Thrifty Label Propagation: Fast Connected Components for Skewed-Degree Graphs

Mohsen Koohi Esfahani
0000-0002-7465-8003

Peter Kilpatrick
0000-0003-0818-8979

Hans Vandierendonck
0000-0001-5868-9259

{mkoohiesfahani01, p.kilpatrick, h.vandierendonck}@qub.ac.uk

Queen's University Belfast, Northern Ireland, UK
https://blogs.qub.ac.uk/GraphProcessing

***Abstract*—Various concurrent algorithms have been proposed in the literature in recent years that mostly focus on the disjoint set approach to the Connected Components (CC) algorithm. However, these CC algorithms do not take the skewed structure of real-world graphs into account and as a result they do not benefit from common features of graph datasets to accelerate processing.**

We investigate the implications of the skewed degree distribution of real-world graphs on their connectivity and we use these features to introduce *Thrifty Label Propagation* as a structure-aware CC algorithm obtained by incorporating 4 fundamental optimization techniques in the Label Propagation CC algorithm.

Our evaluation on 15 real-world graphs and 2 different processor architectures shows that Thrifty accelerates the flow of labels and processes only 1.4% of the edges of the graph.

In this way, Thrifty is up to $16\times$ faster than state-of-the-art CC algorithms such as Afforest, Jayanti-Tarjan, and Breadth-First Search CC. In particular, Thrifty delivers $1.5\times - 19.9\times$ speedup for graph datasets larger than one billion edges.

***Index Terms*—Connected components, Label propagation, High performance computing, Graph Connectivity, Graph algorithms, Graph traversal, Data analysis.**

I. Introduction

Connected Components (CC) is a graph analytic algorithm widely used in different fields of science, industry and technology including biology [1], [2], [3], image processing [4], [5], economy [6], and astronomy [7], and acts also as a preliminary tool in several graph analytics like graph clustering [8], [9], [10], locality optimizing graph relabeling (reordering) [11], graph partitioning and processing [12], [13].

Algorithms for finding connected components in a graph can be placed in one of three classes:

1) **Flood Filling CC** [14], [15], [16], [17] performs breadth-first search (BFS) or depth-first search (DFS) to identify all vertices that are reachable from a chosen starting point. A BFS/DFS search is required for each component.
2) **Label Propagation CC (LP-CC)** [18] iteratively updates the label of each vertex by calculating the minimum value between labels of its neighbours. Label Propagation CC can be specified in terms of generalized Sparse Matrix-Vector (SpMV) multiplication.

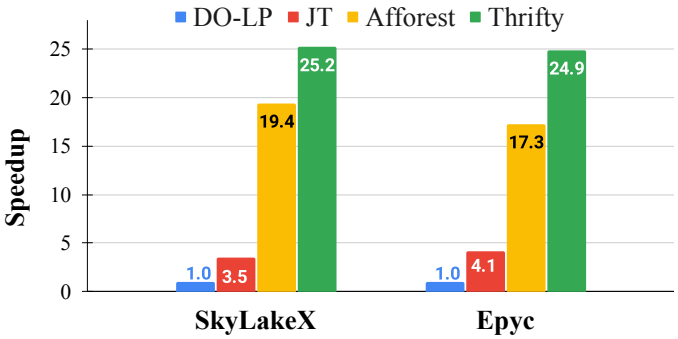


Fig. 1: Average speedup normalized to DO-LP

3) **Disjoint Set CC** [19], [20], [21], [22] uses the disjoint set data structure to group connected vertices in the same set. As sets are often represented as trees, these algorithms have also been called "tree-hooking" algorithms [23].

Some studies have identified that Disjoint Set CC is more efficient than Flood Filling and Label Propagation [22], [24]. This is especially true for graph datasets with high diameters [25], such as road networks. Moreover, in the case of graphs with skewed degree distribution, such as those found in social networks and web graphs, Disjoint Set CC minimizes the number of times each edge is processed. Jayanti and Tarjan process each edge just once [21] while Afforest processes each edge on average slightly more than once [22]. However, Disjoint Set CC is not scalable and has not been effective in distributed processing [26]. In contrast, the Label Propagation CC follows a SpMV model that has been successfully scaled to distributed systems [27], [28], [29].

In this paper, we present a new perspective on the CC algorithm by investigating the implications imposed by real-world graphs on the efficiency of the label propagation process of the LP-CC algorithm. Many real-world graphs derived from social networks, the internet, and the world-wide web show a heavy-tailed skewed degree distribution. In other words, a very small fraction of the vertices are connected to a disproportionately large fraction of edges. This particular relationship between vertices merits special attention.

We identify 4 main problems in LP-CC: (1) repeated wave-fronts, (2) propagating labels to converged vertices, (3) incor-

2168-9253/21/©2021 IEEE
https://doi.org/10.1109/Cluster48925.2021.00042

rect initial label assignment, and (4) weak bootstrapping label propagation. Then, we introduce **Thrifty Label Propagation** as our solution to remedy these problems using four optimization techniques: (1) **Unified Labels Array**, (2) **Zero Convergence**, (3) **Zero Planting**, and (4) **Initial Push**.

We evaluate the Thrifty algorithm against state-of-the-art CC algorithms Afforest [22], Jayanti-Tarjan (JT) [21], BFS-CC [30], and Shiloach-Vishkin (SV) [19]. Moreover, we evaluate Thrifty for 2 different processor architectures: Intel Sky-LakeX and AMD Epyc with 32 and 128 cores, respectively. Figure 1 compares the speedup of different CC algorithms for different architectures. Our evaluation shows that Thrifty is faster than Afforest, JT, BFS-CC, and SV by $1.4\times$, $7.3\times$, $14.7\times$, and $51.2\times$, respectively. In comparison to Direction Optimizing Label Propagation, Thrifty shows $25.2\times$ speedup.

This paper is structured as follows: Section II explains key background materials. Section III explains the main inefficiencies in the Label Propagation algorithm in processing real-world graphs with power-law degree distribution, and Section IV introduces four optimization techniques to solve these problems. Section V evaluates our new Thrifty algorithm and Section VI discusses further related work. Future work is discussed in Section VII.

II. Background

A simple graph or undirected graph $G = (V, E)$ has a set of vertices V, and a set of edges E between these vertices. Edges are unordered pairs of elements of V. N_v is the set of neighbours of vertex v. We consider algorithms for static graphs, which are immutable during the evaluation of the algorithms.

We represent undirected graphs using a compressed sparse (rows or columns) representation [31]. This is a compact representation that is generally assumed in graph processing. A drawback of this representation is that each edge is represented twice: once pointing from a vertex to its neighbour, and once pointing back from the neighbour to the vertex. This representation simplifies information flow across edges in both directions. Afforest also assumes this representation in support of sampling edges incident to specific vertices [22]. Some algorithms, like the Jayanti and Tarjan algorithm, operate correctly on a coordinate representation, where each edge appears precisely once [21].

A frontier F is a data structure that represents a set of active vertices $F.V$ and a set of active edges $F.E$ that is induced by the vertex set: $F.E = \{(v, u) \in E | v \in F.V \wedge u \in N_v\}$. Frontiers may be implemented as worklists (listing specifically the active vertices in $F.V$), or as a bitmap or boolean array (storing a boolean value for each vertex $v \in V$ that indicates if $v \in F.V$). Graph processing systems dynamically switch between these representations depending on the density of the frontier, i.e., the number of vertices and edges it contains compared to the size of the graph [32].

The principle of LP is that each vertex is initially assigned a unique integer label. Each vertex subsequently compares its label to the labels of its neighbours and updates its label to be the smallest among them. This process is repeated for all vertices in the graph during one iteration of the algorithm. Subsequent iterations repeat this process until no further changes are made to the labels. The initially assigned labels can be chosen freely, as long as each vertex has a distinct label.

Algorithm 1: Direction Optimizing LP CC

```
   Input: G(V, E), new_lbls[ ], old_lbls[ ]
1  Frontier old_fr(V), new_fr;
   /* Initial label assignment                    */
2  par_for v ∈ V
3  |  old_lbls_v = new_lbls_v = v;
4  |  old_fr.set(v);
5  do
6  |  new_fr.reset();
   |  /* Identifying direction                    */
7  |  density = (|old_fr.V| + |old_fr.E|)/|E|;
8  |  if density < threshold then
   |  |  /* Push traversal                        */
9  |  |  par_for v ∈ old_fr
10 |  |  |  for u ∈ N_v do
11 |  |  |  |  if atomic_min(&new_lbls_u, old_lbls_v)
   |  |  |  |    then new_fr.set(u) ;
12 |  |  else
   |  |  |  /* Pull traversal                      */
13 |  |  |  par_for v ∈ V
14 |  |  |  |  new_label = old_lbls_v;
15 |  |  |  |  for u ∈ N_v do
16 |  |  |  |  |  if old_lbls_u < new_label then
17 |  |  |  |  |  |  new_label = old_lbls_u;
18 |  |  |  |  if new_label < old_lbls_v then
19 |  |  |  |  |  new_lbls_v = new_label;
20 |  |  |  |  |  new_fr.set(v);
   |  |  /* Synchronizing labels arrays           */
21 |  |  par_for v ∈ V
22 |  |  |  old_lbls_v = new_lbls_v;
23 |  |  swap(new_fr, old_fr);
24 while |old_fr|;
```

A. Direction Optimizing Label Propagation

The direction optimizing graph traversal selects push or pull traversal based on the number of vertices and edges that should be processed [33], [34]. Direction Optimizing Label Propagation (DO-LP) has been implemented in different graph processing frameworks including [35], [25], [28], [36], [37], [38], [39]. We present here a version of the algorithm that is broadly considered as the state of the art Label Propagation algorithm (Algorithm 1).

DO-LP maintains two arrays to store labels of vertices: old_lbls holds the labels derived during the previous iteration, while new_lbls holds the updated labels calculated in the current iteration. DO-LP uses two frontiers to manage active vertices in each iteration: (1) the new_fr collects vertices

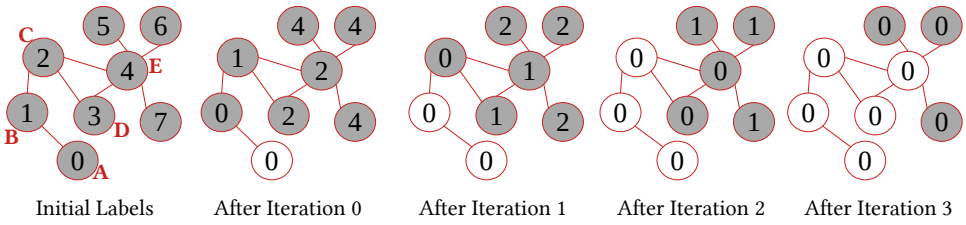
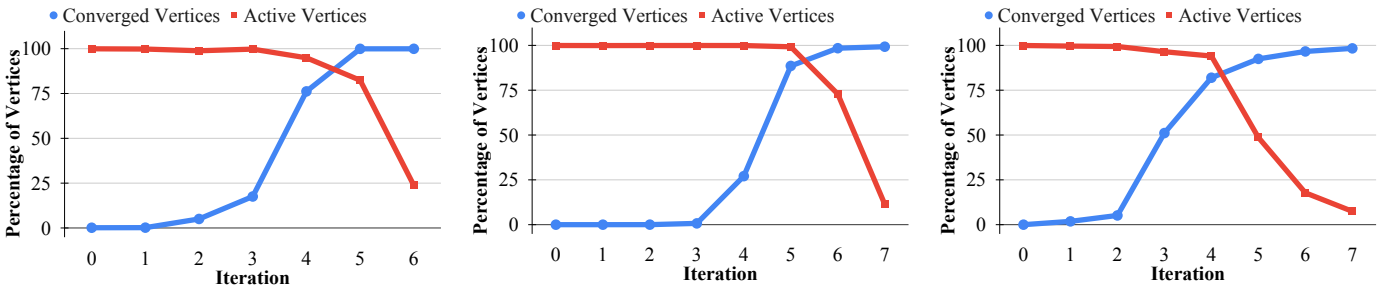


Fig. 2: Label propagation in the Direction Optimizing (gray background indicates an active vertex).



(a) WarWiki (b) Friendster (c) SK-Domain

Fig. 3: Percentage of converged and active vertices in pull iterations of the DO-LP

whose labels have been changed in this iteration and should propagate their new label in the next iteration, and (2) the old_fr contains active vertices in the current iteration, i.e., their label has been changed in the previous iteration.

New and old labels of each vertex are initialized by the vertex ID (Lines 2-4), and then CC iterations are started by identifying the traversal direction using the density of the frontier (Lines 6-7). Values like $\frac{1}{15}$, $\frac{1}{18}$ [34], and 5% [35], [25] are often used as density threshold.

In a sparse iteration, a push traversal is performed: for each vertex v in the frontier, all neighbours are checked. If the new label of a neighbour is greater than the old label of v (Line 10), the neighbour's new label is updated and the neighbour is submitted to the new_fr to be processed in the next iteration. The $atomic\_min()$ uses `compare_and_swap()` to perform an atomic write of old_lbls_v to new_lbls_u, if new_lbls_u is lower than old_lbls_v. The $atomic\_min()$ returns the result of comparison that states if new_lbls_u has been modified by this function.

In a dense iteration, a pull traversal is executed (Lines 13-20). The new label of a vertex is identified as the minimum value between the old labels of the vertex and its neighbours (Lines 16-17). While the pull iteration calculates a new frontier new_fr, it does not consult whether its neighbours are present in old_fr. This is correct as all labels are valid values and improves performance by reducing memory accesses.

An iteration is finished by updating the old labels of vertices to their new values (Lines 21-22). Iterations are continued as long as the label of at least one vertex is modified (Line 24).

III. Drawbacks of Label Propagation in Processing Power-Law Graphs

While DO-LP employs various important inventions in high-performance graph processing, several inefficiencies remain. These inefficiencies are specifically important for power-law graphs.

A. Repeated Wavefronts

Labels are propagated from one vertex to its neighbours. As a consequence, DO-LP propagates a label over one hop distance during one iteration, and two hops distance during two iterations. This causes a wavefront of updates that ripples through the graph and causes changes to the vertex labels. In this way, after each iteration of the DO-LP, a new wavefront is initiated and this new wavefront follows the previous wavefront at a distance.

The DO-LP initiates a new wavefront only at the end of an iteration, when updated labels are committed (Lines 21 and 22 in Algorithm 1). Moreover, wavefronts can propagate over at most one hop during any iteration.

Figure 2 shows an example graph and its label propagation steps. Initially all vertices receive a label as their ID. Following the next iterations, vertices update their labels by comparing labels of their neighbours. Figure 2 shows that DO-LP propagates a label only one hop per iteration. First, label 1 is propagated from vertex B to vertex C and then on to the main part of the graph. This is subsequently repeated by overwriting label 1 with 0. As such, it requires to perform as many iterations as the diameter of the graph (4 iterations) to propagate the lowest label of the component to all vertices of the component. Thus, **label propagation in DO-LP is a slow and repetitive process**.

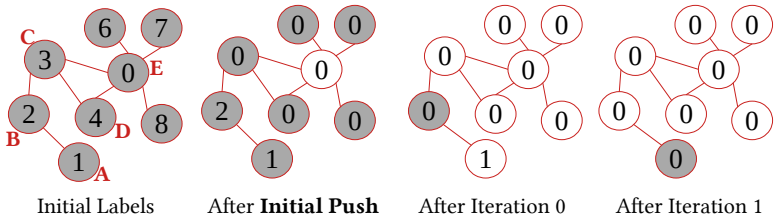


Fig. 4: Label propagation in the Thrifty

B. Preaching to the Converged

The DO-LP algorithm tries to minimize the amount of computation by tracking which vertices had their label changed. In this way, only changed labels are propagated. However, this still causes redundant work. Figure 3 shows the percentage of vertices that are active at the start of pull iterations, as well as the percentage of vertices that have converged to their final value. Converged vertices have reached their final values and do not need to be processed further.

Figure 3 demonstrates that convergence is very slow in the first iterations as well as in the final iterations. In between, convergence occurs very quickly with 30–60% of the vertices converging during one iteration. These are the most effective iterations.

However, during these and the following iterations, there is redundant activity: the number of active vertices is high as well as the number of converged vertices. Hence, most of the active vertices will try to propagate their label to a vertex that has already converged. In other words, **DO-LP performs excess work by processing all edges of the graph in pull iterations as it is not able to identify if vertices have converged**.

C. Inefficient Initial Label Assignment

The propagation of labels is in part driven by the initial label assignment. For instance, in Figure 2, a small label is assigned to the vertex A which is on the fringe of the graph. The label is propagated to the core of the graph over several iterations. During the first iterations, however, other labels are also propagated between vertices like E, D, and C. When A's label reaches C, the traversal inside the core has to be repeated all over.

However, when labels are assigned differently, LP is more efficient. Note the degree of freedom in choosing the initial label assignment: the only constraint is that all vertices initially have distinct labels. If vertex E is initially assigned the smallest label (Figure 4), then the label is first propagated in the core of the network, which then stabilises. The label is subsequently propagated out to vertex E, causing fewer label updates in total. This shows that the initial label assignment affects performance and that **structure-oblivious initial label assignment prevents efficient propagation of the labels**.

D. Eager Bootstrapping Label Propagation

DO-LP starts with propagating the label of each vertex to its neighbours, as indicated by initializing new_fr to contain all vertices. This is necessary initially as we need to compare the label of a vertex at least once to all its neighbours. However, doing this right at the start is inefficient as very few vertices converge to their final label in the very first iterations (Figure 3).

The cause of this inefficiency can be found in the fact that most vertices have a large label. However, in skewed-degree graphs, most vertices also have neighbours with large labels. As such, in the first iterations of the DO-LP, there is little opportunity to reduce the magnitude of the label significantly. Even worse, most updates will be overridden by future wavefronts carrying smaller labels. As such, **the initial pull iterations of the DO-LP are work inefficient**.

IV. Thrifty Label Propagation

We introduce 4 optimization techniques to address the inefficiencies of DO-LP described above. These are implemented in Algorithm 2.

A. Unified Labels Array

DO-LP incurs a slow label propagation, with a wavefront progressing at most one hop per iteration (Section III-A). We address this by employing a **Unified Labels Array** that uses only one array for labels, as opposed to different arrays for the old and new labels. In this way, updated labels can be propagated already within the same iteration as they are calculated, simply by reading the values from the same array (or memory location) as they were written to.

Section V-C1 shows that the number of iterations is reduced by up to 89% and on average by 39%, as a result of accelerating label propagation by using one labels array.

By using one labels array, the label arrays' synchronization in Lines 21-22 of the Algorithm 1 is removed in Algorithm 2. This reduces the execution time of sparse push iterations significantly.

B. Zero Convergence

DO-LP is prone to processing many vertices that have already converged on their final label (Section III-B). As such, we desire to recognise when vertices have converged and, once they have converged, we skip processing these vertices. But how can we know if a vertex has converged? Hereto, we make two observations.

Firstly, the LP algorithm performs the "minimum" arithmetic operation on the labels of each pair of neighbouring vertices. The arguments to the minimum operation are the labels, which are integers. It is important to note that the LP algorithm does not create new labels. It only copies over labels from one vertex to another, in such a way that larger labels are overwritten by smaller labels. As such, we know that the

Algorithm 2: Thrifty Label Propagation

```
   Input: G(V, E), labels[ ]
1  Frontier old_fr(V), new_fr;
   /* Initial label assignment                    */
2  /* Zero Planting                               */
3  par_for v ∈ V
4  |  labels_v = v + 1;
5  |  if v.degree > Max_Degrees_thread_id then
6  |  |  Max_Degrees_thread_id = v.degree;
7  |  |  Max_Ids_thread_id = v;
8  max_degree_id = max(Max_Degrees, Max_Ids);
9  labels_max_degree_id = 0;
10 /* Initial Push                                */
11 par_for v ∈ N_max_degree_id
12 |  labels_v = 0;
13 do
14 |  new_fr.reset();
   |  /* Identifying direction                    */
15 |  density = (|old_fr.V| + |old_fr.E|)/|E|;
16 |  if density < new_threshold then
   |  |  /* Push traversal                        */
17 |  |  par_for v ∈ old_fr
18 |  |  |  for u ∈ N_v do
19 |  |  |  |  if atomic_min(&labels_u, labels_v) then
20 |  |  |  |  |  new_fr.set(u)
21 |  else
   |  |  /* Pull traversal                        */
22 |  |  par_for v ∈ V
23 |  |  |  /* Zero Convergence                    */
24 |  |  |  if labels_v == 0 then
25 |  |  |  |  continue;
26 |  |  |  new_label = labels_v;
27 |  |  |  for u ∈ N_v do
28 |  |  |  |  if labels_u < new_label then
29 |  |  |  |  |  new_label = labels_u;
30 |  |  |  |  |  /* Zero Convergence             */
31 |  |  |  |  |  if new_label == 0 then
32 |  |  |  |  |  |  break;
33 |  |  |  if new_label < labels_v then
34 |  |  |  |  labels_v = new_label;
35 |  |  |  |  new_fr.set(v);
36 |  swap(new_fr, old_fr);
37 while |old_fr|;
```

smallest label that any vertex can ever obtain is the same as the smallest label in the initial assignment of labels. In our case, that is zero. As such, we can safely assume that any vertex holding a zero label has converged – it cannot be updated further.

Our second observation answers the question: *Are there many vertices that will converge to the zero label?* The answer is based on the high connectivity of vertices in real-world skewed-degree graphs. Table I shows the percentage of vertices of each dataset that are in the largest component. It shows

TABLE I: Percentage of vertices in the component containing the vertex with the maximum degree

Dataset	Vertices%	Dataset	Vertices%	Dataset	Vertices%
Pkc	100	WWiki	99.8	LJLnks	99.7
LJGrp	100	Twtr10	100	Twtr	99.8
Wbbs	97.9	TwtrMpi	100	Frndstr	100
SK	100	WbCc	98.9	UKDls	99.3
UU	99.3	UKDmn	99.2	ClWb9	94.5

that more than 94% of vertices of power-law graphs are connected to each other. This corresponds to the notion of a giant component, which forms naturally in skewed-degree graphs [40]. Thus, more than 94% of the vertices can converge to the zero value provided that the zero label is assigned to a vertex in the giant component. Moreover, assuming that initial labels are assigned uniformly at random, the zero label will be assigned to the giant component with a probability of 94%.

The **Zero Convergence** optimization is implemented by adding two branches in Lines 24 and 31 of the Algorithm 2, which check if the vertex has converged to zero. If so, we do not need to process the vertex further. Moreover, the branch at line 31 implies that processing of a vertex terminates immediately, as soon as its label becomes zero.

Section V-C2 shows that **Zero Convergence** tremendously reduces the total processed edges: on average, DO-LP processes each edge 7.7 times, while Thrifty processes only 1.4% of the edges.

C. Zero Planting

In Section III-C we explained that DO-LP assigns initial labels inefficiently which results in long propagation paths and repeated wavefronts. To solve this, we need to ensure that the smallest label is initially placed in the core of the graph, and not on the fringes. Considering also the **Zero Convergence** optimization, we should maximize the chance that the zero label is assigned to the giant component. This is captured in the **Zero Planting** technique.

We employ a simple heuristic to plant the zero label at the start of the algorithm, namely to plant it in the vertex with the highest degree. The rationale is two-fold: In a skewed-degree graph with a giant component, the highest-degree vertex is almost certainly a member of the giant component (if not, the component containing the highest-degree vertex cannot be giant). Secondly, the highest-degree vertex is likely a hub vertex, i.e., it has a high centrality within the graph. As such, it is few hops away from the other vertices in the same component.

The **Zero Planting** technique is implemented in Lines 3-9 of the Algorithm 2. The label of vertex v is initialized by $v + 1$ (instead of v), and the zero label is reserved for the vertex with the maximum degree. In Lines 5-7, each parallel thread (with ID $thread\_id$) finds its local maximum degree and the vertex with the maximum degree (between maximum degrees reported by threads) receives the zero label in Line 9.

Section V-C3 shows that the **Zero Planting** technique provides a very fast convergence rate of 88% of the vertices

after the first pull iteration as a result of removing or cutting short those iterations that are required to propagate the label zero to the hub vertices.

D. Initial Push

We observed that DO-LP starts off poorly in the first iterations as the vast majority of the labels that are propagated in these iterations will later be overwritten (see Section III-D). As such, it starts off too aggressively with pull iterations that propagate labels along all edges. However, identifying which labels are not worth propagating is non-trivial in DO-LP.

The **Zero Planting** optimization enables Thrifty to selectively propagate labels in the initial iterations. In Thrifty, the goal is to make the giant component converge to label zero. As such, we are initially only interested in propagating the zero label. Once the zero label has propagated to a sufficient number of highly connected vertices, it can propagate much more quickly through the giant component. At this stage a full-blown pull iteration (with zero convergence) becomes effective, and will also effect label propagation through the other components. Note that the other components are tiny and hardly contribute to execution time.

The **Initial Push** technique states that the best traversal in the first iteration is a push traversal of the zero label from the vertex with the maximum degree to its neighbours. This push traversal propagates the zero label as much as possible without imposing the cost of processing all edges. The initial push traversal is shown in Lines 11-12 of Algorithm 2.

Thrifty performs only one initial push iteration. This is optimal due to the typical structure of graphs with skewed degree distribution where many high-degree vertices are connected to other high-degree vertices and have many common neighbours. A first initial push iteration thus propagates the zero label to a good number of high-degree vertices, and a second push iteration would traverse many high-degree vertices with many common neighbours. This would replicate much of the work of propagating the zero label. In this way, a pull traversal is more efficient [33] especially with zero convergence check.

Section V-C4 shows that the **Initial Push** technique accelerates the execution time of the first iteration by $5.3\times$.

E. Thrifty Implementation and Data Structures

In this section we present more details of efficient implementation of Thrifty.

In Line 16 of the Algorithm 2 we use $new\_threshold$ to select push or pull traversal. By applying the convergence optimizations that significantly reduce the execution time of pull traversals, we identified 1% acts best as a threshold between push and pull traversals. We evaluate the effect of the threshold in Section V-E.

To accelerate pull iterations, we do not collect a detailed frontier listing all active vertices. At the end of most pull iterations, it suffices to know whether the frontier is dense or sparse. As such, we count active vertices but do not record which vertices are active. In the final pull iteration, prior to switching to sparse iterations, a detailed frontier is necessary. When Thrifty decides to switch to push traversal, it performs a **Pull-Frontier** iteration, which is a pull iteration that also identifies which vertices are active.

Push iterations are sparse and, as such executed quickly. Some web graphs like `UK-Union` and `WebBase-2001` have more than 70 push iterations, and it is necessary to optimize the push iterations. To this end, it is necessary to select data structures carefully [41]. We assign local worklists to each thread to collect its active vertices. We also use a shared byte array between threads that shows if a vertex has been previously added to the local worklist of any thread. The byte array is written and read by all threads and the local worklists are only written by their specific threads but are read by all threads. We do not use atomic instructions to access the shared byte array. In the case that one vertex will be added to two local worklists due to a race condition, then that vertex may be processed twice in the next iteration. This does not affect the correctness of the algorithm. Each thread starts by processing its local worklist and after that steals vertices from the worklists of other threads.

F. Correctness of The Thrifty Algorithm

The Thrifty algorithm uses four optimizations and in this section we show these optimizations do not change the correctness of the algorithm.

The **Unified Labels Array** technique uses one label array for storing labels. It affects Lines 11 and 16 of the Algorithm 1 where new_lbls should be read instead of old_lbls. We assume vertex v reads old_lbls for all of its neighbours except neighbour n. Reading the new_lbls_n can change the correctness of the algorithm only if v can not find any label less than new_lbls_n in the current iteration. In this case v will read new_lbls_n in the next iteration of DO-LP. This shows that reading new_lbls_n instead of old_lbls_n in the current iteration does not affect the correctness of the algorithm.

The **Zero Convergence** technique inserts comparisons to zero to the DO-LP to stop processing edges when reaching the zero label. As zero is the minimum value among all labels, no changes can be applied to a label that has reached zero. This shows that the Zero Convergence technique stops a process that can not change the label of a vertex. In other words, the Zero Convergence technique does not change the correctness of the algorithm.

The correctness of DO-LP is independent of the initial label assignment as long as vertices receive unique initial labels. Therefore the **Zero Planting** technique that plants the zero label in the vertex with maximum degree does not change the correctness of the algorithm.

The **Initial Push** technique can be considered as the application of a different schedule, i.e., the zero label is propagated over one hop before considering other updates. The correctness follows from the same argument as Unified Label Arrays.

TABLE II: Datasets

Dataset	Name	Type	Power-Law	Source	|V| (M)	|E| (B)	|CC|
GBRd	GB Roads	Road Network	No	NR	8	0.016	1
USRd	US Roads	Road Network	No	NR	24	0.058	1
Pkc	Pokec	Social Network	Yes	KN	1.6	0.044	1
WWiki	War Wikipedia Links	Knowledge Graph	Yes	KN	2	0.052	1,245
LJLnks	LiveJournal Links	Social Network	Yes	KN	5	0.098	4,945
LJGrp	LiveJournal Group Memberships	Social Network	Yes	KN	7	0.225	1
Twtr10	Twitter 2010	Social Network	Yes	NR	21	0.530	1
Twtr	Twitter	Social Network	Yes	NR	28	0.956	31,445
Wbbs	WebBase-2001	Web Graph	Yes	LWA	115	1.737	236,185
TwtrMpi	Twitter-MPI	Social Network	Yes	NR	41	2.405	1
Frndstr	Friendster	Social Network	Yes	NR	65	3.612	1
SK	SK-Domain	Web Graph	Yes	LWA	50	3.639	45
WbCc	Web-CC12	Web Graph	Yes	NR	89	3.872	464,919
UKDls	UK-Delis	Web Graph	Yes	LWA	110	6.919	80,443
UU	UK-Union	Web Graph	Yes	LWA	133	9.359	278,716
UKDmn	UK-Domain	Web Graph	Yes	KN	105	6.603	14,333
ClWb9	ClueWeb09	Web Graph	Yes	NR	1,685	15.622	5,642,809

TABLE III: Computing Machines

	SkyLakeX	Epyc
CPU Model	Intel Xeon Gold 6130	AMD Epyc 7702
CPU Frequency	2.10 GHz	2 GHz
Sockets	2	2
NUMA Nodes	2	8
Total CPU Cores	32	128
Hyperthreading	No	No
Total Threads	32	128
L1 Cache	32 KB / 1 core	32 KB / 1 core
L2 Cache	1 MB / 1 core	512 KB / 1 core
L3 Cache	22 MB / 16 cores	16 MB / 4 cores
Total Memory	768 GB	2,048 GB

V. Evaluation

A. Machines and Datasets

We use two different machines listed in Table III for evaluation. The machines use CentOS 7. We use an optimized implementation of CC in the `C` language that deploys the `pthread`, `libnuma`, and `papi` [42] libraries. We use the interleaved NUMA memory policy and apply work-stealing [43] for parallel processing of graph partitions created by vertex and edge partitioning [44], [25].

We create $32 * \#threads$ edge balanced partitions and partitions $[32*t,\ 32*(t+1))$ are initially assigned to thread t. A thread processes its own partitions and then steals partitions from threads on the same NUMA node and finally from threads on other NUMA nodes. In order to preserve locality in processing consequent partitions and increase reuse of cache contents, a thread processes its own partitions in ascending order and steals partitions from other threads in descending order.

We use the master-worker model for managing parallel threads and the `futex` syscall for thread synchronization. We compiled the source code using the `gcc-9.2` compiler with `-O3` optimization flag.

Table II shows the datasets and their sources: "*Konect*" (KN) [45], [46], [47], "*NetworkRepository*" (NR) [48], [49], [50], [51], [52], and "*Laboratory for Web Algorithmics*" (LWA) [47], [53], [49], [54], [55]. Numbers of edges are in billions and numbers of vertices are in millions, counted after removing zero degree vertices because of their destructive effect [25]. Graphs are represented in Compressed Sparse Row/Column [31] with $|V|+1$ index values of 8 bytes per index value and $|E|$ neighbour IDs of 4 bytes each. The column $|CC|$ shows the number of connected components in each dataset. We use 4 bytes data as label of a vertex.

For comparison to other CC algorithms, we use BFS-CC implemented in GraphGrind [30], [25] (commit `5099761`), and the Shiloach-Vishkin and Afforest implementations in GAP [56] (commit `6ac1afd`).

B. Comparison to Prior State of the Art

Table IV compares Thrifty to the prior state-of-the-art CC algorithms. For road networks (`GBRd` and `USRd`) that do not follow a power-law degree distribution SV, JT and Afforest are faster than Thrifty. For graphs larger than `LiveJournal`, Thrifty has the best results on both architectures: on the SkyLake machine, Thrifty provides up to $3.9\times$ and on average $1.6\times$ speedup to Afforest, and $8.4\times - 54.6\times$ speedup compared to the SV, JT, and BFS-CC algorithms. On the EPYC machine Thrifty provides $1.5\times$ speedup over Afforest and $7.3\times - 65.3\times$ over SV, JT, and BFS-CC algorithms.

The importance of Thrifty is not limited to having faster execution time in comparison to Disjoint Set algorithms. Disjoint Set algorithms like SV, Afforest, and JT are concurrent algorithms that do not scale to distributed memory systems. One attempt at distributed disjoint sets notes lack of scalability and net performance loss compared to sequential algorithms [26]. In contrast, the SpMV model of the Label Propagation algorithm allows successful scaling in distributed systems [28], [29].

TABLE IV: CC execution times in milliseconds for Shiloach-Vishkin (**SV**), **BFS-CC**, Direction Optimizing Label Propagation (**DO-LP**), Jayanti-Tarjan (**JT**), **Afforest**, and Thrifty Label Propagation (**Thrifty**)

Dataset	Intel SkyLakeX						AMD Epyc					
	SV	BFS-CC	DO-LP	JT	Afforest	Thrifty	SV	BFS-CC	DO-LP	JT	Afforest	Thrifty
GBRd	32	494	12,131	24	**13**	383	185	875	12,233	**24**	93	677
USRd	120	1,037	38,435	85	**38**	992	268	1,760	20,752	**46**	116	1,843
Pkc	34	24	52	23	**6**	9	28	39	69	40	**6**	20
WWiki	61	28	52	30	**4**	9	60	55	77	116	**6**	28
LJLnks	114	95	111	56	**13**	14	72	133	183	61	**13**	20
LJGrp	928	94	141	58	20	**12**	856	127	118	61	**16**	24
Twtr10	3,849	351	816	634	56	**38**	4,277	215	689	1,012	122	**49**
Twtr	3,017	577	918	307	50	**49**	2,895	1,016	734	270	117	**44**
Wbbs	2,794	2,931	9,822	972	187	**169**	3,208	5,059	6,251	956	182	**143**
TwtrMpi	7,544	723	3,186	1,067	264	**67**	5,740	556	2,793	828	152	**70**
Frndstr	10,405	1,556	11,709	2,559	207	**159**	6,516	677	13,446	2,483	225	**129**
SK	1,985	768	1,631	566	153	**104**	1,518	858	1,095	209	145	**108**
WbCc	25,673	4,267	5,831	1,977	213	**148**	28,790	7,346	6,090	1,419	248	**109**
UKDls	3,587	1,358	4,773	925	299	**234**	2,391	1,913	2,903	310	202	**198**
UU	4,580	3,190	6,128	1,760	343	**246**	3,470	4,879	3,738	708	253	**187**
UKDmn	3,776	1,146	3,781	877	258	**235**	1,934	1,462	2,527	299	213	**178**
ClWb9	170,879	-	47,397	13,942	3,665	**3,138**	210,342	148,586	36,776	11,509	2,470	**1,870**

A second limitation of the Disjoint Set algorithms originates from the fact that concurrent algorithms are very specific solutions to a problem and require great precision in their design and implementation [57]. Concurrent algorithms have limited potential to generalize to other problems. In contrast, the Thrifty algorithm that follows a SpMV model is more generic and conceptually simple. Numerous frameworks have been defined that present a reusable interface and hide numerous performance optimizations behind that interface, out of the concern of the user [27], [28], [29], [37], [58], [39].

C. Has Thrifty Reached Its Goals?

In this section we consider more details of the execution of Thrifty to identify if it has reached the goals we explained in Section IV. In the experiments of this section when we refer to iterations of Thrifty, we count the **Initial Push** as an iteration.

1) Faster Label Propagation and Reducing Number of Iterations: To facilitate faster propagation of the labels, we suggested using the **Unified Labels Array** technique and Table V shows that Thrifty reduces the total iterations by 39%, on average. For `WebBase-2001`, Thrifty reduces total iterations by 89%.

2) Work Reduction: Figure 5 compares the speedup provided by Thrifty in comparison to DO-LP. It also shows the percentage of edges of graphs that are processed by the Thrifty and DO-LP algorithms. It shows that Thrifty reduces the total traversed edges by at least 97%. In fact, Thrifty processes up to only 4.4% of the edges of the graph which shows the **Zero Convergence** can significantly reduce the total work.

Figure 6 also compares Thrifty to DO-LP for reduction in (1) the last level cache misses, (2) memory accesses (load and store memory instructions), (3) branch mis-predictions, and (4) hardware instructions. It shows that Thrifty cuts at least 80% of the redundant work done by DO-LP.

TABLE V: Comparison of the number of iterations required by DO-LP and Thrifty

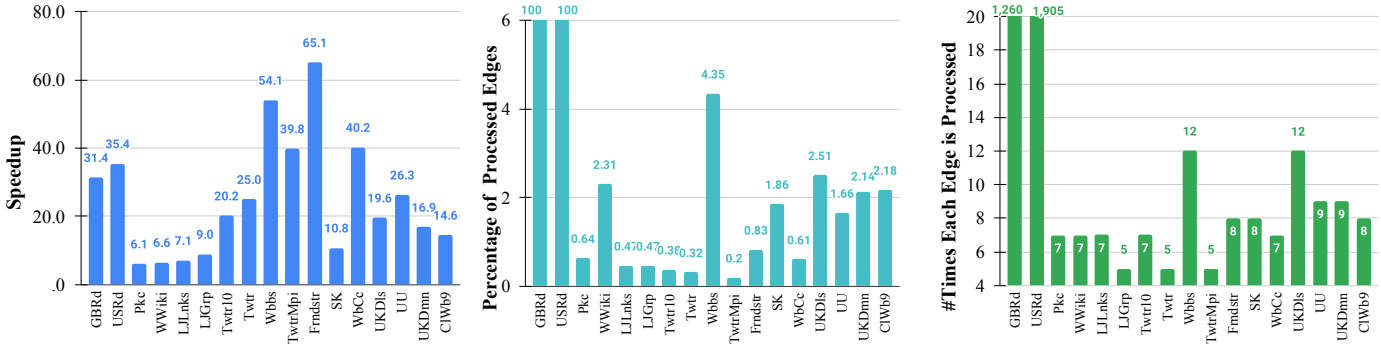
Dataset	DO-LP	Thrifty	Ratio
Pkc	10	5	0.50
WWiki	17	13	0.76
LJLnks	15	6	0.40
LJGrp	7	4	0.57
Twtr10	17	12	0.71
Twtr	15	11	0.73
Wbbs	744	82	0.11
TwtrMpi	15	11	0.73
Frndstr	24	12	0.50
SK	23	20	0.87
WbCc	32	30	0.94
UKDls	79	21	0.27
UU	142	99	0.70
UKDmn	76	41	0.54
ClWb9	79	70	0.89

3) Faster Propagation of The Zero Label: The **Zero Planting** technique is used in the Thrifty algorithm to propagate the zero label from the vertex with the maximum degree. To see its effect on propagation of the zero label, we compare the percentage of converged vertices in the DO-LP and Thrifty in Figures 7 and 8.

It shows that in DO-LP only 34.8% of the vertices are converged after the first four pull iterations, but Thrifty propagates the zero label much faster which results in convergence of 88.3% of the vertices after the first pull iteration.

4) Work Efficient Initial Iteration: To accelerate the first iterations of the Label Propagation algorithm, the **Initial Push** technique uses an initial push to propagate the zero label from the vertex with the maximum degree. To evaluate if this technique is useful, we compare the initial iterations of the DO-LP and Thrifty in Table VI.

If we compare the time spent in the first iteration of DO-LP

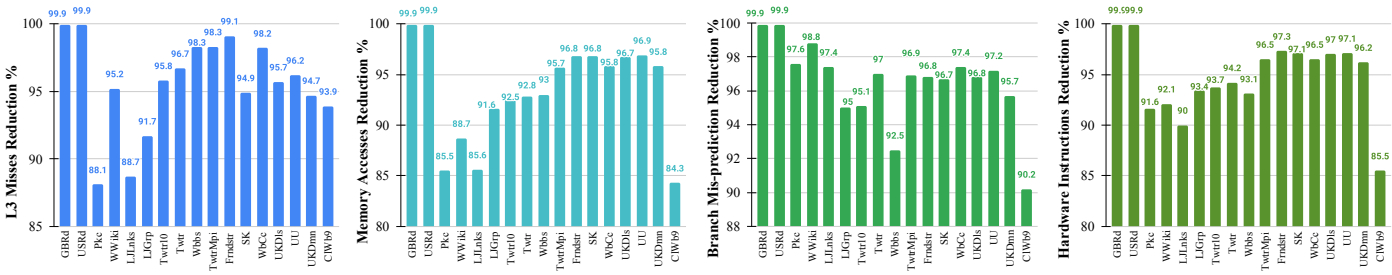


(a) Speedup of Thrifty in comparison to DO-LP

(b) Percentage of edges that are processed by Thrifty

(c) Number of times edges are processed by DO-LP

Fig. 5: Speedup and processed edges on the SkyLakeX machine



(a) Last level cache misses

(b) Memory accesses (load and store memory instructions)

(c) Branch mis-predictions

(d) Hardware instructions

Fig. 6: Reduction (in percent) of hardware events on the SkyLakeX machine for execution of Thrifty in comparison to DO-LP (higher is better)

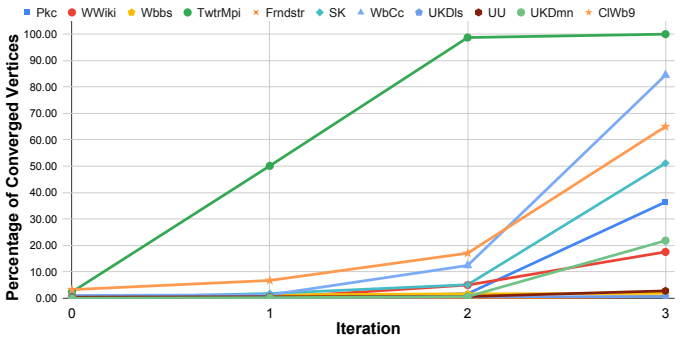


Fig. 7: Percentage of converged vertices in the DO-LP

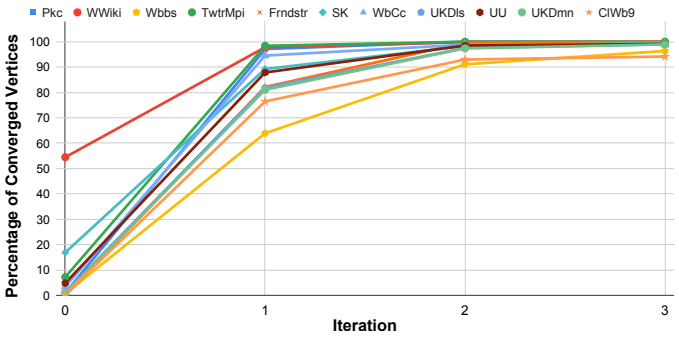


Fig. 8: Percentage of converged vertices in Thrifty

to the sum of the first iteration of Thrifty and its Initial Push, we see that the **Initial Push** technique accelerates the initial iteration of label propagation by $1.9\times - 14.2\times$ and on average by $5.3\times$. Moreover, Figures 7 and 8 show that more vertices are converged in the first pull iteration of Thrifty.

D. Effect of The Optimizations

Figures 9 and 10 show the effect of the Unified Labels Array technique in comparison to the effect of Zero Convergence, Zero Planting and Initial Push techniques. The last three techniques are dependent on each other and therefore we measured the cumulative improvement of them.

For this experiment, we have executed a variant of the DO-LP that also implements the Unified Labels Array technique. Then we have compared the execution time of DO-LP, this new variant, and Thrifty. The difference between execution time of this variant to DO-LP shows the effect of the Unified Labels Array technique, and its difference to Thrifty shows the effect of the Zero Convergence, Zero Planting and Initial Push techniques.

The figures show that, on average, about 65% of Thrifty's improvement is achieved by the Unified Labels Array technique and 35% by the Zero Convergence, Zero Planting and Initial Push techniques.

E. Effect of The Threshold

To explain the effect of threshold for selecting push and pull traversals (Section IV-E), Table VII shows the execution

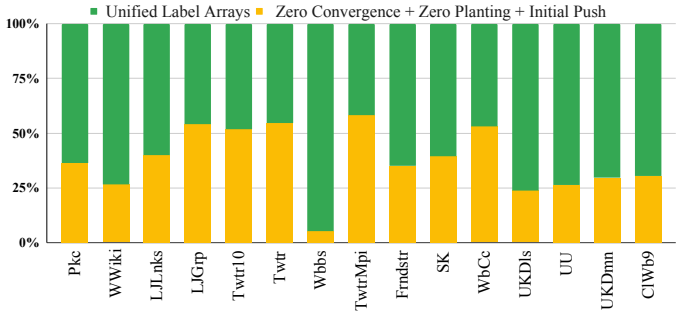


Fig. 9: Contribution of the optimization techniques on the SkyLakeX machine

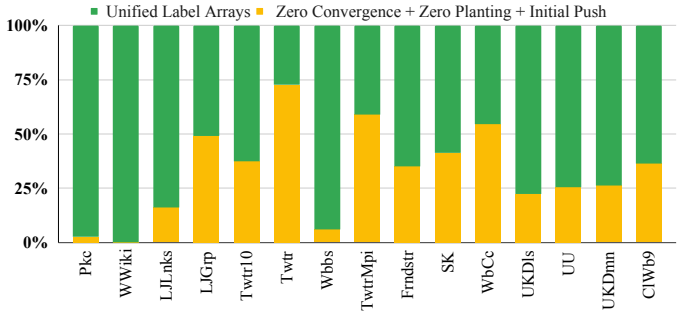


Fig. 10: Contribution of the optimization techniques on the Epyc machine

TABLE VI: Execution time of the first iterations on the SkyLakeX machine in milliseconds

Dataset	**DO-LP**	**Thrifty**		**Speedup**
	Iteration 0	Iteration 0	Iteration 1	
	Pull	Initial Push	Pull with Zero Convergence	
Pkc	6.1	0.9	2.0	2.1
WWiki	5.1	1.5	1.2	1.9
LJLnks	12.6	0.6	4.5	2.5
LJGrp	24.7	0.9	3.5	5.6
Twtr10	106.6	2.0	17.6	5.4
Twtr	170.7	3.3	17.6	8.2
Wbbs	136.2	1.7	71.1	1.9
TwtrMpi	617.4	3.4	40.2	14.2
Frndstr	1,415.9	0.7	107.3	13.1
SK	176.1	3.6	52.2	3.2
WbCc	799.0	5.6	73.7	10.1
UKDls	325.8	1.7	133.3	2.4
UU	452.1	3.2	144.8	3.1
UKDmn	319.5	1.8	122.7	2.6
ClWb9	4,350.2	12.2	1,435.7	3.0

TABLE VII: Effect of the threshold

	Threshold = 1%			**Threshold = 5%**		
Iteration	Traversal	Density	Time (ms)	Traversal	Density	Time (ms)
2	Pull	1.16%	11.7	Pull	1.13%	11.5
3	Pull	0.01%	6.8	Pull Frontier	-	8.8
4	Pull Frontier	-	5.0	Push	-	20.2
5	Push	-	2.3	Push	-	0.8

of the first iterations of the `Friendster` dataset on the Epyc machine. By using 1% as the threshold, iteration 4 is performed in the pull direction that requires 5ms but it is 20ms in the push direction.

It shows that the **Zero Convergence** optimization is able to accelerate the pull iterations which makes it harder for the push traversal to compete. Therefore, it is necessary to have a smaller threshold.

VI. Related Work

The effectiveness of push and pull traversals for different graph analytics is discussed in [59], [33], [34], and the differences between locality of push and pull traversals have been studied in [60]. iHTL [61] optimizes temporal locality by applying push and pull directions in one graph traversal but for different types of vertices. In order to improve the performance of the CC algorithm, we have used the direction optimizing CC as the baseline. DO-LP selects push traversal for sparse iterations (where a small number of vertices are in the frontier) and applies pull traversal for dense iterations.

The Shiloach-Vishkin CC algorithm [19] is the first Disjoint Set CC. It makes a number of iterations, each of which makes a pass over the graph. Each iteration is started by a hook phase that attaches roots of subgraphs based on edges and is followed by a shortcut phase that updates the label of each vertex of subgraphs by the label of its root. BFS is used in [62] to accelerate SV.

FastSV [63] improves performance of LACC [64], however, both algorithms use MIN operator over labels to decide if the label of a vertex should be changed. It makes these algorithms variants of the Label Propagation CC instead of SV. The main difference between LP and SV algorithms is in the observations that result in changing the label of a vertex: SV-based algorithms consider the topology of a graph (i.e., edges between vertices) to change their labels, while LP-based algorithms consider the label of a vertex in relation to labels of its neighbours to identify its new label. Shortcutting technique is used in [65] to accelerate the label propagation CC.

The Jayanti-Tarjan [21] optimizes the SV algorithm based on a linearizable randomized linking strategy. It requires only one traversal of the graph. Afforest [22] uses sampling to reduce the total number of processed edges. ConnectIt [24] extends Afforest by combining various sampling methods with various CC algorithms. We attempted to evaluate ConnectIt but its code repository was under modification and could not be compiled (we are communicating on this with the authors).

VII. Conclusion and Future Work

The conceptual simplicity, in conjunction with the uncontested dominance of Label Propagation in distributed memory systems, prompts us to revisit Label Propagation in shared memory systems. We developed performance optimization techniques to improve the DO-LP algorithm based on the features implied by the structure of real-world graph datasets that follow a scale-free degree distribution:

1) The **Unified Labels Array** technique reduces the number of iterations by 39%.
2) The **Zero Convergence** technique reduces the number of processed edges of the graphs to 1.4% of the edges, on average.
3) The **Zero Planting** technique provides fast convergence of 88% of the vertices after the first pull iteration.
4) The **Initial Push** technique accelerates the initial iteration of label propagation by $5.3\times$, on average.

We introduced **Thrifty Label Propagation**, which deploys these techniques. Our evaluation of Thrifty against state-of-the-art CC algorithms on two different processor architectures shows $1.4\times$ speedup over Afforest, $7.3\times$ over Jayanti-Tarjan, $14.7\times$ over BFS-CC, $51.2\times$ over SV. The Thrifty algorithm is faster than the Direction Optimizing Label Propagation by $25.2\times$.

An important question for future work is how Thrifty applies in a distributed processing setting, where label propagation algorithms are the norm. We plan to apply Thrifty to a distributed processing model like KLA [38]. Moreover, the unordered scheduling of the vertices based on the KLA model can be used in a shared memory system to provide better CPU utilization.

The optimization techniques we expressed for the Label Propagation algorithm are not strictly limited to connected components. In future work we will investigate how these can be generalized to other algorithms expressed in the SpMV model of graph processing. In particular, we wish to explore the connection between the unified arrays optimization and asynchronous execution.

Code Availability

Source code repository and further discussions relating to this paper are available online in https://blogs.qub.ac.uk/graphprocessing/Thrifty-Label-Propagation-Fast-Connected-Components-for-Skewed-Degree-Graphs/.

Acknowledgements

We are grateful for the constructive feedback of our CLUSTER reviewers.

This work is partially supported by the High Performance Computing center of Queen's University Belfast and the Kelvin supercomputer (EPSRC grant EP/T022175/1) and by DiPET (CHIST-ERA project CHIST-ERA-18-SDCDN-002, EPSRC grant EP/T022345/1).

First author is supported by a scholarship of the Department for the Economy, Northern Ireland and Queen's University Belfast.

References

[1] R. Albert, "Scale-free networks in cell biology," *Journal of cell science*, vol. 118, no. 21, pp. 4947–4957, 2005.

[2] Y. Assenov, F. Ramírez, S.-E. Schelhorn, T. Lengauer, and M. Albrecht, "Computing topological parameters of biological networks," *Bioinformatics*, vol. 24, no. 2, pp. 282–284, 11 2007.

[3] F. Hüffner, C. Komusiewicz, A. Liebtrau, and R. Niedermeier, "Partitioning biological networks into highly connected clusters with maximum edge coverage," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 11, no. 3, pp. 455–467, 2013.

[4] M. Dillencourt, H. Samet, and M. Tamminen, "A general approach to connected-component labeling for arbitrary image representations," *J. ACM*, vol. 39, pp. 253–280, 1992.

[5] S. Gazagnes and M. H. Wilkinson, "Distributed connected component filtering and analysis in 2d and 3d tera-scale data sets," *IEEE Transactions on Image Processing*, vol. 30, pp. 3664–3675, 2021.

[6] E. J. Ruiz, V. Hristidis, C. Castillo, A. Gionis, and A. Jaimes, "Correlating financial time series with micro-blogging activity," in *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, ser. WSDM '12. New York, NY, USA: ACM, 2012, p. 513–522.

[7] W. Jang and M. Hendry, "Cluster analysis of massive datasets in astronomy," *Statistics and Computing*, vol. 17, no. 3, pp. 253–262, 2007.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, p. 226–231.

[9] J. Gan and Y. Tao, "DBSCAN revisited: Mis-claim, un-fixability, and approximation," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, p. 519–530.

[10] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger, "Scan: A structural clustering algorithm for networks," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 824–833.

[11] Y. Lim, U. Kang, and C. Faloutsos, "Slashburn: Graph compression and mining beyond caveman communities," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 3077–3089, Dec 2014.

[12] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 472–488.

[13] V. Singh and T. E. Carlson, "Pim-graphscc: Pim-based graph processing using graph's community structures," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 151–154, 2020.

[14] R. Tarjan, "Depth first search and linear graph algorithms," *Siam Journal on Computing*, vol. 1, no. 2, 1972.

[15] S. Orzan, "On distributed verification and verified distribution," Ph.D. dissertation, PhD thesis, Free University of Amsterdam, 2004.

[16] L. Fleischer, B. Hendrickson, and A. Pinar, "On identifying strongly connected components in parallel," in *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, ser. IPDPS '00. Berlin, Heidelberg: Springer-Verlag, 2000, p. 505–511.

[17] G. Slota, S. Rajamanickam, and K. Madduri, "BFS and coloring-based parallel algorithms for strongly connected components and related problems," in *Proceedings - IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS 2014*, ser. Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS. United States: IEEE Computer Society, 2014, pp. 550–559.

[18] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, p. 036106, 2007.

[19] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[20] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '91. New York, NY, USA: ACM, 1991, p. 370–380.

[21] S. V. Jayanti and R. E. Tarjan, "A randomized concurrent algorithm for disjoint set union," *CoRR*, vol. abs/1612.01514, 2016.

[22] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 12–21.

[23] B. Awerbuch and Y. Shiloach, "New connectivity and msf algorithms for shuffle-exchange network and pram," *IEEE Trans. Comput.*, vol. 36, no. 10, p. 1258–1263, Oct. 1987.

[24] L. Dhulipala, C. Hong, and J. Shun, "Connectit: A framework for static and incremental parallel graph connectivity algorithms," *CoRR*, vol. abs/2008.03909, 2020.

[25] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing*. ACM, 2017.

[26] F. Manne and M. M. A. Patwary, "A scalable parallel union-find algorithm for distributed memory computers," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 186–195.

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

[28] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *2009 Ninth IEEE International Conference on Data Mining*, 2009, pp. 229–238.

[29] J. Kepner, D. Bader, A. Buluc, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the graphblas: Seven good reasons," *Procedia Computer Science*, vol. 51, pp. 2453–2462, 2015, international Conference On Computational Science, ICCS 2015.

[30] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Accelerating graph analytics by utilising the memory locality of graph partitioning," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 181–190.

[31] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations - version 2," 1994.

[32] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 78–88.

[33] B.-Y. Su, T. G. Brutch, and K. Keutzer, "Parallel bfs graph traversal on images using structured grid," in *2010 IEEE International Conference on Image Processing*, Sep. 2010, pp. 4489–4492.

[34] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 12:1–12:10.

[35] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. ACM, 2013.

[36] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30.

[37] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, p. 1214–1225, Jul. 2015.

[38] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "KLA: A new algorithmic paradigm for parallel graph computations," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, p. 27–38.

[39] H. Vandierendonck, "Graptor: Efficient pull and push style vectorized graph processing," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: ACM, 2020.

[40] A.-L. Barabaso and E. Bonabeau, "Scale-free networks," *Scientific american*, vol. 288, no. 5, pp. 60–69, 2003.

[41] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2014, pp. 979–990.

[42] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[43] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, p. 720–748, Sep. 1999.

[44] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 183–193.

[45] J. Kunegis, "KONECT – The Koblenz Network Collection," in *Proc. Int. Conf. on World Wide Web Companion*, 2013, pp. 1343–1350.

[46] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, p. 29–42.

[47] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[48] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[49] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 595–602.

[50] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *ICWSM*, Washington DC, USA, May 2010.

[51] F. social network, "Friendster: The online gaming social network," archive.org/details/friendster-dataset-201107.

[52] C. L. Clarke, N. Craswell, and I. Soboroff, "Overview of the trec 2009 web track," DTIC Document, Tech. Rep., 2009.

[53] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive crawling for the masses," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2014, pp. 227–228.

[54] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 587–596.

[55] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, p. 591–600.

[56] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[57] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[58] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018.

[59] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "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*, ser. HPDC '17. New York, NY, USA: ACM, 2017, p. 93–104.

[60] M. Koohi Esfahani, P. Kilpatrick, and H. Vandierendonck, "How do graph relabeling algorithms improve memory locality?" in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 2021, pp. 84–86.

[61] ——, "Exploiting in-Hub Temporal Locality in SpMV-based Graph Processing," in *Proceedings of The 50th International Conference on Parallel Processing*, ser. ICPP '21. Lemont, IL, USA: ACM, 2021.

[62] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru, "An adaptive parallel algorithm for computing connected components," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2428–2439, 2017.

[63] Y. Zhang, A. Azad, and Z. Hu, "Fastsv: A distributed-memory connected component algorithm with fast convergence," in *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2020, pp. 46–57.

[64] A. Azad and A. Buluc, "Lacc: A linear-algebraic algorithm for finding connected components in distributed memory," in *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 2019, pp. 2–12.

[65] S. Stergiou, D. Rughwani, and K. Tsioutsiouliklis, "Shortcutting label propagation for distributed connected components," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, ser. WSDM '18. New York, NY, USA: ACM, 2018, p. 540–546.