GENERATING LANDMARK LABELS FOR SHORT DISTANCE QUERIES IN A DISTRIBUTED SETTING

by
ARDA ŞENER

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
December 2022

Arda Şener 2022 ©

All Rights Reserved

ABSTRACT

GENERATING LANDMARK LABELS FOR SHORT DISTANCE QUERIES IN A DISTRIBUTED SETTING

ARDA ŞENER

Computer Science & Engineering M.S. THESIS, DECEMBER 2022

Thesis Supervisor: Assoc. Prof. Kamer Kaya

Keywords: Graphs, High-Performance Computing, Parallel Algorithms, Distance Queries

Distance queries are a fundamental part of many network analysis applications. Distances can be used to infer the closeness of two users in social networks, the relation between two websites in a web graph, or the importance of the interaction between two proteins or molecules. As a result, being able to answer these queries rapidly has many benefits to the area of network analysis as a whole. Pruned landmark labeling is a technique used to generate an index for a given graph that allows the shortest path queries to be completed in a fraction of the time when compared to a standard BFS (Breadth First Search) based algorithm. PSL (Parallel Shortest-distance Labeling) is a pruned landmark labeling algorithm that is designed to be implemented in a multithreaded environment and works particularly well on social networks. Unfortunately, even for a medium-size, 50 million vertex graph, the index size can be as large as 300GB. On the same graph, a single CPU core takes more than 12 days to generate the index. This thesis aims to implement PSL in a distributed environment by partitioning the input graph and distributing the partitions to the nodes. Our method can provide improvements in both the execution time and the memory consumption by distributing both across multiple nodes of a cluster. Furthermore, we develop techniques and conduct experiments that can help increase the performance of the PSL algorithm.

ÖZET

DAĞITIK ORTAMDA EN KISA YOL SORGULARI İÇİN YER İŞARETİ ETİKETLERİ OLUŞTURMA

ARDA ŞENER

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, ARALIK 2022

Tez Danışmanı: Doç. Dr. Kamer Kaya

Anahtar Kelimeler: Çizgeler, Yüksek Başarımlı Hesaplama, Paralel Algoritmalar, Uzaklık Sorguları

Uzaklık sorguları ağ analiz işlemlerinin önemli ve temel bir parçasıdır. Bu sorgular sosyal ağlarda kullanıcıların yakınlığının öğrenilmesi, internet üzerinde sitelerin ilişkilerinin karşılaştırılması, biyolojik ağlarda moleküllerin birbiriyle etkileşimlerinin incelenmesi gibi alanlarda kullanılabilir. Dolayısıyla, bu sorguların hızlı bir şekilde cevaplanabilmesi ağ analizi alanına genel olarak yarar sağlamaktadır. PLL (Pruned Landmark Labeling) adı verilen algoritma, bu sorguların çok daha kısa sürede cevaplanabilmesini sağlayan yer işaretleri oluşturmak için literatürde sıklıkla kullanılmaktadır. PSL (Parallel Shortest-distance Labeling) algoritması PLL tabanlı paralel hesaplama yapılabilen ortamlarda kullanılmak üzere tasarlanmış ve özellikle sosyal ağlarda kullanılan bir algoritmadır. Fakat PLL tabanlı algoritmaların hafıza karmaşıklığı oldukça fazladır. Örneğin, orta boyutlu çizgeler için bile oluşturulan yer işaretleri hafızada 300GB üzerinde yer kaplayabilmektedir. Bununla beraber, orta boyutlu çizgelerde, modern bir CPU çekirdeği ile yer işaretlerini oluşturmak için 12 günden uzun süre harcayabilmektedir. Bu tez PSL algoritmasının dağıtık bir ortamda uygulanmasının çizgenin bölünmesi ve dağıtılması aracılığı ile uygulanması üzerinedir. Bu teknik ile hem zaman, hem kullanılan hafıza açısından önemli kazanımlar sağlanmıştır. Ek olarak, bu tez, PSL algoritmasının performansının artırılmasına yönelik deney ve teknikler de içermektedir.

ACKNOWLEDGEMENTS

I want to thank;

My thesis advisor Dr. Kamer Kaya for his help and guidance throughout this project and my education;

My colleagues from the Sabanci University HPC lab who shared their experiences and knowledge with me;

My friends and family who were there for me and helped me deal with the stress and pressure;

IT4Innovations National Supercomputing Center of the Technical University of Ostrova for providing us access to the Karolina cluster which is used for most of the experiments provided in this text.

This work was supported by the Scientific and Technological Research Council of Turkey (TUBİTAK) and EuroHPC Joint Undertaking through grant agreement No. 220N254. The numerical calculations reported in this text were partially performed at TUBİTAK ULAKBİM, High Performance and Grid Computing Center.

Dedicated to my mother.

TABLE OF CONTENTS

LIST OF TABLES .... **x**

LIST OF FIGURES .... **xii**

1. INTRODUCTION **1**

2. NOTATION AND TERMINOLOGY **3**

3. PROBLEM STATEMENT AND BACKGROUND **5**

3.1. Problem Statement 5
3.2. 2-Hop Labeling 6
3.3. Sequential Approaches 7
3.4. Parallel Approaches 9
3.5. Distributed Approaches 11
3.6. Compression Methods 12
3.7. Bit-Parallel Labels 13

4. METHODOLOGY **16**

4.1. Leaf Elimination 16
4.2. Partitioning 17
4.3. Vertex Separation 18
4.4. Ranking the Vertex Separator 20
4.5. Synchronization 20
4.6. Summary and Hypothesis 22
4.7. Technical Details 23

5. EXPERIMENTAL RESULTS **26**

5.1. Experimental Setup 26
5.2. Shared Memory Experiments 28
5.2.1. Performance Evaluation of NPSL 29
5.2.2. Bit-Parallel Label Experiments 31

5.2.3. Compression Experiments 34
5.2.4. Ranking Experiments 35
5.2.5. Vertical Scalability Experiments 37
5.3. Distributed Memory Experiments 41
5.3.1. Performance Evaluation of DPSL 41
5.3.2. Comparison to Previous Work 47
5.4. Threats to Validity 50
6. CONCLUSION 52
7. FUTURE WORK 53
BIBLIOGRAPHY 56
APPENDICES 58

LIST OF TABLES

Table 5.1. Specifications of the systems used for the experiments 26
Table 5.2. Graphs used for the experiments 27
Table 5.3. Parameters for DPSL and NPSL 28
Table 5.4. Compression levels for DPSL and NPSL 28
Table 5.5. [Karolina] Comparison of indexing times between PSL and NPSL with CL=2 on hard and moderately hard graphs. The first two columns show the indexing times in seconds for PSL and NPSL respectively. The last column shows the speedup of NPSL with respect to PSL. 29
Table 5.6. [Karolina] Indexing time (in seconds) comparison of NPSL with compression levels 2 and 3 on hard and moderately hard graphs with the speedup obtained when going from CL=2 to CL=3. 34
Table 5.7. [Karolina] Comparison of memory consumption of labels between compression levels 2 and 3 of NPSL 35
Table 5.8. [Gandalf] Memory used to store the labels (in gigabytes) with different ranking methods on easy and moderately easy graphs. Degree refers to degree-based ranking. Degree + VS refers to degree-based ranking where the ranks of the vertex separator (for a 4 node partitioning) has been increased. Reduction shows the percentage decrease in memory usage when going from Degree to Degree + VS. .. 37
Table 5.9. [Gandalf] Memory used to store the labels (in gigabytes) with different ranking methods on easy and moderately easy graphs. BC refers to betweenness centrality-based ranking. BC + VS refers to betweenness centrality-based ranking where the ranks of the vertex separator (for a 4 node partitioning) has been increased. Reduction shows the percentage decrease in memory usage when going from BC to BC + VS. 37
Table 5.10. [Gandalf] LLC miss rate of NPSL on some moderately easy graphs with varying thread counts with BP labels. 40

Table 5.11. [Gandalf] LLC miss rate of NPSL on some moderately easy graphs with varying thread counts without BP labels. 40
Table 5.12. [Karolina] Indexing time (seconds) of DPSL on hard and moderately hard graphs on a varying number of nodes 44
Table 5.13. [Karolina] Speedup of DPSL on hard and moderately hard graphs with respect to single node execution (NPSL) 44
Table 5.14. [Karolina] Total memory consumption (in GBs) of the label cover in 2,4,8 node DPSL when compared to NPSL (denoted as 1 node) for hard and moderately hard graphs. For DPSL, the values are the summation of the memory used to store the label sets across every node.. 46
Table 5.15. [Karolina] Maximum per node memory consumption (in GBs) of the label cover in 2,4,8 node DPSL when compared to NPSL (denoted as 1 node) for hard and moderately hard graphs. For DPSL, the values are the memory consumption result from the node that stores the largest number of labels.................................... 46
Table 5.16. [Gandalf] Size of the label cover in gigabytes using degree ranking for hard and moderately hard graphs. The first column is an unmodified NPSL execution. The second column is an NPSL execution where the vertex separator vertices are increased in rank as in Tables 5.8 and 5.9. The third column is a 4 node DPSL execution where the generated label covers are merged with the duplicates removed. The fourth column shows the increase from the second column to the third column. 46
Table 5.17. [Karolina] Indexing time of DPSL in comparison to the prior work DVCPLL on 4 nodes (32 threads per node) on hard and moderately hard graphs. 49

Table A1. [Karolina] Indexing time (seconds) of NPSL on moderately easy graphs when different OpenMP thread scheduling methods are used. . 58

LIST OF FIGURES

Figure 3.1. A diagram showing the label set and label cover terms with respect to a small example graph 6

Figure 3.2. A figure representing the various terms used for BP labels. The green vertex is the root, the blue vertices are sub-roots, and the red vertex is any vertex in the graph........ 14

Figure 4.1. An example showing the effect of leaf elimination. The Green vertex is the leaf, the yellow vertex is the parent of the leaf, gray vertices are eliminated meaning they will not be processed during indexing. The numbers on the vertices represent their ranks. We can see that leaf elimination causes an additional vertex to be eliminated. 17

Figure 4.2. A diagram showing the various steps of our distributed PSL algorithm. The numbers on the vertices indicate their ranks. The red vertices are processed by all nodes and are synchronized. The blue and green vertices are each processed by a different node. 22

Figure 4.3. Comparison of adjacency list (left) and CSR (right) sparse data storage formats. 23

Figure 5.1. [Karolina] Comparison of indexing times for NPSL and PSL using different compression levels on easy and moderately easy graphs. CL=3 is not implemented in PSL and is omitted as a result. 30

Figure 5.2. [Karolina] Effect of Bit-Parallel Label's root count (BPR) on the indexing time of NPSL on moderately easy and moderately hard graphs. 32

Figure 5.3. [Karolina] Effect of Bit-Parallel Label's root count (BPR) on the memory consumption of NPSL when processing the graphs TOPC and DBLP. 33

Figure 5.4. [Karolina] Indexing time (in seconds) comparison of NPSL with compression levels 2 and 3 on easy and moderately easy graphs. 34

Figure 5.5. [Nebula] Effect of shuffling a portion of the vertex ranks on the indexing time on moderately easy graphs. 35
Figure 5.6. [Nebula] Memory consumption of the labels generated by NPSL when different ranking methods are used on moderately easy graphs. 36
Figure 5.7. [Nebula] Indexing time of NPSL when different ranking methods are used on moderately easy graphs. 36
Figure 5.8. [Karolina] Indexing time of NPSL with varying thread counts on moderately easy grahps. 38
Figure 5.9. [Karolina] Indexing time of NPSL with varying thread counts on hard and moderately hard graphs 38
Figure 5.10. [Karolina] Indexing time of NPSL with varying thread counts on hard and moderately hard graphs without using BP labels 40
Figure 5.11. [Gandalf, NT=15] Indexing time of the different levels of NPSL and DPSL when processing DBLP. The first four series are the 4 processes used for DPSL, DPSL is the maximum of these four (the other processes wait for the slowest one). 41
Figure 5.12. [Gandalf, NT=15] Label counts added on each level for NPSL and DPSL when processing DBLP. The first four series are the 4 processes used for DPSL, DPSL total is the sum of those 4 processes, DPSL Max. Per Node is the maximum value among all the processes. 42
Figure 5.13. [Karolina] Indexing time of DPSL on moderately easy graphs with and without compression on a varying number of nodes 43
Figure 5.14. [Karolina] 128 thread NPSL and 32 thread 4 node DPSL execution comparison on hard and moderately hard graphs in terms of indexing time. 44
Figure 5.15. [Karolina] Breakdown of the execution times of 4 node DPSL executions on hard and moderately hard graphs. 45
Figure 5.16. [Karolina] Indexing time of DPSL in comparison to the prior works DVCPLL and DPLANT on 4 nodes (32 threads per node) on moderately easy graphs. 47
Figure 5.17. [Karolina] Indexing time of DPSL in comparison to the prior works DVCPLL and DPLANT on a varying number of nodes (32 threads per node) on the graph DBLP 48
Figure 5.18. [Karolina] Indexing time of DPSL in comparison to the prior works DVCPLL and DPLANT on 4 nodes with a varying number of threads on the graph DBLP 49

Figure A1. [Karolina] Effect of Bit-Parallel Label's root count (BPR) on the memory consumption of NPSL when processing moderately easy graphs. 59
Figure A2. [Karolina] The experiment from Figure 5.18 repeated on the graph FLIX 60

1. INTRODUCTION

Given a graph, a distance is the length of a path between two given vertices where the path is chosen based on a certain feature. For the shortest path distance, the length of a shortest path between two vertices is taken. The problem of computing the distance between only two specific vertices of a graph is referred to as the point-to-point shortest-path distance (PPSD) problem.

Simple traversal-based algorithms can be used to answer queries for such distances (Bellman, 1958; Dijkstra, 1959). However, running such an algorithm on every query may not be feasible for all applications. For example, ranking searches and finding influential communities in social networks require answering these queries faster than the current traversal-based algorithms can achieve (Li, Wang, Deng, Yang, Sellis & Yu, 2017; Vieira, Golgher, Fonseca, Reis, Damazio & Ribeiro-Neto, 2007).

A trivial solution to this problem is to use an algorithm like Dijkstra's Algorithm (Dijkstra, 1959) to compute PPSD for all pairs of vertices in the graph and store the results in a lookup table. Then answering the queries can be done in constant time by simply returning the result from the lookup table. However, many graphs used in the previously mentioned applications are quite large, reaching billions of vertices. As a result, the quadratic space consumption of this method makes it infeasible for such applications.

One popular solution to this problem is the use of 2-hop labeling. With this approach, vertices store labels that are composed of a target vertex and the distance to the target vertex. The combination of these labels forms a label cover of the graph. Querying under this method involves finding a common vertex in the labels of the vertices on the query. This querying method reduces the number of labels needed and as a result, this method scales better to large graphs. However, finding the minimal set of labels to be able to answer queries correctly is NP-hard (Cohen, Halperin, Kaplan & Zwick, 2003). Several approaches to compute nearly optimal label sets have been proposed in the literature.

Pruned Landmark Labeling (PLL) (Akiba, Iwata & Yoshida, 2013) is a very popular

approach and one that is the basis of most algorithms discussed in this text. In this approach, the cover is generated by executing a traversal algorithm from each vertex adding the label associated with the vertex to every other vertex. As the algorithm's name suggests, labels that would not improve the querying precision are pruned. This method works better than many of its predecessors in terms of speed, label set size, and its applicability to many different types of graphs. Several parallel implementations of PLL exist but are often limited in performance due to dependency issues caused by the pruning operation requiring previously generated labels e.g., Jin, Peng, Wu, Dragan, Agrawal & Ren (2020); Lakhotia, Dong, Kannan & Prasanna (2019); Qiu, Zhao, Zhu, Wang, Yuan & Wolf (2018).

Parallel Shortest-Distance Labeling (PSL) (Li, Qiao, Qin, Zhang, Chang & Lin, 2021) is an algorithm derived from PLL that is designed specifically for parallelism and use with low-diameter graphs (such as social networks and web graphs). In this approach, the label set is generated by each vertex taking labels from its neighborhood repeatedly. Similar to PLL, a pruning step is applied to only take the necessary labels. The label set generated by this method is identical to PLL. Li et al. (2021) suggested several compression methods to reduce the size of this set.

The contributions of this thesis are as follows:

- A new implementation of the PSL algorithm with improvements on the indexing time,
- Application of a new compression technique to the PSL algorithm which has a small but consistent effect on the memory usage,
- A distributed implementation of the PSL algorithm utilizing graph partitioning,
- Experiments on various parameters of the PSL algorithm like bit-parallel label size and ranking method.

2. NOTATION AND TERMINOLOGY

We describe a graph informally as a set of objects referred to as vertices connected by another set of objects referred to as edges. Formally, we can denote a graph as $G = (V, E)$ where $V = \{v_0, v_1, v_2, \cdots, v_N\}$ is the set of vertices and $E \subseteq V \times V$ is the set of edges. The vertex and edge sets can optionally be weighted by assigning additional numerical values to each vertex and edge respectively. This work is mostly focused on unweighted graphs which contain neither edge nor vertex weights. Graphs can also be directed or undirected. An undirected graph has the property $(v_x, v_y) \in E \implies (v_y, v_x) \in E$ for any distinct $v_x, v_y \in V$. This property does not exist on directed graphs. This work is mostly focused on undirected graphs.

When referring to the members of the set V of the graph $G = (V, E)$ we will exclusively use the term "vertex". A common synonym for "vertex" is "node" which is exclusively used in this text to refer to the computation nodes present in a compute cluster.

The neighborhood of a vertex v on an undirected graph $G = (V, E)$ is defined as a set $ngh(v)$ such that, $(v, u) \in E$ for all $u \in ngh(v)$. The degree of a vertex v ($deg(v)$) is simply the number of vertices in its neighborhood.

Some of the symbols and notations used in this text:

- $G = (V, E)$: The graph where V is the set of vertices and E is the set of edges.
- N : The number of vertices in the graph. (i.e., $|V|$)
- M : The number of edges in the graph. (i.e., $|E|$)
- B : The number of bit-parallel label roots used in the PSL and DPSL algorithms.
- ϵ : The user-defined balance constraint of the partitioning algorithm.
- $ngh(v)$: The neighborhood of the vertex v.
- $rank(v)$: The rank of the vertex v.

- $deg(v)$: The degree of the vertex v (i.e., $|ngh(v)|$)
- $L(v)$: The label set for the vertex v. When all label sets for vertices in V are combined they form a label cover of the graph $G = (V, E)$. This cover will be denoted by L.
- $D(v, w)$: The distance between vertices v and w stored as a part of the label in $L(v)$

3. PROBLEM STATEMENT AND BACKGROUND

3.1 Problem Statement

Given an undirected, unweighted graph $G = (V, E)$, a path between two vertices $v_x, v_y \in V$ is defined as a list of edges $P(v_x, v_y) = e_0, e_1, \cdots, e_n$ such that $v_x \in e_0$, $v_y \in e_n$ and any consecutive edges $e_m, e_{m+1} \in P$ must share a vertex such that the second vertex of e_m is the first vertex of e_{m+1} for $0 \leq m < n$.

The point-to-point shortest path on an undirected, unweighted graph $G = (V, E)$ given two vertices $v_x, v_y \in V$ is the path from v_x to v_y that contains the least amount of edges. The point-to-point shortest path distance (PPSD) problem is concerned with finding the length (i.e., the number of edges) of this path.

PPSD is not a new problem and research on it is abundant. PPSD can be solved using well-known traversal-based algorithms. Depending on the type of the input graph Bellman-Ford (Bellman, 1958), Dijkstra (Dijkstra, 1959) or Delta-Stepping (Meyer & Sanders, 2003) algorithms could be used. A simple algorithm based on Breadth-First Search (BFS) could suffice if the graph is unweighted. The literature also contains optimizations and parallel implementations of these algorithms (Dhulipala, Blelloch & Shun, 2017; Firoz, Zalewski, Kanewala & Lumsdaine, 2019; Leiserson & Schardl, 2010; Nguyen, Lenharth & Pingali, 2013). Furthermore distributed BFS algorithms can also be found (Makki, 1996; Ueno, Suzumura, Maruyama, Fujisawa & Matsuoka, 2017).

The solutions presented above all require the graph to be traversed in some manner. However, this may not be feasible in a scenario where PPSD queries need to be answered within a short period of time. Such queries are useful in certain applications such as ranking searches in social networks (Vieira et al., 2007; Yahia, Benedikt, Lakshmanan & Stoyanovichy, 2008), finding influential communities in social networks

(Li et al., 2017), fact-checking in knowledge graphs (Shiralkar, Flammini, Menczer & Ciampaglia, 2017) and finding close points on road networks (Abeywickrama & Cheema, 2017).

A trivial solution for answering PPSD queries is storing the answers to all possible queries. However, for a graph with N vertices this method requires $O(N^2)$ memory. The problem we are interested in is reducing this memory requirement by storing just enough information to answer the query in a reasonable amount of time.

3.2 2-Hop Labeling

A popular solution to the problem laid out in Section 3.1 is the 2-hop labeling method (Cohen et al., 2003). As part of this method, each vertex stores a set of labels. Each label is composed of two elements; an integer identifying a vertex in the graph and an integer denoting the PPSD distance to that vertex from the vertex that is storing this label. We will refer to the composition of all label sets $L(v)$ for $v \in V$ for a graph $G = (V, E)$ as the label cover of that graph. This notation is illustrated in Figure 3.1. The process of generating the label cover will be referred to as indexing.

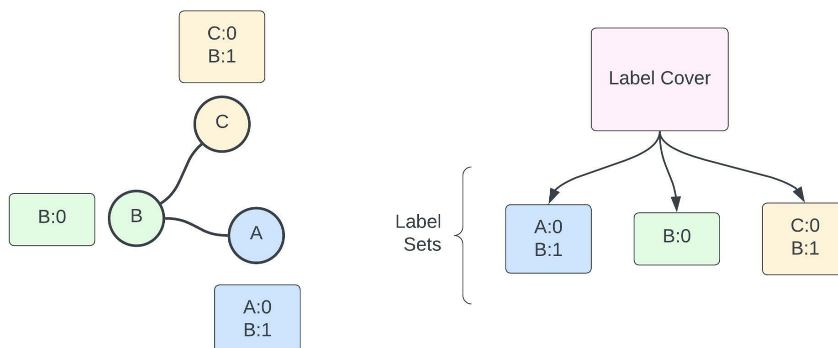


Figure 3.1 A diagram showing the label set and label cover terms with respect to a small example graph

Given an undirected graph $G = (V, E)$, and a label cover, the PPSD distance between two arbitrary vertices $v, u \in V$ can be calculated by finding a vertex $w \in V$ such that $w \in L(v) \cap L(u)$ and summing the distances $D(v, w)$ and $D(u, w)$ which are stored in those label sets. However, for queries to be answered correctly enough of these

labels should be generated. Since space efficiency is a big portion of this problem, we want to avoid generating more than the necessary amount of labels. Unfortunately, finding the minimum number of labels required for successful queries is NP-hard (Cohen et al., 2003). We will discuss some solutions to generating a nearly minimal label cover in the following sections.

3.3 Sequential Approaches

Pruned landmark labeling is an algorithm used for generating 2-hop covers of graphs originally introduced by Akiba et al. (2013). A simplified pseudo-code for the algorithm can be seen in Algorithm 1.

Algorithm 1 PLL (Pruned Landmark Labeling) Algorithm

Input: The input graph $G = (V, E)$

Output: The label cover L

```
procedure PLL(G)
    for u in V do
        PUSH(u)

procedure PUSH(u)
    Q ← a queue
    dist ← an array of size N
    dist(v) ← ∞ for all v in V
    dist(u) ← 0
    add u to Q
    while Q is not empty do
        Dequeue v from Q
        if not PRUNE(u, v, d) then
            add u to L(v)
            D(u,v) ← dist(v)
            for w in ngh(v) do
                enqueue w to Q
                dist(w) ← dist(v) + 1

procedure PRUNE(u, v, d)
    for w in L(v) ∩ L(u) do
        d_v ← D(v,w)
        d_u ← D(u,w)
        if d_v + d_u <= d then return true
    return false
```

We classify PLL as a push-based algorithm where a push operation is defined to be a traversal algorithm that inserts the source vertex to the label sets of the visited vertices. Completing a push operation from each vertex of the graph generates a complete 2-hop label set.

During the push operation, a pruning operation is performed to reduce the number of labels. The label sets of the visited vertex and the source vertex are compared to find a common vertex. If such a vertex is found, the distances in both label sets are added together to get a 2-hop distance between the source and the visited vertex. If this distance is less than the depth of the BFS operation the label is pruned so it

is not added to the visited vertex's label set. To optimize this operation the label set of the source vertex is densified into a single array (i.e., distance cache) which allows the intersection operation in the pruning procedure to be done with a single loop over $L(v)$.

To acquire a minimal label cover, the vertices of the graph should be indexed in a particular order. This is achieved by **ranking** the vertices based on some metric and processing them from the highest rank to the lowest rank. Processing central vertices first is advantageous as they are likely to be in many shortest paths. As a result, vertices are usually ranked using their degrees or betweenness centrality.

PLL and any other push-based algorithms are inherently difficult to parallelize. As explained in the previous paragraph indexing vertices in a certain order is advantageous. If multiple vertices are indexed simultaneously the order between them is lost. Furthermore, the pruning algorithm requires the label cover generated up to that point to be able to prune effectively. Indexing multiple vertices simultaneously creates a dependency issue where these simultaneous processes can not prune labels based on the label sets generated by each other.

Another commonly used sequential solution is the Pruned Highway Labeling (PHL) algorithm (Akiba, Iwata, Kawarabayashi & Kawata, 2014). This algorithm is primarily focused on indexing road networks and does not perform well on social networks, web graphs, and other low-diameter graphs which is the primary focus of this study.

3.4 Parallel Approaches

The PLL algorithm discussed in Section 3.3 is not designed with parallelism in mind. Despite this, several parallelized variants of the PLL algorithm exist in the literature.

Qiu et al. (2018) presented a work called ParaPLL where different vertices of the input graph are distributed among the threads, each thread performing the push operation shown in Algorithm 1. Processing multiple vertices at the same time in the PLL algorithms causes inefficiency in the pruning stage as the labels generated simultaneously in other threads are not available to the pruning algorithm. As a result, this algorithm often produces a larger label set than desired.

Lakhotia et al. (2019) suggested the Label Construction and Cleaning (LCC) and Global Local Labeling (GLL) algorithms which follow a similar structure to ParaPLL but incorporate a cleaning stage to reduce the label set size.

Jin et al. (2020) suggested an approach called VC-PLL which implements the PLL algorithm using a scatter-gather approach. With this approach, they were able to generate label sets with identical sizes to those generated by PLL.

Parallel Shortest-distance Labeling (PSL) is another 2-hop labeling algorithm, originally suggested by Li et al. (2021). Given the same graph and parameters, PSL can create an identical label set to PLL even when running in parallel. A simplified pseudo-code for the algorithm can be seen in Algorithm 2.

Algorithm 2 PSL (Parallel Shortest-distance Labeling) Algorithm

Input: The input graph $G = (V, E)$
Output: The label cover L

```
procedure PSL
    d ← 2
    while new labels were added in the last iteration do
        for u in V do
            Pull(u, d)
        d ← d + 1

procedure Pull(u, d)
    for v in ngh(u) do
        for w in L(v) where D(v,w) = d − 1 do
            if not Prune(u, w, d) then
                add w to L(v)
                D(v,w) ← d
```

We classify PSL as a pull-based algorithm with a pull operation defined as a vertex taking labels for a particular distance from its neighbor vertices. Labels of distance 0 (vertices themselves) or 1 (immediate neighbor vertices) are initialized at the beginning of the algorithm. Then a pull operation is performed from every vertex (this will be referred to as a level from here on), starting with a distance of 2 and increasing after every vertex is processed once. This operation continues until no improvement can be made to the label sets. Since the distances keep increasing, the number of iterations is limited by the diameter of the input graph. This limits the usage of the algorithm on large-diameter graphs such as road networks.

During the pull operation, every vertex checks the label sets of each of its neighbors and considers adding each label to its own label set (i.e., pull the label) through the same pruning operation as in PLL. The distance cache optimization explained in Section 3.3 is also applied here. When a label is pulled, its distance is increased by 1 since the path length represented by that label is now 1 longer. Due to the nature of the algorithm, each distance is processed in order (i.e., each distance is processed at a different level). As a result, when the distance d labels are being added we are only concerned with $d-1$ distance labels of the neighbor vertices as they are the only ones that can result in a distance of d when pulled. Furthermore, when the distance d labels are being added, we only use labels with less than d distance for pruning since the pruning operation is only successful if it finds a shorter distance than d. To summarize, when a d distance label is pulled, other d distance labels can not affect the result. This allows each level of the algorithm to be internally independent. So during a level, each vertex can be processed simultaneously without any dependency problems such as the ones we saw with PLL.

3.5 Distributed Approaches

Several of the works based on PLL mentioned in Section 3.4 also present distributed approaches. We will discuss some of these in this section. To our knowledge, there is no distributed implementation of the PSL algorithm in the literature.

Qiu et al. (2018) presented a distributed variant of their ParaPLL work (DParaPLL) where the vertices are distributed among the nodes and are processed in parallel. For efficient pruning, the labels should be synchronized. DParaPLL does this synchronization step at certain intervals to reduce communication bottlenecks. As a result, the frequency of this synchronization has a direct effect on the number of labels generated.

Lakhotia et al. (2019) presented multiple distributed algorithms. DGLL which is a distributed variant of their Global Local Labeling (GLL) algorithm utilizes a synchronization and cleaning step to remove unnecessarily generated labels. As is the case with DParaPLL, these unnecessary labels are often generated due to the inefficiency of the pruning operation when the full label cover is not available. They

also provided a hybrid algorithm with DGLL and PLaNT (which we referred to as DPLANT) where the indexing starts using the PLaNT algorithm and eventually switches to DGLL after a trigger.

Jin et al. (2020) presented a distributed variant of their VC-PLL work (which we referred to as DVCPLL). In this variant, the vertices are modeled using a master-mirror notion. The label set of each vertex is only stored on a single node, in this particular node, the vertex is labeled as a master. Every other node contains a mirror for this vertex and relies on information sent from the master for pruning operations.

3.6 Compression Methods

Several compression methods can be used to decrease the number of labels generated by the PSL and PLL algorithms. For instance, Li et al. (2021) suggested 2 compression methods for their PSL algorithm.

The first method involves **merging identical vertices** in the input graph. Given two vertices u and v they are considered identical if one of the following is true:

- $ngh(u)$ is identical to $ngh(v)$
- $ngh(u)/v$ is identical to $ngh(v)/u$

Such vertices can be merged into a single vertex and queries made with either vertex can simply be redirected to this single vertex. This method is applicable to both PSL and PLL algorithms.

The second method is **local minimum elimination** which involves eliminating vertices whose rank is the minimum among the ranks of their neighborhood. More formally, given a vertex v, v is said to be the local minimum if $rank(v) < rank(u)$ for every $u \in ngh(v)$. Such vertices are not removed from the graph entirely but simply ignored during the indexing operation of PSL which results in their label sets remaining empty. During querying, the label set of these vertices can be reconstructed by visiting the neighboring vertices.

Anirban, Wang, Islam, Kayesh, Li & Huang (2022) suggested some well-performing compression methods as well. However, their Compressed Distance Labeling (CDL)

technique is only applicable to directed graphs and their Linear Set Compression (LC) technique is more suitable for road networks neither of which is the focus of this study.

3.7 Bit-Parallel Labels

Bit-Parallel (BP) labels are an optimization for the PLL algorithm introduced by Akiba et al. (2013). The general idea is to store some commonly used label information in a dense manner to allow for additional and faster pruning. This optimization also works on the PSL algorithm and is used by Li et al. (2021). This section will feature an explanation of this optimization in detail as it is an important parameter of the algorithm and we performed several experiments to find its optimal value. These experiments are discussed in Section 5.2.2.

BP labels are generated before the actual indexing operation of the algorithms begins. They are generated directly from the raw unprocessed graph, meaning that other parameters of the algorithm such as compression, and the number of threads do not affect BP labels. However, they are affected by the ranking method used.

BP label generation starts by choosing the highest-ranked vertex r in the graph as the root vertex. A fixed number of neighbors of the root vertex are also chosen as sub-roots which we will denote as the set Sr. This choice is again made by favoring the highest-ranked vertices. Then for each vertex v in the graph, we compute the following information:

1. Shortest-path distance between r and v.
2. For each vertex s in Sr, the shortest-path distance between v and s subtracted from the shortest-path distance between v and r.

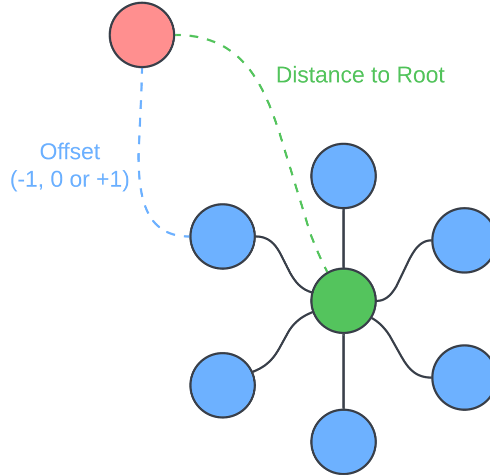


Figure 3.2 A figure representing the various terms used for BP labels. The green vertex is the root, the blue vertices are sub-roots, and the red vertex is any vertex in the graph.

Computing the first value can simply be done by the use of a BFS algorithm. This information is stored as an integer value. If the graph in question is a low-diameter graph such as a social network or a knowledge graph, we can store this value using a lower bit-rate to save space since the expected distances will be small.

Computing the second value (which will be referred to as the offset) can again be done by the use of a BFS algorithm with the results being subtracted from the first value. For each vertex v and sub-root s, the offset can be stored using only 2-bits. This is because of the fact that this subtraction can only result in 3 values which are 0, -1, and $+1$. This is a direct result of the sub-roots being neighbors of the root meaning any shortest-path distance originating from these vertices can only differ by 1 at most. By setting the sub-root count as 64, the offset of all sub-roots can be stored as two 64-bit integers. This can be done by the following process:

1. Initialize both integers to 0.
2. Set the bits of the first integer to 1 when the sub-root at that index has an offset of 0.
3. Set the bits of the second integer to 1 when the sub-root at that index has an offset of -1.

Once generated, the BP labels can be used during the pruning stages of the algorithm. For example, when considering the addition of the label w to the label set of vertex v, we can get the distance to both vertices from the root vertex using BP

labels. Summing these two distances gives an upper bound to the actual shortest-path distance between w and v. Furthermore, by performing bitwise operations on the two integers representing the sub-roots, we can find a tighter upper bound.

This method can be repeated for multiple root vertices. The memory consumption of BP labels is directly proportional to the product of the number of roots and the number of vertices of the graph. Since BP labels are used in the pruning stages of PSL and PLL algorithms, they directly have an effect on which labels get added. As a result, they are required during the query operation and become an integral part of the final labels.

4. METHODOLOGY

4.1 Leaf Elimination

We define a leaf as a vertex with degree one that is connected to a vertex with a higher degree. More formally, given an undirected graph $G = (V, E)$ if for $v \in V$, there exists only one $u \in V$ such that $(v, u) \in E$ then v is a leaf if and only if $deg(u) > 1$.

The label set of a leaf can be generated using the label set of its parent due to Lemma 4.1.1. This situation can also be handled during the query with minimal performance impacts by performing the query operation using the parent vertex and then incrementing the results by 1. Furthermore, since the only vertex connected to a leaf is its parent, it can not affect the rest of the graph. As a result, eliminating the leaf vertices from the graph is possible without affecting the final query results.

Lemma 4.1.1. *A leaf's label set can only contain itself and the labels of its parent.*

Proof. Due to the nature of the PSL algorithm, labels can only propagate through neighboring vertices. As a result, for a vertex v to add a certain label w to its label set, w should exist in another vertex u in the neighborhood of v. If v is a leaf, then there is only a single vertex in its neighborhood and as a result, u must be the parent vertex. □

The local minimum elimination discussed in Section 3.6 is often sufficient to eliminate the leaves. If we assume a degree-based ranking strategy, the rank of the parent vertex will be higher than the leaf by the very definition of the leaf. This means that the leaf vertex is a local minimum and will be eliminated when local minimum elimination is used. With another ranking strategy, this is not guaranteed, however, in most centrality-based metrics the leaves are extremely likely to be

a local minimum as well. In fact, for closeness centrality and betweenness centrality rankings, the parent vertex will always be ranked higher than a neighbor leaf vertex. However, we found that applying leaf elimination before local minimum elimination is actually beneficial. The reason for this is that when leaves are present they prevent their parents from being selected as local minimum vertices. Since each parent is connected to a low-rank leaf vertex, they can not have the lowest rank in their neighborhood. Applying leaf elimination first allows parent vertices to be eliminated as local minimum vertices. This process is illustrated in Figure 4.1.

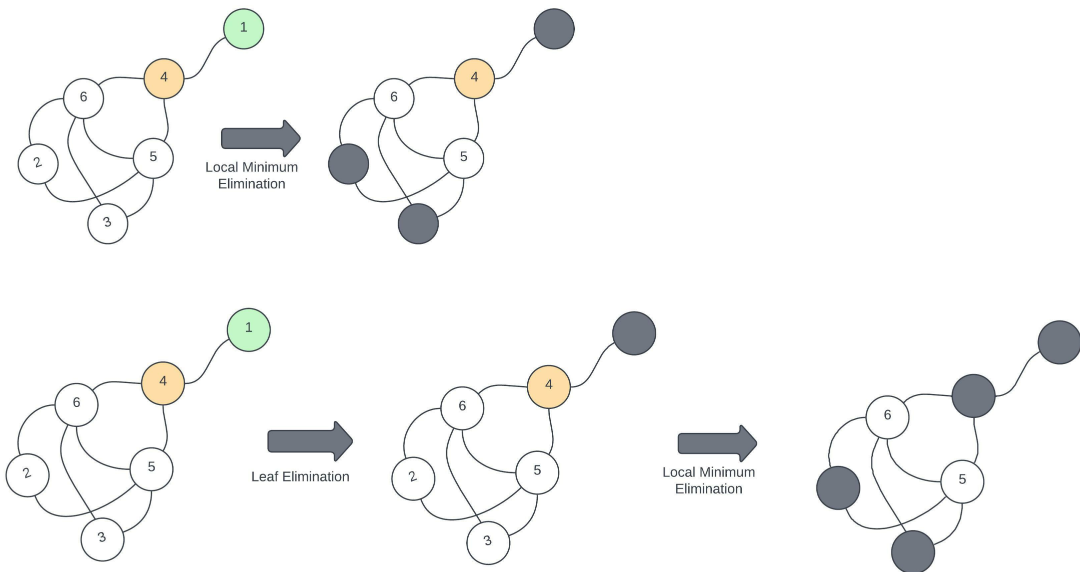


Figure 4.1 An example showing the effect of leaf elimination. The Green vertex is the leaf, the yellow vertex is the parent of the leaf, gray vertices are eliminated meaning they will not be processed during indexing. The numbers on the vertices represent their ranks. We can see that leaf elimination causes an additional vertex to be eliminated.

4.2 Partitioning

Given a graph $G = (V, E)$, the act of partitioning the graph is defined as creating a number of disjoint sets (called partitions) whose union should result in the set V. The balanced graph partitioning problem can then be defined as creating these

disjoint sets such that the sum of the vertex weights in any of the sets is within a certain range of the average denoted by ϵ (Lasalle, Mostofa, Patwary, Satish, Sundaram, Karypis & Dubey, 2015).

Our goal is to use one of the many existing partitioning algorithms to partition the graph into multiple partitions. These partitions can then be distributed among a set of nodes each running the PSL algorithm independently. However, there would need to be some communication in between to ensure the correctness of the results. Our expectation from this approach was the following:

- Ability to scale better to a high number of cores by having those cores reside on different machines with independent memory structures which we cover in this thesis.
- Ability to process larger graphs due to the better scalability of memory when multiple machines are involved.
- Ability to use multiple accelerators like GPUs (Graphical Processing Units) by providing each accelerator with a partition to process which is a future work.

As discussed previously, for the correctness of the results there was a necessity for communication between the nodes. If this was not implemented, it would not be possible to accurately answer queries that contain two vertices on different partitions. This is because answering a query when the two vertices u and v are reachable from each other involves finding a common vertex w in the label sets of u and v. However, since partitions are by definition disjoint, if u and v reside in different partitions then the common vertex w can be in the same partition with at most one of u and v. As a result, without some form of synchronization between the nodes, it would not be possible to answer all potential queries.

4.3 Vertex Separation

In order to deal with the synchronization requirement mentioned in Section 4.2, we used a vertex separation-based approach. A vertex separator is defined as a set of vertices which when removed from the graph, causes the graph to become multiple disconnected graphs (Lasalle & Karypis, 2013a). Some graph partitioners (Lasalle & Karypis, 2013b)

For our algorithm instead of removing the vertex separator from the graph, we duplicated it on each partition. As a result, each node had a copy of the vertices on the vertex separator. We used the vertex separator as a synchronization zone where at the end of each level of the PSL algorithm the labels generated by each node can be unified. This is possible due to Lemma 4.3.2, as any label that is propagated between two different partitions would have to go through the vertex separator.

Lemma 4.3.1. *Given a graph with a vertex separator-based partitioning, no vertex outside of the vertex separator can be connected to a vertex from a different partition in the original graph.*

Proof. Consider a graph $G = (V, E)$, let the vertex separator be denoted as W and W splits the graph into n partitions denoted as $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ ... $G_n = (V_n, E_n)$. Contrary to the lemma, assume there exists a vertex $v_i \in V_i$ and $v_i \notin W$ such that $(v_i, v_j) \in E$ for some $v_j \in V_j$ and $v_j \notin W$. However, if this is the case, removing W from the original graph G would leave the edge (v_i, v_j) since neither vertex is in W. This suggests that the partitions are not disconnected which is against the definition of the vertex separator. As a result, our assumption must have been wrong and no such v_i should exist. □

Lemma 4.3.2. *If the PSL algorithm is applied to a graph with a vertex separator-based partitioning, a vertex can contain the label for a vertex from a different partition if and only if that label also exists in a vertex in the vertex separator.*

Proof. Due to the nature of the PSL algorithm, labels are propagated only to neighboring vertices. As a result of this, a vertex can contain the label for another vertex if and only if there is a path between those two vertices and the label should be included on all vertices along this path. Since by Lemma 4.3.1, only the vertices on the vertex separator can be connected to vertices from different partitions, paths between two vertices from different partitions should always pass through a vertex on the vertex separator. □

The pruning stage of the PSL algorithm causes some trouble with this approach. Pruning occurs in a similar fashion to querying, meaning the label sets of both the currently processed vertex and the label vertex being considered are required for cross-comparison. However, if these two vertices reside in different partitions then only one of them will be available to the node. As a result, allowing propagation of all labels over the vertex separator is not feasible as it would require an unreasonable amount of data to be replicated.

4.4 Ranking the Vertex Separator

As explained in Section 3.3, the PSL algorithm features a rank-based pruning operation to reduce the number of labels generated. We made use of this feature to filter the labels that are propagated over the vertex separator. Essentially, our goal was to limit the labels on the vertex separator to only consist of vertices on the vertex separator. Since the vertices of the vertex separator and their labels are already duplicated on all partitions, this would solve the pruning problem discussed in Section 4.3. This can be achieved by setting the rank of each vertex on the vertex separator to a high value such that no vertex outside of the vertex separator has a higher rank. Due to the rank-based pruning stage of the PSL algorithm discussed in Section 3.3, a vertex can only contain labels for vertices that have a higher rank than its own rank. When the ranks of the vertex separator vertices are increased in this way, we ensure that they can only contain each other as labels. When this is coupled with Lemma 4.3.2, we find that no vertex outside of the vertex separator can be a label in a different partition than its own.

There is a caveat to this approach. Modifying the ranks of the vertices could have performance impacts as it directly affects the number of labels generated and the effectiveness of the pruning operation. This will be discussed further in Section 5.2.4.

4.5 Synchronization

Synchronization of the vertex separator needs to occur after every level (every iteration of the outermost loop shown in Algorithm 2) for a correct set of labels to be generated. Since the vertex separator is replicated across all nodes, each node contains a set of labels for each vertex on the vertex separator. The synchronization step involves computing the union of all sets for each of these vertices. In the end, every node should contain identical label sets for every vertex on the vertex separator. Furthermore, we want to avoid any duplication of labels as this can cause some unnecessary computation to occur at later levels.

To get the union of the label sets we use Algorithm 5. Our initial approach involved

using a single node to perform this operation handling each vertex in a separate thread. Since this algorithm has significantly lower complexity than PSL, this step would not be the bottleneck of the algorithm. However, we found the communication load to cause significant slowdowns. Handling everything in a single node involves all other nodes sending their data to this one node and overloading the network capacity of that node.

Algorithm 3 Label Set Union

Input: The input graph $G = (V, E)$ and the label cover of the vertex separator $L = [L_0, L_1, \cdots, L_k]$

Output: The union label cover Γ

```
seen ← a boolean array of size N
seen(v) ← false for all v ∈ V
for v in V do
    for L_i in L do
        for w in L_i(v) do
            if not seen(w) then
                seen(w) ← true
                add w to Γ(v)
```

As a result, we decided to distribute to load among the nodes. Each node was assigned a set of vertices on the vertex separator to apply the union operation. To reduce the communication load, we developed a round-robin tournament-based approach (the algorithm is included in the appendix). Specifically, we used a temporarily dense single round-robin (DSRR) approach where the nodes are paired with each other in multiple rounds such that each node pairing is encountered once and each node is paired only once for every round. Limiting ourselves to an even node count of k, we end up with $k(k-1)$ pairings and $k-1$ rounds (Henz, Müller & Thiel, 2004). At each pairing, the two nodes exchange the information they need from each other to perform the union. Once every union is created, they are broadcasted to all of the nodes.

4.6 Summary and Hypothesis

Various steps of our distributed PSL method are illustrated in Figure 4.2. Our hypothesis was that our distributed method would generate more labels overall and require more memory as a result. However, this memory requirement would be distributed among multiple machines. If queries are to be performed on a single machine, the labels can be combined removing any copies and this operation would result in a set of labels closer in size to those generated by PSL. Furthermore, as a result of having more computational power available, the indexing operation would take less time.

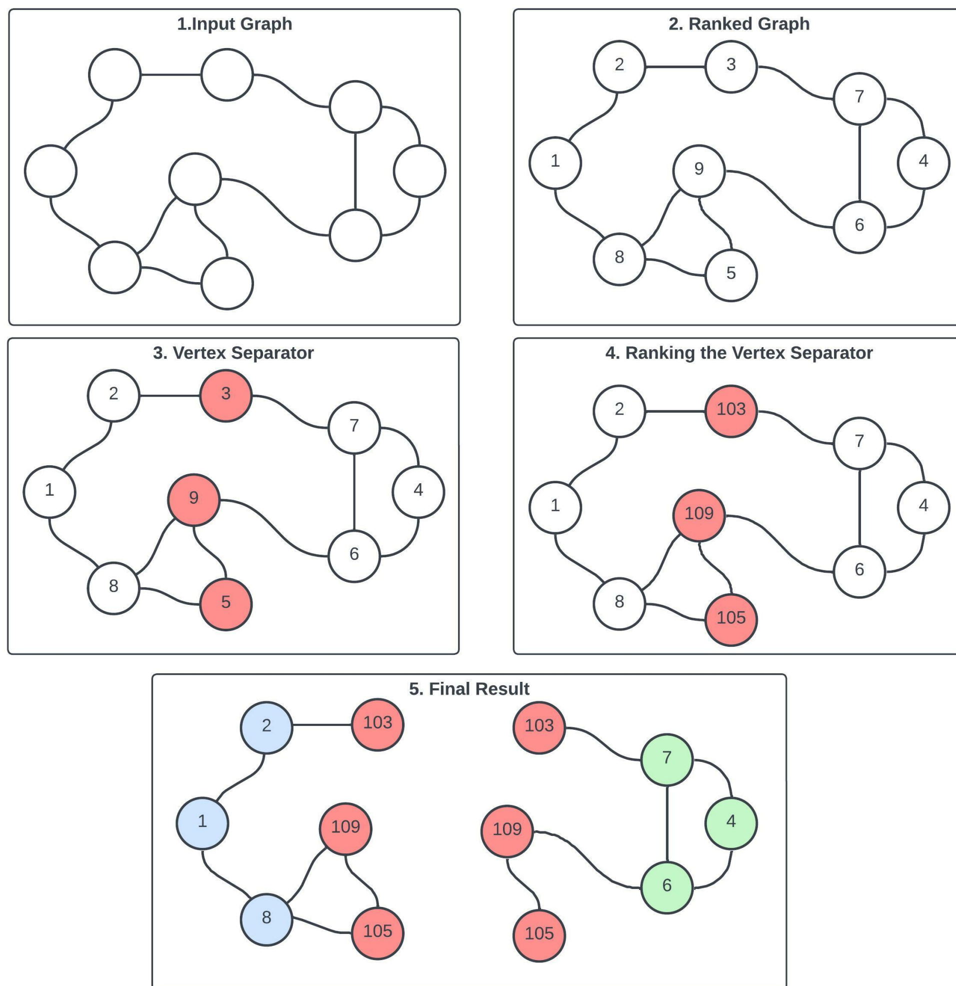


Figure 4.2 A diagram showing the various steps of our distributed PSL algorithm. The numbers on the vertices indicate their ranks. The red vertices are processed by all nodes and are synchronized. The blue and green vertices are each processed by a different node.

4.7 Technical Details

Both our implementation of PSL on shared memory and our partitioning-based distributed method was implemented in C++17 using OpenMP and OpenMPI. We used the `constexpr` feature of C++ to ensure only the necessary sections of the codes are compiled for the parameters selected. For example, when the leaf elimination feature discussed in Section 4.1 is turned off, all code sections related to it are not compiled and as a result, it can not affect the execution time in any way. We used the PIGO library for faster input of graph files (Gabert & Çatalyürek, 2021). For OpenMP, we used dynamic thread scheduling with a chunk size of 256 (some comparisons for this parameter are included in the Appendix).

To achieve better cache utilization, all of our implementations represent the input graph using a Compressed Sparse-Row (CSR) format. CSR stores the adjacency lists of the vertices in a single continuous array rather that the multiple disjoint arrays used in the original PSL implementation. The difference is illustrated in Figure 4.3.

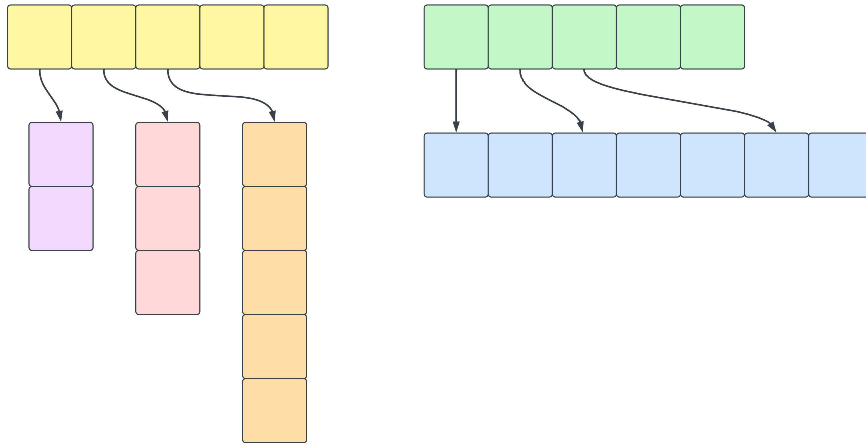


Figure 4.3 Comparison of adjacency list (left) and CSR (right) sparse data storage formats.

To represent the label sets, we used a data structure similar to a CSR. For each vertex, we store two arrays. The first one contains the vertices of the labels for this vertex. The second one stores the start and end indices of the distances. Since labels are generated in order of their distances, there is no need to sort the arrays at any point. The exact data structure is included in the appendix. This is quite different than the method used in the original PSL implementation which stores the distance and the vertices bitwise in a single integer. However, this method creates a

limit on maximum distance and graph size. Specifically, a large graph with a high number of vertices can not be indexed precisely since the number of bits left for the distance information is reduced. Our method avoids this issue and according to our tests does not cause any performance issues.

To obtain the vertex separator explained in Section 4.3, we used off-the-shelf partitioning tools. Some of these tools such as `mtmetis` (Lasalle & Karypis, 2013a) provide built-in procedures for finding vertex separators. However, we found constructing the vertex separator from a standard partitioning (edge cut) to be a better approach. The method we used for this task is shown in Algorithm 4. For any edge, where the two endpoints are on different partitions we choose the one with the higher rank to add to the vertex separator. We believe this rank-based choice gives a better result due to the modification we make to the ranks of the graph explained in Section 4.4.

Algorithm 4 Algorithm to obtain the vertex separator from a standard partitioning

Input: The input graph G=(V,E) and the partitioning array P
Output: The vertex separator S

```
for each edge (u,v) in E do
    if P(u) is not P(v) then
        if u not in S and v not in S then
            if rank(u) > rank(v) then
                Insert u into S
            else
                Insert v into S
```

We experimented with different partitioners and different parameters. The best and most consistent results we obtained came from `Mt-KaHyPar` with its default configuration (Gottesbüren, Heuer, Sanders & Schlag, 2021). `Mt-KaHyPar` consistently provided small vertex separators and better load balancing. There is also a quality focussed configuration of `Mt-KaHyPar` (Gottesbüren, Heuer, Sanders & Schlag, 2022). However the time it took to partition the graph with this configuration was too high for our purposes and could often be higher than the indexing time. As a result, we used the default configuration of `Mt-KaHyPar` for our results.

We found that balancing the load between the partitions was problematic especially when the compression methods discussed in Sections 3.6 and 4.1 are used. We also found that using the logarithm of the degrees of the vertices as weights when partitioning provided a better load balancing for most graphs we tested when compared to uniform and degree weights. Our proposed reason for this is that label counts and

processing times of vertices are not strongly correlated with their degrees or rank. However, our research did indicate a weak correlation between degrees and label counts. Using the logarithm of the degree allowed us to use degrees as weights with a lower influence. To fix the problem with compression, we decreased the weights of compressed and eliminated vertices by a flat amount when partitioning. While this approach did not eliminate the load balancing issue, it did reduce it.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

We evaluated our work using multiple systems, the specifications of which can be found in Table 5.1. The machine used for each experiment is indicated in the caption of the corresponding figures and tables. For comparisons, we used the Karolina system as it was the most powerful and up-to-date system available to us. Nebula and Gandalf were mostly used for experiments where the execution time was not the primary focus. All source codes by previous works and us were compiled with `-O3` optimization flag using the compilers `g++` and `mpic++` with the versions specified on Table 5.1. All distributed codes running on the Karolina cluster are set to use InfiniBand HDR (200 Gb/s) for communication.

Name	**Number of Nodes**	**CPU**	**GCC Version**	**OpenMPI Version**	**Operating System**
Karolina	72	2 x (AMD 7763, 64 cores, 2,45 GHz)	10.2.0	4.1.1	CentOS 7 (Core), Linux 3.10.0
Nebula	1	2 x (Intel Xeon E5-2620, 8 cores, 2.10 GHz)	9.3.0	4.0.3	Ubuntu 20.04.2, Linux 5.4.0
Gandalf	1	Intel Xeon E7-4870, 60 Cores, 2.30 GHz	11.3.0	4.1.2	Ubuntu 22.04.1, Linux 5.15.0

Table 5.1 Specifications of the systems used for the experiments

We used multiple different graphs for our experiments, these are listed in Table 5.2. Our algorithms are designed for undirected and unweighted graphs as a result none of these graphs feature weights and they are all interpreted as undirected graphs.

Group	Abbreviation	Name	N	M	Source
Easy	DELI	soc-delicous	536,109	2,731,922	Rossi & Ahmed (2015)
	LAST	soc-lastfm	1,191,806	9,038,660	Rossi & Ahmed (2015)
	DIGG	soc-digg	770,800	11,814,264	Rossi & Ahmed (2015)
Moderately Easy	FLIX	soc-flixster	2,523,387	15,837,602	Rossi & Ahmed (2015)
	CITE	coPapersCiteseer	434,103	32,073,440	Davis & Hu (2011)
	DBLP	coPapersDBLP	540,487	30,491,458	Davis & Hu (2011)
	TOPC	wiki-topcats	1,791,490	57,016,282	Davis & Hu (2011)
Moderately Hard	FBA	socfb-A-anon	3,097,166	47,334,788	Rossi & Ahmed (2015)
	FBB	socfb-B-anon	2,937,613	41,919,708	Rossi & Ahmed (2015)
Hard	POKE	Pokec	1,632,804	61,245,128	Davis & Hu (2011)
	LIVE	LiveJournal	5,363,261	155,983,028	Davis & Hu (2011)

Table 5.2 Graphs used for the experiments

The implementations tested in this chapter are abbreviated for ease of reading. PSL refers to the original PSL implementation by Li et al. (2021). NPSL refers to our shared-memory implementation of the PSL algorithm using the improvements discussed in Chapter 4. DPSL refers to our partitioning-based distributed implementation of the PSL algorithm.

Our experiments mainly focus on 2 metrics: the indexing time and the final memory consumption of the generated labels. The indexing time is measured without taking the preprocessing steps into account including Bit-Parallel label generation. The final memory consumption of the generated labels also excludes the memory consumption of the Bit-Parallel labels. This is done since the generation time and size of Bit-Parallel labels are directly related to the input variables N and B. They are not influenced by any of the changes introduced by us. A more in-depth investigation of Bit-Parallel labels is included in Section 5.2.2. Other preprocessing steps such as partitioning, ranking, and compression are also excluded from the indexing times but are measured separately and will be referred to when necessary. All of the reported timings are calculated by taking the average of a minimum of 5 runs. In cases where the variation between the runs was too high, the number of runs was increased.

NPSL and DPSL support various parameters. These are described in Table 5.3. Unless specified otherwise the default values listed in Table 5.3 are used for all experiments. Table 5.4 displays possible values for the compression level (CL) parameter. This parameter directly corresponds to the techniques introduced by Li et al. (2021). Specifically, PSL corresponds to a compression level of 0, PSL+ corresponds to a compression level of 1, and PSL* corresponds to a compression level of 2. Compression level 3 adds the leaf elimination technique we suggested in Section 4.1 to compression level 2.

Abbreviation	**Name**	**Description**	**Default Value**
NT	OpenMP Thread Count	Number of threads to be used in OpenMP sections.	32
SCHE	OpenMP Schedule	Scheduling strategy used in OpenMP sections.	dynamic, 256
BPR	BP Root Count	Number of roots used during bit-parallel label construction.	15
RNKM	Ranking Method	Method (or metric) used to order and rank the vertices.	degree
CL	Compression Level	Determines the compression and elimination methods to be used. Please see Table 5.4.	3
LBO	Load Balance Offset	Reduces the weight of the eliminated/compressed vertices by this amount.	100
MMSG	Maximum Message Size	Determines the maximum amount of data sent during synchronization. Larger data will be sent in chunks.	1.07E+09

Table 5.3 Parameters for DPSL and NPSL

Level	Identical Vertex Compression	Local Minimum Elimination	Leaf Elimination
0	No	No	No
1	Yes	No	No
2	Yes	Yes	No
3	Yes	Yes	Yes

Table 5.4 Compression levels for DPSL and NPSL

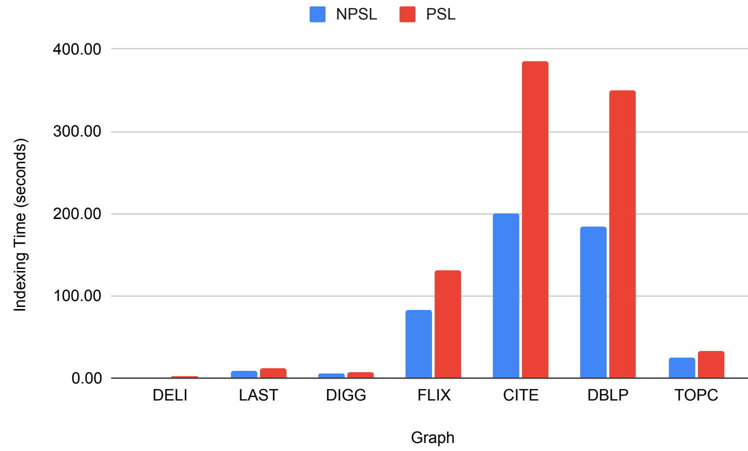
5.2 Shared Memory Experiments

5.2.1 Performance Evaluation of NPSL

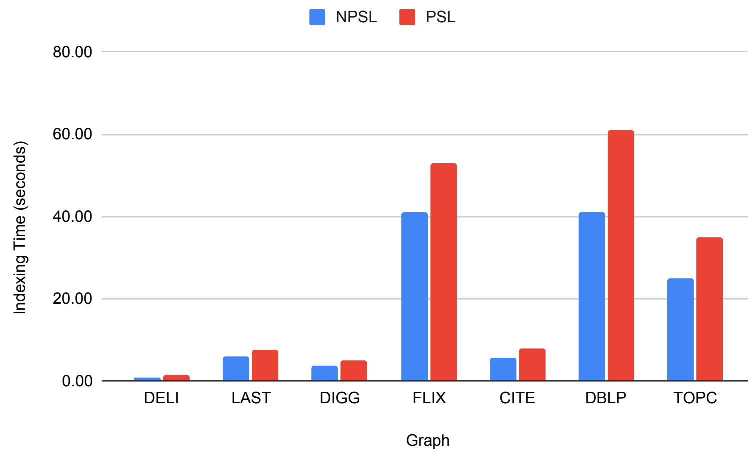
Figure 5.1 and Table 5.5 show the performance of NPSL with respect to the original PSL implementation by Li et al. (2021). It can be seen that on all the graphs tested NPSL was faster. Since the differences between the implementations are numerous, we can not point out the concrete source of the speedup. However, we are aware that the data structures used for the input graph and label sets as discussed in Section 4.7 contributed positively to the performance.

	PSL Indexing Time	**NPSL Indexing Time**	**NPSL Speedup**
FBB	742.69	630.72	1.18
FBA	1063.29	872.90	1.22
POKE	2289.61	2069.76	1.11
LIVE	3312.06	3203.23	1.06

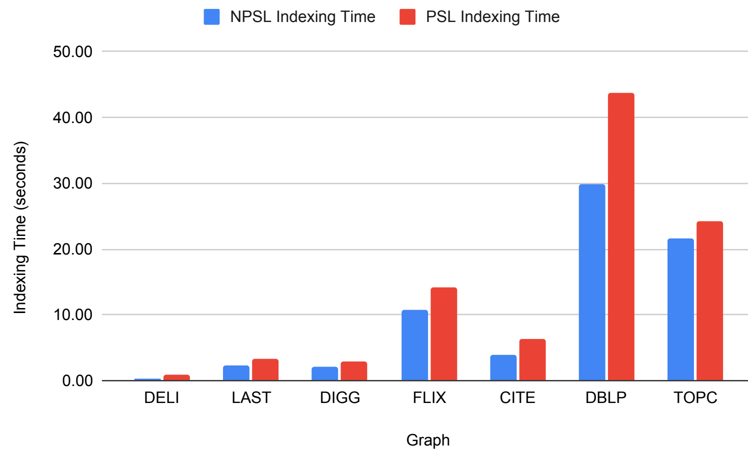
Table 5.5 [Karolina] Comparison of indexing times between PSL and NPSL with CL=2 on hard and moderately hard graphs. The first two columns show the indexing times in seconds for PSL and NPSL respectively. The last column shows the speedup of NPSL with respect to PSL.



(a) Using CL=0



(b) Using CL=1



(c) Using CL=2

Figure 5.1 [Karolina] Comparison of indexing times for NPSL and PSL using different compression levels on easy and moderately easy graphs. CL=3 is not implemented in PSL and is omitted as a result.

5.2.2 Bit-Parallel Label Experiments

The experiments in this section show the effect of varying the number of roots used during BP label generation (see Section 3.7). We chose to use 0, 15, 30, 45, and 60 as the number of roots during these experiments. These translate to 0, 255, 510, 765, and 1020 bytes per vertex respectively. With padding introduced by the C++ compiler, these values become 0, 256, 512, 768, and 1024 bytes per vertex respectively.

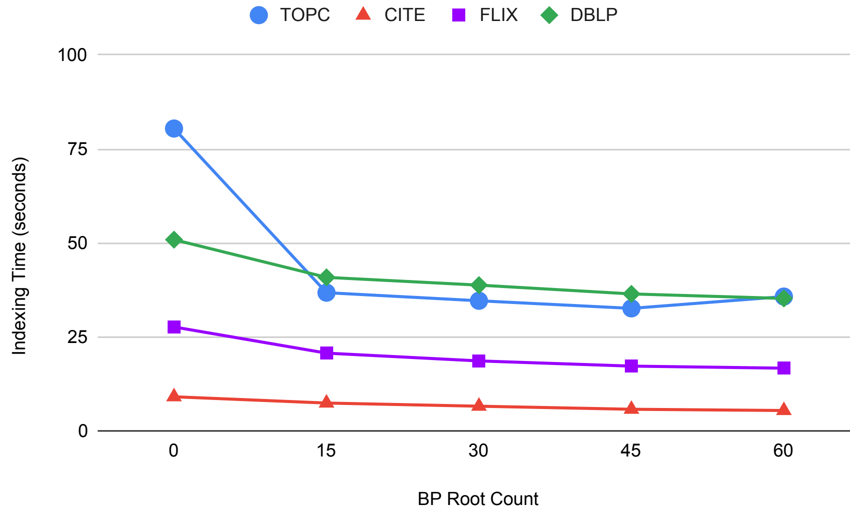
Figure 5.2 demonstrates the change in indexing time with the number of roots used. On TOPC, using 15 roots produced a speedup of 220% when compared to not using BP labels. On FBA, we also saw a significant speedup of 154%. While not as extreme, all the graphs we tested exhibited some speedup when BP labels are introduced. As a result, we can state that BP labels are a crucial component of the performance of NPSL.

However, increasing the number of roots beyond 15 did not improve the indexing times significantly. On TOPC, using 30 roots produced a speedup of only 106%, and on FBA, we actually saw a slight increase in indexing times. We have two possible explanations for this situation:

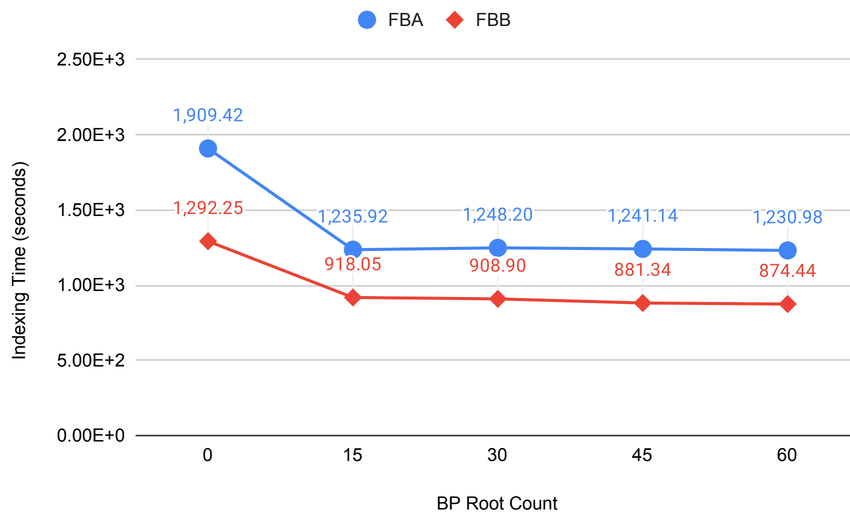
1. While for each root vertex the sub-roots are processed in parallel using bitwise operations, each root vertex is processed sequentially. As a result, increasing the number of roots also increases the pruning time. Up to a certain number of roots, this is balanced by the reduction of the labels generated and other pruning operations that need to be performed.

2. Increasing the root count increases the amount of memory required to store the BP labels. During the pruning stage, these labels are accessed in a pseudo-random manner which puts more load on the last-level cache (LLC) of modern CPUs by Intel and AMD.

Figure 5.3 demonstrates the breakdown of the memory usage of the generated labels. As stated in Section 3.7, BP labels are an integral part of the final labels and are required for querying. We saw that increasing the number of roots to 30 and beyond often resulted in a larger memory footprint.

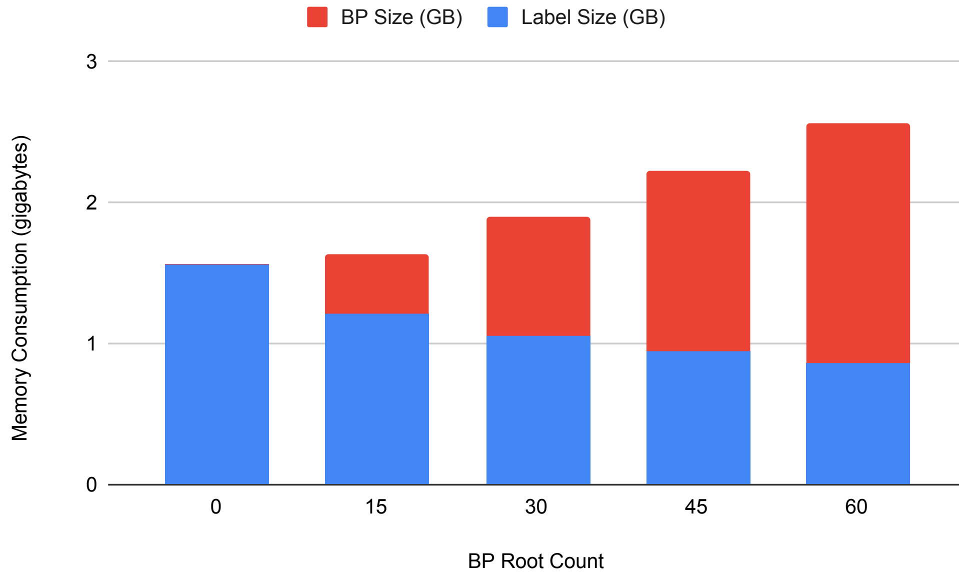


(a) Moderately Easy Graphs

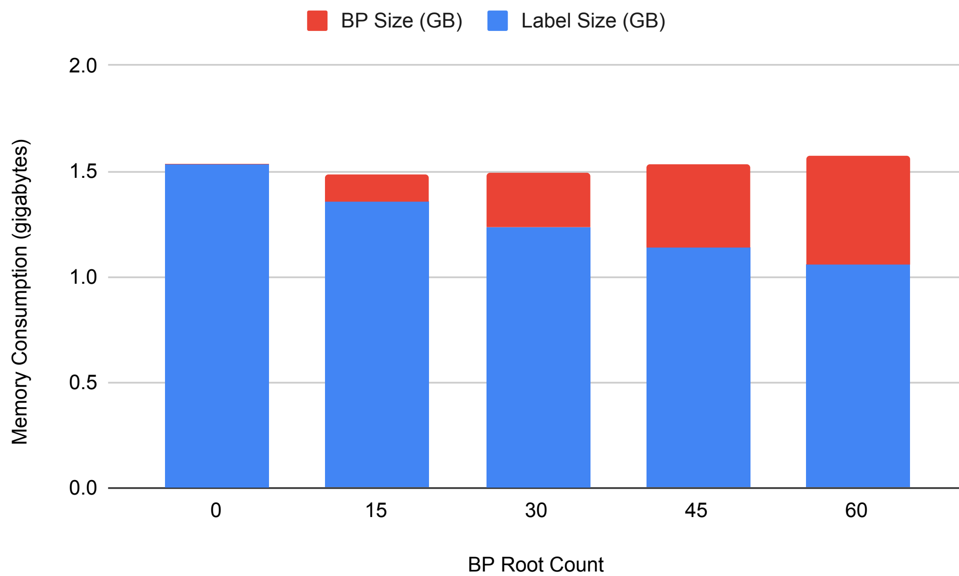


(b) Moderately Hard Graphs

Figure 5.2 [Karolina] Effect of Bit-Parallel Label's root count (BPR) on the indexing time of NPSL on moderately easy and moderately hard graphs.



(a) TOPC



(b) DBLP

Figure 5.3 [Karolina] Effect of Bit-Parallel Label's root count (BPR) on the memory consumption of NPSL when processing the graphs TOPC and DBLP.

To sum up this section, as a result of the larger memory footprint and the decrease in performance we suggest using a minimal number of roots for BP labels with implementations of the PSL algorithm. PLL-based implementations may have different results as the order of operations are quite different between the two algorithms.

5.2.3 Compression Experiments

In this section, we evaluated the performance of our leaf elimination optimization described in Section 4.1. The compression methods introduced by Li et al. (2021) are also used in our implementation of NPSL. However since these are explored in detail in the original paper, we decided not the evaluate them further.

Figure 5.4 and Table 5.6 demonstrate the effect of leaf elimination on the indexing time of NPSL. While the effect is not greatly pronounced, it is consistent across all the graphs we tested and indicates a speedup of up to 106%.

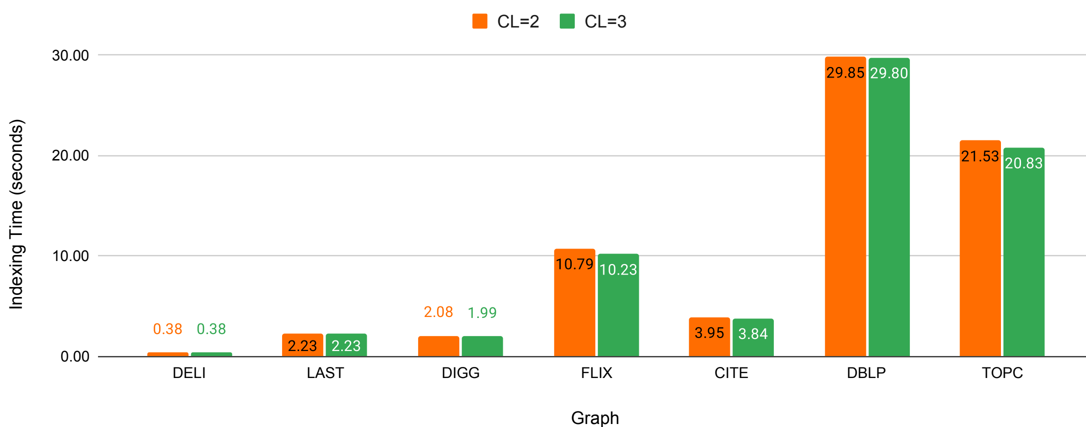


Figure 5.4 [Karolina] Indexing time (in seconds) comparison of NPSL with compression levels 2 and 3 on easy and moderately easy graphs.

	CL=2	CL=3	Speedup
FBA	872.90	822.03	1.06
FBB	630.72	625.41	1.01
POKE	2,069.76	1,985.23	1.04
LIVE	3203.23	3144.05	1.02

Table 5.6 [Karolina] Indexing time (in seconds) comparison of NPSL with compression levels 2 and 3 on hard and moderately hard graphs with the speedup obtained when going from CL=2 to CL=3.

Table 5.7 demonstrates the effect of leaf elimination on the memory consumption of the labels on NPSL. We found that leaf elimination had a consistent reduction on the final label size across all the graphs we tested.

	CL=2	CL=3	Reduction
DELI	0.04	0.04	9.59%
LAST	0.18	0.18	1.95%
DIGG	0.21	0.20	7.27%
FLIX	0.66	0.62	6.66%
CITE	0.41	0.40	3.12%
DBLP	1.37	1.34	1.78%
TOPC	1.22	1.21	0.08%
FBA	13.97	13.89	0.55%
FBB	10.46	10.38	0.76%
POKE	26.77	26.38	1.46%
LIVE	44.16	41.53	5.95%

Table 5.7 [Karolina] Comparison of memory consumption of labels between compression levels 2 and 3 of NPSL

5.2.4 Ranking Experiments

Figure 5.5 shows an experiment we performed to demonstrate the importance of ranking. The vertices are initially ranked based on their degrees. Then a varying percentage of the highest-ranked vertices are shuffled among themselves. The figure displays the effect of this shuffling operation on the indexing time of NPSL. We can see that on TOPC shuffling of just 1% of the vertices caused the indexing time to triple.

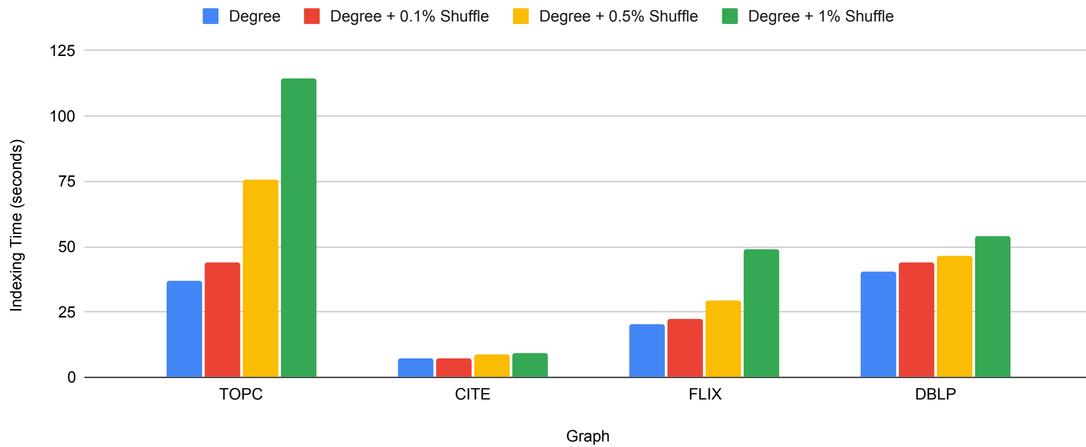


Figure 5.5 [Nebula] Effect of shuffling a portion of the vertex ranks on the indexing time on moderately easy graphs.

Figures 5.6 and 5.7 display the memory consumption and indexing time respectively of NPSL under different ranking methods. These results are the averages of a minimum of 5 runs and we checked the standard deviation of the results to keep the variation under control. Our results confirm the tests performed by Li et al. (2021) where betweenness centrality outperformed degree.

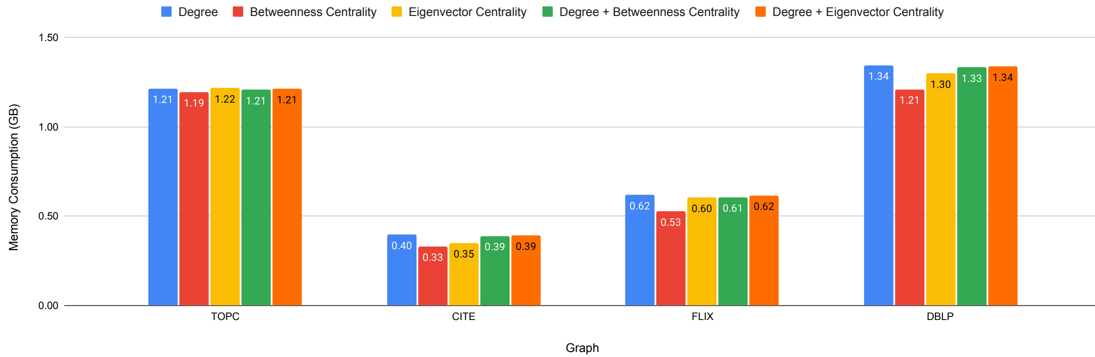


Figure 5.6 [Nebula] Memory consumption of the labels generated by NPSL when different ranking methods are used on moderately easy graphs.

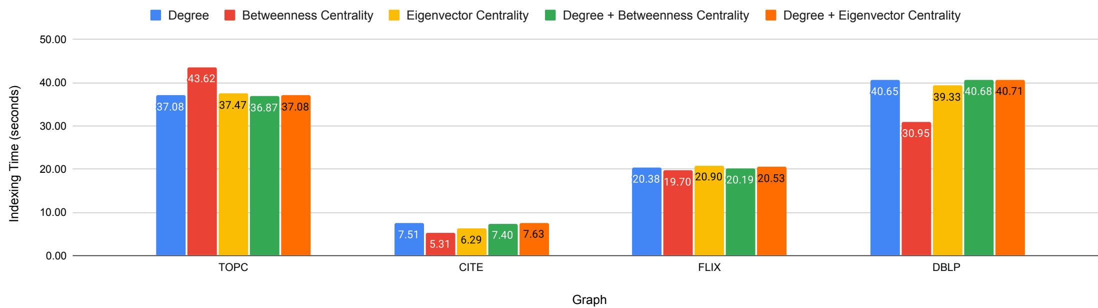


Figure 5.7 [Nebula] Indexing time of NPSL when different ranking methods are used on moderately easy graphs.

Despite the better performance of betweenness centrality, many of the experiments we performed used degrees for ranking since they were performed before we obtained these results. Furthermore, degree-based ranking is commonly used in the literature and specifically is used in the prior works we compared our results to.

The rank modification explained in Section 4.4 is a fundamental part of our distributed method. To see the effect it has on the label cover size in isolation, we applied the same modification in a single node setting. This was done by creating a vertex separator using the `Mt-KaHyPar` default preset (Gottesbüren et al., 2021) for 4 partitions and modifying the rank of the vertex separator in the same way as the distribution method. The results can be seen in Tables 5.8 and 5.9 and show that the

modification is beneficial when applied to a degree-based ranking in most graphs in terms of label cover size. Conversely, when applied to a betweenness centrality-based ranking the method often causes a small increase in label cover size.

	Degree	Degree + VS	Reduction
DELI	0.040	0.040	-0.29%
LAST	0.177	0.168	5.09%
DIGG	0.198	0.193	2.75%
TOPC	1.214	1.206	0.68%
CITE	0.396	0.354	10.67%
FLIX	0.618	0.589	4.70%
DBLP	1.344	1.282	4.59%

Table 5.8 [Gandalf] Memory used to store the labels (in gigabytes) with different ranking methods on easy and moderately easy graphs. Degree refers to degree-based ranking. Degree + VS refers to degree-based ranking where the ranks of the vertex separator (for a 4 node partitioning) has been increased. Reduction shows the percentage decrease in memory usage when going from Degree to Degree + VS.

	BC	BC + VS	Reduction
DELI	0.044	0.043	0.71%
LAST	0.158	0.152	3.80%
DIGG	0.182	0.178	1.99%
TOPC	1.183	1.184	-0.15%
CITE	0.317	0.337	-6.12%
FLIX	0.543	0.525	3.44%
DBLP	1.204	1.246	-3.55%

Table 5.9 [Gandalf] Memory used to store the labels (in gigabytes) with different ranking methods on easy and moderately easy graphs. BC refers to betweenness centrality-based ranking. BC + VS refers to betweenness centrality-based ranking where the ranks of the vertex separator (for a 4 node partitioning) has been increased. Reduction shows the percentage decrease in memory usage when going from BC to BC + VS.

5.2.5 Vertical Scalability Experiments

Vertical scaling is defined as the process of increasing the resources (usually CPU and memory) available to a single node (Barzu, Carabas & Tapus, 2017). As such, the algorithm is said to be vertically scalable if it can adequately benefit from such an increase. In this section, we evaluated the vertical scalability of our NPSL implementation by measuring and comparing its indexing time on varying thread counts. Increasing the thread count will cause the program to use more of the available cores of the CPU. As these experiments were carried out on Karolina (a system with 128 physical CPU cores per node), we limited the maximum thread count to 128.

Figure 5.8 demonstrates the indexing time of our implementation with respect to different thread counts. We found that up to a thread count of 16, the implemen-

tation scaled near-linearly. However, with higher thread counts the scalability was sublinear. This is also supported by Figure 5.9 which features larger graphs. These experiments were repeated on the original PSL implementation by Li et al. (2021) and we saw similar results.

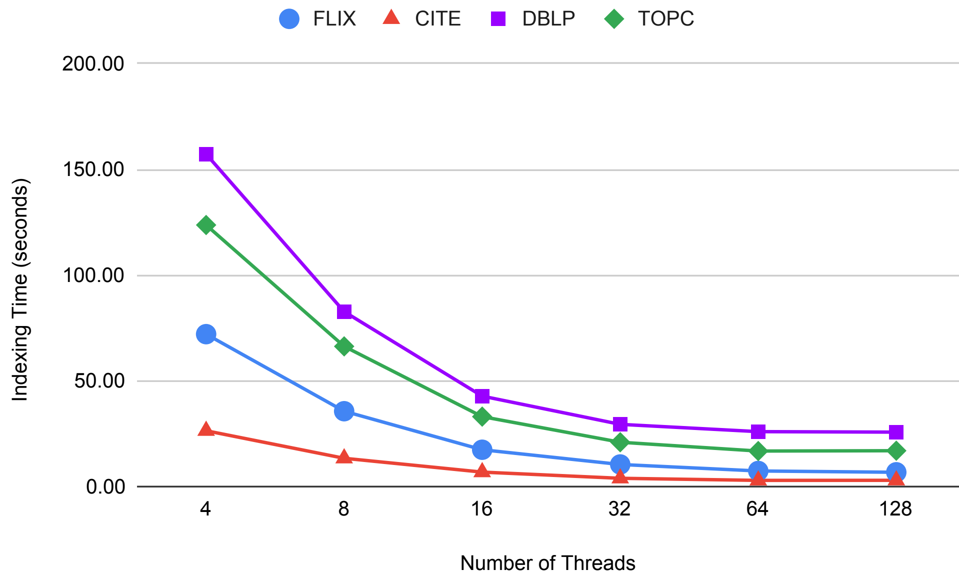


Figure 5.8 [Karolina] Indexing time of NPSL with varying thread counts on moderately easy grahps.

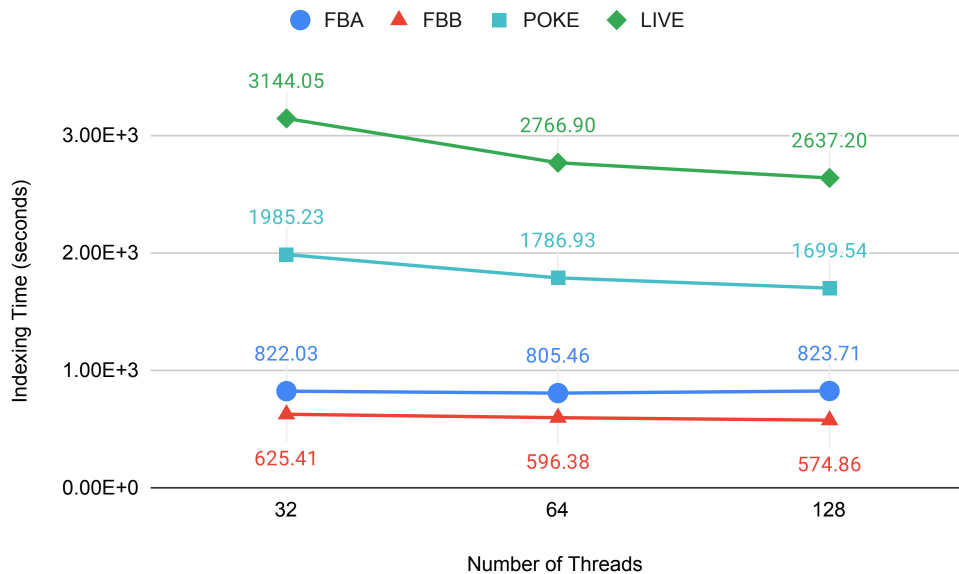


Figure 5.9 [Karolina] Indexing time of NPSL with varying thread counts on hard and moderately hard graphs

We suggest that this problem stems from Last-Level Cache (LLC) usage. The CPUs

used in Karolina feature LLCs of 256MB. Each node has 2 CPUs which results in 512MB of LLC memory. This amount is the same regardless of the number of threads used. LLC is utilized heavily in NPSL. Specifically, LLC is used by the following:

- As explained in Section 3.4, PSL uses a dense array to store the distances for the vertex currently being processed. When the algorithm considers adding a vertex v to a vertex u' label set, the pruning procedure performs lookups on this array for the labels of vertex v. Since the labels of a vertex are hard to predict and are not contiguous, this causes a lot of unaligned memory accesses and as a result, fills LLC. Since each thread processes a distinct vertex in PSL, there needs to be a distinct instance of this array per thread. Increasing the thread count increases the total memory used for this purpose and fills LLC more quickly.
- As explained in Section 5.2.2, BP labels can occupy a lot of memory and are also accessed in an unaligned manner.

While increasing the number of threads allows for more computational power, we propose that this is offset by the amount of cache misses caused by the higher traffic in LLC. To illustrate this point, we repeated the experiment from Figure 5.9 without using BP labels. The results can be seen in Figure 5.10 and indicate a speedup closer to linear than in Figure 5.9. We can also see that LIVE, POKE, and FBA were processed faster by a small margin on 128 threads when BP was not used. As explained in Section 5.2.2, BP almost always has a significant positive effect on the indexing time. However, contrary to the general assumption within the literature, these results show that this is not the case when a high number of threads is used.

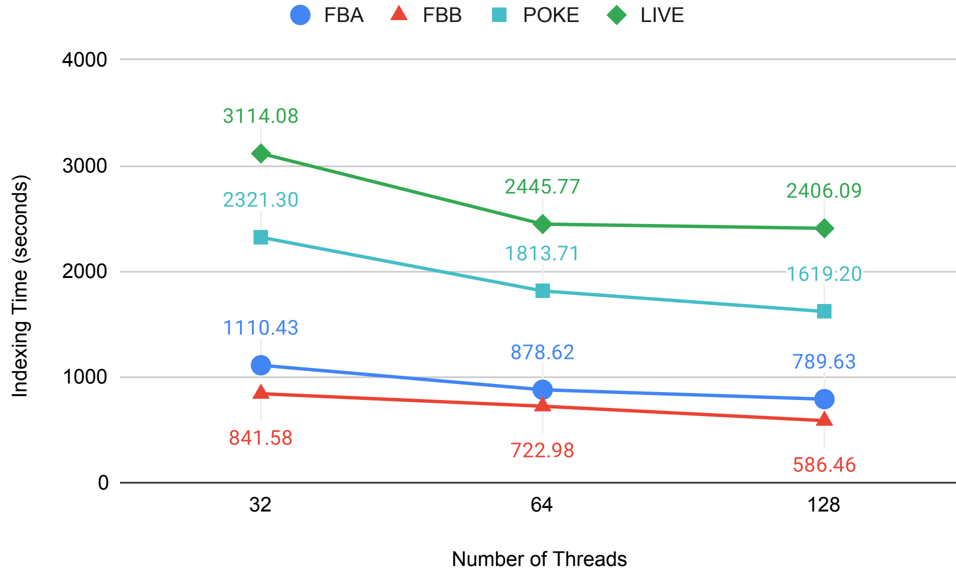


Figure 5.10 [Karolina] Indexing time of NPSL with varying thread counts on hard and moderately hard graphs without using BP labels

The previous result is supported by Tables 5.10 and 5.11 where the LLC miss rate of NPSL was measured using the Linux tool `perf`. Two situations can be observed in these results. Firstly, in most cases LLC miss rate increases when the thread count increases. Secondly, LLC miss rate consistently decreases when BP labels are turned off. Gandalf has a lower core count than Karolina and is overall slower. As a result, unlike Karolina, it can not reach the point where the algorithm no longer scales. We would like to repeat these experiments on Karolina however are unable to do so due to having no access to the hardware counters. Despite this, the results here make a strong argument for the negative effect of BP labels when the thread count is high.

Graph	Thread Count	LLC Miss Rate (%)
DBLP	15	40.18
DBLP	30	42.78
DBLP	60	43.63
TOPC	15	18.03
TOPC	30	18.89
TOPC	60	20.19
FLIX	15	21.8
FLIX	30	21.27
FLIX	60	23.7

Table 5.10 [Gandalf] LLC miss rate of NPSL on some moderately easy graphs with varying thread counts with BP labels.

Graph	Thread Count	LLC Miss Rate (%)
DBLP	15	31.83
DBLP	30	33.29
DBLP	60	35.26
TOPC	15	11.45
TOPC	30	12.19
TOPC	60	13.35
FLIX	15	17.38
FLIX	30	17.94
FLIX	60	18.64

Table 5.11 [Gandalf] LLC miss rate of NPSL on some moderately easy graphs with varying thread counts without BP labels.

5.3.1 Performance Evaluation of DPSL

Figures 5.11 and 5.12 show the correctness of the hypothesis we presented in Section 4.6. In Figure 5.11, we can see that DPSL provides an overall lower indexing time per level when compared to NPSL when identical parameters are used for both. DPSL indexing time is equivalent to the indexing time of the slowest MPI process as other processes need to wait for the slowest one to finish for synchronization. It should be noted however that this run excludes the synchronization steps and as a result the network cost which is demonstrated later. In Figure 5.12, we can see that each node generates and therefore stores fewer labels than the single node NPSL execution. However, it can be seen that DPSL uses more memory overall when the total across all nodes is taken into account. This total can be reduced significantly by removing the duplicate labels that reside in the vertex separator.

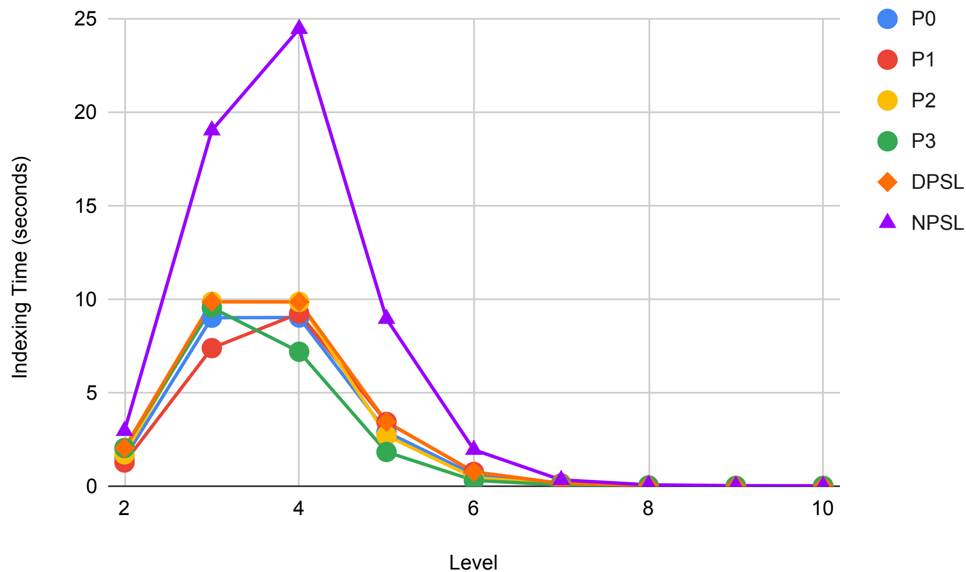


Figure 5.11 [Gandalf, NT=15] Indexing time of the different levels of NPSL and DPSL when processing DBLP. The first four series are the 4 processes used for DPSL, DPSL is the maximum of these four (the other processes wait for the slowest one).

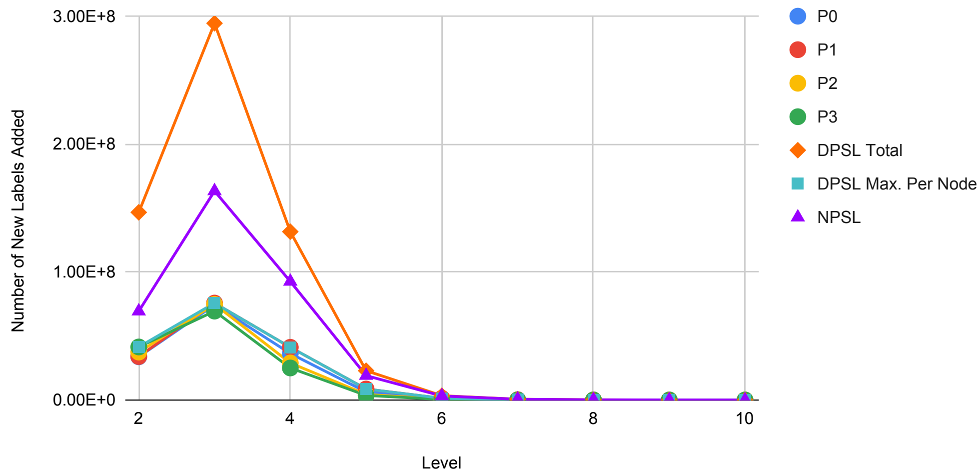
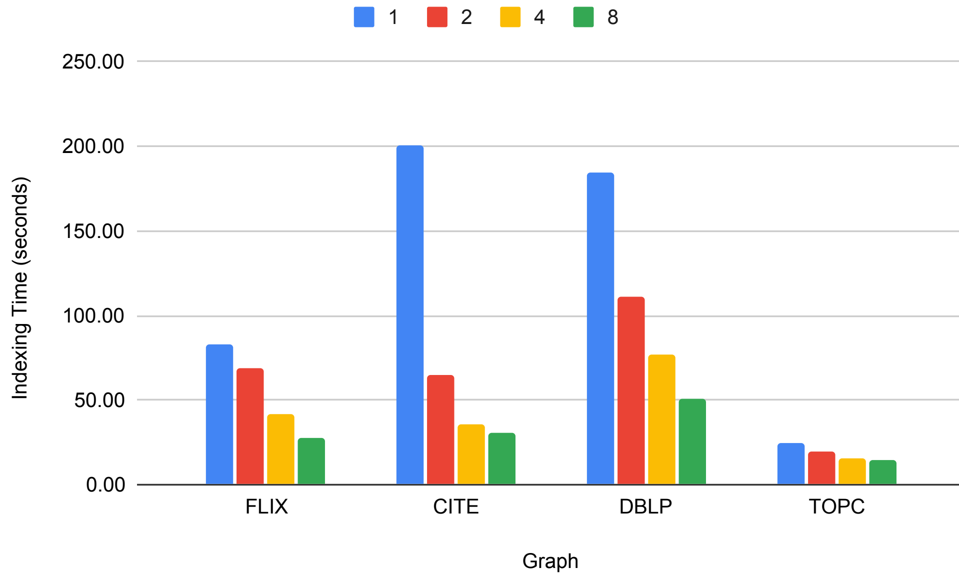
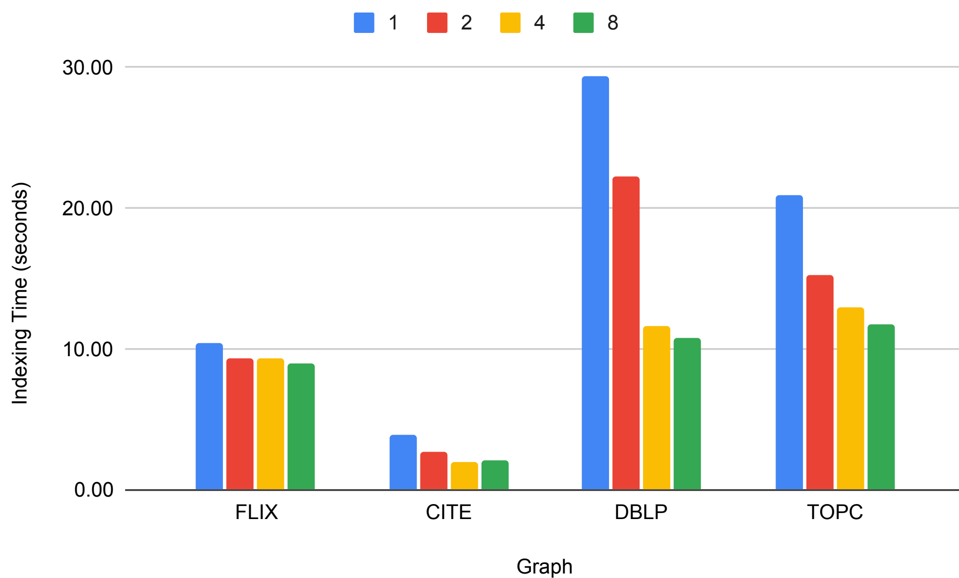


Figure 5.12 [Gandalf, NT=15] Label counts added on each level for NPSL and DPSL when processing DBLP. The first four series are the 4 processes used for DPSL, DPSL total is the sum of those 4 processes, DPSL Max. Per Node is the maximum value among all the processes.

Figure 5.13 shows the indexing time of DPSL on 2,4 and 8 nodes with and without compression. The single node results mentioned are provided using NPSL. The results without compression indicated a higher speedup than the results with compression. This is largely related to the load balancing issue discussed in Section 4.7. For CITE the run without compression indicates a higher than 2x speedup for the 2 node runs. This can be explained by the vertex separator ranking DPSL performs as explained in Section 5.2.4.



(a) CL=0



(b) CL=3

Figure 5.13 [Karolina] Indexing time of DPSL on moderately easy graphs with and without compression on a varying number of nodes

Tables 5.12 and 5.13 show the indexing time and speedup of DPSL respectively on some larger graphs.

	1	2	4	8
FBA	822.03	785.62	625.26	720.75
FBB	625.41	425.40	417.27	560.95
POKE	1,985.23	1,623.47	1,427.47	1,321.55
LIVE	3,144.05	1,719.25	1,476.37	1,347.11

Table 5.12 [Karolina] Indexing time (seconds) of DPSL on hard and moderately hard graphs on a varying number of nodes

	1	2	4	8
FBA	1.00	1.05	1.31	1.14
FBB	1.00	1.47	1.50	1.11
POKE	1.00	1.22	1.39	1.50
LIVE	1.00	1.83	2.13	2.33

Table 5.13 [Karolina] Speedup of DPSL on hard and moderately hard graphs with respect to single node execution (NPSL)

The speedups obtained from DPSL, coupled with the vertical scalability problem explained in Section 5.2.5 allow DPSL to outperform NPSL when the same number of cores is used for both as shown in Figure 5.14.

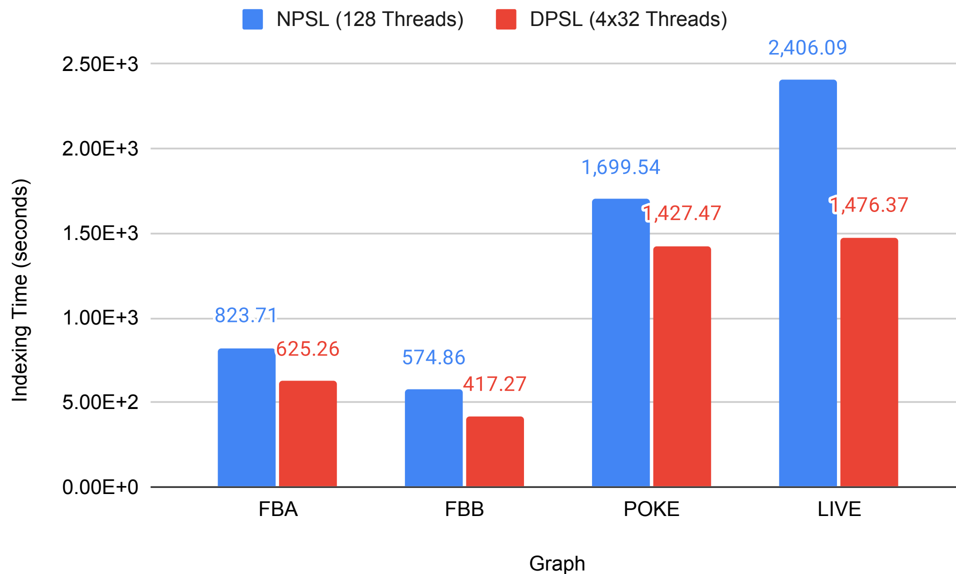


Figure 5.14 [Karolina] 128 thread NPSL and 32 thread 4 node DPSL execution comparison on hard and moderately hard graphs in terms of indexing time.

Figure 5.15 shows the breakdown of the execution time of DPSL. When processing hard and moderately hard graphs, both the partitioning and synchronization times constituted less than 2.4% of the execution time.

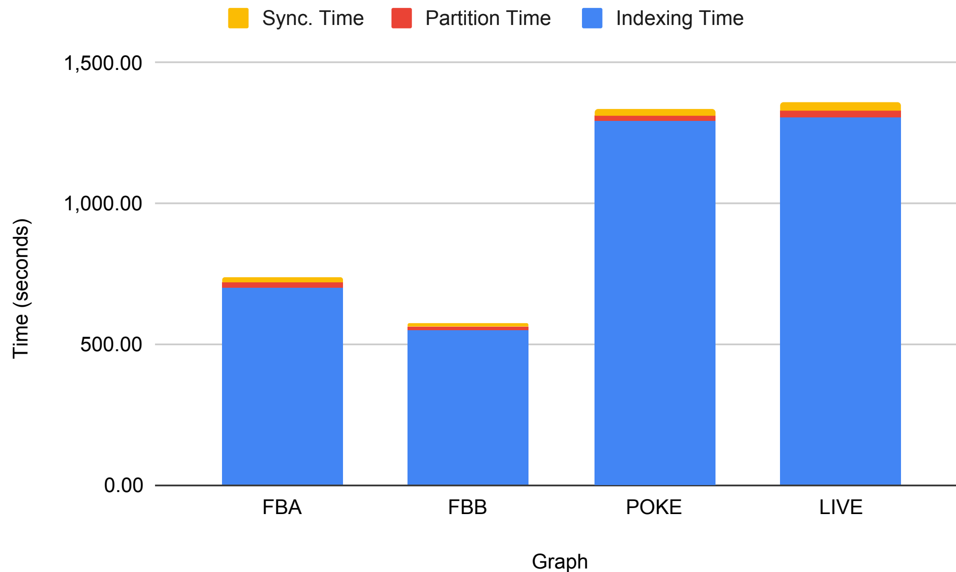


Figure 5.15 [Karolina] Breakdown of the execution times of 4 node DPSL executions on hard and moderately hard graphs.

Tables 5.14 and 5.15 show two different memory consumption interpretations for the DPSL algorithm. Table 5.14 shows that the memory consumed across all nodes increases with the number of nodes used. This is to be expected due to the duplicated labels on the vertex separator. Table 5.15 shows the highest memory consumption among the nodes and a decrease as the number of nodes increases can be observed. As a result, DPSL can potentially process graphs that NPSL can not (due to the limited amount of memory on each node) by distributing the memory load among the nodes. These results are highly dependent on the quality of the partitioning. The size of the vertex separator is crucial as it determines the number of labels that need to be replicated. The balance of the partitions is also very important since the largest graphs we can process with this method are limited by the maximum memory consumption shown in Table 5.15, meaning that the partition with the heaviest load will dictate this variable. As shown in Tables 5.15 and 5.12, the partitions for FBA and FBB on the 8 node runs were not of sufficient quality and as a result, an increase in the memory consumption and indexing time was observed.

	1	2	4	8
FBA	13.89	20.33	38.68	53.54
FBB	10.38	14.07	25.85	54.07
POKE	26.38	36.76	68.26	137.76
LIVE	41.20	48.77	78.34	142.35

Table 5.14 [Karolina] Total memory consumption (in GBs) of the label cover in 2,4,8 node DPSL when compared to NPSL (denoted as 1 node) for hard and moderately hard graphs. For DPSL, the values are the summation of the memory used to store the label sets across every node.

	1	2	4	8
FBA	13.89	10.64	10.02	10.16
FBB	10.38	7.43	6.72	6.91
POKE	26.38	19.23	17.62	17.47
LIVE	41.20	25.04	22.01	19.16

Table 5.15 [Karolina] Maximum per node memory consumption (in GBs) of the label cover in 2,4,8 node DPSL when compared to NPSL (denoted as 1 node) for hard and moderately hard graphs. For DPSL, the values are the memory consumption result from the node that stores the largest number of labels.

We explained in Section 5.2.4, that due to the ranking modification on the vertex separator, our DPSL method generates a label cover with a different size. This effect could be negative or positive depending on the ranking method used. Due to the synchronization method used, DPSL is not able to prune the labels on the vertex separator as effectively as NPSL. As a result, the size of the label cover is slightly larger than what was suggested in Section 5.2.4. In our tests, this slight increase did not exceed 0.36% as can be seen in Table 5.16.

	NPSL	NPSL (VS Order)	DPSL (4 Nodes)	Increase
FBA	13.89	13.12	13.13	0.08%
FBB	10.38	9.73	9.75	0.21%
POKE	26.38	25.86	25.88	0.07%
LIVE	41.20	36.89	37.02	0.36%

Table 5.16 [Gandalf] Size of the label cover in gigabytes using degree ranking for hard and moderately hard graphs. The first column is an unmodified NPSL execution. The second column is an NPSL execution where the vertex separator vertices are increased in rank as in Tables 5.8 and 5.9. The third column is a 4 node DPSL execution where the generated label covers are merged with the duplicates removed. The fourth column shows the increase from the second column to the third column.

5.3.2 Comparison to Previous Work

To our knowledge, no distributed implementation of the PSL algorithm exists other than this work. We compared our implementation against two distributed PLL implementations. The first implementation will be denoted as DPLANT and is introduced by Lakhotia et al. (2019). We used their hybrid implementation for comparisons as it reportedly achieves better performance and scalability. The second implementation will be denoted as DVCPLL and is introduced by Jin et al. (2020). DVCPLL also uses the BP label optimization explained in Section 3.7. We used the same BP root count of 15 for all runs of DVCPLL and DPSL. Preprocessing steps such as BP label generation and ranking were not included in the indexing time calculations.

Figure 5.16 shows the indexing time of all the implementations on different graphs when running on Karolina with 4 nodes. Each node is set to use 32 threads for all implementations. DVCPLL and DPLANT do not utilize any form of compression on the input graph as a result we also included results using a compression level of 0 for DPSL. These should provide a fairer comparison. Some of the compression methods mentioned in this text are also applicable to the PLL algorithm and could be used with DVCPLL and DPLANT. We can report that DPSL had a better performance on all the graphs tested with respect to DVCPLL and DPLANT in this setting.

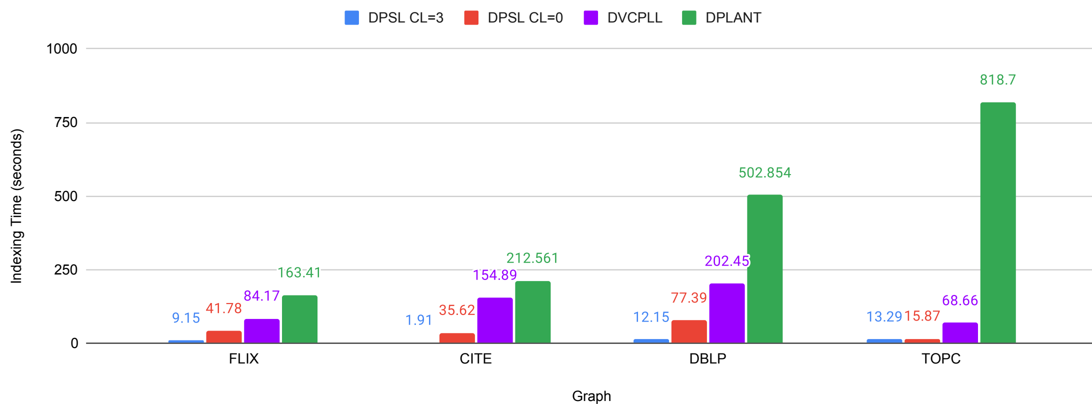


Figure 5.16 [Karolina] Indexing time of DPSL in comparison to the prior works DVCPLL and DPLANT on 4 nodes (32 threads per node) on moderately easy graphs.

Figure 5.17 demonstrates the scalability of the algorithms when the number of nodes used on the Karolina cluster increases. Each node is again set to use 32 threads for all implementations. We found that despite its initial low performance in Figure 5.16, DPLANT exhibited better scalability than DPSL and DVCPLL. As a result, it could

outperform the others on larger clusters with more nodes. In fact, Lakhotia et al. (2019) suggested that their algorithms can achieve super-linear speedup when the number of nodes is increased due to a release of pressure on the memory management system. Figure 5.18 shows the scalability of the algorithm when the number of threads used on each node is increased. Again, DPLANT scaled better despite being overall slower than the others.

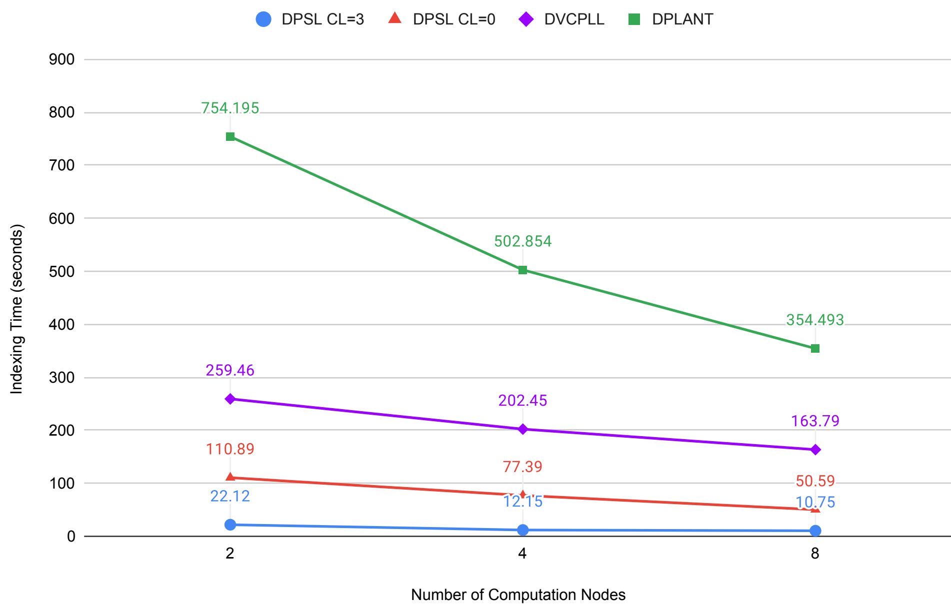


Figure 5.17 [Karolina] Indexing time of DPSL in comparison to the prior works DVCPLL and DPLANT on a varying number of nodes (32 threads per node) on the graph DBLP

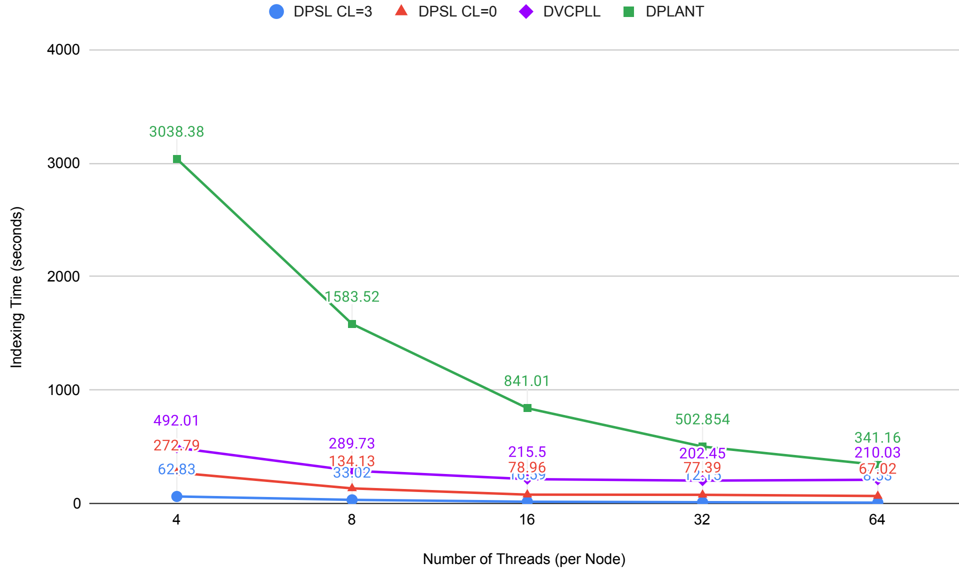


Figure 5.18 [Karolina] Indexing time of DPSL in comparison to the prior works DVCPLL and DPLANT on 4 nodes with a varying number of threads on the graph DBLP

We also performed indexing time comparisons between DVCPLL and DPSL on 4 nodes using hard and moderately hard graphs as shown in Table 5.17. DPLANT was omitted from this experiment as it was outperformed by DVCPLL within the parameters of this experiment. The results from this experiment confirm the superior performance of DPSL. DVCPLL performs better than DPSL CL=0 on FBB, likely as a result of the relatively low average degree of FBB.

	DPSL CL=3	DPSL CL=0	DVCPLL
FBA	625.26	1,596.14	1,610.91
FBB	417.27	1,243.18	1,056.84
POKE	1,427.47	1,710.88	4,572.89
LIVE	1,476.37	2,621.28	8,327.82

Table 5.17 [Karolina] Indexing time of DPSL in comparison to the prior work DVCPLL on 4 nodes (32 threads per node) on hard and moderately hard graphs.

It should be noted that, both DVCPLL and DPLANT contain additional features that DPSL does not support in its current form. Specifically, DPLANT is reportedly compatible with directed and weighted graphs and DVCPLL while focusing on undirected graphs is said to be extendible to directed and weighted graphs as well. However, Li et al. (2021) suggest that the PSL algorithm can be extended to directed graphs by splitting the label set in two, we believe this approach would be

applicable to DPSL as well. Furthermore, DPSL does not perform well on high-diameter graphs such as road networks which DVCPLL and DPLANT can process without issue. Since this is an issue with the nature of the algorithm itself, it is unlikely to be resolved.

5.4 Threats to Validity

There are several threats to the validity of this research and these experiments. Some of these are listed below and will be addressed in this section.

1. The specified experiments are not enough to indicate the correctness of the algorithms presented.
2. The results are taken from a limited set of data. More specifically different graph types (for example, road networks) are not included in the experiments.
3. The performance results are not reliable due to the low number of repeated experiments.

To address the first threat, we tested the algorithms further in terms of the correctness of their results. Both DPSL and NPSL are tested using randomized queries and the results are compared against a sequential Breadth First Search (BFS) algorithm. This experiment was repeated on multiple graphs for different Bit-Parallel label sizes, ranking methods, and with all the compression techniques mentioned. For a selection of smaller graphs, we also performed queries for all possible pairs of vertices and again compared the results to a sequential BFS algorithm. For NPSL only, we performed comparison-based tests against the original PSL code. In these tests, the numbers of the labels generated by the algorithms were compared. This comparison was done for each level of the algorithms on each vertex separately. In all of these tests, we found no indication that either implementation was generating incorrect label sets. Furthermore, we believe the explanations and proofs provided in Chapter 4 provides solid reasoning for the correctness of the methods.

To address the second threat, we tried to test the algorithms with different graphs of different sizes acquired from different sources. However, the algorithm is designed for low-diameter graphs like social networks. There is an upper limit on the distances the algorithm can index. As a result, on high-diameter graphs like road networks, the algorithm will stop once it hits this upper limit. The query results obtained

would not be reliable since any distance larger than the upper limit can not be accounted for. Furthermore, certain optimizations are not designed for such large distances and can cause the results to be incorrect due to overflows. As a result, we decided to avoid any performance results involving such graphs. The other works mentioned in this text, DPLANT (Lakhotia et al., 2019) and DVCPLL (Jin et al., 2020) are designed to work with various types of networks and should be preferred when working with high-diameter graphs.

To address the third threat, we measured the standard deviation of all timing-related results mentioned throughout this text. For both DPSL and NPSL repeated executions on the same graph with the same setting had a standard deviation of less than 11%. Specifically, on the graph POKE we saw a maximum standard deviation of 8.51% and on the graph LIVE we saw a maximum standard deviation of 6.41%.

6. CONCLUSION

In this thesis, we presented several optimizations and extensions around the PSL algorithm. Our implementation based on the CSR data structure increased the speed of the indexing. Furthermore, the results we presented in Sections 5.2.2, 5.2.4, 5.2.5 should help with the fine-tuning of the parameters of the algorithm.

We also argued and showed that a leaf elimination algorithm could increase the speed of the indexing and reduce the final label size. This method can also be combined with other compression methods discussed in Section 3.6.

Finally, we introduced a partition-based distributed implementation of the PSL algorithm. With this implementation, we were able to reduce the indexing time of the algorithm by distributing the computational load to multiple nodes. As discussed in Section 5.2.5, we had trouble scaling the PSL algorithm to CPUs with large amounts of cores. Our method allows using several less powerful CPUs to exceed the performance of a larger CPU such as the one in the Karolina cluster we used. Furthermore, our method could allow the processing of larger graphs by reducing the memory requirements as shown in Section 5.3.1.

7. FUTURE WORK

We had success using the leaf elimination technique discussed in Section 3.6. Other methods for eliminating certain vertices or compressing the graph could be investigated. We suspect that the linear set compression technique suggested by Anirban et al. (2022) could be used and might allow the algorithm to perform better on large-diameter graphs such as road networks.

Our research indicates that predicting the number of labels that will be generated for each vertex is not a trivial task. Specifically, we found no strong correlation between the number of labels and the ranks, degrees, and various centrality metrics of vertices. Similarly, the indexing time of each individual vertex is hard to predict and is not strongly correlated with any other data we collected. However, we suspect a more detailed analysis (perhaps making use of machine learning models) could help with such a prediction. Being able to predict these metrics would allow for better load balancing through the partitioning algorithm and should increase the performance of DPSL.

Our distributed method should be applicable to GPUs as well. In a multi-GPU setting, the partitions can be distributed among the GPUs. A hybrid GPU-CPU algorithm should also be possible by assigning a partition to the CPU as well. However, this requires the implementation of the PSL algorithm on GPUs. We worked on this problem and found that there were several issues that could affect such an implementation. Most importantly, due to the limited amount of memory and the high number of cores on a typical modern GPU it is not feasible to create a distance cache (described in Section 4.7) for each thread if we set the thread count to be equivalent to the number of cores. Creating fewer threads would cause the performance to drop. We suspect this can be resolved by assigning a group of threads to each vertex and thus allowing them to share the distance cache but this has the downside of requiring more synchronization and complexity. Furthermore, our CPU PSL implementation makes high use of dynamically allocated memory

and we found memory allocation within GPU kernels to be slow. We suspect this could be solved by pre-allocating all the memory and coming up with a system to distribute it among the threads as necessary.

We exclusively focussed on uniform partitionings where each partition had the same target weight. This was done as the nodes on the cluster were identical and thus we required equivalent computational loads. This may not necessarily be the case on all systems. Many of the partitioning tools we referred to in this thesis allow non-uniform partitionings to be created which should allow a set of heterogenous nodes to share the work more fairly. This could also be used with the hypothetical GPU implementation we mentioned previously. In cases where the machine contains multiple different GPUs or a hybrid CPU-GPU method is used, the computational power of each unit is likely to be different and should benefit from non-uniform partitioning. Furthermore, even if the GPU implementation fails to reach the performance of the CPU implementation, it can still be useful in the hybrid case by simply being assigned a smaller partition requiring less computation.

In our method, we created a single vertex separator for the whole graph even in cases where more than 2 partitions were created. We suspect this might be partially responsible for the method's poor performance on 8 nodes. Implementing a recursive bi-partitioning-based method could bring better results on higher node counts. In such a partitioning, the graph would be recursively divided in two, each division creating a different vertex separator. This could improve communication in the synchronization algorithm as it can be done recursively. However, as mentioned in Section 4.5, we did not have performance problems with our synchronization algorithm. The main benefits we would expect from this approach are a reduction in duplicated labels and better load balancing.

A hybrid approach could be applied to DPSL where the labels on the vertex separator are precomputed in a shared memory setting. This theoretically would remove the necessity for the synchronization step of DPSL. However, it would reduce the amount of pruning that can be done on the vertex separator and can increase the final label count. It should be noted that Lakhotia et al. (2019) saw success with a similar approach in their hybrid algorithm.

We developed a distributed querying algorithm to test the correctness of the labels generated by DPSL. However, due to time constraints, we did not optimize this algorithm or tested any other approach to querying in a distributed setting. DPSL can still be used by combining the label sets generated on each machine and performing the queries on a single machine. However, the development of an efficient distributed querying algorithm would make it more substantial.

Li et al. (2021) developed an improved ranking method based on betweenness centrality which takes local minimum vertices into account. We were not able to test this on DPSL due to time constraints. Such a ranking method may help improve the load balancing of the algorithm if it is used during the partitioning stage.

BIBLIOGRAPHY

Abeywickrama, T. & Cheema, M. A. (2017). Efficient landmark-based candidate generation for knn queries on road networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *10178 LNCS*, 425–440.

Akiba, T., Iwata, Y., Kawarabayashi, K.-I., & Kawata, Y. (2014). Fast shortest-path distance queries on road networks by pruned highway labeling.

Akiba, T., Iwata, Y., & Yoshida, Y. (2013). Fast exact shortest-path distance queries on large networks by pruned landmark labeling.

Anirban, S., Wang, J., Islam, M. S., Kayesh, H., Li, J., & Huang, M. L. (2022). Compression techniques for 2-hop labeling for shortest distance queries. *World Wide Web*, *25*, 151–174.

Barzu, A. P., Carabas, M., & Tapus, N. (2017). Scalability of a web server: How does vertical scalability improve the performance of a server? *Proceedings - 2017 21st International Conference on Control Systems and Computer, CSCS 2017*, 115–122.

Bellman, R. (1958). On a routing problem. *Quart. Appl. Math.*, *16*, 87–90.

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

Davis, T. A. & Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, *38*.

Dhulipala, L., Blelloch, G., & Shun, J. (2017). Julienne: A framework for parallel graph algorithms using work-efficient bucketing. volume Part F129316, (pp. 293–304). Association for Computing Machinery.

Dijkstra, E. W. (1959). A note on two problems in connection with graphs. *Numerische Mathematik 1*, 269–296.

Firoz, J. S., Zalewski, M., Kanewala, T., & Lumsdaine, A. (2019). Synchronization-avoiding graph algorithms. (pp. 52–61). Institute of Electrical and Electronics Engineers Inc.

Gabert, K. & Çatalyürek, U. V. (2021). PIGO: A parallel graph input/output library. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (pp. 276–279). IEEE.

Gottesbüren, L., Heuer, T., Sanders, P., & Schlag, S. (2022). Shared-memory n-level hypergraph partitioning. In *24th Workshop on Algorithm Engineering and Experiments (ALENEX 2022)*. SIAM.

Gottesbüren, L., Heuer, T., Sanders, P., & Schlag, S. (2021). Scalable shared-memory hypergraph partitioning. In *23rd Workshop on Algorithm Engineering and Experiments (ALENEX 2021)*, (pp. 16–30). SIAM.

Henz, M., Müller, T., & Thiel, S. (2004). Global constraints for round robin tournament scheduling. *European Journal of Operational Research*, *153*(1), 92–101.

Jin, R., Peng, Z., Wu, W., Dragan, F., Agrawal, G., & Ren, B. (2020). Parallelizing pruned landmark labeling: Dealing with dependencies in graph algorithms. Association for Computing Machinery.

Lakhotia, K., Dong, Q., Kannan, R., & Prasanna, V. (2019). Planting trees for scalable and efficient canonical hub labeling.

Lasalle, D. & Karypis, G. (2013a). Efficient nested dissection for multicore architectures. In *27th IEEE International Parallel & Distributed Processing Symposium, 2013*.

Lasalle, D. & Karypis, G. (2013b). Multi-threaded graph partitioning.

Lasalle, D., Mostofa, M., Patwary, A., Satish, N., Sundaram, N., Karypis, G., & Dubey, P. (2015). Improving graph partitioning for modern graphs and architectures. In *5th Workshop on Irregular applications: Architectures and Algorithms, Supercomputing, 2015*.

Leiserson, C. E. & Schardl, T. B. (2010). A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). (pp. 303–314).

Li, J., Wang, X., Deng, K., Yang, X., Sellis, T., & Yu, J. X. (2017). Most influential community search over large social networks. *Proceedings - International Conference on Data Engineering*, 871–882.

Li, W., Qiao, M., Qin, L., Zhang, Y., Chang, L., & Lin, X. (2021). Distance labeling: on parallelism, compression, and ordering. *VLDB Journal*.

Makki, S. A. M. (1996). Efficient distributed breadth-first search algorithm. *computer communications ELSEVIER Computer Communications*, *19*, 628–636.

Meyer, U. & Sanders, P. (2003). -stepping: A parallelizable shortest path algorithm. *Journal of Algorithms*, *49*, 114–152.

Nguyen, D., Lenharth, A., & Pingali, K. (2013). A lightweight infrastructure for graph analytics. (pp. 456–471).

Qiu, K., Zhao, J., Zhu, Y., Wang, X., Yuan, J., & Wolf, T. (2018). Parapll: Fast parallel shortest-path distance query on large-scale weighted graphs. Association for Computing Machinery.

Rossi, R. A. & Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *AAAI*.

Shiralkar, P., Flammini, A., Menczer, F., & Ciampaglia, G. L. (2017). Finding streams in knowledge graphs to support fact checking. *Proceedings - IEEE International Conference on Data Mining, ICDM*, *2017-November*, 859–864.

Ueno, K., Suzumura, T., Maruyama, N., Fujisawa, K., & Matsuoka, S. (2017). Efficient breadth-first search on massively parallel and distributed-memory machines. *Data Science and Engineering*, *2*, 22–35.

Vieira, M. V., Golgher, P. B., Fonseca, B. M., Reis, D. C. D., Damazio, R., & Ribeiro-Neto, B. (2007). Efficient search ranking in social networks. *undefined*, 563–572.

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

Source Code 1 C++ data structure for storing the labels for a vertex. Each vertex in the graph has a LabelSet associated with it.

```
// Stores the labels for each vertex
struct LabelSet {
  // IDs of the vertices in order of distances
  vector<IDType> vertices;
  // Indices for the start and end of distances for the vertices vector
  vector<IDType> dist_ptrs;
};
```

Algorithm 5 Round Robin Tournament

Input: Number of nodes k (must be even)

$C \leftarrow$ an array of size k

$C(i) \leftarrow i$ for $i \in 0, 1...k$

for i in $0, 1...k-1$ **do**

  for j in $0, 1...k$ **do**

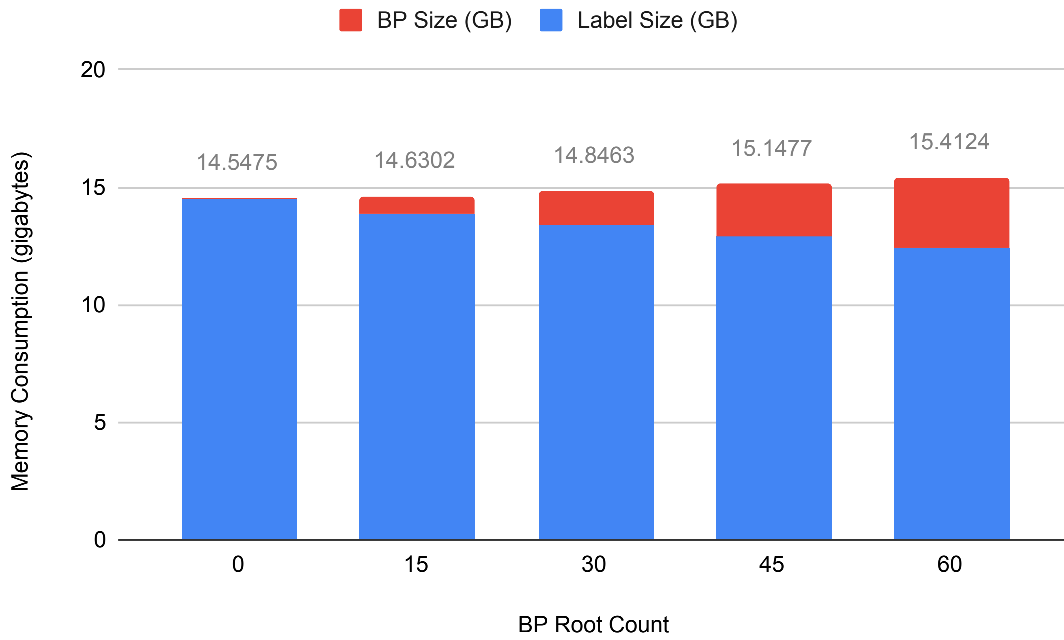
    $p \leftarrow (k - j - 1) \bmod k$

    pair $C(j)$ with $C(p)$

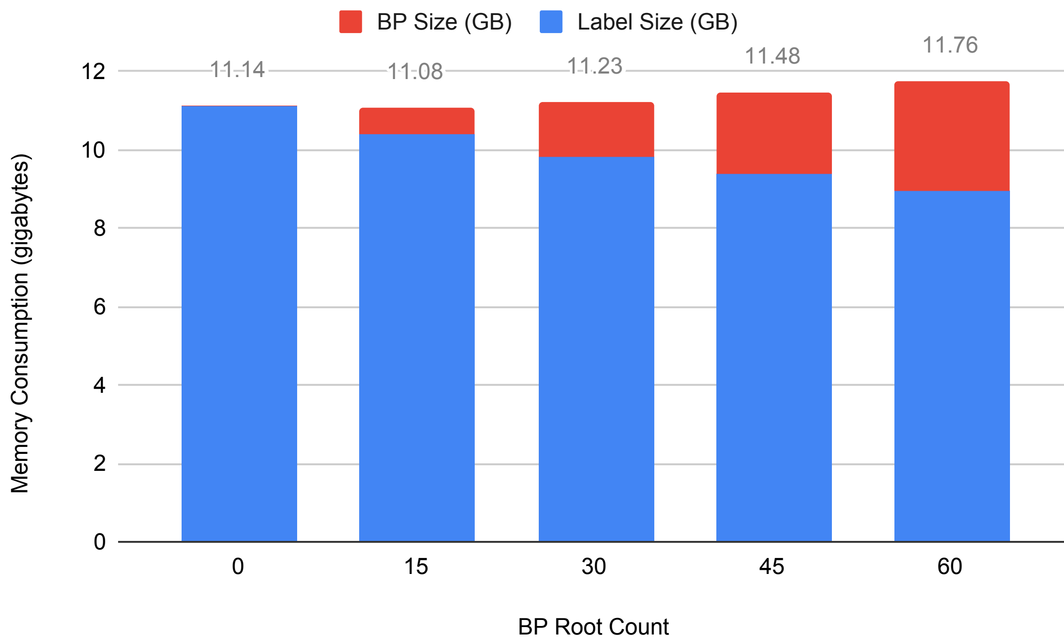
  rotate C right except for the first element

	Static, 1	Static, 256	Dynamic, 1	Dynamic, 256	Guided, 1	Guided, 256
FLIX	10.85	10.80	11.80	10.13	10.66	10.61
CITE	4.43	4.11	5.07	3.79	5.31	5.31
DBLP	33.42	33.56	29.91	28.39	42.02	44.16
TOPC	21.49	21.99	21.38	21.27	19.55	20.79

Table A1 [Karolina] Indexing time (seconds) of NPSL on moderately easy graphs when different OpenMP thread scheduling methods are used.



(a) FBA



(b) FBB

Figure A1 [Karolina] Effect of Bit-Parallel Label's root count (BPR) on the memory consumption of NPSL when processing moderately easy graphs.

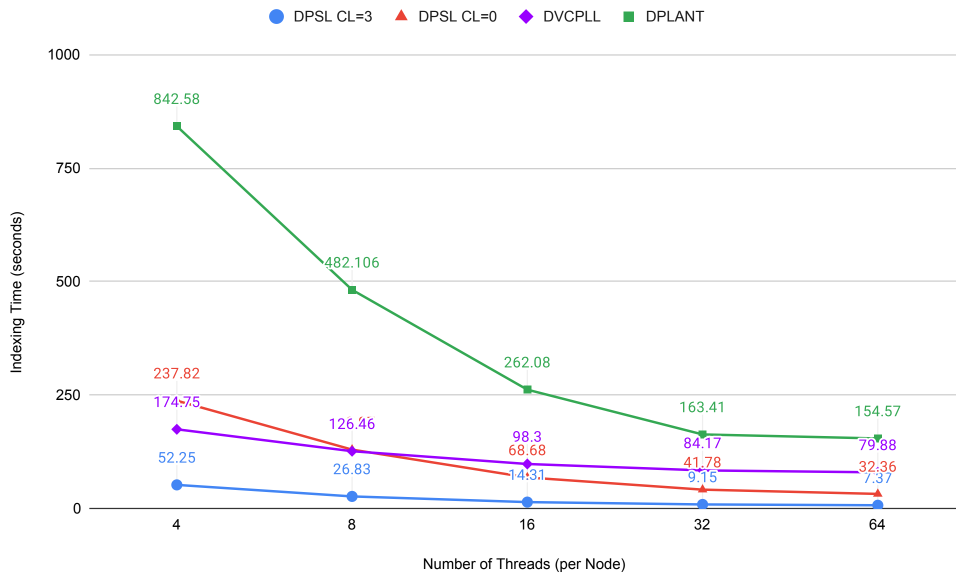


Figure A2 [Karolina] The experiment from Figure 5.18 repeated on the graph FLIX