Parallel Cluster-BFS and Applications to Shortest Paths*

Letong Wang[†] Guy Blelloch[‡] Yan Gu [†] Yihan Sun [†]

Abstract

Breadth-first Search (BFS) is one of the most important graph processing subroutines, especially for computing the *unweighted distance*. Many applications may require running BFS from multiple sources. Sequentially, when running BFS on a cluster of nearby vertices, a known optimization is using *bit-parallelism*. Given a subset of vertices of size k and the distance between any pair of them is no more than d, BFS can be applied to all of them in a total work of $O(dm(k/w+1))$, where w is the length of a word in bits and m is the number of edges. We will refer to this approach as **cluster-BFS (C-BFS)**. Such an approach has been studied and shown effective both in theory and in practice in the sequential setting. However, it remains unknown how this can be combined with thread-level parallelism.

In this paper, we focus on designing efficient *parallel* C-BFS based on BFS to answer unweighted distance queries. Our solution combines the strengths of bit-level parallelism and thread-level parallelism, and achieves significant speedup over the plain sequential solution. We also apply our algorithm to real-world applications. In particular, we identified another application (landmark-labeling for the approximate distance oracle) that can take advantage of parallel C-BFS. Under the same memory budget, our new solution improves accuracy and/or time on all the 18 tested graphs.

1 Introduction

Breadth-First Search (BFS) is one of the most important graph processing subroutines. Given a graph $G = (V, E)$ and a vertex $s \in V$, BFS visits all vertices in V in increasing order of (hop) distance to s. BFS can be used for many purposes. One of the most common use scenarios for BFS is to compute the *unweighted*

distance from the source. In this paper, we focus on designing efficient *parallel* approaches based on BFS to answer unweighted distance queries. Throughout the paper, we use $n = |V|$ and $m = |E|$, and use "distance" to refer to the hop distance on an unweighted graph.

Many applications may require running BFS from multiple sources. Examples of this are using BFS for low-diameter decomposition [30], all-pairs shortest paths (APSP), or oracles for exact or approximate APSP. A key observation is that *bit-parallelism* [16] can be used effectively when running BFS on a cluster of nearby vertices [1, 16]. Given a subset of vertices with size k and the distance between any pair of them is no more than d, BFS can be applied to all of them in total $O(dm(k/w + 1))$ work (number of operations), where w is the length of a word in bits and m is the number of edges [16]. Since a machine word must hold at least $\Omega(\log n)$ bits to store a pointer, this means $w = \Omega(\log n)$. We will refer to this approach as *cluster-BFS (C-BFS)*, and present more details in Sec. 3. Chan [16] used this idea to develop an all-pair shortest-path algorithm that runs in $O(mn/w)$ work (when $m = \Omega(n \log n \log \log \log n)$). In addition to saving time, C-BFS also saves space: it only uses $O(d)$ words per w vertices instead of a word (or at least enough bits to store a distance) per vertex as in standard BFS. Akiba et al. [1] used this idea in the exact two-hop distance oracle but only considered the special case for $d = 2$ (a star-shaped cluster: a vertex and its neighbors). We refer to this algorithm as the AIY algorithm. Both of the previous papers focus on the sequential setting.

While the C-BFS with bit-level parallelism has shown to be effective in sequential settings, surprisingly, we know of no previous work combining it with thread-level parallelism. BFS is one of the most well-studied parallel graph processing problems, and state-of-the-art solutions have been highly optimized using techniques such as direction optimizations [6, 38]. To be practical, any C-BFS would have to compete with these. Our goal is to develop an efficient C-BFS with high parallelism such that it (1) achieves the same level of parallelism as the standard parallel BFS, with additional benefits by using clusters, (2) supports a clean interface that is flexible for different parameter settings (i.e., varying d

---

and k), and (3) facilitates various real-world applications. In this paper, we provide a systematic study of parallel C-BFS and achieve all three goals above.

To achieve *high performance*, we design an efficient parallel algorithm. Our algorithm is work-efficient (i.e., it has the same asymptotical work as the sequential counterpart). It has the same span as regular parallel BFS algorithms (e.g., [38]), which $\tilde{O}(D)$ for graph diameter D, and thus is best suited for small-diameter graphs, such as social networks, computer networks, or web graphs. Our algorithm uses the *directional optimization* that has been shown to be useful for parallel BFS. By doing this, our algorithm achieves the strengths of both bit-level and thread-level parallelism: it has high parallelism as in the state-of-the-art parallel BFS algorithms and obtains additional performance gain by using bit-level parallelism.

To achieve a *flexible interface*, we designed our algorithm for general k and d, easily integrating into various applications with user-defined parameters.

To understand how C-BFS can *facilitate real-world applications*, we study two Distance Oracle (DO) techniques that can benefit from C-BFS: the exact DO as in [1] and the Landmark Labeling (LL) for an approximate DO. As far as we know, our work is the first to use C-BFS to accelerate LL.

We implemented our C-BFS algorithm and the two applications. We compare our algorithm with multiple baselines to study the performance gain in depth, and test it on 18 graphs with various types and sizes on a 96-core machine. Compared to standard sequential BFS, our algorithm employs both bit-parallelism (on clusters) and thread-level parallelism (along with optimizations used in parallel BFS) to improve performance. In the simplest setting where $k = 64$ and $d = 2$, the combination of them enables up to 1119× speedup (500× on average) compared to the plain sequential BFS, where bit-level and thread-level parallelism each contributes about 20× speedup. Interestingly, by comparing with the performance of existing work (Ligra [38], where only thread-parallelism is used, and AIY [2], where only bit-level parallelism is used), we observed that both bit-level parallelism and thread-level parallelism and work very well in synergy. Each of them still fully contributes to the performance when the other is present, achieving the same level of improvement as when used independently (see Fig. 1). Therefore, we believe our work on an efficient implementation combining thread-level and bit-level parallelism fills the gap in the existing study of both C-BFS and parallel BFS.

We also studied the performance of our C-BFS with different parameters. Typically, k is set to be $\Theta(w)$, and the work (and space) is proportional to d. Our result
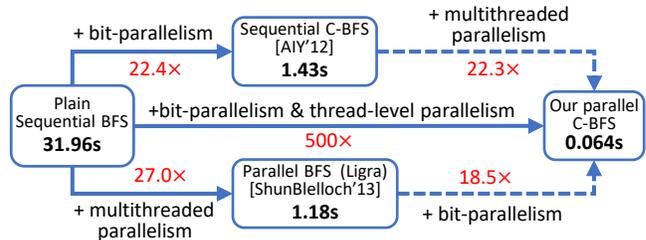

Figure 1: Performance comparison with existing work. We test the running time of BFSs from a cluster of 64 vertices. The baselines are Ligra [38] that only uses thread-level parallelism and AIY'12 [2] that only uses bit-level parallelism. The numbers are geometric means across 18 graphs. Full results are shown in Tab. 2 and Fig. 3.

shows that the running time increases almost linearly with value of d, especially when d is small. This explains why existing work (e.g., [2]) tends to choose the smallest $d = 2$ case in real-world applications.

Applying our algorithm also gives significant improvement to the aforementioned applications. For the 2-hop distance oracle, our parallel implementation outperforms the sequential AIY algorithm [1] by 9–36×, and can process much larger graphs than the AIY algorithm. For landmark labeling (LL), with a fixed memory budget, C-BFS improved regular LL in either accuracy or preprocessing time on all 18 tested graphs, and improved *both* on 14/18 graphs. This is due to the saving in space allowing more landmarks to be used for C-BFS. We observed that using $d = 2$ achieved better overall performance in accuracy, time and space. Due to the page limit, we present more results in our full version.

2 Preliminaries

Notations. Let $G = (V, E)$ be an unweighted graph. We use $n = |V|$, $m = |E|$, and use D to denote the diameter of the graph. Let $N(v) = \{u \in V \mid (v, u) \in E\}$ be the set of neighbors of vertex $v \in V$. In directed graphs, $N^+(v)$ and $N^-(v)$ represent outgoing and incoming neighbors, respectively. We use $\delta(u, v)$ to denote the shortest distance between u and v. We assume machine word size $w = \Omega(\log n)$, such that the vertex and edge IDs are within constant words. Let $S = \{s_1, s_2, ..., s_k\}$ represent a cluster, where k is the cluster size. Let d be the diameter (maximum distance between any pair) of the cluster. We summarize the notations in Tab. 1.

Computational Model. We use the binary fork-join parallel model [8, 17], with work-span analysis [10, 21]. We assume a set of threads that access a shared memory. A thread can `fork` two child threads to work in parallel, and then waits. When both children complete, the parent thread continues. A parallel for-loop can be simulated by recursive `fork`s in logarithmic

$G = (V, E)$: the input graph. $n = |V|$ and $m = |E|$.

$S = \{s_1, ..., s_k\}$: the source cluster for the BFS.

k : the cluster size, i.e., $k = |S|$.

d : the diameter of the cluster.

w : the length of a word in bits. $w = \Omega(n)$.

D : the diameter of the graph.

$\delta(u, v)$: the shortest distance between u and v.

Table 1: Notations in the paper.

levels. The **work** of an algorithm is the total number of instructions, and the **span** is the length of the longest sequence of dependent instructions. We can execute the computation using a randomized work-stealing scheduler [10, 20].

We assume two unit-cost *atomic* operations. COMPARE_AND_SWAP(p, v_{old}, v_{new}) atomically reads the memory location pointed to by p, and writes value v_{new} to it if the current value is v_{old}. It returns *true* if it succeeds and *false* otherwise. FETCH_AND_OR(p, v_{new}) atomically reads the memory location pointed to by p, takes the bitwise OR operation with value v_{new}, and stores the results back. It returns *true* if v_{new} successfully sets any bit stored in p to be 1, and *false* otherwise. Most machines directly support these instructions.

Parallel BFS. We briefly review parallel BFS, because it is one of our baselines, and some of the concepts are also used in our cluster-BFS. Parallel BFS starts from a single source $s \in V$. The algorithm maintains a **frontier** of vertices to explore in each round, starting from the source, and finishes in at most D rounds. In round i, the algorithm processes (visits their neighbors) of the current frontier $\mathcal{F}_i$, and puts all their (unvisited) neighbors in the next frontier $\mathcal{F}_{i+1}$. If multiple vertices in $\mathcal{F}_i$ attempt to add the same vertex to $\mathcal{F}_{i+1}$, a COMPARE_AND_SWAP is used to guarantee that only one will succeed.

One widely-used optimization for parallel BFS is directional optimization [6, 38]. At a high level, when the frontier size $|\mathcal{F}_i|$ is large, the algorithm will not process $\mathcal{F}_i$, but instead visit each unprocessed vertex v, and determine if v has an incoming neighbor in $\mathcal{F}_i$. If so, v will be put in $\mathcal{F}_{i+1}$. Such an optimization is observed to be effective, especially on small-diameter graphs. We present more details in our full version paper.

3 Parallel Cluster-BFS

Cluster-BFS (C-BFS) runs BFS from a cluster of sources $S \subseteq V$. If the sources have diameter d (maximum distance between any pair), then all distances from S to any vertex $v \in V$ will differ by at most d. This means that if we run a set of BFSs from S, syn-

chronously, every $v \in V$ will appear in at most $d+1$ consecutive frontiers. Cluster-BFS takes advantage of this by representing all the sources in S that can visit a given vertex, on a given round, as a vector of booleans (bits). In this way, each vertex will be visited at most $d+1$ times instead of $|S|$ times if all searches are performed separately. More details are described in Sec. 3.1. Importantly, if the bit-vector fits in $O(1)$ words, the distances of all $|S|$ sources can be handled (propagated from a vertex in the current frontier to a neighbor) with $O(1)$ bitwise logical operations. Since a machine word must hold at least $\Omega(\log n)$ bits (so it can represent a pointer), this means the algorithm can save a factor of $\Omega(\log n/d)$ work.

It appears that Chan first described this idea [16], but he did not go into any details of the implementation, but just saying that this is possible. Akiba et al. [1] later showed a concrete implementation based on this idea, with the limitations that it is sequential and only works on the cluster of a star with $d = 2$ (a center vertex and its neighbors). To the best of our knowledge, there has been no previous work on developing a parallel implementation of the cluster-BFS algorithm. Indeed it is challenging to achieve high performance given how BFS has been widely studied with numerous optimizations both sequentially and in parallel. In this paper, we propose our algorithm, given as Alg. 1, which is an efficient parallel cluster-BFS implementation with a general interface and low coding effort. In the following sections, we will first introduce the bitwise representation to maintain the distances in cluster-BFS and then give our parallel cluster-BFS algorithm.

3.1 Cluster Distance Representations

Given a set S of source vertices with diameter d, cluster-BFS computes a compact representation of the shortest distance from every source in S to every vertex in V. The idea of cluster-BFS is based on the following fact.

FACT 3.1. *On an unweighted graph, if the distance betweeen vertex s_1 and s_2 is d, then for any vertex $v \in V$, $|\delta(s_1, v) - \delta(s_2, v)| \leq d$.*

For example, if s_1 and s_2 are neighbors, the distances from a vertex v to them can differ by at most 1. We can further extend Fact 3.1 to Cor. 3.1, which says if a cluster of vertices S has diameter d, the distances from v to vertices in S differ by at most d.

COROLLARY 3.1. *On an unweighted graph, given a set S of vertices with diameter no more than d, for any vertex $v \in V$, we have*

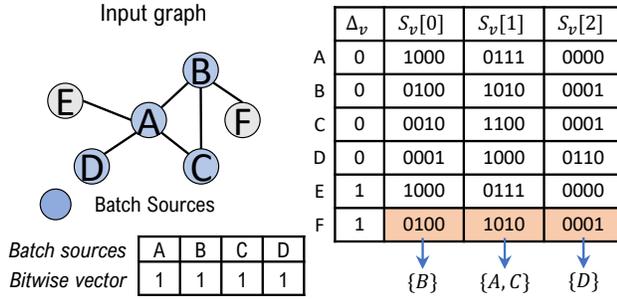$$\max_{s \in S} \delta(s, v) - \min_{s \in S} \delta(s, v) \leq d$$

Input graph

Batch Sources

Batch sources	A	B	C	D
Bitwise vector	1	1	1	1

	Δ_v	$S_v[0]$	$S_v[1]$	$S_v[2]$
A	0	1000	0111	0000
B	0	0100	1010	0001
C	0	0010	1100	0001
D	0	0001	1000	0110
E	1	1000	0111	0000
F	1	0100	1010	0001

$\{B\}$ $\quad$ $\{A,C\}$ $\quad$ $\{D\}$

Figure 2: Illustration of bitwise representation. The batch set S is $\{A, B, C, D\}$. 4-bit bit-subsets are used to represent subsets of S. Δ_v is the smallest shortest distance from any vertex in S to v. The subset $S_v[i]$ is defined as $\{s \in S | \delta(s,v) = \Delta_v + i\}$.

Therefore, for each vertex v, we can classify the sources in S by their distances to v. Let $\Delta_v = \min_{s \in S} \delta(s,v)$ be the smallest distance from any source in S to v. According to Cor. 3.1, the distance between v and any $s \in S$ must be in range $[\Delta_v, \Delta_v + d]$. This divides all vertices in S in $d+1$ different subsets based on their distances to v. Let $S_v[i]$ be the subset of sources in S that has a distance to v as $\Delta_v + i$. More formally,

$$S_v[i] = \{s \in S \mid \delta(s,v) = \Delta_v + i\}$$

Then for a vertex v, the distances between v and all sources in S can be represented by the $(d + 2)$-tuple $\langle S_v[0..d], \Delta_v\rangle$, which we call the *cluster distance vector* of v to S.

Note that if $|S| = w$, we can use a one word bit-vector to represent any subset of $S' \subseteq S$: bit i is 1 iff. the i-th element in S is also in S'. We call such a representation of a subset of S a *bit-subset*. In this way, a cluster distance vector only takes $d+1$ words for bit-subsets and one byte to store the shortest distances from v to $|S|$ sources (assuming $D < 256$).

An illustration for the bit-subset and cluster distance vector is shown in Fig. 2. In this example, S is the set $\{A, B, C, D\}$, and the diameter of the subgraph is $d = 2$. We need subsets $S_v[0..2]$ for each vertex v, which are represented by the bit-subsets, each with four bits. A to D are represented by the four bits from left to right. From the cluster distance vector, we can recover the shortest distance from all the sources $s \in S$ to each vertex $v \in V$ by the fact that each source in $S_v[i]$ has distance $\Delta_v + i$ to v. For example, for vertex F, $S_F[1]$ is 1010, which represents the subset $\{A, C\}$, we can infer $\delta(B, A) = \delta(B, C) = \Delta_F + 1 = 2$.

The main idea of cluster-BFS is to use bitwise operations on bit-subsets to quickly compute the union/intersection of the sets, allowing us to use the cluster distance vector of v to compute the cluster distance vectors of its neighbor u in constant time. In the following, we elaborate on our parallel cluster-BFS algorithm.

Algorithm 1: Cluster-BFS search from S

Input:
A graph $G = (V, E)$, a cluster $S \subseteq V$ with diameter d
Output:
cluster distance vectors $\langle S_v[0..d], \Delta_v\rangle$ for all $v \in V$.
Maintains:
i: the current round number, initialized to 0
$S_{seen}[\cdot], S_{next}[\cdot]$: array of bit-subset for each $v \in V$
$r[v]$: the lastest round v is in the frontier
$\mathcal{F}_i$: frontier vertices in round i

// Initialization
1 **ParallelForEach** $v \in V$ **do**
2 $\quad$ $S_{seen}[v] \leftarrow \emptyset$, $S_{next}[v] \leftarrow \emptyset$
3 $\quad$ $\Delta_v \leftarrow \infty$
4 $\quad$ $r[v] \leftarrow \infty$
5 **for** $s \in S$ **do** $S_{seen}[s] \leftarrow \{s\}$
6 $i \leftarrow 0$
7 $\mathcal{F}_0 \leftarrow S$
// Traversing
8 **while** $\mathcal{F}_i \neq \emptyset$ **do**
9 $\quad$ **ParallelForEach** $u \in \mathcal{F}_i$ **do**
10 $\quad\quad$ $S_{new} \leftarrow S_{next}[u] \setminus S_{seen}[u]$
11 $\quad\quad$ **if** $\Delta_u = \infty$ **then** $\Delta_u \leftarrow i$
12 $\quad\quad$ $S_u[i - \Delta_u] \leftarrow S_{new}$
13 $\quad\quad$ $S_{seen}[u] \leftarrow S_{seen}[u] \cup S_{new}$
14 $\quad$ **ParallelForEach** $u \in \mathcal{F}_i$ **do**
15 $\quad\quad$ **ParallelForEach** $v \in N(u)$ and $i - \Delta_v < d$ **do**
16 $\quad\quad\quad$ **if** FETCH_AND_OR$(S_{next}[v], S_{seen}[u])$
17 $\quad\quad\quad\quad$ **if** COMPARE_AND_SWAP$(r[v], r[v], i)$
18 $\quad\quad\quad\quad\quad$ $\mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \cup \{v\}$
19 $\quad$ $i \leftarrow i + 1$
20 **return** $\langle S_v[1..d], \Delta_v\rangle$ for all $v \in V$

3.2 Our Parallel Algorithm

In this section, we introduce our parallel algorithm to compute the cluster distance vector for all vertices given a source cluster S, which is $\langle S_v[0..d], \Delta_v\rangle$ for all $v \in V$. The pseudocode of our cluster-BFS is shown in Alg. 1.

Our cluster-BFS algorithm is based on the following fact: if u and v are neighbors and there is a path from a source $s \in S$ to u with length $i - 1$, then there must exist a path from s to v with length i. In round i, u records all the sources in S that reach it in round i by a bit-subset. When u visits its neighbor v, u propagates this bit-subset to v by taking a bitwise OR operation with the bit-subset representing the vertices reaching v in round $i+1$ (Alg. 1: line 16). According to Cor. 3.1, all the vertices in S will visit v at least once during round Δ_v to round $\Delta_v + d$; in other words, all the vertices will be put into the frontier for $d + 1$ times. Therefore, we need to record the number of times v has been put in the frontier. When v has been put in the frontier $d + 1$ times, since all sources in S must have already visited v, we do not need to process v anymore. Otherwise, we

will process v and put it to the next frontier since other sources in S may visit v in the future.

In our algorithm, we use a boolean array $S_{seen}[\cdot]$ to store whether a vertex has been visited in all previous rounds by any sources in S, and $S_{next}[v]$ includes the vertex if it is also in the current frontier (i.e., visited by any vertex in the current round). We denote Δ_v as the first round that any vertex from S touches vertex v. Alg. 1 has two stages: initialization and traversing.

Initialization This step is relatively simple. We initialize the arrays of $S_{seen}[\cdot]$, $S_{next}[v]$, and Δ_v. We use another, array $r[\cdot]$, to avoid duplication of vertices in the frontier. Later in the traversing stage, when multiple vertices want to add v to the next frontier at the same time, only one can successfully set $r[v]$ to the current round number by atomic operation COM-PARE_AND_SWAP (line 17), and the successful vertex will put v to the next frontier.

Traversing At the beginning, the algorithm puts all the sources $s \in S$ into the first frontier $\mathcal{F}_0$. Then, we visit all vertices by frontiers. In each round, we process frontiers in two stages, where the first stage processes vertices and the second stage processes edges. In the first stage (line 9 to line 13), we first compute the sources that newly visited u by $S_{new} \leftarrow S_{next}[u] \setminus S_{seen}[u]$ (line 10). Note that the bit-subset S_{new} contains sources whose distances are i, which is also $S_u[i - \Delta_u]$ (line 12). Then we update $S_{seen}[u]$ to include newly visited vertices (line 13), and set Δ_u to the current round number if it has not been set yet (line 11). In the second stage (line 14 to line 18), we process the neighbor vertices of the current frontier that have not been visited for d times already (line 15). For an edge from $u \in \mathcal{F}_i$ to its neighbor v, we propagate the sources seen so far by u, $S_{seen}[u]$, to v (line 16). In general, if any source $s \in S$ visited u in the previous round, s should also visit v in this round, and should be included in the $S_{next}[v]$ for v in this round. If the $S_{next}[v]$ is changed, which means there are new sources visiting v, v should be added to the next frontier. To avoid duplication in the next frontier, only the one that can successfully set $r[v]$ to i (line 17) by COMPARE_AND_SWAP will put v to the next frontier. Note that we can further benefit from the directional optimization that is commonly used in parallel BFS. Additional details about the directional optimization are given in our full version paper.

The efficiency of the algorithm relies on using bit-operations to compute the union and difference of two bit-subsets, stated below.

LEMMA 3.1. *Given the bit-subset subsets S_1 and S_2 of a set S with size k, we can compute the bit-subset representation of $S_1 \cup S_2$, $S_1 \setminus S_2$ in $O(k/w + 1)$ work and $O(\log(k/w) + 1)$ span, where w is the word length.*

Proof. A bit-subset with k bits needs $\lceil k/w \rceil$ words. These $\lceil k/w \rceil$ words can be processed in parallel. With the constant cost for bitwise or/not operations for a single word, the work and span for $S_1 \cup S_2$, $S_1 \setminus S_2$ are $O(k/w + 1)$ and $O(\log(k/w) + 1)$. $\square$

We now show the cost analysis of the cluster-BFS algorithm.

THEOREM 3.1. *Given a set S of k vertices with diameter d, we can compute the cluster distance vector from S to every vertex in V in $O(dm(k/w + 1))$ work and $O((D + d) \log n)$ span.*

Proof. We will analyze the work and span for the traversal stage, which dominates the cost of the initialization stage. The traversal consists of two phases: the first phase processes vertices in the frontier, and the second phase processes edges from the frontier. The cost of processing a single edge and a vertex is asymptotically the same, as it involves applying a constant number of set operations, which results in $O(k/w + 1)$ work and $O(\log(k/w)+1)$ span, as described in Lem. 3.1. Assuming $n < m$, the cost of traversal is primarily determined by edge processing. Therefore, the rest of the proof will focus on analyzing the cost of edge processing.

For the work, since each vertex is in the frontier for at most d times, each edge is processed at most d times, leading to a total work of $O(dm(k/w + 1))$. For the span, recall that D is the diameter of the graph, and there are at most $D + d$ rounds in the traversal stage. The span for each round is $O(\log(k/w)+1+\log n)$, which accounts for the cost of generating $O(m)$ parallel tasks (costing $O(\log n)$ in the binary fork-join model) and the cost of processing a single edge (costing $O(\log(k/w)+1)$ as shown in Lem. 3.1). Since k/w is smaller than n, the span for each round simplifies to $O(\log n)$. Thus, the total span for the $D + d$ rounds of the traversal stage is $O((D + d) \log n)$. Therefore, the total work and span for our parallel cluster-BFS are $O(dm(k/w + 1))$ and $O((D + d) \log n)$, respectively. $\square$

If we take $k = \Theta(w)$, such that each bit-subset fits within a constant number of words, the work simplifies to $O(dm)$ and the span remains $O((D+d) \log n)$, which matches the work and span of a single BFS. Since $w = \Omega(\log n)$, this means that we can compute $O(\log n)$ more BFSs with asymptotically the same cost.

4 Applications

In this section, we show two applications that can benefit from our new parallel Cluster-BFS algorithm. Both applications are distance oracles (DO) for unweighted graphs. A ***distance oracle (DO)*** is an index designed

to answer the shortest distances between two vertices on a graph. Although such a query can always be answered by computing the distance on the fly (e.g., running a BFS from one of the query vertices), this can be inefficient for applications requiring low latency or requesting multiple queries. A distance oracle aims to store information generated during preprocessing to accelerate the distance queries.

From the perspective of accuracy, distance oracles can be classified into approximate distance oracles (ADO) and exact distance oracles (EDO). ADOs may not answer the accurate distance but are cheaper in preprocessing time, query time, and index space, and thus scale to large graphs. EDOs always give the exact distance but can be more expensive to compute. Therefore, EDOs are usually used in applications that are on small graphs but more sensitive to accuracy. In this section, we will introduce how to apply our cluster-BFS to the two existing distance oracles: a 2-hop labeling-based EDO and a landmark labeling-based ADO. Since both applications work on undirected graphs, we assume the graph to be undirected in this section. Our cluster-BFS algorithm works for general directed graphs.

4.1 An EDO based on 2-Hop Labeling

Here, we consider the EDO constructed by Akiba et al. [2], called *2-hop labeling*, and we apply the idea of C-BFS to this algorithm. Their original algorithm is sequential, and we refer to it as the *AIY* algorithm. We can replace the component for 2-hop labeling in their algorithm with our parallel C-BFS to achieve better performance. For completeness, we describe their algorithm in the full version paper. In the experiments, we compared our parallel C-BFS with the component of C-BFS in their sequential code. We also parallelized the entire algorithm for 2-hop labeling using our parallel C-BFS, and present a comparison in our full version paper.

4.2 An ADO based on Landmark Labeling

In this paper, we mainly focus on this application that can benefit from C-BFS, which we believe is new. The application is an ADO based on landmark labeling [22, 32, 39, 42, 43]. As mentioned, ADOs sacrifice accuracy to get lower running time and index space. Here, as is common, we assume a one-sided error, such that the distance reported by the ADO cannot be smaller than the actual distance. To measure the loss in accuracy, for a distance query on $u, v \in V$, we use **distortion** $\psi(u, v)$ as the ratio between the answer from an ADO (denoted as $query(u, v)$) over true distance $\delta(u, v)$, i.e., $\psi = query(u, v)/\delta(u, v)$. We define the distortion of an ADO as the average distortion over all

---

Algorithm 2: Framework of Landmark Labeling

1 The algorithm maintains $L[\cdot][\cdot]$ as the index with size $n \times |S|$. $L[v][i]$ is the distance between vertex v and landmark h_i, initialized as ∞

2 Function CONSTRUCT_INDEX(G, H) // $G = (V, E)$
    // Each $h_i \in H$ is a vertex in the plain LL, and is a cluster in C-BFS-based LL
3   **for** $h_i \in H$ **do**
      // In our algorithm, we replace BFS with C-BFS
4 $t[1..n] \leftarrow$ BFS(G, h_i)
5   **foreach** $v \in V$ **do** $L[v][i] \leftarrow t[v]$
6   **return** L
7 Function QUERY(u, v)
8 $ans \leftarrow \infty$ // the answer of the query
9   **for** $i \leftarrow 0$ to $|H| - 1$ **do**
      // Our algorithm computes the shortest distance via all vertices in all clusters
10 $dis \leftarrow L[u][i] + L[v][i]$
11 $ans \leftarrow \min(ans, dis)$
12   **return** ans

---

pairs. As ψ is always greater than 1, for simplicity, we use ϵ to describe the distortion, where $1 + \epsilon = \psi$.

Landmark labeling (LL) is one of the widely-used approaches of ADOs and probably the simplest. The basic idea is to select a subset H of vertices as landmarks, and precompute the distances between each landmark $h \in H$ and all the vertices $u \in V$. When the distance between two vertices, u and v, is queried, query(u, v) answers the minimum $\delta(u, h) + \delta(h, v)$ over all the landmarks $h \in H$ as an estimation. We show the high-level idea of LL in Alg. 2.

Generally, the distortion of a query depends on how far the actual shortest paths are from their nearest landmark. If the actual shortest paths contain any landmark vertex, the query answer is equal to the true distance. Therefore, adding more landmarks can decrease the distortion, but it also needs more space and time to store and compute the index. In this paper, we propose to use C-BFS to optimize LL. In particular, we select clusters of vertices as landmarks instead of single vertices. As discussed, a C-BFS on a cluster of size $O(\log n)$ has costs (both time and space) asymptotically the same as running BFS on one vertex. In this way, we can select w times more landmarks with asymptotically the same cost as the plain LL. We note that the quality of the w landmarks in one cluster may not be as good as choosing them independently, since they are highly correlated. However, we experimentally observe that with the same memory limit, using cluster-BFS in landmark LL significantly improves the performance both in distortion and preprocessing time (see Sec. 5.3).

To apply C-BFS to LL, we need two subroutines: 1) selecting landmarks in clusters with low distortion,

and 2) answering the queries from the cluster distance vectors computed by C-BFS.

Selecting Landmarks in Clusters. The landmark selection is crucial in LL as it affects the query quality. A typical way is to prioritize vertices with high degrees [1, 26, 31]. We also employ this approach. In this step, we aim to identify clusters with a specified size $k = w$ and diameter d. Within these constraints, the selection of landmarks should prioritize vertices with higher degrees. To find a cluster, we first identify the vertex with the highest degree. Among all its $\lfloor d/2 \rfloor$-hop neighbors, we then select $k - 1$ additional vertices with the highest degrees. For instance, if $k = 64$ and $d = 2$, we select the vertex with the highest degree and 63 of its neighbors with the next highest degrees to create the first cluster. Once we select a cluster, we will mark all the vertices in the cluster, and will not select them again. We repeat this process until we select r clusters (giving rw landmarks in total). One can also select landmarks using other heuristics [27, 31]. After selecting clusters, we apply C-BFS to all selected clusters using Alg. 1. By doing this, we obtain the cluster distance vectors between each vertex and each cluster.

Answering Queries with Clustered Landmarks. To extend the original landmark idea to work with C-BFS, we need to show how to use the cluster distance vectors to answer the queries. A query answers the shortest distances between two vertices through any landmark. When all landmarks are independent vertices, $query(u, v)$ returns the smallest $\delta(u, l) + \delta(l, v)$ over all landmark vertices $l \in L$. In our case, the landmarks are grouped into clusters. Therefore, we first compute, within each cluster S, the smallest value $\delta(u, s) + \delta(s, v)$ for all $s \in S$. Then we will take the minimum among all clusters.

The problem boils down to finding $\delta(u, s) + \delta(s, v)$ for all sources s in a given cluster S. Recall that for each cluster S, both vertices u and v have obtained their cluster distance vector from C-BFS, denoted as $\langle S_u[0..d], \Delta_u \rangle$ and $\langle S_v[0..d], \Delta_v \rangle$. The possible shortest distances passing through S is in the range $[\Delta_u + \Delta_v, \Delta_u + \Delta_v + 2d]$. For two bit-subset, $S_u[i]$ and $S_v[j]$, if their intersection is not empty, it means there is a path connecting u and v with distance $\Delta_u + \Delta_v + i + j$ passing through any source vertex in the intersection. We check the intersections from the lowest possible distances (e.g., $S_u[0] \cap S_v[0]$) to higher possible distances, until we find a nonempty intersection, and return their distance sum. The complexity of the query grows in a quadratic manner as d grows. For the simplest case, $d = 2$, we only need to check three intersections for each cluster to get the answer.

We have shown another optimization, bidirectional

searching, for answering queries that can reduce distortion without much more overhead in querying time. Since this optimization is independent to C-BFS itself, due to the page limit, we put the description and experiments in our full version paper.

5 Experiments

Setup. We run our experiments on a 96-core (192 hyperthreads) machine with four Intel Xeon Gold 6252 CPUs and 1.5 TB of main memory. We implemented all algorithms in C++ using ParlayLib [7] for fork-join parallelism and parallel primitives (e.g., sorting). We use `numactl -i all` for parallel tests to interleave the memory pages across CPUs in a round-robin fashion.

We tested 18 undirected graphs, which are either social or web graphs with low diameters. Graph information is given in Tab. 2. All graphs are from commonly used open-source graph datasets [13, 14, 24, 36]. When comparing the average running times or speedups across all the graphs, we use the geometric mean.

Baseline Algorithms. We compare our algorithm to two existing implementations: 1) ***Ligra***, the parallel BFS in the Ligra [38] library (only using thread-level parallelism), and 2) ***AIY***, which is the C-BFS component from Akiba et al. [2] (sequential, only using bit-level parallelism). We also compared AIY for the 2-hop distance oracle as one of the applications.

5.1 Microbenchmarks for Cluster-BFS

We start with testing our cluster-BFS (C-BFS) as a building block. We first test the simplest case where $d = 2$ and $k = w = 64$, where each cluster is a star (a vertex and its up to 63 neighbors). We present a detailed experimental study for this simple case because one of our baselines, AIY, only supports sources as star-shaped clusters. Another reason is that in previous work (as well as new results in this paper), we observed that using $d = 2$ gives the best overall performance for the two applications discussed in this paper. We present some studies about varying d at the end of this subsection.

Recall that our new C-BFS benefits from two aspects: 1) using bit-level parallelism with the idea of clustering to compute the results from $O(w)$ sources simultaneously, and 2) using thread-level parallelism and known parallel techniques for optimizing BFS (e.g., directional optimization). In our test, we choose ten different clusters and report the average running time of them, as well as the plain sequential BFS as the simplest baseline. The plain sequential BFS processes all 64 sources independently, and the running time is in the column "Seq-BFS time" in Tab. 2. Our final running time of C-BFS using all techniques is provided in

Dataset	Graph Information			Seq-BFS Time(s)	Related Work		Parallel C-BFS		Self-Speedup	
	n	m	Notes		AIY	Ligra	Final	Time(s)	C-BFS	Ligra
EP	75.9K	811K	Epinions1 [3]	0.18	20.6×	4.02×	102×	0.002	4.39×	9.44×
SLDT	77.4K	938K	Slashdot [25]	0.21	18.8×	3.91×	94.1×	0.002	3.47×	9.55×
DBLP	317K	2.10M	DBLP [44]	0.77	20.3×	6.22×	183×	0.004	10.2×	17.1×
YT	1.13M	5.98M	com-youtube [44]	3.30	22.9×	17.8×	445×	0.007	20.6×	31.6×
SK	1.69M	22.2M	skitter [15, 36]	6.56	21.0×	30.4×	496×	0.013	26.7×	33.1×
IN04	1.38M	27.6M	in_2004 [13, 14]	4.08	20.9×	4.00×	171×	0.024	10.1×	17.8×
LJ	4.85M	85.7M	soc-LiveJournal1 [3]	39.8	23.7×	61.8×	1017×	0.039	47.7×	53.9×
HW	1.07M	112M	hollywood_2009 [14, 36]	18.7	20.9×	89.7×	928×	0.020	32.4×	48.7×
FBUU	58.8M	184M	socfb-uci-uni [34, 36, 41]	268	32.0×	49.6×	973×	0.276	54.4×	52.8×
FBKN	59.2M	185M	socfb-konect [34, 36, 41]	176	27.9×	38.8×	712×	0.247	53.1×	51.8×
OK	3.07M	234M	com-orkut [44]	61.6	19.8×	102×	1119×	0.055	49.0×	65.4×
INDO	7.41M	301M	indochina [11, 13, 36]	38.8	21.9×	12.4×	452×	0.086	25.7×	35.9×
EU	11.3M	521M	eu-2015-host [12–14]	119	23.9×	26.6×	821×	0.145	18.5×	41.3×
UK	18.5M	523M	uk-2002 [13, 14]	91.8	22.7×	30.7×	687×	0.134	42.1×	46.7×
AR	22.7M	1.11B	arabic [13, 14]	147	22.5×	10.7×	461×	0.319	18.0×	33.8×
TW	41.7M	2.41B	Twitter [23]	861	20.6×	157×	856×	1.006	56.3×	60.2×
FT	65.6M	3.61B	Friendster [44]	2084	20.4×	187×	813×	2.563	59.4×	64.6×
SD	89.2M	3.88B	sd_arc [29]	1898	25.0×	80.3×	945×	2.008	55.7×	62.5×
GeoMean				32.0	22.4×	27.0×	500×	0.064	24.8×	35.4×

Table 2: Tested graphs and microbenchmarks on different BFS algorithms from a cluster of vertices with size 64. The numbers followed by '×' are speedups, higher is better. Others are running time, lower is better. The columns "AIY", "Ligra" in related work and "Final" show the speedup over the "Seq-BFS". "AIY" is referred to sequential C-BFS from [1], "Ligra" is referred to parallel single BFS [38], and "Final" is referred to our parallel C-BFS. The "self-speedup" is the speedup running the algorithm in parallel over running it in sequential.
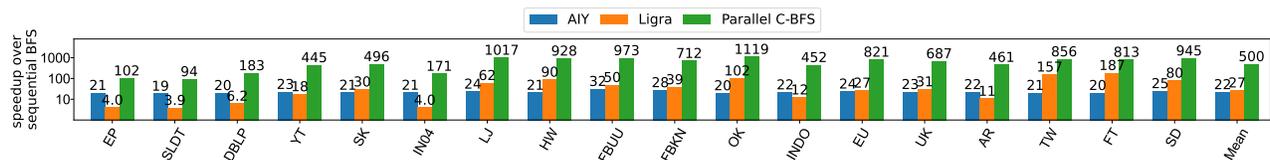

Figure 3: Speedup of parallel Ligra BFSs and parallel C-BFS over the standard sequential BFS on cluster with size 64. y-axis is the speedup over sequential regular BFS in log-scale, higher is better. Each group of bars represents a graph, except the last group, which represents the average across all graphs. The numbers on the bar are the speedup of parallel algorithms over the standard sequential algorithm.

the column "Par-Time (s)" in Tab. 2. To evaluate the performance gain by both techniques, we compared C-BFS with both Ligra and AIY. AIY only supports C-BFS on star-shaped clusters. The column "AIY" and "Ligra" in Tab. 2 provide the speedups over the plain sequential BFS. To better illustrate the results, we show the speedups relative to the plain version "Seq-BFS" in Fig. 3. Essentially, the column "AIY" means the speedup that can be achieved by applying bit-level parallelism on a cluster-BFS in the sequential setting. Similarly, the column "Ligra" provides the speedup that can be achieved by applying thread-level parallelism for running $k = 64$ regular (non-cluster) BFS.

As shown in the column "AIY", using clusters and bit-parallelism gets up to 18.8–32.0× improvement, which is uniform on different graphs. Note that here, we have $k = 64$ sources, so the maximum speedup can be 64×. Since C-BFS is more complicated than the plain BFS, there are some constant overheads, resulting in an

average 22.4× speedup in a sequential setting.

For applying thread-level parallelism, the improvement on different graphs varies greatly, from 3.91× (on smallest graphs) to up to 187× on a 96-core machine with hyperthreads. The benefit on certain graphs (e.g., OK, TW, and FT) is significant, which is over 100×. One reason for the difference in improvement is the directional optimization. On some dense graphs, the backward step can be applied across many of the rounds and thus save significant work (see our full version paper for more details). Another reason is that there is more parallelism available on larger graphs. This is consistent with the observations in prior work [6, 38]. On average, effectively utilizing parallelism gives 27.0× speedup over plain sequential BFS.

The columns "Par-Time(s)" and "Final" show our parallel C-BFS running time and overall improvement over "Seq-BFS". Our algorithm combines the strengths of both bit-level and thread-level parallelism. Our so-
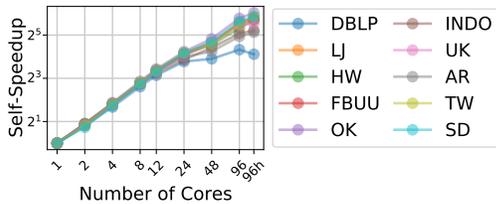
Figure 4: The scalability curve on different number of processors for C-BFS. The y-axis is the self speedup. The C-BFS running on one core is always 1. The x-axis is the number of cores. 96h represents 96 cores with hyperthreads.

lution is always better than any of the baselines. Compared to the plain sequential baseline, the improvement is 94.1–1119×, and 500× on average.

For comparing the techniques and improvements of all baselines, we show a summary figure in Fig. 1. The time and speedup numbers are average on all tested graphs. Compared to Ligra, our algorithm improves the performance by 18.5× by utilizing clusters and bit-level parallelism. Compared to AIY, our algorithm improves the performance by 22.3× by utilizing thread-level parallelism. This indicates that our combination of bit- and thread-level parallelism works very well in synergy. Each of them still (almost) fully contributes to the performance, achieving the same level of improvement as when used independently. Therefore, we believe our work on an efficient implementation combining thread- and bit-level parallelism fills the gap in the existing study of both C-BFS and parallel BFS.

Self-relative Speedup and Scalability. In addition to the aforementioned set of baselines, we further tested C-BFS in the sequential setting to study the self-relative speedup (in column "Self-Spd."). The speedup numbers are from 9.44× (on smallest graphs) to more than 40× (on most large graphs). In summary, the self-speedup of applying thread-level parallelism is 35.4× on average.

In addition to the overall self-relative speedup on 96 cores with hyperthreads, the scalability curve, as shown in Fig. 4, presents the self-relative speedup results across different number of cores. The curve demonstrates that our algorithm achieves nearly linear speedup for most of the graphs, indicating efficient parallel scalability. One exception is the DBLP graph, which deviates from this pattern due to its smaller size.

Influence of Cluster Diameter d in C-BFS. There are no existing implementations supporting C-BFS with general d. Recall that our C-BFS supports general clusters with diameter d instead of star-shaped clusters (a vertex and its neighbors, where $d = 2$). As shown in Thm. 3.1, the work is proportional to d. We tested different d from 2 to 6 on all the graphs. We choose 10 representative graphs and show the C-BFS running time on clusters with different d in Fig. 5. The full
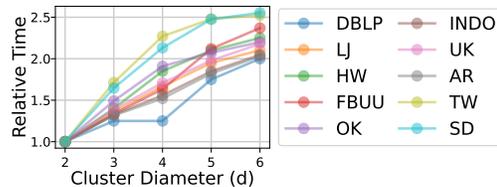

Figure 5: The running time of C-BFS on various cluster diameter d. The y-axis shows the relative running time over $d = 2$. The x-axis shows the cluster diameter d.

running time is shown in Tab. 4 in Sec. 5.4. The running time increases as d grows. A large d allows for better flexibility for the shape of the cluster (the vertices can be further from each other), but significantly affects the performance. In Sec. 5.4, we will further show that using a large d in LL incurs overhead in space and time, and therefore, using $d = 2$ is almost always more effective in applications.

5.2 2-Hop Distance Oracle

As mentioned in Sec. 4, we can replace the C-BFS in Akiba et al. to accelerate their algorithm for an EDO. Due to the page limit, we present the results in our full version paper. To do this, we also need to parallelize the other parts in their algorithm. In a nutshell, our algorithm with thread-level parallelism accelerates their algorithm by 9–36×, and can also process much larger graphs than they can do.

5.3 Approximate Landmark Labeling

We now show how C-BFS can significantly improve the landmark labeling (LL) approach with respect to both running time and accuracy. In general, more landmarks will lead to better accuracy for the distance queries, as the landmarks are more likely to be on or close to the shortest path between the queried vertices. Hence, by using the clusters as landmarks, we can drastically increase the number of landmarks by a factor of w with $O(d)$ overhead in time and space. For simplicity, we start by considering clusters with diameter $d = 2$, and later discuss clusters with $d > 2$. Following the optimization mentioned in Sec. 4.2, we use each cluster as a vertex and its $w - 1$ neighbors, and only store a distance and d bit-subsets in the cluster distance vector.

We study the effectiveness of our approach by comparing our C-BFS-based LL with a standard solution where each landmark is one vertex. We limit the total memory usage for both algorithms to with a parameter of t bytes per vertex for each graph, and construct an LL-based index within this memory budget. For C-BFS, memory usage per cluster includes the distance (1 byte) and d bit-subsets ($w/8$ bytes each) per vertex, while the memory usage for a regular LL is one byte (the distance)
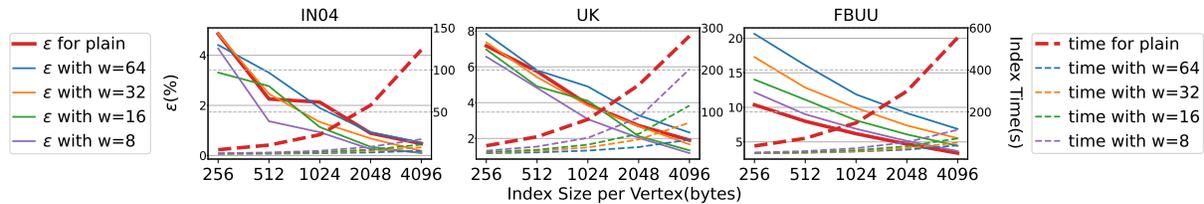
Figure 6: Tradeoffs between index size and distortion/construction time The x-axis is the memory limits per vertex in bytes, and is in log-scale. The y-axis on the left shows the $(1+\epsilon)$ distortion. The y-axis on the right shows the preprocessing time. For both preprocessing time and distortion, lower is better. For the algorithms compared here, 'plain' is the regular LL, others are the C-BFS-based LL that choose clusters with size w as landmarks.

Data	Index Time (s)			ϵ (%)		
	Plain	$w=64$	$w=8$	Plain	$w=64$	$w=8$
EP	1.26	0.02	0.08	0.4	0.1	0.1
SLDT	1.15	0.02	0.07	0.7	0.1	0.1
DBLP	3.57	0.08	0.25	2.5	2.2	1.0
YT	9.22	0.23	0.59	0.3	0.3	0.1
SK	13.4	0.55	1.77	1.4	0.7	0.4
IN04	20.0	0.96	3.88	2.1	1.9	0.9
LJ	36.2	1.72	5.63	5.0	4.3	3.5
HW	12.4	0.93	4.10	10.6	5.6	7.1
FBUU	138	11.3	27.0	6.2	11.9	6.9
FBKN	127	10.5	24.9	6.2	11.9	6.9
OK	26.3	2.87	10.1	8.7	7.7	7.3
INDO	83.2	5.44	29.8	3.1	1.5	1.3
EU	87.3	7.01	34.9	2.6	1.3	1.7
UK	80.4	8.28	38.8	3.9	4.9	3.1
AR	148	17.6	86.8	2.6	4.0	2.2
TW	112	31.0	99.3	1.5	1.4	1.1
FT	251	61.1	193	16.8	12.4	12.8
SD	318	75.6	255	0.6	0.3	0.3

Table 3: The index construction time, $(1+\epsilon)$ distortion, and query time for ADO based on landmark labeling. The "Plain" is the plain LL algorithm that each landmark is a single vertex. The "$w=64$" and "$w=8$" C-BFS-based LL that landmarks are in clusters with size w. The memory budget is 1024 bytes per vertex. For both index time and ϵ, lower is better.

per vertex. For example, C-BFS with $w=64$ and $d=2$ needs 17 bytes to store a cluster distance vector, and C-BFS with $w=8$ and $d=2$ only needs 3 bytes. Thus, the memory usage depends on the word size and number of clusters we choose. For fair comparisons, we fix the memory usage per vertex in the index for different baselines. With the same memory budget, a larger w results in fewer (independent) clusters, typically allowing faster preprocessing. As discussed in Sec. 4.2, the landmarks in the same cluster are highly correlated to each other, and may not bring the same benefit as independent ones. Thus, a larger w may also result in less accuracy.

In this experiment, we tested on different w from $\{8, 16, 32, 64\}$ and $d=2$. In Tab. 3, we show the two extremes of using $w=64$ and $w=8$. Due to page limits, we put the full information of all tested w in our full version paper.

We show the full result of using memory limit as $t = 1024$ bytes per vertex in Tab. 3, where "plain" refers to using simple parallel BFS, "$w=64$" is to use C-BFS with $w=64$ (17 bytes per cluster), and "$w=8$" is C-BFS with $w=8$ (3 bytes per cluster). With the memory limit of 1024 bytes per vertex, we can choose 1024 landmarks for regular LL, 341 clusters for CC8, or 60 clusters for CC64. For regular LL, each landmarks are chosen based on prioritizing the high-degree vertices. We compare the index time and the *error* ϵ shown in percentage. It means that the ADO has $(1+\epsilon)$ distortion (defined in Sec. 4.2). We take the average of 100,000 pairs to estimate the error on all the graphs, except for the largest two graphs FT and SD. We only compute the distance of 10,000 pairs on these two graphs, since generating the ground truth is expensive on large graphs.

On all 18 tested graphs, C-BFS with $w=64$ always gives a lower running time. When $w=8$, the running time is 2.4–5.5× higher than $w=64$, but still mostly faster than the plain version. With the same memory budget, $w=64$ roughly processes 5.68× fewer clusters than $w=8$. Similarly, comparing $w=64$ and the plain version, the number of (clustered) BFSs performed by $w=64$ is 17× fewer than the number of (single) BFSs by regular LL. The running time can be up to 53× faster, but on average, it is around 10×—each C-BFS is still more expensive than a single BFS, but the numbers indicate that the overhead is small.

Regarding distortions, $w=8$ generally gives better accuracy than $w=64$, but it can also be worse in several instances. That is because $w=64$ selects more landmarks than $w=8$ (3840 vs. 2728) but fewer independent clusters (60 vs. 341). Therefore, the loss of using fewer clusters may or may not be compensated by more (correlated) landmarks. Empirically, the results still suggest that $w=8$ gives overall better accuracy, as it is more accurate on 11 out of 18 tested graphs. Both $w=8$ and $w=64$ are more accurate than the plain version on at least 16 out of 18 graphs. This indicates that the increased number of landmarks, although less independent, still positively affects the accuracy on most of the 18 scale-free networks we tested here.

It is worth noting that the query times for all graphs are similar, and we show the average (geometric mean) query time at the last line in Tab. 3. This is because the query time is completely determined by the number of landmarks, instead of the graph size.

The experimental results suggest that, if the primary objective is to reduce preprocessing time, using C-BFS with $w = 64$ will always give a more efficient version than the plain version. The precision is also superior in most cases. When achieving better precision is the top priority, C-BFS with a smaller w can significantly reduce the distortion. When using $w = 8$, the distortion is almost always better than both $w = 64$ and the plain version, and the running time also improves over the plain version on most of the graphs.

We also show the trade-off between preprocessing time and distortion in Fig. 6 on three representative graphs, particularly including those that $w = 8$ and 64 may perform poorly on. We include results using $w = 16$ and $w = 32$, with different memory budgets (t bytes per vertex, shown in the x-axis). In general, it is still true that large w results in faster preprocessing but larger distortion. The red line shows the baseline of regular LL. For both preprocessing time and distortion, lower is better. Overall, using $w = 16$ or $w = 32$ provides a compromise for both time and distortion, and can be a more stable choice across all graphs.

For both preprocessing time and distortion, using C-BFS provides a significant improvement on almost all graphs. This verifies the effectiveness of our C-BFS to achieve a practical distance oracle.

C-BFS with Diameter $d > 2$. Unlike the (sequential) AIY algorithm that can only apply to $d = 2$ case (a vertex and its neighbors), our C-BFS is general and works for any given d. Hence, it is interesting to understand how the performance and quality are affected by varying d. Note that larger d gives us flexibility in selecting the sources. It can generally reduce the correlations between the sources, and even sometimes enable a larger k if no vertices in the graph have large degrees (although this is rare in the scale-free networks tested in this paper). Therefore, we tested the time and distortion of LL on clusters with larger d and present the results in Sec. 5.4. The takeaway is that, although larger d may provide higher source quality, the space and time consumption are also linearly proportional to d. Hence, given the same memory limit, choosing clusters with $d > 2$ does not help with decreasing distortion (other than the graph OK, where $d = 3$ improves the distortion over $d = 2$ by 3%). However, generally, the faster running time is due to fewer clusters that can be selected for the same memory budget.

Data	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
EP	0.002	0.002	0.003	0.003	0.003
SLDT	0.002	0.002	0.002	0.002	0.003
DBLP	0.004	0.005	0.005	0.007	0.008
YT	0.007	0.011	0.012	0.014	0.015
SK	0.013	0.017	0.020	0.023	0.026
IN04	0.024	0.027	0.029	0.032	0.033
LJ	0.039	0.053	0.065	0.076	0.082
HW	0.020	0.028	0.037	0.042	0.045
FBUU	0.276	0.368	0.451	0.585	0.654
FBKN	0.247	0.318	0.388	0.501	0.564
OK	0.055	0.082	0.105	0.114	0.121
INDO	0.086	0.114	0.134	0.159	0.176
EU	0.145	0.195	0.239	0.284	0.290
UK	0.134	0.186	0.229	0.265	0.292
AR	0.319	0.420	0.485	0.581	0.647
TW	1.006	1.723	2.284	2.494	2.539
FT	2.563	4.412	5.880	6.594	6.815
SD	2.008	3.305	4.280	4.970	5.128
GeoMean	0.064	0.086	0.104	0.120	0.131

Table 4: The parallel C-BFS time (seconds) for one cluster with size 64 on different cluster diameter d.

5.4 Further Study on the $d > 2$ Case in Approximate Distance Oracle

Here, we provide additional information for the $d > 2$ case for the landmark labeling, and continue the discussion from Sec. 5.3. Recall d is the diameter of clusters. We first study its influence in C-BFS. Then, we study the influence of d in the application Landmark Labeling introduced in in Sec. 4.2.

The Performance of C-BFS on Different d. According to Thm. 3.1, the work of C-BFS is proportional to d. In order to know how to choose proper d in different applications, we need to first know how different d affect the running time of C-BFS in practice. We tested the running time of C-BFS with clusters of different diameter d. The results are shown in Tab. 4. The running time increases as d grows larger, as shown in Tab. 4, and space usage ($O(d)$ space to store the distances of a cluster to a vertex) also increases. When applying C-BFS to other applications, we need to weigh the benefits brought by more general clusters and the overhead on time and space costs. Tab. 4 provides a reference for overhead on running time.

The Performance of Different d on Landmark Labeling. Our C-BFS is the first implementation of C-BFS that supports general clusters. It gives us a chance to study the performance and quality of LL for clusters with larger d. Clusters with larger d have fewer correlations for vertices in the same cluster, but the computational cost for C-BFS also increases. We are interested in whether it is worth using clusters with larger d in the LL application.

We show the full result of using memory limit as

Data	Construction Time (s)				Distortion ϵ (%)			
	Plain	$d=2$	$d=3$	$d=4$	Plain	$d=2$	$d=3$	$d=4$
EP	1.26	0.03	0.02	0.02	0.4	0.1	0.2	0.2
SLDT	1.15	0.02	0.02	0.02	0.7	0.1	0.4	0.6
DBLP	3.57	0.09	0.07	0.06	2.5	2.2	3.4	4.0
YT	9.22	0.23	0.19	0.17	0.3	0.3	0.6	0.8
SK	13.4	0.58	0.41	0.37	1.4	0.7	1.5	2.1
IN04	20.0	1.06	0.70	0.58	2.1	1.9	2.4	2.6
LJ	36.2	1.79	1.41	1.26	5.0	4.3	5.5	6.0
HW	12.4	0.99	0.75	0.63	10.6	5.6	6.5	7.0
FBUU	138	11.7	9.31	8.26	6.2	11.9	13.4	15.9
FBKN	127	10.8	8.82	7.52	6.2	11.9	13.5	16.0
OK	26.3	3.05	2.65	2.32	8.7	7.7	7.5	7.6
INDO	83.2	6.09	3.92	3.40	3.1	1.5	2.2	2.6
EU	87.3	8.00	6.33	5.55	2.6	1.3	1.9	2.2
UK	80.4	9.06	6.63	5.90	3.9	4.9	5.3	5.6
AR	148	19.4	13.4	11.7	2.6	4.0	6.8	7.1
TW	112	31.0	27.8	23.9	1.5	1.4	1.5	1.8
FT	251	61.1	52.0	45.8	16.8	12.4	13.7	14.6
SD	318	75.6	61.2	53.1	0.6	0.3	0.8	0.9

Table 5: The Approximate Landmark Labeling Time and Distortion for cluster with size 64 and memory limits 1024 bytes per vertex on different cluster diameter d.

$t = 1024$ bytes per vertex and word size $w = 64$ in Tab. 5, where "plain" refers to using simple parallel BFS (1 bytes per landmark), "$d = 2$" is to use C-BFS with $d = 2$ (17 bytes per cluster), "$d = 3$" is C-BFS with $d = 3$ (25 bytes per cluster) and "$d = 4$" is C-BFS with $d = 4$ (33 bytes per cluster). With memory limits 1024 bytes per vertex, we can choose 1024 landmarks for regular LL, 60 clusters for $d = 2$, 40 clusters for $d = 3$, and 31 clusters for $d = 4$. Landmarks are chosen based on prioritizing the high-degree vertices, similar to the setting previously.

The construction time is reversely proportional to d, since larger d leads to fewer sources and clusters. However, for distortion, other than the graph OK, larger d always leads to lower accuracy. For the graph OK, the improvement is only about 3%. In conclusion, since for distance oracles, generally the space usage and accuracy are the most crucial—given the same memory budget, choosing clusters from diameter $d > 2$ does not help with accuracy on most graphs. However, since our C-BFS algorithm is highly parallel and efficient, it provides opportunities for researchers in the future to study other applications on whether more general sources with $d > 2$ can be more effective than star-size ones with $d = 2$.

6 Related Work

This paper mainly focuses on scale-free (small-diameter) networks, and we refer the audience to an excellent survey [4] of algorithms for large-diameter graphs (e.g., road networks). In this paper, we discuss the approaches based on landmark labeling, and here, we re-

view other approaches for distance queries. We first review the approximate solutions. The concept of approximate distance oracles (ADOs) was proposed by Thorup and Zwick [40], and has later been theoretically studied in dozens of papers. Practically, papers [18, 22, 31] discuss how to select the best "landmarks" for these type of sketch-based solutions, and showed that degrees or betweenesses are reasonable metrics. Other solutions include embedding-based solutions [28, 33, 37, 46] (embedding graph metric into other simpler ones such as Euclidean space), tree-based approaches [9, 45], and some recent attempts using deep learning [35]. Most of these approaches are more complicated than the landmark-based labeling mentioned in this paper.

Regarding the exact distance queries, Pruned Landmark Labeling (PLL) [1] is among the latest solutions for scale-free networks. Li et al. [26] showed another implementation, but they did not release their code, so we cannot compare their running time with ours. Other techniques, such as contraction hierarchies (CH) [19] and transit nodes routing [5], focus more on large-diameter graphs like road networks.

7 Conclusion

In this paper, we present parallel implementations for cluster-BFS, which runs BFS from a cluster of vertices with diameter d. Our algorithm is work-efficient in theory, and also leads to high parallelism on low-diameter graphs as we tested in the experiments. We employ both bit-level and thread-level parallelism to optimize the performance. Both of them lead to significant speedup. Especially, we observed that bit-level and thread-level parallelism work well in synergy. We also show that the combination of the techniques also leads to performance improvement in two applications in distance oracles in multiple measurements of preprocessing time and accuracy, and allows our implementation to scale to much larger graphs than a sequential algorithm. Besides, our C-BFS is the first implementation that supports general clusters with diameter d instead of star-shaped clusters, which give us a chance to study the benifits and overhead of choosing clusters with larger d in C-BFS and in its applications.

Acknowledgments

References

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 349–360. 1, 2, 3, 7, 8, 12

[2] Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. 2012. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *ACM International Conference on Extending Database Technology*. 144–155. 2, 6, 7

[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 44–54. 8

[4] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias M üller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2016. Route Planning in Transportation Networks. In *Algorithm Engineering - Selected Results and Surveys*, Lasse Kliemann and Peter Sanders (Eds.). Lecture Notes in Computer Science, Vol. 9220. 19–80. https://doi.org/10.1007/978-3-319-49487-6_2 12

[5] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. 2007. Fast routing in road networks with transit nodes. *Science* 316, 5824 (2007), 566–566. 12

[6] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 1–10. 1, 3, 8

[7] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509. 7

[8] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102. 2

[9] Guy E. Blelloch, Yan Gu, and Yihan Sun. 2017. Efficient Construction of Probabilistic Tree Embeddings. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. 12

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748. 2, 3

[11] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *spe* 34, 8 (2004), 711–726. 8

[12] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2014. BUbiNG: Massive Crawling for the Masses. In *www*. International World Wide Web Conferences Steering Committee, 227–228. 8

[13] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *www*. 587–596. 7, 8

[14] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *www*. ACM Press, Manhattan, USA, 595–601. 7, 8

[15] CAIDA. [n.d.]. Skitter. http://caida.org/tools/measurement/skitter/. 8

[16] Timothy M Chan. 2012. All-pairs shortest paths for unweighted undirected graphs in o (mn) time. *talg* 8, 4 (2012), 1–17. 1, 3

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press. 2

[18] Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. 2010. A sketch-based distance oracle for web-scale graphs. In *wsdm*. 401–410. 12

[19] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46, 3 (2012), 388–404. 12

[20] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60. 3

[21] Yan Gu, Omar Obeya, and Julian Shun. 2021. Parallel In-Place Algorithms: Theory and Practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 114–128. 2

[22] Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. 2010. Fast and accurate estimation of shortest paths in large graphs. In *ACM International Conference on Information and Knowledge Management*. 499–508. 6, 12

[23] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600. 8

[24] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. (2014). 7

[25] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *im* 6, 1 (2009), 29–123. 8

[26] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling distance labeling on small-world networks. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1060–1077. 7, 12

[27] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment* 11, 4 (2017), 445–457. 7

[28] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. Grecs: Graph encryption for approximate shortest distance queries. In *ACM SIGSAC Conference on Computer and Communications Security*. 504–517. 12

[29] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web Data Commons — Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph. 8

[30] Gary L Miller, Richard Peng, and Shen Chen Xu. 2013. Parallel graph decompositions using random shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1

[31] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *ACM International Conference on Information and Knowledge Management*. Association for Computing Machinery. 7, 12

[32] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. 2012. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE Transactions on Knowledge and Data Engineering* 26, 1 (2012), 55–68. 6

[33] Satish Rao. 1999. Small distortion and volume preserving embeddings for planar and Euclidean metrics. In *ACM Symposium on Computational Geometry (SoCG)*. 300–306. 12

[34] Veronica Red, Eric D Kelsic, Peter J Mucha, and Mason A Porter. 2011. Comparing community structure to characteristics in online collegiate social networks. *SIAM review* 53, 3 (2011), 526–543. 8

[35] Fatemeh Salehi Rizi, Joerg Schloetterer, and Michael Granitzer. 2018. Shortest path distance approximation using deep learning techniques. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 1007–1014. 12

[36] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *aaai*. https://networkrepository.com 7, 8

[37] Yuval Shavitt and Tomer Tankel. 2008. Hyperbolic embedding of internet graph for distance estimation and overlay construction. *IEEE/ACM Transactions on Networking* 16, 1 (2008), 25–36. 12

[38] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 135–146. 1, 2, 3, 7, 8

[39] Liying Tang and Mark Crovella. 2003. Virtual landmarks for the internet. In *ACM SIGCOMM conference on Internet measurement*. 143–152. 6

[40] Mikkel Thorup and Uri Zwick. 2005. Approximate distance oracles. *J. ACM* 52, 1 (2005), 1–24. 12

[41] Amanda L Traud, Peter J Mucha, and Mason A Porter. 2012. Social structure of facebook networks. *sma* 391, 16 (2012), 4165–4180. 8

[42] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Ba ñuelos, Jaak Vilo, and Marlon Dumas. 2011. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *ACM International Conference on Information and Knowledge Management*. 1785–1794. 6

[43] Monique V Vieira, Bruno M Fonseca, Rodrigo Damazio, Paulo B Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. 2007. Efficient search ranking in social networks. In *ACM International Conference on Information and Knowledge Management*. 563–572. 6

[44] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213. 8

[45] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2023. LiteHST: A Tree Embedding based Method for Similarity Search. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 1, 1 (2023), 1–26. 12

[46] Xiaohan Zhao and Haitao Zheng. 2010. Orion: shortest path estimation for large social graphs. In *Workshop on Online social networks*. 12