ParaPLL on the graph GNUT (the only graph we are able to run for ParaPLL, as it throws an error of `Segmentation Fault` with the other graphs). For this graph, we can see that BVC-PLL is, in general, more than one order of magnitude faster than ParaPLL (even for non-SIMD version).

7 RELATED WORK

Many existing efforts aim to efficiently parallelize graph algorithms. Some of them most closely related to our work are discussed here.

Parallel PLL: Multiple parallel PLL approaches [19, 20, 42, 58] allow processing multiple vertices simultaneously. However, they cannot produce the same (and compact) label sets as the original PLL. They also cannot process large graphs as in our implementation. For instance, PLaNT [42] produces a superset of labels initially, so it needs label cleaning after construction. Dong *et al.* combine intra-node parallelism (as proposed in original PLL paper [3]) and inter-node parallelism – the latter not leading to the same results as the sequential computation. We are also made aware that Li *et al.* propose Parallel Shortest-distance Labeling (PSL) that replaces PLL's node-order dependency with a shortest-distance dependence [43]. Our basic vertex-centric algorithm, VC-PLL [33], is discovered independently as PSL. Moreover, we provide linear algebra analysis and combine the vertex-centric model and batched design to guarantee a smaller memory access cost than PLL.

Graph Processing Paradigms: *Batch processing* is a general idea that is also explored within the context of graph processing [76]. However, this work was focused on addressing problems like skewed distributions and high density. The use of block processing to improve the 2-hop labeling approach is original to our work. From the theoretical side, we can prove that it reduces the number of distance check operations (as shown in supplementary materials aforementioned). It also significantly improves the data locality of remote distance check operations and restricts the overall memory usage.

Our BVC-PLL adopts a *synchronous* paradigm (Bulk Synchronous Parallel (BSP) execution [73]) because its key designs to accelerating the batched processing (like bit operation, and compact data structure) rely on this property. It can be beneficial to extend BVC-PLL to incorporate asynchronous execution ideas (like *k*-level asynchronous (KLA) [21], and synchronization-avoiding algorithms [22]) to further improve its scalability and performance in the future.

Finally, as we mentioned, new graph processing frameworks (like GraphBLAS(T) [37] and GraphMat [72]) exploit efficient implementations of SpMV and SpGEMM from HPC community. As we also explained, due to the complicated masking operator it appears inefficient to implement BVC-PLL in this linear algebra form.

Graph Framework Implementations: Many popular graph processing engines and frameworks have been developed in recent years. Some of them focus on processing in-memory datasets on one node (e.g., Galois [52], Ligra [68], Polymer [77], GraphGrind [71], etc.), or disk-resident datasets on one node, (Graphchi [41], X-Stream [61], etc.) or performing distributed memory processing (Pregel [49], GraphLab [46], PowerGraph [27], etc.). The in-memory frameworks focus on improving shared memory parallelism and addressing NUMA issues, the out-of-core ones aim to reduce disk traffic, and the distributed ones concern how to efficiently partition graphs, store partitions on multiple machines, and perform low-cost communication. Many graph frameworks are also designed for GPUs [23, 28, 29, 31, 39, 45, 50, 51, 54, 66, 74, 79, 80]. For instance, works like CuSha [39] and Gunrock [74] target on load balance and memory coalescing optimizations, while works like GraphReduce [66] and Graphie [28] focus on reducing CPU-GPU traffic for the processing of large graphs not fitting in the GPU memory. Certain efforts were also specific to Xeon Phi [5, 11, 32, 56, 69]. In addition, certain graph processing frameworks are designed for hybrid CPU and coprocessors [10, 16, 26, 30, 47, 65]. Moreover, certain compiler-based efforts offer either high-level intermediate representations or domain-specific languages to support high-performance and high-productivity graph programming, such as GraphIt [78] on CPU (and, similarly, IrGL [54] on GPU). To the best of our knowledge, PLL has not been the target of any of these efforts, and there is no previous work supporting a scalable and exact parallel implementation of PLL.

8 CONCLUSION

In this paper, we proposed VC-PLL, which, to the best of our knowledge, is the first scalable parallelization of Pruned Landmark Labeling (PLL) that is able to produce the same result as the sequential method. We have achieved this by developing new insights that enable mapping the algorithm to a vertex-centric model. We also introduced a new batched execution mechanism for VC-PLL to better support message filtering and remote memory access. The resulting BVC-PLL algorithm can even run faster than the original PLL sequentially. Our experimental results further demonstrate the parallel efficiency and scalability of BVC-PLL and shows its superiority over the most recent ParaPLL algorithms on weighted graphs (using a straightforward extension of BVC-PLL). In our future work, we plan to further investigate how to optimize BVC-PLL on weighted graphs and how to extend it for out-of-core graphs. We also plan to investigate the possibility of implementing the cost-saving mechanism in BVC-PLL for other graph algorithms.

ACKNOWLEDGMENTS

The authors acknowledge William & Mary Research Computing (https://www.wm.edu/it/rc) for providing computational resources

and/or technical support that have contributed to the results reported within this paper. The work was also partially supported by the Ohio Supercomputer Center under Grant no. PGS0218.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2011. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Experimental Algorithms*, Panos M. Pardalos and Steffen Rebennack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–241.

[2] Takuya Akiba. 2014. Pruned Labeling Algorithms: Fast, Exact, Dynamic, Simple and General Indexing Scheme for Shortest-Path Queries. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14 Companion)*. Association for Computing Machinery, New York, NY, USA, 1339–1340. https://doi.org/10.1145/2567948.2582735

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. 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, 349–360.

[4] Maxim Babenko, Andrew V. Goldberg, Haim Kaplan, Ruslan Savchenko, and Mathias Weller. 2015. On the Complexity of Hub Labeling (Extended Abstract). In *Mathematical Foundations of Computer Science 2015*. Springer Berlin Heidelberg, 62–74.

[5] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 32–41.

[6] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 93–104.

[7] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology* 25, 2 (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249 arXiv:https://doi.org/10.1080/0022250X.2001.9990249

[8] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. Grarep: Learning Graph Representations with Global Structural Information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 891–900.

[9] Tim Carnes, Chandrashekhar Nagarajan, Stefan M. Wild, and Anke van Zuylen. 2007. Maximizing Influence in a Competitive Social Network: A Follower's Perspective. In *Proceedings of the Ninth International Conference on Electronic Commerce (ICEC '07)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/1282100.1282167

[10] Linchuan Chen, Xin Huo, Bin Ren, Surabhi Jain, and Gagan Agrawal. 2015. Efficient and simplified parallel graph processing over cpu and mic. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 819–828.

[11] Linchuan Chen, Peng Jiang, and Gagan Agrawal. 2016. Exploiting Recent SIMD Architectural Advances for Irregular Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 47–58.

[12] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and S Yu Philip. 2006. Fast computation of reachability labeling for large graphs. In *International Conference on Extending Database Technology*. Springer, 961–979.

[13] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S Yu. 2008. Fast computing reachability labelings for large graphs with high compression rate. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 193–204.

[14] Jiefeng Cheng, Jeffrey Xu Yu, and Philip S. Yu. 2011. Graph Pattern Matching: A Join/Semijoin Approach. *IEEE Trans. Knowl. Data Eng.* 23, 7 (2011), 1006–1021.

[15] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and distance queries via 2-hop labels. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete algorithms*. 937–946.

[16] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 752–768.

[17] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. 2014. Robust exact distance queries on massive networks. *Microsoft Research, USA, Tech. Rep* 2 (2014).

[18] Q. Dong, K. Lakhotia, H. Zeng, R. Karman, V. Prasanna, and G. Seetharaman. 2018. A Fast and Efficient Parallel Algorithm for Pruned Landmark Labeling. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547548

[19] Qing Dong, Kartik Lakhotia, Hanqing Zeng, Rajgopal Karman, Viktor Prasanna, and Guna Seetharaman. 2018. A fast and efficient parallel algorithm for pruned landmark labeling. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[20] Damir Ferizovic. 2015. *Parallel Pruned Landmark Labeling for Shortest Path Queries on Unit-Weight Networks*. Ph.D. Dissertation. National Research Center.

[21] Adam Fidel, Nancy M Amato, Lawrence Rauchwerger, et al. 2014. Kla: A new algorithmic paradigm for parallel graph computations. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 27–38.

[22] Jesun Sahariar Firoz, Marcin Zalewski, Thejaka Kanewala, and Andrew Lumsdaine. 2018. Synchronization-avoiding graph algorithms. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 52–61.

[23] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: eXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 121–131.

[24] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*. 319–333.

[25] Alan George, John R Gilbert, and Joseph WH Liu. 2012. *Graph theory and sparse matrix computation*. Vol. 56. Springer Science & Business Media.

[26] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques, PACT*. ACM, 345–354.

[27] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[28] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT),*. IEEE, 233–245.

[29] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. 2017. MultiGraph: Efficient Graph Processing on GPUs. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 27–40.

[30] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 78–88.

[31] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.

[32] Peng Jiang, Linchuan Chen, and Gagan Agrawal. 2016. Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 16.

[33] Ruoming Jin, Zhen Peng, Wendell Wu, Feodor Dragan, Gagan Agrawal, and Bin Ren. 2019. Pruned Landmark Labeling Meets Vertex Centric Computation: A Surprisingly Happy Marriage! *arXiv preprint arXiv:1906.12018* (2019).

[34] R. Jin, N. Ruan, Y. Xiang, and V. E. Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD*.

[35] Ruoming Jin, Ning Ruan, Bo You, and Haixun Wang. 2013. Hub-Accelerator: Fast and Exact Shortest Path Computation in Large Social Networks. *CoRR* abs/1305.0507 (2013).

[36] Ruoming Jin and Guan Wang. 2013. Simple, fast, and scalable reachability oracle. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1978–1989.

[37] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. 1–9. https://doi.org/10.1109/HPEC.2016.7761646

[38] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. Vol. 22. SIAM.

[39] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 239–252.

[40] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 395–410.

[41] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

[42] Kartik Lakhotia, Rajgopal Kannan, Qing Dong, and Viktor Prasanna. 2019. Planting Trees for Scalable and Efficient Canonical Hub Labeling. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 492–505. https://doi.org/10.14778/3372716.3372722

[43] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling Distance Labeling on Small-World Networks. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for

Computing Machinery, New York, NY, USA, 1060–1077. https://doi.org/10.1145/3299869.3319877

[44] Ye Li, Man Lung Yiu, Ngai Meng Kou, et al. 2017. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment* 11, 4 (2017), 445–457.

[45] Hang Liu and H Howie Huang. 2018. Simd-x: Programming and processing of graph algorithms on gpus. *arXiv preprint arXiv:1812.04070* (2018).

[46] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A New Framework for Parallel Machine Learning. *arXiv preprint arXiv:1408.2041* (2014).

[47] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 527–543.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data (SIGMOD '10)*.

[49] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[50] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 201–213.

[51] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 117–128.

[52] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.

[53] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 709–724.

[54] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 1–19.

[55] Scott Beamer Krste Asanovic David Patterson. 2012. Direction-Optimizing Breadth-First Search. *SC12, November* (2012), 10–16.

[56] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. 2018. GraphPhi: Efficient Parallel Graph Processing on Emerging Throughput-oriented Architectures. In *2018 International Conference on Parallel Architecture and Compilation (PACT)*. ACM.

[57] Qiongwen Xu, Xu Zhang, Jin Zhao, Xin Wang, and T. Wolf. 2016. Fast shortest-path queries on large-scale graphs. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. IEEE, 1–10.

[58] Kun Qiu, Yuanyang Zhu, Jing Yuan, Jin Zhao, Xin Wang, and Tilman Wolf. 2018. ParaPLL: Fast Parallel Shortest-path Distance Query on Large-scale Weighted Graphs. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2.

[59] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. NScale: Neighborhood-Centric Large-Scale Graph Analytics in the Cloud. *The VLDB Journal–The International Journal on Very Large Data Bases* 25, 2 (2016), 125–150.

[60] Neil Robertson and Paul D. Seymour. 1986. Graph minors. II. Algorithmic aspects of tree-width. *Journal of algorithms* 7, 3 (1986), 309–322.

[61] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.

[62] Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet. 2006. Distance join queries on spatial networks. In *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems (GIS '06)*.

[63] J. Sankaranarayanan, H. Samet, and H. Alborzi. 2009. Path oracles for spatial networks. *PVLDB* 2 (August 2009). Issue 1.

[64] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. 2004. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *Advances in Database Technology - EDBT 2004*, Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–255.

[65] Robert Searles, Stephen Herbein, and Sunita Chandrasekaran. 2016. A portable, high-level graph analytics framework targeting distributed, heterogeneous systems. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE, 79–88.

[66] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: Processing Large-scale Graphs on Accelerator-based Systems. In *2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. IEEE, 1–12.

[67] P. Shiralkar, A. Flammini, F. Menczer, and G. L. Ciampaglia. 2017. Finding Streams in Knowledge Graphs to Support Fact Checking. In *2017 IEEE International Conference on Data Mining (ICDM)*. 859–864. https://doi.org/10.1109/ICDM.2017.105

[68] Julian Shun and Guy E Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[69] Milan Stanic, Oscar Palomar, Ivan Ratkovic, Milovan Duric, Osman Unsal, Adrian Cristal, and Mateo Valero. 2014. Evaluation of vectorization potential of graph500 on intel's xeon phi. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 47–54.

[70] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article Article 16, 10 pages. https://doi.org/10.1145/3079079.3079097

[71] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2017. GraphGrind: addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[72] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241* (2015).

[73] Leslie G Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[74] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 11.

[75] Mathias Weller. 2014. Optimal Hub Labeling is NP-complete. *CoRR* abs/1407.8373 (2014).

[76] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[77] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-Aware Graph-Structured Analytics. *ACM SIGPLAN Notices* 50, 8 (2015), 183–193.

[78] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance DSL for Graph Analytics. *arXiv preprint arXiv:1805.00923* (2018).

[79] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552.

[80] Wenyong Zhong, Jianhua Sun, Hao Chen, Jun Xiao, Zhiwen Chen, Chang Cheng, and Xuanhua Shi. 2016. Optimizing graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (2016), 1149–1162.

[81] Lei Zou, Lei Chen, and M. Tamer Özsu. 2009. Distance-join: pattern match query in a large graph database. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 886–897.