Karlsruhe Institute of Technology

# Parallel Pruned Landmark Labeling for Shortest Path Queries on Unit-Weight Networks

Bachelor Thesis of

Damir Ferizovic

At the Department of Informatics
Institute of Theoretical Informatics

| | |
|---|---|
| Reviewer: | Prof. Dr. rer. nat. Peter Sanders |
| Second reviewer: | Prof. Dr. Dorothea Wagner |
| Advisor: | Prof. Dr. rer. nat. Peter Sanders |
| Second advisor: | Prof. Guy Blelloch |

Duration: November 1st, 2014 – February 28th, 2015

---

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

**Karlsruhe, February 25, 2015**

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
(**Damir Ferizovic**)

# Contents

**Abstract** vii

**1 Introduction** 1

**2 Preliminaries** 3

2.1 Graph . . . 3

2.2 Notations . . . 3

2.3 The problem . . . 3

2.4 Breadth-first search (BFS) . . . 4

2.5 Labeling . . . 4

**3 Related work** 5

3.1 Exact methods . . . 5

3.2 Approximate methods . . . 5

3.3 Pruned landmark labeling[AIY13] . . . 5

3.3.1 Landmark labeling . . . 6

3.3.2 Pruned landmark labeling . . . 6

3.3.3 Bit parallel labeling . . . 7

3.3.4 Results . . . 8

**4 Parallel pruned landmark labeling** 11

4.1 Vertex order . . . 11

4.2 Phase I . . . 12

4.2.1 Bit-parallel search from any k vertices . . . 13

4.3 Phase 2 . . . 14

4.4 Query . . . 16

4.5 Implementation . . . 16

4.5.1 Initialization . . . 16

4.5.2 Pruning of the frontier . . . 17

4.5.3 Vertex order induced cache-efficiency . . . 17
4.5.4 Locks . . . 17
4.6 Variations . . . 17
4.6.1 Lower memory consumption . . . 17
4.6.2 Directed edges . . . 18
4.6.3 Weighted edges . . . 18

**5 Experimental evaluation** **19**
5.1 Environment . . . 19
5.2 Datasets . . . 19
5.3 Performance of PLL . . . 20
5.4 Our performance results . . . 21
5.4.1 Single threaded . . . 21
5.4.2 Multi-threading . . . 21
5.4.2.1 Speed-ups with the memory-light version . . . 23
5.4.2.2 Speed-ups within the individual phases . . . 24
5.4.2.3 Size of the constructed index . . . 25
5.4.2.4 Query-performance . . . 26
5.4.2.5 Asymptotic analysis of the memory usage . . . 26

**6 Future work** **27**

**7 Conclusion** **29**

**Appendix** **31**
A Failed methods . . . 31
A.1 Generalised bit-parallel labeling . . . 31
A.2 Parallel Breadth-first-search . . . 31
A.3 Multi-source . . . 31

**8 Acknowledgments** **33**

**Bibliography** **35**

# Abstract

While using graphs inside of an application, there is often a need to repeatedly answer shortest-path distance queries between pairs of vertices. There are numerous ways to solve this problem. In this thesis, we present an approach that utilizes the research done within labeling-based methods, which represent a compromise between online-search-based methods (e.g., Breadth-First-Search) and transitive-closure-based methods (e.g., by using Floyd-Warshall), and thread-parallelism. Our focus within this thesis is to introduce an approach that scales well in a multi-threaded environment. It is based on a two-phased labeling method as found in [AIY13]; this method also forms a baseline for our comparisons. Additionally, we present general performance measures and discuss the scaling behaviour of our approach.

# 1. Introduction

Being able to answer shortest-path distance queries between any two vertices represents one of the fundamental problems on graphs and its usage spans over a wide range of applications. It is prevalent in most scenarios where data is represented in a graph-like manner. Bioinformatics, databases, and social networking are some of them.

For example, the distance between two users within social networks gives us information about the social relationship they have.
If we model web pages as vertices and the links between them as edges, the length of the path between two web pages indicates the relevance of one to another.

Different types of graphs require different types of techniques. This work sets its focus on complex networks (e.g., social networks). These graphs are small in diameter and dense in their construction. Another type of interesting real-world networks is road networks. They are not within the scope of this thesis as they require the utilization of different techniques; which are not efficiently usable together with complex networks.

There are numerous approaches to the problem stated in the above paragraphs. There is the possibility of answering the distance queries by computing them without any preprocessed information about the graph. This can be done by simply executing a breadth-first search (BFS) for every query. The downside to this solution is the time performance on larger graphs, which is not acceptable if there is a great amount of queries that have to be answered. The other extreme is to memorize the distances between all pairs of vertices by storing them within an *index* – possibly a two-dimensional array. Queries would be answered instantly. The critical problem with this approach is that any feasible methods have a high memory consumption and take a considerably amount of time to construct the index. A solution to this problem lies in combining the search with a precomputed index, as is done within [AIY13], [Wei10], [ADGW12].

The resulting index of our approach is based on a *2-hop cover* [CHKZ02]; we assign a vertex-set $S_v$ to each vertex $v$ in the graph in such a way that for every pair of vertices $u$ and $w$, $S_u \cap S_w$ contains a vertex that lies on a shortest path between $u$ and $w$. If for every pair of vertices $x$ and $r \in S_x$, the distance $dist(x, r)$ in the graph is known, the distance between two vertices can be computed as $\min_{r \in S_u \cap S_w} dist(u, r) + dist(r, w)$. The list of pairs $\{(w, dist(v, w)) : w \in S_v\}$ is called a *label* for vertex $v$. That is why the other name for creating a 2-hop cover is also the *labeling method*.

In the upcoming sections, we introduce existing techniques to compute and utilize these

labels. Most of these techniques use breadth-first search (BFS), which is the state-of-the-art method to compute distances within our graph type. Special attention is given to the pruned landmark labeling (PLL) from [AIY13]. It combines extremely fast distance computation with fast preprocessing. Closer look within its implementation shows possibilities for high parallelism. After introducing the existent research, we show the needed changes on PLL to integrate thread-parallelism and finally analyse the results over various datasets.

# 2. Preliminaries

Knowledge of some fundamental concepts is required in order to fully understand the research in this thesis. The following section introduces the needed topics and clarifies the notations we use.

## 2.1 Graph

Let $V$ be an arbitrary finite set and $E \subseteq V \times V$. A graph is defined as an ordered pair $G = (V, E)$. The edges of $G$ are called *undirected* if for every $(v, w) \in E$, $(w, v) \in E$ also holds, otherwise they are defined as *directed*. Undirected edges are written as unordered sets, that is: $\{u, v\} \in E$ instead of $(u, v) \in E$ and $(v, u) \in E$. Graphs that only contain undirected edges are also called undirected graphs.

## 2.2 Notations

Further, for simplicity, we define the following notations:

- $b$ - the word size.
- $N_G(v) = \{w : \{v, w\} \in E\}$ - the set of neighbors of vertex $v$.
- $\mathcal{D}_G(u, v)$ - the distance between $u$ and $v$ inside the graph $G$.
- $L(v) = \{(w, \mathcal{D}_G(v, w)) : w \in S \subseteq V\}$ - label of $v$.

## 2.3 The problem

Given an undirected graph $G$, construct an index that efficiently answers distance queries. The construction should scale well in terms of multi-threading and the resulting index should be similar to the one form [AIY13].

## 2.4 Breadth-first search (BFS)

As the utilization of breadth-first-searches is extensively done in our thesis, a short overview is given here.

BFS is an exploration algorithm that is commonly used within undirected graphs. One of its mayor usages lies in the distance computation from a chosen vertex to all other vertices in the graph – therefore solving the single-source shortest path problem. Throughout its execution a set of vertices is maintained, called the *frontier*. After the BFS has done $d$ steps, the frontier contains the vertices whose shortest path to the chosen vertex has the distance $d$. In order to avoid visiting some vertices multiple times, an array is used for marking the visited vertices.

**Algorithm 1** Breadth-first search (BFS) on $G = (V, E)$ from vertex $r \in V$

```
 1: procedure BFS(G, v)
 2:     Q ← an empty queue                          ▷ Initialization of the frontier.
 3:     P[v] ← ∞ for all v ∈ V                      ▷ Initialization of the distance array.
 4:     P[r] ← 0
 5:     Enqueue r onto Q
 6:     while Q is not empty do
 7:         Dequeue v from Q
 8:         for all u ∈ N_G(v) do
 9:             if P[u] = ∞ then
10:                 P[u] ← P[v] + 1
11:                 Enqueue r onto Q
```

## 2.5 Labeling

The index resulting from our construction is composed of a list of labels. That is, for each vertex $v \in V$, it contains a label $L(v)$. The process of creating these labels is referred as *labeling* the vertices. Each label contains a list of pairs $(u, dist_G(v, u))$ and with them the distance can be found by:
$\mathcal{D}_L(v, w) = min\{d_1 + d_2 : (r, d_1) \in L(v), (r, d_2) \in L(w)\}$

If there is no common vertex for $L(v)$ and $L(w)$, $w$ is unreachable from $v$ ($\mathcal{D}_L(v, w) = \infty$).

The bit-parallel labels, that are introduced in the upcoming sections, compute the distance in a different way. However, this computation takes essentially the same approach as the computation of the normal labels does.

# 3. Related work

The next two sections describe the main groups of approaches previously considered for this problem: *exact* and *approximate*. Due to the close relation of [AIY13] to our approach, we present a detailed discussion at this point.

## 3.1 Exact methods

In the group of exact methods, most prevalent are methods based on 2-hop covers [CHKZ02], possible examples include: decomposition of the graph into a tree [Wei10], highway-centric labeling [JRXL12] (creating a spanning tree and using it as a "highway" for answering queries), and hierarchical hub-labeling [ADGW12].

## 3.2 Approximate methods

For some applications, approximative answers to the queries are sufficient. This setting can be used to develop methods that are more time-efficient. Landmark-labeling is the most prominent among the approximation approaches.
In landmark-labeling, a subset of vertices is selected and the distances from them to all other vertices is computed and stored as their labels. More formally, for a selected subset of vertices $S \subseteq V$, the distances $\mathcal{D}_G(l, w), l \in S, w \in V$ are computed. Since it is possible that no shortest path between two vertices goes through any of the selected roots, distance queries on these labels are not necessarily exact. The initial selection of the vertex-set $S \subseteq V$ has severe impact on the approximative quality; centric vertices represent in most scenarios a good choice – more discussion about this is done in 4.1.

## 3.3 Pruned landmark labeling[AIY13]

The pruned landmark labeling (PLL) combines the advantages of several previous methods to approach this problem. Its key idea is similar to approximate landmark-labeling. Central vertices are prioritized and used to reduce the size of all further searches. This is done, as seen in more detail in 3.3.2, with *pruning*.

Pruned landmark labeling consists of two phases. In the first phase, pruning is ignored and several landmarks are processed by using bit-parallelism. After a number of vertices are processed this way, the algorithm switches to the second phase. The key-element of

this phase is *pruning* – during its labeling of the vertices, it does not traverse those vertices whose distance can already be computed from the existing labels.

The following sections introduce the phases in more detail, with 3.3.2 and 3.3.3 corresponding to the second and first phase, respectively (starting with the second phase leads to a better understanding of the first one).

### 3.3.1 Landmark labeling

For answering shortest-path distance queries between two vertices, precomputed information can be used for a better time-performance. Suppose that $b$ lies on a shortest path between $a$ and $c$, and that we know the distance between $a$ and $b$, $d_1$ and the distance between $b$ and $c$, $d_2$. With this, one can infer the distance between $a$ and $c$:
$\mathcal{D}_G(a, c) = \mathcal{D}_G(a, b) + \mathcal{D}_G(b, c) = d_1 + d_2$.

This is the core idea of landmark labeling. Landmarks are selected and the distance between them and all other vertices is computed. For all pairs of vertices on whose shortest path a landmark was selected, the shortest path can be computed. In order to achieve exact answers, all vertices have to be processed as roots – and this is also done here. The distance between $x$ and $y$ can be then computed by $\mathcal{D}_G(x, y) = \min_{r \in L}\{\mathcal{D}_G(x, r) + \mathcal{D}_G(r, y)\}$, where L is the set of selected landmarks. That is, for every pair of vertices $a$ and $c$, it is possible to find a landmark-vertex $b$ that lies on one of the shortest paths between $a$ and $c$.

**Example:**
Consider figure 3.1. We see that the vertex $r$ covers all shortest paths between the vertex sets $\{a, b\}$ and $\{c, d, e\}$ (as it is the only vertex connecting both sets). In order to compute the distance between, for example, $a$ and $e$, it is enough to add the corresponding distances to $r$ together, that is: $\mathcal{D}_G(a, e) = \mathcal{D}_G(a, r) + \mathcal{D}_G(r, e) = 3$.

### 3.3.2 Pruned landmark labeling

The next problem lies in reducing the number of label-entries that are being added to the label-lists of all vertices in the graph. In case we have to store the distance to every other vertex in $G$, we end up requiring $O(|V|^2)$ memory space. The introduction of pruning prevents this.

Assuming all vertices in $G$ have been labeled with some landmarks $L \subseteq G$, it can be seen that for the next vertex that is being selected as a landmark, not all vertices have to contain the distance to that landmark.

Let $r \in V$ be the next landmark we want to process, $v \in V$ a vertex to whose label we want to add $r$, and $d_{BFS}$ be the distance from $r$ to $v$ (computed by the $BFS$). If no vertices have already been pruned, two cases can happen: $\mathcal{D}_L(r, v) \leq d_{BFS}$ and $\mathcal{D}_L(r, v) > d_{BFS}$. When $\mathcal{D}_L(r, v) > d_{BFS}$ holds, we do not prune $v$ – the solution of the query is improving by adding $(r, d_{BFS})$ to the label $L(v)$. For the other case, $\mathcal{D}_L(r, v) \leq d_{BFS}$, however, pruning is feasible – no distance can be shortened by continuing this search.

**Example:**
Consider again figure 3.1. Let $c$ be the next root we add to our landmarks. As BFS is executed, at some point the frontier-state $\{r, d, e\}$ is reached. At this stage, pruning is possible. The exact distance to vertex $r$ is already known due to the label-entries that were produced while taking vertex $r$ into the set of landmarks. Therefore, $r$ can be pruned, and, consequently, vertices $a$ and $b$ do not get additionally processed.

It is obvious that the order in which the vertices are processed influences the number of pruned vertices. The vertices being processed earlier influence the pruning-quality for the

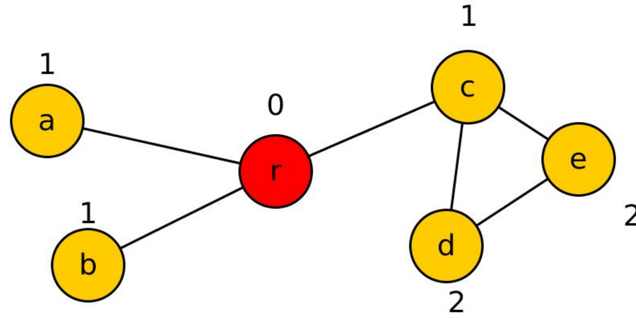


Figure 3.1: Graphical demonstration of the pruned landmark labeling method. The values over the vertices represent the distances to the root-vertex.

upcoming searches. Good choices include central vertices: high-degree vertices or the ones through which a large number of shortest paths go. The approach taken here is sorting the vertices according to their degrees – from the highest to the lowest.

Algorithm 2 illustrated the given approach.

**Algorithm 2** Pruned landmark labeling

1: **procedure** PRUNED-LANDMARK-LABELING($G$)
2: $G \leftarrow$ RearrangeVertices($G$) ▷ Sort vertices according to their degrees
3: $L[v] \leftarrow \emptyset$ for all $v \in V$
4: **for all** $v \in V$ **do**
5: Pruned-Landmark-Labeling-BFS($G, L, v$)
6: **return** L
7:
8: **procedure** PRUNED-LANDMARK-LABELING-BFS($G, L, r$)
9: $P[v] \leftarrow \infty$ for all $v \in V$ ▷ Holds the distance from the root to $v$.
10: $Q \leftarrow$ an empty queue
11: $P[r] \leftarrow 0$
12: Enqueue $r$ onto $Q$
13: **while** $Q$ is not empty **do**
14: Dequeue $v$ from $Q$
15: **if** $P[v] \geq \mathcal{D}_L(r, v)$ **then** ▷ Check if $v$ should be pruned
16: **continue**
17: $L[v] \leftarrow L[v] \cup \{(r, P[v])\}$
18: **for all** $u \in N_G(v)$ **do**
19: **if** $P[u] = \infty$ **then**
20: $P[u] \leftarrow P[v] + 1$
21: Enqueue $u$ onto $Q$

### 3.3.3 Bit parallel labeling

Not many prunes are made in the first phase of creating the index. Therefore, we could avoid pruning at all for the first few searches. Avoiding them gives us the possibility to improve our approach in another way. For this purpose, we use another kind of labels. They allows us to process several landmarks with the same overhead as processing only one. The key idea for this approach is bit-parallelism. We take a root $r \in V$ and up to 64 vertices (or $b$ vertices if the word size is different) adjacent to $r$. With these vertices,

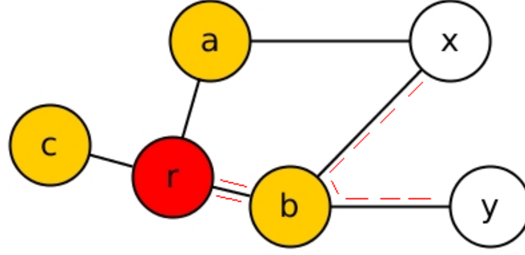


Figure 3.2: Graphical demonstration of approximating the distance between $x$ and $y$ by using the bit-parallel labels that are created from the root $r$. Computation of the distance between $x$ and $y$ by using the bit-parallel labels: $\mathcal{D}_S(x,y) = (\mathcal{D}_G(x,r)-1)+(\mathcal{D}_G(y,r)-1)$

we create the ordered set $S = (v_0, ..., v_{63})$. For every vertex $v$ in the graph, we compute the label-entry $(r, \mathcal{D}_G(v,r), S_r^{-1}(v), S_r^0(v), S_r^1(v))$. The three elements $S_r^{-1}(v), S_r^0(v)$, and $S_r^1(v)$ represent sets. We partition vertices from $S$ into them according to their relative distance to $v$; compared to the distance from $r$ to $v$. This gives us a fast way to compute the distance from $v$ to any of the vertices in $\{r\} \cup S$.
Internally, $S_r^i(v)$ is a bit-mask and the $j-th$ bit is set to 1, if the distance from $v$ to $v_j$ changes by $i$ compared to the distance the distance from $v$ to $r$. That is: $\mathcal{D}_G(v, v_j) = \mathcal{D}_G(v,r) + i$.

As the bitwise-OR of $S_r^{-1}(v), S_r^0(v)$, and $S_r^1(v)$ covers all bits, knowing $S_r^{-1}(v)$ and $S_r^0(v)$ is enough to infer $S_r^1(v)$.

For a bit-parallel root $r \in V$ and any pair of vertices $x, y \in V$, let $d_r = \mathcal{D}_G(x,r) + \mathcal{D}_G(r,y)$. In order to know the distance of the shortest path between $x$ and $y$ **that is going through** $S$, from now on defined as $\mathcal{D}_S(x,y)$, it is sufficient to look at the bit-parallel-label-entries $(r, S_r^{-1}(x), S_r^0(x))$ and $(r, S_r^{-1}(y), S_r^0(y))$. Three cases have to be considered. Note: $|$ is the bitwise-OR operator. If

- $S_r^{-1}(x) \mid S_r^{-1}(y) \neq 0$, then $\mathcal{D}_S(x,y) = d_r - 2$,
- $S_r^{-1}(x) \mid S_r^0(y) \neq 0$ or $S_r^0(x) \mid S_r^{-1}(y) \neq 0$, then $\mathcal{D}_S(x,y) = d_r - 1$,
- otherwise $\mathcal{D}_S(x,y) = d_r$.

**Example:**
In figure 3.2 a simple example is given for approximating the distance between $x$ and $y$ through the bit-parallel root $r$ and its adjacent vertices $a, b, c$. We see that the distance from $x$ and $y$ to $r$ equals 2, and that their distance to $b$ changes by $-1$ compared to the distance to the root (therefore is $b$ in their $S_r^{-1}$ set). The approximative distance, then, is: $\mathcal{D}_S(x,y) = \mathcal{D}_S(x,b) + \mathcal{D}_S(y,b) = (\mathcal{D}_G(x,r) - 1) + (\mathcal{D}_G(y,r) - 1)$. As the distance to the root is stored inside the labels, $\mathcal{D}_G(x,r)$ and $\mathcal{D}_G(y,r)$ are already known and, therefore, the distance of the shortest path between $x$ and $y$ (through $S$!) is computable.

*Algorithm 3* shows the construction of the labels in $O(|V| + |E|)$ for a selected root $r$ and a set of up to 64 adjacent vertices, $S_r$.

### 3.3.4 Results

In the domain of complex networks, PLL is more efficient than other approaches by a large margin. Table 3.3, taken from [AIY13], showcases that performance difference. Hierarchical Hub Labeling represents the work done from [ADGW12] and Tree Decomposition is an improved version of [Wei10] ([ASK12]).

---

**Algorithm 3** Bit-parallel BFS from $r \in V$ and $k \in N_0$ $S_r \subseteq N_G(r)$ [AIY13]

---

1: **procedure** BP-BFS$(G, r, S_r)$
2: $(P[v], S_r^{-1}[v], S_r^0[v]) \leftarrow (\infty, \emptyset, \emptyset)$ for all $v \in V$
3: $(P[r], S_r^{-1}[r], S_r^0[r]) \leftarrow (0, \emptyset, \emptyset)$
4: $(P[v], S_r^{-1}[v], S_r^0[v]) \leftarrow (1, v, \emptyset)$ for all $v \in S_r$
5: $Q_0, Q_1 \leftarrow$ an empty queue
6: Enqueue $r$ onto $Q_0$
7: Enqueue $v$ onto $Q_1$ for all $v \in S_r$
8: **while** $Q_0$ is not empty **do**
9: $E_0 \leftarrow \emptyset$ and $E_1 \leftarrow \emptyset$
10: **while** $Q_0$ is not empty **do**
11: Dequeue $v$ from $Q_0$
12: **for all** $u \in N_G(v)$ **do**
13: **if** $P[u] = \infty \vee P[u] = P[v] + 1$ **then**
14: $E_1 \leftarrow E_1 \cup \{(v, u)\}$
15: **if** $P[u] = \infty$ **then**
16: $P[u] \leftarrow P[v] + 1$
17: Enqueue $u$ onto $Q_1$
18: **else if** $P[u] = P[v]$ **then**
19: $E_0 \leftarrow E_0 \cup \{(v, u)\}$
20: **for all** $(v, u) \in E_0$ **do**
21: $S_r^0[u] \leftarrow S_r^0[u] \cup S_r^{-1}[v]$
22: **for all** $(v, u) \in E_1$ **do**
23: $S_r^{-1}[u] \leftarrow S_r^{-1}[u] \cup S_r^{-1}[v]$
24: $S_r^0[u] \leftarrow S_r^0[u] \cup S_r^0[v]$
25: $Q_0 \leftarrow Q_1$ and $Q_1 \leftarrow$ an empty queue
26: **return** $(P, S_r^{-1}, S_r^0)$

---

Even for networks with several millions of vertices, indexing is fast, queries remain quick, and memory consumption stays in an acceptable range.

To the best of our knowledge, no thread-parallel index construction for the chosen problem and graph-type exists. We decided to research the parallelism within PLL as it is one of the main methods of constructing an index for complex networks and also shows potential for good parallelism.

| Dataset | Pruned Landmark Labeling | | | | Hierarchical Hub Labeling | | | | Tree Decomposition | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | IT | IS | QT | LN | IT | IS | QT | LN | IT | IS | QT |
| Gnutella | 54 s | 209 MB | 5.2 $\mu$s | 437+16 | 245 s | 380 MB | 11 $\mu$s | 1,275 | 209 s | 68 MB | 19 $\mu$s |
| Epinions | 1.7 s | 32 MB | 0.5 $\mu$s | 7+16 | 495 s | 93 MB | 2.2 $\mu$s | 256 | 128 s | 42 MB | 11 $\mu$s |
| Slashdot | 6.0 s | 48 MB | 0.8 $\mu$s | 14+16 | 670 s | 182 MB | 3.9 $\mu$s | 464 | 343 s | 83 MB | 12 $\mu$s |
| NotreDame | 4.5 s | 138 MB | 0.5 $\mu$s | 29+16 | 10,256 s | 64 MB | 0.4 $\mu$s | 41 | 243 s | 120 MB | 39 $\mu$s |
| WikiTalk | 61 s | 1.0 GB | 0.6 $\mu$s | 9+16 | DNF | - | - | - | 2,459 s | 416 MB | 1.8 $\mu$s |
| Skitter | 359 s | 2.7 GB | 2.3 $\mu$s | 123+64 | DNF | - | - | - | DNF | - | - |
| Indo | 173 s | 2.3 GB | 1.6 $\mu$s | 133+64 | DNF | - | - | - | DNF | - | - |
| MetroSec | 108 s | 2.5 GB | 0.7 $\mu$s | 19+64 | DNF | - | - | - | DNF | - | - |
| Flickr | 866 s | 4.0 GB | 2.6 $\mu$s | 247+64 | DNF | - | - | - | DNF | - | - |
| Hollywood | 15,164 s | 12 GB | 15.6 $\mu$s | 2,098+64 | DNF | - | - | - | DNF | - | - |
| Indochina | 6,068 s | 22 GB | 4.1 $\mu$s | 415+64 | DNF | - | - | - | DNF | - | - |

Figure 3.3: Performance comparison between PLL and previous methods for the real-world data sets. IT denotes indexing time, IS denotes index size, QT denotes query time, and LN denotes average label size for each vertex. DNF means it did not finish in one day or run out of memory. Taken from [AIY13]. **Note:** The results are taken with a different machine. Details about the corresponding machines can be found in the paper from where the table originates.

# 4. Parallel pruned landmark labeling

In its core, the algorithm in this thesis is based on PLL from [AIY13]. The algorithm uses differentiation between two phases and the two kinds of labels that come with that: the *normal* and *bit-parallel* labels. The main structure of the approach is depicted in *Algorithm 4*.

The first step is to rearrange all vertices. Processing specific vertices first impacts the performance greatly as they influence the pruning throughout the second phase.

**Algorithm 4** Parallel pruned landmark labeling of $G$, taking $k$ vertices as roots for the bit-parallel labeling

```
1: procedure PPLL(G, k)
2:     G ← RearrangeVertices(G)          ▷ sorts vertices in a specific order
3:     (L_BP, L) ← (∅, ∅)
4:
5:     L_BP ← L_BP ∪ Phase_I(G, k)
6:     L ← L ∪ Phase_II(G, L_BP)
7:     return (L_BP, L)
```

Having reordered the vertices, we proceed to create the labels for all vertices. For the first $k$ searches, we run the first phase. They do not include any pruning, but do allow us to process considerably more vertices. Lastly, in the second phase, we create the normal labels by using the bit-parallel and normal labels to prune the spanning of a BFS. These pruning mechanisms, which are used within this phase, keep the memory consumption of the index and the time-performance low.

The outlined structure until now is identical to the one used within PLL. Modifications within individual sections make our approach different and enables the utilization of thread-parallelism.

## 4.1 Vertex order

As already mentioned, the order of processing the vertices greatly influences the pruning quality. Experiments and previous work [PBCG09][AIY13] has shown that vertices lying in more ”central” regions represent good choices. In these ”central” regions, the selected vertices cover a great number of shortest paths between all the pairs of vertices. Selecting

vertices at random intuitively does not lead to a good performance, as the selected vertices are not going to necessarily have large number of shortest paths going through them.

The order we use in our method is based on the degree property of a vertex – we sort the vertices descending by their degrees. Another feasible approach is to sort the vertices by their approximate *clossenes centrality*. In [AIY13] both orders have been tested, and the conclusion is that the order according to the degrees performs marginally better than the one based on the approximate clossenes. This has also been verified from our side with experimental tests.

No parallelism has been integrated in this section. The reordering can easily be done with an $O(|V|log(|V|) + |E|)$ algorithm: sort the vertices descending by their degrees, create a mapping of the vertices, and then update the structure that is managing the adjacency properties. Implementing parallel sorting at this point does not lead to any performance improvements because it consumes only a small part of the computation time.

## 4.2 Phase I

**Algorithm 5** Phase 1

1: **procedure** PHASE$_I(G, k)$
2: **for** $i := 1$ to $k$ **in parallel do**
3: $id_{mem} \leftarrow$ UnusedMemoryId() ▷ Getting a free index for memory access.
4: $r \leftarrow$ NextRoot()
5: $M \leftarrow$ up to $b$ vertices from $N_G(r)$
6: $L_{BP}[i] \leftarrow$BP-BFS$(G, r, M)$
7: unset$(id_{mem})$ ▷ Reinitializing the used memory for the next searches.
8: **return** $L_{BP}$

Intuitively, not many vertices get pruned within the first searches. The reason to this lies within the fact that in order to prune a vertex, specific entries are needed within the labels. These entries are more probable to exist, the more searches we have accomplished. Because of that, ignoring the pruning in the beginning motivates us to approach the labeling with a bit-parallel search; which is not well suited alongside with the pruning techniques.

In the bit-parallel labeling, we execute $k$ searches in order to accomplish the needed labeling. For each search we select a vertex-root $r \in V$ and up to $b$ vertices from the set $N_G(r)$. The algorithm then takes the selected vertices and labels with them all vertices inside the graph in one BFS-like search.

**Parallelism**
The first impression one might get by looking at the way the bit-parallel roots are computed, is that the update over the labels is difficult to parallelize. However, a closer look shows that they are easily made independent by assigning an unique index to each labeling (as already done by using $r$ within algorithm 3). This finding immediately motivates us to consider executing multiple searches in parallel.

In order to achieve thread-safety, an unique id, $id_{mem}$, is given to each thread. This ensures that the thread can access memory that is specifically assigned to it. Assigning memory anew every time a thread starts slows performance down. This effect is induced by the synchronisation the threads have to accomplish on every allocation. Also, adding such memory-management-layer enables us to easily reinitialize the memory for the next searches.

For finding a root that has not been processed, an integer $root_{last}$ pointing to the last selected root is used. When a new, unprocessed, root $r$ is needed, an incremental search is executed from $root_{last}$ to $|V|$. When found, $root_{last}$ is updated to $r$.

The section for finding $root_{last}$ and $id_{mem}$ are protected within their own locks. Exploring $N_G(r)$ can be also easily made thread-safe by either protecting it wholly under a lock or adding a lock for each individual vertex – the choice has no effects on the performance. It is also interesting to note that a race condition exists here. Two roots might share an adjacent vertex. Depending which root takes it first, different bit-parallel labels can be made. As these changes are minor, the performance does not get affected from this.

Updating the index with the new label-entries does not require any further mechanisms for multi-threading. By using the unique index that is assigned to each labeling in the beginning, no access has to be done between the individual searches – labels are easily accessed independently of each other.

It is important to note that this phase is executed until $k$ roots have been processed. The selection of $k$ influences the efficiency of this and the upcoming phases. Higher values of $k$ make the first phase slower and the upcoming one faster, and vice-versa. In [AIY13], values $\leq 64$ were sufficient, but as we have acquired high parallelism in this phase, higher values for $k$ can be also taken into consideration.

The main structure of this phase is very similar to that of algorithm 3. Algorithm 5 depicts the steps needed to synchronize the constructed labels. The only line that does not require any lock is 6.

### 4.2.1 Bit-parallel search from any k vertices

An important area that we investigated was if there is an efficient way to generalise the bit-parallel labeling. We checked if there is a way to make the labeling from any $b$ vertices possible – removing the requirement that the vertices share an adjacent vertex.

Our approach to this was to select a set of $b$ vertices and assign to each of them an index between 0 and $b-1$ (used to identify them in a bit-mask). Then, a search is executed that is similar to a multi-source BFS. The idea is to merge the search from several roots into one and label each reached vertex with the merged set in one step. In order to do that, to each element in the frontier a bit-mask is added.

After the search expanded its frontier $d$ times, the $i$-th index in the bit-mask is set to 1 if the $i$-th selected root reaches the vertex in $d$ steps. In order to know the set of vertices that have already visited a vertex in a search, an additional bit-mask is attached to each vertex in the graph; which are used to filter out the vertices that have already visited it. All necessary operations can be done with simple bitwise operations.

In order to compute the distance between $x \in V$ and $y \in V$, we assign an integer to each label-entry, which identifies the chosen set of $b$ roots. Then, we compare the label-entries between $x$ and $y$ with the same identifier. Let $(id, mask_x(i), \mathcal{D}_x(i))$ be the label-entry for $x$ and $(id, mask_y(i), \mathcal{D}_y(i))$ for $y$ (index $i$ is used to distinguish between the different entries with the same identifier). The distance between $x$ and $y$ is then computable by: $\min_{id,i}\{D_x(i) + D_y(i) : mask_x(i) \;\&\; mask_y(j) \neq 0\}$; where & represents the bitwise-AND operator. Some important optimizations can be done here: We kept the label-entries sorted by $id$. This enabled us to easily select the set of entries with a specific identifier in $x$'s and $y$'s labels. Experiments have also shown that the number of entries a vertex owns for a fixed $id$ is very small. For **Epinions** that is 2.6, for **Skitter** 3.5 (5.1). However, this still imposes a bigger label size than the one that is achieved in the bit-parallel labeling from [AIY13]. This also increases the time that is needed to compute the distance between

two vertices – slowing down the pruning in the second phase. Unfortunately, this leads to an overall worse performance.

## 4.3 Phase 2

As the second phase produces labels that are different from the ones the first phase produces, the pruning phase is also split into two phases. First, a pruning-check is made by using the bit-parallel labels. If the check is unsuccessful, another pruning-check is executed on the normal labels. For a vertex surviving both checks, a new label-entry $(root, d_{BFS})$ is added to its list.

**Parallelism**
Parallelism in this phase is accomplished by executing multiple searches at the same time. Selecting the next root and the memory space that is required for this phase is identical to the one from the first phase. The section that requires more attention lies within the introduced pruning. As already stated, this phase produces different kind of labels compared to the first phase, called *normal labels*. While new entries are being added to the normal labels, care has to be taken with the sections that might also access the labels. To make this execution safe, an array of spin-locks is used. Only one segment within the algorithm is allowed to access a vertex's label at a time. This also eliminates the possibility of adding unnecessary entries to the labels, as the section where the labels are computed and used for pruning belong within the same lock.

As all searches are done in parallel, race conditions occur often during the labeling process. The reason to this lies within the section where pruning and insertions on the normal labels are done. A thread might outrun another thread and insert its label before another does. Therefore, the order of the label-entries may be mixed up. This also means that the received order is not necessarily increasing by the vertex-id. This property is crucial for integrating a cache-efficient query procedure (4.4).

**Creating ordered labels:**
To acquire an ordered list for all labels at the end, a simple sorting algorithm can be used. We decided to use insertion-sort. Per vertex, the size of the normal label is generally small and fits in cache. Also, the displacement of each entry is, intuitively, not far from its original place, as it is only based on the race conditions. This also implies that the changes induced by the race conditions are overall small. While repairing the order, vertices are processed in parallel.

**Algorithm 6** Phase 2

```
1: procedure PHASE_II(G, L_BP)
2:     G ← RearrangeVertices(G)        ▷ sort vertices according to their degrees
3:     L[v] ← ∅ for all v ∈ V
4:     for all not processed v ∈ V in parallel do
5:         id_mem ← UnusedMemoryId()
6:         Phase_II-BFS(v, G, L_BP, L))
7:         unset(id_mem)
8:     fixOrder(L)
9:     return L
10:
11: procedure PHASE_II-BFS(r, G, L_BP, L)
12:     P[v] ← ∞ for all v ∈ V
13:     Q ← an empty queue
14:     P[r] ← 0
15:     Enqueue r onto Q
16:     while Q is not empty do
17:         Q_nxt ← an empty queue
18:
19:         for all v ∈ Q do
20:             if P[v] ≥ D_{L_BP}(r, v) then
21:                 remove v from Q
22:             else
23:                 lock.acquire()
24:                 if P[v] ≥ D_L(r, v) then
25:                     remove v from Q
26:                 else
27:                     L[v] ← L[v] ∪ {(r, P[v])}
28:                 lock.release()
29:
30:         for all v ∈ Q do
31:             for all u ∈ N_G(v) do
32:                 if P[u] = ∞ then
33:                     P[u] ← P[v] + 1
34:                     Enqueue u onto Q_nxt
35:         Q ← Q_nxt
```

## 4.4 Query

One can take several approaches to find the distance between two pairs of vertices by using the labels. Additionally, as two kinds of labels exist, two phases exist. Algorithm 7 and 8 illustrate how to estimate the distance with the bit-parallel and normal labels, respectively.

Another approach to answer the queries is to search for a matching pair in a merge-like fashion. It has to be ensured, however, that the labels have the same relative order between all vertices. This approach, compared to algorithm 8, is cache efficient, and our tests have shown that this improves the performance of a query by several factors (in some cases over 20).

Interesting to note: Algorithm 8 is not only useful for the queries, but also for the pruning mechanism within the second phase. The array $D$ can be initialized in the beginning and used throughout the search – distance computation becomes significantly faster.

**Algorithm 7** Estimating the distance between $u, v \in V$ with the bit-parallel labels $L_{BP}$

1: **procedure** QueryBP($L_{BP}, v, w$)
2: $res \leftarrow \infty$
3: **for** $i = 1$ **to** $k$ **do**
4: $d \leftarrow L_{BP}[i].P[v] + L_{BP}[i].P[w]$
5: **if** $L_{BP}[i].S_r^{-1}[v]$ & $L_{BP}[i].S_r^{-1}[w]$ **then**
6: $d \leftarrow d - 2$
7: **else if** $L_{BP}[i].S_r^{-1}[v]$ & $L_{BP}[i].S_r^{0}[w]$ **or** $L_{BP}[i].S_r^{0}[v]$ & $L_{BP}[i].S_r^{-1}[w]$ **then**
8: $d \leftarrow d - 1$
9: **if** $res > d$ **then**
10: $res \leftarrow d$
11: **return** $res$

**Algorithm 8** Estimating the distance between $u, v \in V$ with the normal labels $L$

1: $D[v] \leftarrow \infty$ for all $v \in V$
2: **procedure** Query($L, v, w$)
3: **for all** $(r, d) \in L[v]$ **do**
4: $D[r] \leftarrow d$
5: $res \leftarrow \infty$
6: **for all** $(r, d) \in L[w]$ **do**
7: **if** $D[r] + d < res$ **then**
8: $res \leftarrow D[r] + d$
9: **for all** $(r, d) \in L[v]$ **do**
10: $D[r] \leftarrow \infty$
11: **return** res

## 4.5 Implementation

The implementation requires attention within several segments in order to acquire an efficient algorithm. Most of the upcoming sections can be also found within [AIY13]. The same is true for the variations of our implementation within section 4.6.

### 4.5.1 Initialization

While executing the searches inside the second phase, it is possible that some searches reach only a small amount of vertices. This fact can be used in order to speed up the array initialization for the next search – we keep track of the changed fields and revert them back to the initial value after the search is finished.

### 4.5.2 Pruning of the frontier

In order to speed up the pruning section in the second phase, three things can be done.

- Looking at algorithm 6, we see that on line 20 the distance is always computed to the root $r$. Algorithm 8 shows that the changes on $D$ are sufficient to be made at the beginning and in the end of each search. With this, we spend $|L[r]|$ steps for each search, instead of $cnt_{nodes} \cdot |L[r]|$ steps, where $cnt_{nodes}$ is the number of reached vertices in the search. This alleviates a high amount of unneeded work.
- If the vertex that we compare to the current root was already processed (taken as a landmark), $\mathcal{D}_L(r, v)$ equals to zero. Therefore, the overhead that is given with that query can be avoided by simply checking if the vertex was already selected as a landmark.
- We can also use the fact that we compute $\mathcal{D}_L(r, v)$ by constantly updating an upper bound, $\delta$ (for example, $res$ in 7 and 8). If we find a $\delta$, such that $P[v] \geq \delta \geq \mathcal{D}_L(r, v)$ holds, the computation of $\mathcal{D}_L(r, v)$ can be interrupted.

### 4.5.3 Vertex order induced cache-efficiency

As already stated, an ordered set of all the vertices is created. In our case that order requires the vertices to be sorted from the one with the highest degree to one with the lowest.

There are two straightforward ways to utilize this in our construction. We can either relabel all vertices (giving the vertex with the highest degree the lowest index) and update their adjacency lists accordingly or leave the vertices as they are and simply execute the searches in order.

Using the first method represents a better choice. Processing the vertices increasingly by their index utilizes the spatial locality in the memory efficiently. A strong support for this argument is the content that the labels receive – roots are ordered increasingly by their index. Making the distance computation with the labels more cache-efficient.

### 4.5.4 Locks

The usage of spin-locks for particular critical sections made the performance of our implementation better. This specifically holds for the sections in the second phase, where parallel access is done on the index-structure.

For the other sections, like the assignment of the memory-id, no significant difference has been noticed – mutexes and spin-locks can be used.

## 4.6 Variations

### 4.6.1 Lower memory consumption

As the bit-parallel index construction allocates $O(|V| + |E|)$ memory per thread, we also developed a reduced memory version with $O(|V|)$ memory consumption. To achieve this, we introduced a simple modification on Algorithm 3.

The idea is to go twice over the frontier instead of once. This way we are able to avoid the computation of $E_0$ and $E_1$. Instead of adding entries to $E_0$, we compute the changes done on $S_r^0[u]$ right away. Then, we iterate again over the frontier to process the changes that are made with $E_1$.

**Algorithm 9** Memory-reduced version of the bit-parallel labeling

1: **while** $Q_0$ is not empty **do**
2: $//E_0 \leftarrow \emptyset$ and $E_1 \leftarrow \emptyset$
3: **for all** $v \in Q_0$ **do**
4: **for all** $u \in N_G(v)$ **do**
5: **if** $P[u] = \infty \vee P[u] = P[v] + 1$ **then**
6: $// \ E_1 \leftarrow E_1 \cup \{(v, u)\}$
7: **if** $P[u] = \infty$ **then**
8: $P[u] \leftarrow P[v] + 1$
9: Enqueue $u$ onto $Q_1$
10: **else if** $P[u] = P[v]$ **then**
11: $S_r^0[u] \leftarrow S_r^0[u] \cup S_r^{-1}[v]$
12: $// \ E_0 \leftarrow E_0 \cup \{(v, u)\}$
13: **while** $Q_0$ is not empty **do**
14: Dequeue $v$ from $Q_0$
15: **for all** $u \in N_G(v)$ **do**
16: **if** $P[u] = P[v] + 1$ **then**
17: $S_r^{-1}[u] \leftarrow S_r^{-1}[u] \cup S_r^{-1}[v]$
18: $S_r^0[u] \leftarrow S_r^0[u] \cup S_r^0[v]$
19: $Q_0 \leftarrow Q_1$ and $Q_1 \leftarrow$ an empty queue

Algorithm 9 illustrates the modifications. Sections that did not require any attention have been left out.

The reason we did not introduce this approach right away is the worse time performance it produces.

### 4.6.2 Directed edges

In order to create an equivalent index for graphs with directed edges, a small change suffices. We split up all labels (bit-parallel and normal) into two groups: $L_{IN}$ and $L_{OUT}$. We create $L_{IN}$ by executing the same steps as for undirected graphs, while taking only ingoing edges in consideration. The same has to be done for $L_{OUT}$, but with the outgoing edges.

### 4.6.3 Weighted edges

Although the bit-parallel labeling is not usable for weighted edges, it is still possible to modify this algorithm for weighted edges. Instead of running a BFS, Dijkstra's algorithm can be run – the pruning mechanism remains the same. The same kind of parallelism can be used by executing multiple searches at the same time.

# 5. Experimental evaluation

## 5.1 Environment

The implementation of the method we presented in this thesis has been implemented in `C++`. We compiled it using the Intel `C++` Compiler, icpc/icc (version 14.0.2), with the -O3 flag.

**Hardware:** The machine that was used for evaluating the presented implementation was a NUMA Intel machine. It consisted of four Xeon E7-8870 with 2.40GHz per core and a 1066Mhz bus. This gave us 40 physical and 80 logical cores (with hyper-threading). The total main memory that was available for each test-run was 256GiB, where 64GiB belonged to each processor.

## 5.2 Datasets

We evaluated the presented approach on a set of various real-life datasets, modelling different complex structures. Special focus within the analysis was set on the behaviour within a multi-threaded environment. In order to achieve this, we analyzed the behaviour and the speed-ups resulting from our implementation.

All datasets used for evaluating our implementation are given in table 5.1.

| Dataset | $\|V\|$ | $\|E\|$ |
|---|---:|---:|
| Gnutella | 62 562 | 147 878 |
| Slashdot | 70 069 | 358 647 |
| Epinions | 75 888 | 405 740 |
| Google | 916 428 | 4 322 051 |
| WikiTalk | 2 394 385 | 4 659 565 |
| Skitter | 1 696 416 | 11 095 298 |
| Flickr | 1 715 256 | 15 555 042 |
| eu-2005 | 862 665 | 16 639 895 |
| Hollywood | 1 139 906 | 57 515 616 |
| Indochina | 7 414 866 | 153 487 303 |

Table 5.1: Used datasets

**Datasets - description**

**Gnutella:** A peer-to-peer network for file-sharing. Snapshot from August 2002 [RFI02].

**Slashdot:** Friend/foe network from Slashdot. Snapshot from Februrary 2009 [LLDM08].

**Epinions:** A who-trust-whom online social network of a general consumer review site Epinions.com. Members of the site can decide whether to "trust" each other [RAD03].

**Google:** Web pages represented with vertices and a hyperlink between them with an edge. Dataset was released inside a Google Programming Contest [Goo02], [LLDM08].

**WikiTalk:** Vertices represent members. There is an edge from $x$ to $y$ if user $x$ has edited at least once a talk-page of user $y$ [LHK10b][LHK10a].

**Skitter:** An internet topology graph created from traceroutes [LKF05].

**Flickr:** Photo-sharing website (www.flickr.com) [MMG+07].

**eu-2005:** A small crawl of the .eu domain [BV04],[BRSV10],[BCSV04].

**Hollywood:** Actors are represented with vertices and there is an edge between them if they have been in the same movie (at least once) [BV04],[BRSV10],[BCSV04].

**Indochina:** Web graph network of the Indochina region [BV04], [BRSV10], [BCSV04].

Note: We treated directed edges as undirected in the datasets with directed edges.

## 5.3 Performance of PLL

The table 5.2 shows the performance of PLL. The number of bit-parallel roots has been selected by considering the resources (time and memory) that the datasets required. Tests have shown that our selections for the number of bit-parallel roots were also beneficial for PLL.

For example, in **Indochina**, by taking 64 bit-parallel roots, we would get an average of ca. 2000 normal labels. Taking 2048 bit-parallel roots gives us a much better performance with approximately the same memory usage. These values have been also verified experimentally.

The number of bit-parallel roots that have been considered in the analysis were from the set of numbers that are a power of two. This is due to the time that is needed to evaluate a test-run and also due to the fact that testing other values does not make a significant contribution to our analysis.

| Dataset | IT [$s$] | LN | QT [$\mu s$] |
|---|---|---|---|
| Gnutella | 23.7 | 151+ 256 | 4.0 |
| Slashdot | 2.1 | 26+ 32 | 1.1 |
| Epinions | 1.3 | 16+ 32 | 0.9 |
| Google | 142.0 | 36+ 256 | 2.9 |
| WikiTalk | 59.3 | 9+ 64 | 1.4 |
| Skitter | 494.4 | 47+ 256 | 3.8 |
| Flickr | 934.1 | 80+ 512 | 5.8 |
| eu-2005 | 334.0 | 33+ 512 | 4.3 |
| Hollywood | 4 403.7 | 78+2 048 | 18.0 |
| Indochina | 7 336.6 | 166+ 512 | 7.0 |

Table 5.2: Performance of PLL in our environment (single threaded). IT denotes the indexing time, QT denotes the query time, and LN denotes the average label size (first number being the number of normal and second of bit-parallel labels).

We also included a table from [AIY13]: Table 3.3 depicts their achieved results in comparison to other competitors. It can be seen that within their selected set of test data, PLL performs significantly better.

## 5.4 Our performance results

Section 5.4.1 shows the overall performance of our implementation without multi-threading, while section 5.4.2 depicts the overall performance of our implementation with multi-threading. An analysis from different perspectives is made and a discussion over the achieved results is given.

### 5.4.1 Single threaded

By comparing table 5.2 and 5.3, we see that the amount of time needed to construct the indices is similar. As both implementations follow the same logic in a single-threaded environment, the computed index has the same contents and, therefore, the same index size.

Although the same steps are being executed using a single thread, there are some marginal differences regarding the time performance. Those differences are based on the actual implementation of both techniques - our approach contains mechanisms that enable it to scale well on multiple threads and is differently implemented within some sections.

| Dataset | IT [$s$] | LN | QT [$\mu s$] |
|---|---|---|---|
| Gnutella | 27.3 | 151+ 256 | 4.1 |
| Slashdot | 1.9 | 26+ 32 | 1.1 |
| Epinions | 1.4 | 16+ 32 | 1.0 |
| Google | 147.1 | 36+ 256 | 3.0 |
| WikiTalk | 59.4 | 9+ 64 | 1.6 |
| Skitter | 381.0 | 47+ 256 | 3.7 |
| Flickr | 994.2 | 80+ 512 | 5.9 |
| eu-2005 | 284.1 | 33+ 512 | 4.4 |
| Hollywood | 4 454.1 | 78+2 048 | 18.2 |
| Indochina | 6 901.3 | 166+ 512 | 6.8 |

Table 5.3: Performance of our implementation (single threaded). IT denotes the indexing time, QT denotes the query time, and LN denotes the average label size (first number being the number of normal and second of bit-parallel labels).

The last part that requires our remarks is the query performance. The time required to answer a query is also mostly the same for both implementations. This result is induced by the similar query-implementation as compared to PLL.

### 5.4.2 Multi-threading

Table 5.4 shows that the multi-threaded index construction can be several times faster than a single-threaded execution. The speed-up factors that were achieved with the parallel execution enabled us to process networks considerably faster.

| Dataset | Number of threads 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| | IT [s] | Speed-up | | | | | |
| Gnutella | 27.3 | 1.7 | 3.4 | 6.6 | 9.0 | **17.4** | 16.3 |
| Slashdot | 1.9 | 1.5 | 2.4 | 3.5 | **4.3** | 3.1 | 1.5 |
| Epinions | 1.4 | 1.5 | 2.0 | 2.9 | **3.4** | 2.3 | 1.9 |
| Google | 147.1 | 1.6 | 3.0 | 5.7 | 7.5 | 11.1 | **17.4** |
| WikiTalk | 59.4 | 1.5 | 2.7 | 4.2 | 5.8 | 6.3 | **6.5** |
| Skitter | 381.0 | 1.7 | 3.2 | 6.0 | 8.8 | 10.2 | **20.4** |
| Flickr | 994.2 | 1.8 | 3.5 | 6.9 | 12.0 | 13.0 | **23.1** |
| eu-2005 | 284.1 | 1.7 | 3.2 | 6.1 | 10.8 | 17.4 | **22.2** |
| Hollywood | 4 454.2 | 1.8 | 3.3 | 6.5 | 11.6 | 19.9 | **25.3** |
| Indochina | 6 901.3 | 1.8 | 3.5 | 6.9 | **12.3** | 10.5 | DNF |

Table 5.4: Running time of the parallel implementation. Only the column for a single thread contains the actual time (in seconds); other columns contain only the speed-up. DNF means it run out of memory. IT denotes the indexing time.

Throughout our experiments it is evident that the achieved speed-up is not solely dependent on the size of the graph, but rather its structure too. **Indochina** has a significantly larger structure than **Hollywood** (see 5.1), yet it performs better in terms of parallelism. Overall, a correlation between the size of the graph and the speed-ups exists, though.

### Anomalies within the speed-ups

Extreme changes in the speed-up are also noticeable on some occasions – an example for this is **Gnutella**. These fluctuations arise from the internal thread-management algorithms; which we left to the operating system to choose. As we see in 5.5, utilizing the memory in a different way influences these anomalies. The same applies to the cases where the speed-up goes up, down, and up again.

### Number of bit-parallel roots

The effect the selected number of bit-parallel roots has on the construction is also analyzed with figure 5.1. Because the parallelism performs so well in the first phase (5.4), it is good to choose a larger number of bit-parallel roots. Furthermore, increasing $k$ too much worsens the performance. In our example with **Gnutella**, we found that this is case from $k = 1024$ to $k = 4096$.

Therefore, in order to reduce the index size, a lower number of bit-parallel roots can be considered without having to lose the benefits on the construction time – this applies to all datasets.

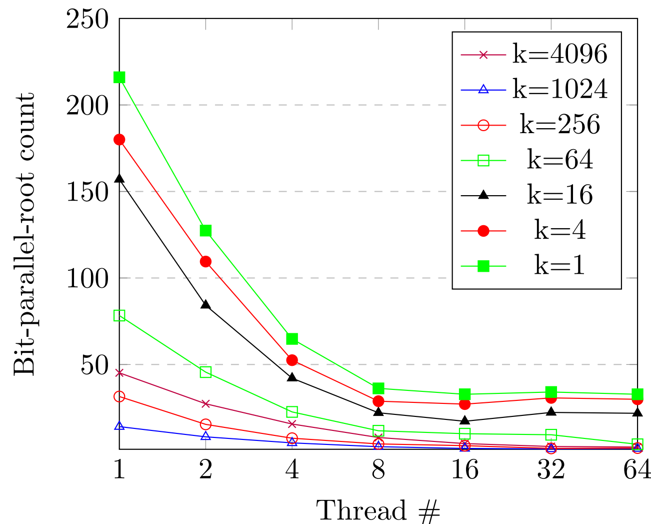


Figure 5.1: Influence of the selected number of bit-parallel roots in a multi-threaded environment. Evaluation done on **Gnutella**.

#### 5.4.2.1 Speed-ups with the memory-light version

As each thread allocates $O(|V| + |E|)$ space in the first phase, the memory consumption becomes very high for graphs with hundreds of millions of edges. Because of this, some limits exist over our main approach. For example, **Indochina** fails to construct the index with 64 threads for that reason.

| Dataset | Number of threads 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| | IT [s] | Speed-up | | | | | |
| Gnutella | 28.7 | 1.8 | 3.5 | **6.6** | 5.6 | 4.8 | 4.8 |
| Slashdot | 4.2 | 1.7 | 2.9 | **3.9** | 2.4 | 3.6 | 2.5 |
| Epinions | 1.7 | 1.6 | 2.1 | 2.9 | 3.3 | **3.7** | 2.7 |
| Google | 166.5 | 1.7 | 3.2 | 5.8 | 8.0 | 10.3 | **16.0** |
| WikiTalk | 68.8 | 1.7 | 2.8 | 4.2 | 5.7 | 5.9 | **6.4** |
| Skitter | 433.7 | 1.7 | 3.2 | 6.1 | 8.5 | 9.3 | **20.5** |
| Flickr | 1 096.9 | 1.7 | 3.5 | 6.9 | 11.9 | 11.9 | **23.7** |
| eu-2005 | 337.8 | 1.7 | 3.2 | 6.4 | 10.6 | 15.4 | **21.5** |
| Hollywood | 5 664.2 | 1.8 | 3.4 | 6.4 | 11.6 | 19.2 | **25.5** |
| Indochina | 7 419.7 | 1.8 | 3.4 | 6.8 | **12.0** | 10.0 | 11.0 |

Table 5.5: Running time of the parallel implementation with memory optimization in a multi-threaded environment. Only the column for a single thread contains the actual time (in seconds); other columns contain only the speed-up. IT denotes the indexing time.

Within the memory-light variation, that we introduced in 4.6.1, only $O(|V|)$ gets allocated per thread. Thus, when used, **Indochina** is executed without any problems. This improvement greatly eliminates the limits posed from the memory's side. An evaluation of the memory-light implementation is given with table 5.5.

It is evident that the additional time that is needed within the memory-light variation remains proportionally constant and very small (see figure 5.2). Therefore, it is also

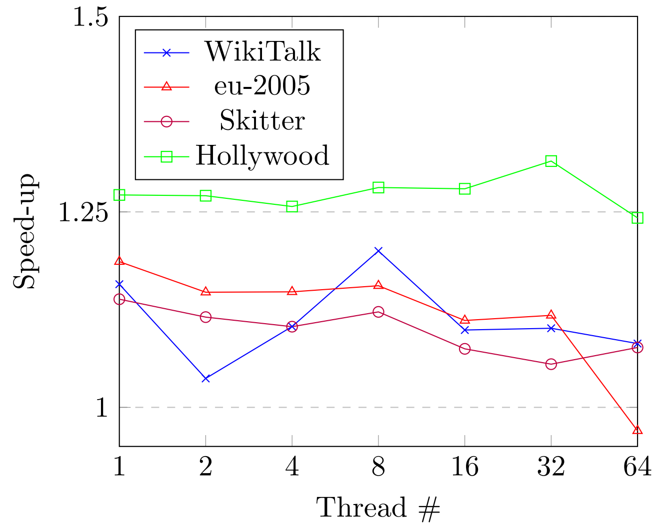


Figure 5.2: Speed-up without memory optimizations compared to the implementation with the optimizations.

acceptable to consider the memory-light variation equal to the main approach that was introduced here. The decision to differ between these variations was made due to the focus on time efficiency.

#### 5.4.2.2 Speed-ups within the individual phases

We also conducted an evaluation of the individual phases. Figure 5.3 shows the performance of the first phase. It is noticeable that the performance-increase drops when the thread count reaches a certain amount (mostly 32 in our cases). The second phase, though, gains on performance even when the thread-count reaches 64 threads (as seen in figure 5.4).

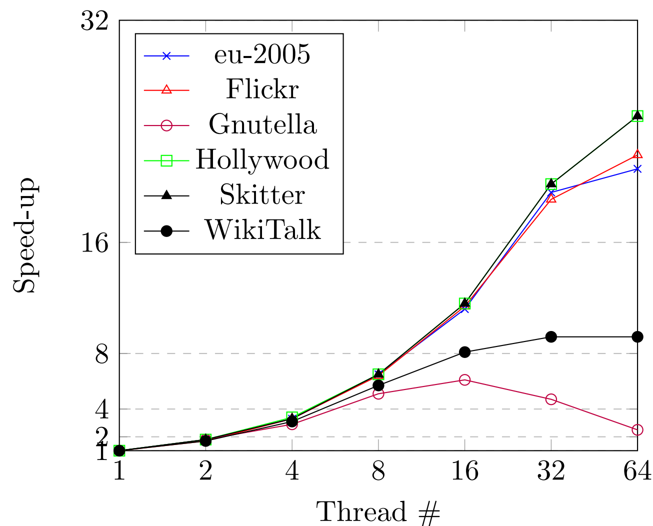


Figure 5.3: Depicting the speed-up factors of the first phase.

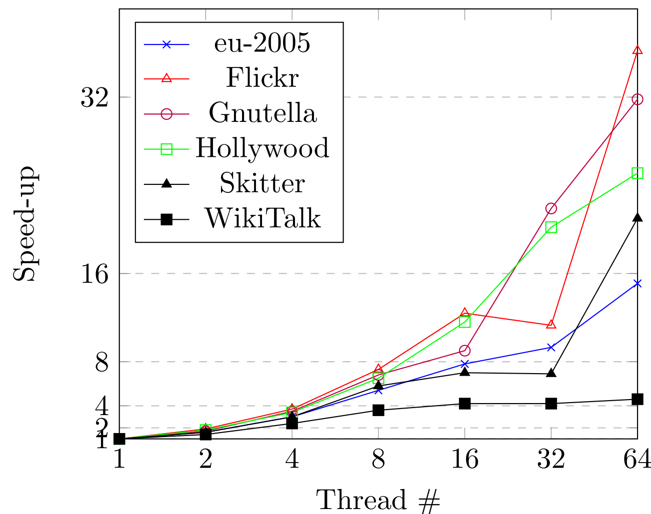


Figure 5.4: Depicting the speed-up factors of the second phase.

#### 5.4.2.3 Size of the constructed index

The index's size itself is not significantly changing between our and the PLL's implementation. For a single-threaded execution we get identical labels, while for a multi-threaded execution only minor differences exist – induced by the race conditions. Table 5.6 illustrates these differences on a selected subset of datasets. We see that these changes are in most cases minor. Surprisingly, they also often go in our favour. Extremes exist, as it is the case with **WikiTalk**.

| | Avg. label size per thread | | | | | | |
|---|---|---|---|---|---|---|---|
| Dataset | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| Slashdot | 26.4 | 26.4 | 26.4 | 26.2 | 26.2 | 25.9 | 26.9 |
| Epinions | 16.5 | 15.4 | 16.2 | 16.9 | 16.9 | 14.3 | 15.1 |
| Google | 36.1 | 36.1 | 36.1 | 36.1 | 36.4 | 37.3 | 38.1 |
| WikiTalk | 9.6 | 9.5 | 9.3 | 9.4 | 10.1 | 8.7 | 4.8 |
| Skitter | 47.4 | 45.9 | 46.0 | 45.8 | 44.9 | 44.8 | 45.0 |
| eu-2005 | 33.2 | 33.3 | 33.5 | 33.8 | 35.7 | 38.7 | 38.2 |
| Indochina | 166.7 | 172.4 | 174.6 | 173.3 | 174.8 | 181.4 | 174.0 |

Table 5.6: Effects of multi-threading on the average number of normal labels a vertex has.

#### 5.4.2.4 Query-performance

As we have seen, different indices can be acquired from the race conditions. This finding motivated us to also benchmark the performance of the shortest-path distance queries with the resulting labels. Unfortunately, we found that we got slower results in this domain – up to 20% in some cases. Table 5.7 showcases this finding.

| | Query time [$\mu s$] | | | | | | |
|---|---|---|---|---|---|---|---|
| Dataset | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| Slashdot | 1.1 | 1.2 | 1.2 | 1.2 | 1.3 | 1.2 | 1.3 |
| Epinions | 1.0 | 1.0 | 1.1 | 1.2 | 1.1 | 1.1 | 1.1 |
| Google | 3.0 | 3.8 | 4.0 | 4.1 | 4.1 | 4.0 | 4.0 |
| WikiTalk | 1.6 | 2.3 | 2.2 | 2.1 | 2.2 | 2.2 | 2.2 |
| Skitter | 3.8 | 4.6 | 4.9 | 4.9 | 4.8 | 4.8 | 4.5 |
| eu-2005 | 4.4 | 5.6 | 6.0 | 6.0 | 6.1 | 6.0 | 5.6 |
| Indochina | 6.8 | 8.1 | 7.7 | 7.8 | 7.9 | 7.9 | 7.8 |

Table 5.7: Relation of the query performance (in nanoseconds) to the given number of threads.

#### 5.4.2.5 Asymptotic analysis of the memory usage

The interesting part to note regarding memory consumption lies within the amount of space that is used during the computation (excluding the index). Let $p$ be the number of threads working on the index construction. From the asymptotic perspective, $O(p \cdot (|V| + |E|))$ memory-space is utilized for the construction of an index. As the number of edges can get very high, the memory-improvement has also been proposed; this leads to $O(p \cdot |V| + |E|)$ memory usage.

# 6. Future work

Theoretically, both phases of our approach have a linear speed-up in the number of threads that are assigned to it.

It has to be noted that race conditions exist within both phases. The flow of the execution often diverges from previous runs and may construct a slightly different index each time. This also means that the time-performance may be different each time. However, a large number of tests have shown that these changes are minor. In order to make such conditions thread safe, various mechanisms were implemented. These mechanisms and the overhead caused by the internal workflow of the system induce some slowdown on the performance. Therefore, we believe that improvements within our approach would mainly include technical details.

An interesting idea to explore for better time-performance is to utilize the research that has been done in the area of concurrent hash-tables. There is often the need to access an array concurrently, and it may prove useful to use a hash-table for that. Supportive for this argument is also the random access on an array on which only a few fields may ever get accessed. This way, cache-efficiency may be better utilized. We believe that this may open new areas within this work.

Further research could be also done on our attempt to generalize the bit-parallel labeling. The difficulty we see here lies in the utilization of the resulting labels. We were not able to make the querying over them efficiently enough, which, if possible, could lead to a better coverage for the roots, smaller indices, and faster distance queries.

# 7. Conclusion

As demonstrated, high parallelism can be achieved for the landmark-labeling method as in PLL. Network-instances, for which PLL showcased impractical time performance, can be executed several factors faster and, thus, enter in the domain of practical usage. We were also able to maintain the benefits that PLL offered: small query times and small index size. The main method does use more memory throughout its construction, but with the memory-improved version this remains in an acceptable range.

We introduced concurrency in PLL by executing multiple searches in its phases in parallel. The main challenge that existed, and for which we offered a solution, were the mechanisms needed to make it thread-safe.

We proved its practical usage by benchmarking its performance on several real-life networks from various scenarios. In every instance that we tested, speed-ups were achieved. Theoretically, a linear speed-up can be achieved with this approach; due to the overhead introduced by multi-threading this theoretical bound could not be reached.

It is essential to note that using our approach leads to a slightly worse performance with regard to the queries (as seen in 5.7). The significance of this drawback has to be determined on a case-to-case basis.

# Appendix

## A Failed methods

Throughout our research on this topic, several other approaches have been tested to improve the performance of the index construction. In the following sections we summarize some of them and briefly explain the difficulties that were encountered.

### A.1 Generalised bit-parallel labeling

Addressed in 4.2.1: We were not able to improve the performance of the bit-parallel labeling by introducing an approach that is able to use roots without a common, adjacent, vertex.

### A.2 Parallel Breadth-first-search

Instead of running several searches in parallel, the search itself can be also made concurrent. This can be done by introducing parallelism within the part where the new frontier gets computed, which is usable in both phases. Unfortunately, results have shown that this approach is, by far, slower. Using the threads to execute multiple searches in parallel leads to a much better performance. The problem with this approach was within the pruning section in each search.

It is also important to note, however, that this approach does lead to some speed-up (rarely exceeding the one-digit domain).

The experiments included the usage of the Ligra framework [SB13] (a graph processing framework for shared memory).

### A.3 Multi-source

In order to maximize the throughput of each search, the selection of multiple roots was tested. The problem here was the overhead of having to maintain multiple arrays for each root individually. As stated in 6, concurrent hash-tables with cache-efficiency may represent useful for handling these drawbacks.

32

# 8. Acknowledgments

In this section, I would like to express my gratitude to professor Guy Bleloch and Ph.D. student Julian Shun from Carnegie Mellon University (CMU). They supported my research throughout my stay at CMU; weekly meetings with them, in which we discussed both my results and new ideas, proved essential. They also provided me with access to several high-quality machines for testing my algorithm.

Additionally, my gratitude goes to professor Peter Sanders, Niklas Baumstark from Karlsruhe Institute of Technology and Madeline Lemberg from Yale University. Professor Sanders supervised my thesis, gave the initial topic, and offered some of his thoughts regarding it. Niklas helped me from the technical side and introduced me to the new working environment at CMU. Madeline Lemberg edited several sections of my thesis.

34

# Bibliography

[ADGW12] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," Tech. Rep. MSR-TR-2012-46, April 2012. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=163231

[AIY13] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," pp. 349–360, 2013. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465315

[ASK12] T. Akiba, C. Sommer, and K.-i. Kawarabayashi, "Shortest-path queries for complex networks: Exploiting low tree-width outside the core," in *Proceedings of the 15th International Conference on Extending Database Technology*, ser. EDBT '12. New York, NY, USA: ACM, 2012, pp. 144–155. [Online]. Available: http://doi.acm.org/10.1145/2247596.2247614

[BCSV04] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Softw. Pract. Exper.*, vol. 34, no. 8, pp. 711–726, Jul. 2004. [Online]. Available: http://dx.doi.org/10.1002/spe.587

[BRSV10] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," *CoRR*, vol. abs/1011.5425, 2010.

[BV04] P. Boldi and S. Vigna, "The webgraph framework i: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 595–602. [Online]. Available: http://doi.acm.org/10.1145/988672.988752

[CHKZ02] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 937–946. [Online]. Available: http://dl.acm.org/citation.cfm?id=545381.545503

[Goo02] Google, "Google programming contest 2002," 2002.

[JRXL12] R. Jin, N. Ruan, Y. Xiang, and V. Lee, "A highway-centric labeling approach for answering distance queries on large sparse graphs," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 445–456. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213887

[LHK10a] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Predicting positive and negative links in online social networks," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10.

New York, NY, USA: ACM, 2010, pp. 641–650. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772756

[LHK10b] ——, "Signed networks in social media," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 1361–1370. [Online]. Available: http://doi.acm.org/10.1145/1753326.1753532

[LKF05] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: ACM, 2005, pp. 177–187. [Online]. Available: http://doi.acm.org/10.1145/1081870.1081893

[LLDM08] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *CoRR*, vol. abs/0810.1355, 2008.

[MMG+07] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *IMC'07*, 2007.

[PBCG09] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, ser. CIKM '09. New York, NY, USA: ACM, 2009, pp. 867–876. [Online]. Available: http://doi.acm.org/10.1145/1645953.1646063

[RAD03] M. Richardson, R. Agrawal, and P. Domingos, "Trust management for the semantic web," in *The Semantic Web - ISWC 2003*, ser. Lecture Notes in Computer Science, D. Fensel, K. Sycara, and J. Mylopoulos, Eds. Springer Berlin Heidelberg, 2003, vol. 2870, pp. 351–368. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-39718-2_23

[RFI02] M. Ripeanu, I. Foster, and A. Iamnitchi, "Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design," *arXiv preprint cs/0209028*, 2002.

[SB13] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2517327.2442530

[Wei10] F. Wei, "Tedi: Efficient shortest path query answering on graphs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807181