An Evaluation of Parallel Eccentricity Estimation Algorithms on Undirected Real-World Graphs

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

ABSTRACT

This paper presents efficient shared-memory parallel implementations and the first comprehensive experimental study of graph eccentricity estimation algorithms in the literature. The implementations include (1) a simple algorithm based on executing two-pass breadth-first searches from a sample of vertices, (2) algorithms with sub-quadratic worst-case running time for sparse graphs and non-trivial approximation guarantees that execute breadth-first searches from a carefully chosen set of vertices, (3) algorithms based on probabilistic counters, and (4) a well-known 2-approximation algorithm that executes one breadth-first search per connected component. Our experiments on large undirected real-world graphs show that the algorithm based on two-pass breadth-first searches works surprisingly well, outperforming the other algorithms in terms of running time and/or accuracy by up to orders of magnitude. The high accuracy, efficiency, and parallelism of our best implementation allows the fast generation of eccentricity estimates for large graphs, which are useful in many applications arising in large-scale network analysis.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Experimentation, Measurement, Performance

1 Introduction

The *eccentricity* of a vertex in a graph is defined to be the largest distance from the vertex to any other reachable vertex. Computing the eccentricities of vertices in a graph is a well-studied problem due to its many applications in the analysis of networks (see, e.g., [49]). For example, the eccentricity of a vertex can be used to compute its *eccentricity centrality* score, defined to be the inverse of its eccentricity. This can be used to measure a vertex's accessibility in the graph (i.e., "central" vertices tend to have a higher eccentricity centrality score). Eccentricities have been used to study characteristics of routing networks [32, 33]—a low average eccentricity indicates that the devices are mostly near each other, whereas a high average eccentricity implies that devices are spread out. A vertex's eccentricity value in a routing network could be indicative of a device's worst-case response time [46]. Graph eccentricities also find applications in biological networks [38], location analysis [12], and state-transition graphs of finite-state machines that arise in formal hardware verification [35]. The eccentricity distributions of graphs have been analyzed to classify the roles of vertices [28, 48, 46]. In addition, for certain algorithms on matrices, it is desirable to identify a starting vertex with a certain eccentricity criteria (see, e.g., [26]).

KDD'15, August 10-13, 2015, Sydney, NSW, Australia.
© 2015 ACM. ISBN 978-1-4503-3664-2/15/08 ...$15.00.
DOI: http://dx.doi.org/10.1145/2783258.2783333.

The exact eccentricity of every vertex in an unweighted graph can be computed simply by executing a breadth-first search (BFS) starting from each vertex, taking a total of $O(mn)$ work. More generally, computing the graph eccentricities can be solved exactly using an all-pairs shortest paths (APSP) algorithm. However, these algorithms inherently require $\Omega(n^2)$ work (see, e.g., [51, 15, 50] and the references therein), which is prohibitive for large graphs. A natural question to ask is whether there exists sub-quadratic work algorithms for graph eccentricity. There have been several papers describing methods for efficiently approximating all of the eccentricities of a graph without resorting to APSP, while most other papers focus on efficiently computing or approximating the diameter of a graph. In this paper, we focus on the problem of approximating all eccentricities as this has more applications than the diameter problem. Due to the large sizes of current graphs, we are also interested in parallel solutions to the problem.

For a connected, undirected graph, it is well-known that a 2-approximation for all vertex eccentricities can be achieved by performing a BFS from an arbitrary vertex. As far as we know, the only algorithms with a sub-quadratic worst-case work complexity for sparse graphs that provide a better provable approximation guarantee for eccentricities are the ones by Roditty and Vassilevska Williams [40] and Chechik et al. [16]. Roditty and Vassilevska Williams [40] present an algorithm for undirected, unweighted graphs that generates eccentricity estimates $\hat{e}(v)$ for each vertex v, such that $(2/3)e(v) \leq \hat{e}(v) \leq (3/2)e(v)$, where $e(v)$ is the true eccentricity of v. The algorithm requires $O(m\sqrt{n \log n})$ work. Chechik et al. [16] describe an algorithm for undirected, weighted graphs that generates an estimate $\hat{e}(v)$ for each vertex v, such that $(3/5)e(v) \leq \hat{e}(v) \leq e(v)$, and requires $O((m \log m)^{3/2})$ work. *This paper presents the first empirical evaluation of parallel implementations of (variants of) these two algorithms.*

As for empirical work on eccentricity computation, Kang et al. [28] present HADI, a parallel MapReduce algorithm based on the Flajolet-Martin counters for estimating the number of distinct elements in a multiset [23]. The algorithm is more general in that it can be used to estimate the neighborhood sizes of vertices [36]. Boldi et al. [10] present an improved algorithm for estimating neighborhood sizes using the HyperLogLog counters of Flajolet et al. [24], and their algorithm can be used for eccentricity estimation. This paper will study these two algorithms. We note that the problem of estimating the eccentricity distribution (i.e., estimate the counts of vertices with each eccentricity value) has also been studied [19, 46]; however, the techniques are not applicable to the more general problem of generating eccentricity estimates for all vertices.

In addition to the estimation algorithms described above, we also study a simple method of running two-pass BFS's from a sample of vertices simultaneously, which we refer to as k-BFS. *We show that k-BFS generates surprisingly good estimates with high efficiency, and describe bit-level optimizations to improve its performance.*

We give shared-memory parallel implementations of all of the algorithms using the recent Ligra graph processing framework [41], and empirically evaluate them on a variety of large-scale undirected, unweighted real-world graphs. *As far as we know, this is the first comprehensive comprehensive study comparing all of the different eccentricity algorithms in the literature.* For all of the algorithms, we present experiments showing their running time, accuracy, and parallel scalability on a modern multicore machine. For k-BFS and the algorithms based on probabilistic counters [36, 28, 10], we study the running time versus the accuracy by varying the number of BFS's or number of counters used. We show that k-BFS is much more accurate (at most 0.01% error on the real-world graphs for which we could compute the true eccentricities) for a given running time than the algorithms based on probabilistic counters. k-BFS is also much more accurate than the simple 2-approximation algorithm on the real-world graphs, although the 2-approximation algorithm achieves reasonable accuracy and is faster. Compared to the algorithms of Roditty and Vassilevska Williams [40] and Chechik et al. [16], k-BFS achieves comparable accuracy while being orders of magnitude faster. In addition, k-BFS is orders of magnitude faster than two exact parallel eccentricity algorithms that we implement, and achieves up to 38x parallel speedup on a 40-core machine with hyper-threading. Due to the efficiency, parallelism, and high accuracy of our implementation of k-BFS, we are able to use it to quickly generate eccentricity distribution plots for several of the largest publicly available real-world graphs, and produce estimates that are useful for other applications in graph analytics.

2 Preliminaries

We denote an unweighted graph by $G(V,E)$, where V is the set of vertices and E is the set of edges in the graph. The number of vertices in a graph is $n = |V|$, and the number of edges is $m = |E|$. The vertices are assumed to be indexed from 0 to $n-1$.

We use $d(v,w)$ to refer to the shortest distance from vertex v to vertex w in G ($d(v,w) = \infty$ if unreachable). For a set of vertices S, we define $d(v,S)$ to be $\max_{s \in S} d(v,s)$; in other words, it is the maximum distance from v to any vertex in S. We define $e(v) = \max_{w \in V | d(v,w) \neq \infty} d(v,w)$ to be the ***eccentricity*** of vertex v in G, $D = \max_{v \in V} e(v)$ to be the ***diameter*** of the graph, and $R = \min_{v \in V} e(v)$ to be the ***radius*** of the graph. We will use $\hat{e}(v)$ to refer to an estimate of the eccentricity of vertex v.

A ***compare-and-swap (CAS)*** is an atomic instruction supported on modern multicore machines that takes three arguments—a memory location (*loc*), an old value (*oldV*), and a new value (*newV*); if the value stored at *loc* is equal to *oldV* it atomically stores *newV* at *loc* and returns *true*, and otherwise it does not modify *loc* and returns *false*. An ***AtomicOR*** takes two arguments, a location x and a value y, performs a bitwise-OR of the value at location x and the value y, and stores the result in x. It can be implemented using a loop with a CAS until the result of the bitwise-OR is successfully stored at the location or until the result is equal to the value already at location x. Throughout the paper, we use the notation $\&x$ to denote the memory location of variable x, and "|" to denote the bitwise-OR operator.

Algorithms in this paper are analyzed in the work-depth model [7], where ***work*** is equal to the number of operations required and ***depth*** is equal to the number of time steps required. Concurrent reads and writes are allowed in the model, with which CAS can be simulated. We make the standard assumption that $\Theta(\log n)$ bits fit in a word.

We will use the basic parallel primitives, prefix sum and filter [7]. ***Prefix sum*** takes an array X of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus x = x$ for any x, and returns the array $(\perp, \perp \oplus X[0], \perp \oplus X[0] \oplus X[1], \ldots, \perp \oplus X[0] \oplus X[1] \oplus \ldots \oplus X[n-2])$, as well as the overall sum $\perp \oplus X[0] \oplus X[1] \oplus \ldots \oplus X[n-1]$. ***Filter*** takes an array X of length n and a predicate function f, and returns an array X' of length $n' \leq n$ containing the elements in $x \in X$ such that $f(a)$ returns true, in the same order that they appear in X. Filter can be implemented using prefix sum, and both require $O(n)$ work and $O(\log n)$ depth [7].

A ***breadth-first search (BFS)*** algorithm takes an unweighted graph $G(V,E)$ and a source vertex $r \in V$, and computes $d(r,v)$ for all vertices v reachable from r. A standard parallel implementation of BFS takes $O(m+n)$ work and $O(\min(n, D\log n))$ depth by proceeding in iterations and in iteration h visiting all vertices at distance h away from r [7].

A well-known 2-approximation algorithm for graph eccentricity works as follows: For each component in the graph, run a BFS from an arbitrary vertex and use the maximum distance found as the eccentricity estimate for all vertices in the component. Using the triangle inequality, the estimates $\hat{e}(v)$ for each vertex v can be shown to satisfy $(1/2)e(v) \leq \hat{e}(v) \leq 2e(v)$.

2.1 Ligra Framework

Our implementations are written using the recent Ligra shared-memory graph processing framework [41]. We choose to use Ligra because it is a high-level framework that allows graph traversal algorithms to be expressed easily, and implementations using Ligra have been shown to outperform other high-level graph processing frameworks. The Ligra framework itself is very lightweight, incurring minimal overheads compared to code written without outside of a high-level framework, and the simplicity of the implementations makes them easier to understand and more accessible. Many of the implementation ideas described in this paper can be applied to fully hand-written implementations of the algorithms as well. For the implementations in this paper, we found the performance overhead of the Ligra system itself to be at most 5% (usually much less) compared to equally-optimized code that was fully hand-written. Since the largest publicly-available real-world graphs fit in shared memory, we did not find it necessary to use distributed-memory frameworks. Furthermore, graph algorithms in shared-memory have been shown to be more efficient than distributed-memory on a per-core, per-dollar, and per-joule basis.

Ligra supplies a ***vertexSubset*** data structure used for representing a subset of the vertices, and provides two simple functions, one for mapping over vertices and one for mapping over edges. **VERTEXMAP** takes as input a vertexSubset U and a function F, and applies F to all vertices in U.[1] F can side-effect data structures associated with the vertices. **EDGEMAP** takes as input a graph $G(V,E)$, vertexSubset U, boolean update function F, and boolean conditional function C; it applies F to all edges $(u,v) \in E$ such that $u \in U$ and $C(v) = $ *true* (call this set of edges E_a), and returns a vertexSubset U' containing vertices v such that $(u,v) \in E_a$ and $F(u,v) = $ *true*. Again, F can side-effect data structures associ-

[1] This is actually a less general version of VERTEXMAP, which suffices for the implementations in this paper. The more general version of VERTEXMAP takes a boolean function F, and returns the vertices for which applying F returned *true*.

```
1: Dist = {∞, . . . , ∞}                                      ▷ ∞ indicates unexplored
2: procedure UPDATE($s$, $d$)
3:     return (CAS(&Dist[$d$], ∞ , Dist[$s$] + 1 ))            ▷ atomically visit neighbor
4: procedure COND($v$)
5:     return (Dist[$v$] == ∞)                                ▷ check if neighbor has been visited
6: procedure BFS($G$, $r$)                                    ▷ $r$ is the root
7:     Dist[$r$] = 0
8:     vertexSubset Frontier = {$r$}
9:     while (size(Frontier) > 0) do
10:        Frontier = EDGEMAP($G$, Frontier, UPDATE, COND)
11:    return Dist
```

Figure 1: Pseudocode for BFS in Ligra.

ated with the vertices. The programmer must ensure the parallel correctness of the functions passed to VERTEXMAP and EDGEMAP.

A key feature of Ligra that makes it efficient is that it has two implementations of EDGEMAP, a version for sparse input vertexSubsets that writes data from just the vertices in the input, and a version for dense input vertexSubsets that reads data from all vertices in the graph satisfying the conditional function. Ligra automatically switches between the two versions of EDGEMAP based on the size of the input vertexSubset. This optimization is very beneficial for graph traversal algorithms where the size of the active vertex set changes over time. The idea was first used in BFS by Beamer et al. [5]. Ligra also supports graph compression, which leads to improvements in space usage as well as performance [43]. We refer the reader to [41, 43] for implementation details of Ligra. Ligra compiles with either Cilk Plus or OpenMP.

BFS in Ligra. We describe how to implement BFS in Ligra (pseudocode shown in Figure 1). The distances of all vertices are initialized to ∞, indicating that they have not yet been visited (Line 1). The starting vertex r has its distance set to 0 (Line 7) and is placed on the initial frontier, which is represented as a vertexSubset (Line 8). An EDGEMAP is applied on the frontier vertices until the frontier becomes empty (Lines 9–10). The EDGEMAP uses an Update function that visits the unexplored neighbors by updating their distance values atomically using a compare-and-swap (Lines 2–3). The Cond function (Lines 4–5) simply checks if a vertex has been visited. Newly visited vertices in each iteration are placed on the next frontier. The algorithm terminates when the frontier becomes empty, as this means that all vertices reachable from r have been visited. The implementation can be shown to take $O(m+n)$ work and $O(\min(n, D\log n))$ depth, matching that of the standard parallel BFS algorithm.

2.2 Exact Eccentricity Algorithm

Takes and Kosters describe an algorithm for exact eccentricity computation, and show that it is faster than APSP in practice [46]. We describe their algorithm for undirected graphs, and refer to it as ***TK***. We assume a connected graph; otherwise, the algorithm is separately run on each connected component. The algorithm is based on repeatedly selecting a vertex, executing a BFS from it to compute its eccentricity, and using the result to bound the eccentricity of the remaining vertices with the following property: for all vertices $v, w \in V$, $\max(e(w) - d(w,v), d(w,v)) \leq e(v) \leq e(w) + d(w,v)$.

Each vertex v maintains a lower bound $e_L(v)$ and an upper bound $e_U(v)$ on its eccentricity. A set W of vertices is initialized to contain all vertices. The algorithm proceeds in rounds. In each round, a vertex $w \in W$ is selected and a BFS is executed from w. Then for all $v \in W$, $e_L(v)$ is updated to be $\max(e_L(v), e(w) - d(w,v), d(w,v))$, and $e_U(v)$ is updated to be $\min(e_U(v), e(w) + d(w,v))$. Afterward, vertices v in W where $e_L(v) = e_U(v)$ are removed, as their exact eccentricities have been determined. The algorithm terminates when W becomes empty. In the worst case, the overall work is $O(mn)$, however Takes and Kosters show that it is much lower in practice [46]. In each round, the vertex w to execute a BFS from can be selected using a heuristic, and the heuristic shown to work best is to alternate between a vertex with the highest $e_U(v)$ value and a vertex with the lowest $e_L(v)$ value [45].

This algorithm can easily be parallelized by executing each BFS in parallel, and performing the updates to the lower/upper bounds in parallel. Selecting the vertex to perform a BFS from can be done with a prefix sum computation over the e_U or e_L values. Removing vertices from W can be done with a parallel filter. We implement this algorithm in Ligra using the BFS procedure in Figure 1, and use it as a baseline to compare with the eccentricity estimation algorithms. A GPU implementation of a similar algorithm (for diameter computation) is described in [22].

3 k-BFS

This section describes ***k-BFS***, an eccentricity estimation algorithm that performs two phases of executing multiple BFS's simultaneously. The algorithm assumes an undirected, unweighted graph. We describe the algorithm for a single connected component; if the graph is not connected, then the algorithm is run on each component, and this will be discussed in more detail at the end of this section.

Conceptually, the algorithm is very simple. Define S to be an initial set of k randomly sampled vertices, and call each of these vertices a ***source***. The algorithm proceeds in two phases. The first phase computes $d(v,S)$ for all vertices $v \in V$ (the maximum distance from v to any vertex in S). This can be accomplished by performing a BFS from each source vertex, and keeping track of the current level of the BFS. $d(v,S)$ is then equal to the highest level of a BFS that visits v. Then define S' to be the k vertices with the largest $d(v,S)$ values. The second phase of the algorithm computes eccentricity estimates $\hat{e}(v)$ for all v, defined to be the maximum distance from v to a vertex in $S \cup S'$. Computing $d(v,S')$ can again be accomplished by performing a BFS from each vertex in S', and keeping track of the levels of each BFS. Then $\hat{e}(v) = \max(d(v,S), d(v,S'))$. The sources in the second phase are likely to produce good eccentricity estimates as they are likely to be far from many vertices. This algorithm is similar to the double-sweep BFS technique used for diameter estimation [17, 31], except that we execute k BFS's together and also compute eccentricity estimates for the vertices.

A naive implementation of this algorithm simply executes each of the first k BFS's independently in parallel, and then the next k BFS's independently in parallel. Overall, the BFS's take $O(km)$ work and $O(\min(n, D\log n))$ depth. Finding the k BFS sources for the second phase can be accomplished with a parallel integer sort [39] in $O(n)$ work and $O(\log n)$ depth with high probability.[2] The total work of the algorithm is $O(km)$ and depth is $O(\min(n, D\log n))$ with high probability. For our implementation, we combine the k BFS's together to improve practical performance, leveraging the fact that there is shared work among the BFS's.

Implementation. We describe our implementation of the first phase of the multiple-BFS algorithm in detail (pseudocode shown in Figure 2). The second phase is similar, except that the BFS source vertices are determined from the first phase instead of being random. The implementation is an extension of the eccentricity estimation algorithm in [41].

For each vertex, we need to keep track of which BFS's have visited it, which is done by keeping one bit per BFS source. In particular, for a word size of w and sample size of k, each vertex maintains $\lceil k/w \rceil$ words, which we refer to as a ***bit-vector***. When the i'th BFS visits vertex v, the $(i - w\lfloor i/w \rfloor)$'th bit of the $\lfloor i/w \rfloor$'th word of v's bit-vector will be set to 1. We will take advantage of bit-level parallelism to set multiple bits together by advancing the frontiers of all BFS's simultaneously. The words in each bit-vector

[2] Probability at least $1 - 1/n^c$ for any constant $c > 0$.

```
1: Visited = $\{\{0,\dots,0\},\dots,\{0,\dots,0\}\}$          ▷ words initialized to all 0
2: NextVisited = $\{\{0,\dots,0\},\dots,\{0,\dots,0\}\}$      ▷ words initialized to all 0
3: Ecc = $\{\infty,\dots,\infty\}$                             ▷ initialized to all $\infty$
4: round = 0
5: procedure UPDATE($s$, $d$)
6:     Changed = false
7:     for $j = 0$ to $\lceil k/w \rceil - 1$ do
8:         if (Visited[$d$][$j$] $\neq$ Visited[$s$][$j$]) then
9:             ATOMICOR(&NextVisited[$d$][$j$], Visited[$d$][$j$] | Visited[$s$][$j$])
10:            oldEcc = Ecc[$d$]
11:            if (Ecc[$d$] $\neq$ round) then
12:                Changed = Changed | CAS(&Ecc[$d$], oldEcc, round)
13:    return Changed
14: procedure COPY($i$)
15:     parfor $j = 0$ to $\lceil k/w \rceil - 1$ do
16:         Visited[$i$][$j$] = NextVisited[$i$][$j$]
17: procedure INIT($i$)
18:     Set unique bit in Visited[$i$] and NextVisited[$i$] to 1
19:     Ecc[$i$] = 0
20: procedure COMPUTE-ECC($G$)
21:     vertexSubset Frontier = $k$ randomly sampled vertices
22:     VERTEXMAP(Frontier, INIT)                               ▷ initializes frontier
23:     while (size(Frontier) > 0) do
24:         round = round + 1
25:         Frontier = EDGEMAP($G$, Frontier, UPDATE, $C_{true}$)
26:         VERTEXMAP(Frontier, COPY)
27:     return Ecc
```

Figure 2: Pseudocode for k-BFS (first phase)

are all initialized to 0, except for the vertices in the sample, each of which have a unique bit set in their bit-vector. To allow our implementation to be as portable across architectures as possible, we use a word size of 64 bits, supported on all modern machines. We note that certain architectures support vector operations on a larger number of bits, and using them may improve the performance of our implementation on these architectures.

The implementation proceeds in iterations, where each iteration advances the search frontier of all k BFS's by one level. The implementation uses two arrays of bit-vectors, Visited and NextVisited (Lines 1–2), as well as an array Ecc to keep track of the eccentricity estimates for all vertices (Line 3). Initially, k random vertices are placed on the frontier, represented as a vertexSubset (Line 21). They are initialized on Line 22 with a VERTEXMAP using the Init function (Lines 17–19), which sets a unique bit in their bit-vectors in Visited and NextVisited to 1, and their eccentricity estimate to 0.

In each iteration, the implementation applies an EDGEMAP to the frontier vertices to visit their unvisited neighbors and advances all BFS searches by one level (Line 25). C_{true}, a function that always returns *true*, is passed to EDGEMAP because vertices can be visited multiple times, unlike in a single BFS. The Update function (Lines 5–13) checks for all $j \in \{0,\dots,\lceil k/w \rceil - 1\}$ whether the j'th word of the bit-vector of the frontier vertex s is different from the corresponding word of its neighbor d's bit-vector in the Visited array (Line 8); if so, this means that at least one BFS has found an unvisited neighbor d, and it performs a bitwise-OR of s's word with d's word and passes the result to d using an AtomicOR (Line 9). Atomicity is needed because multiple frontier vertices could visit the same neighbor in parallel. The result is stored in the NextVisited array, which keeps the state of the bit-vector of d for the next iteration. This is done to prevent a BFS from visiting vertices more than one hop away in a single iteration (which may happen if d is also on the frontier and processed after s). In addition, the eccentricity estimate of d is updated to be the current round number, which is equal to the current level of each BFS (Lines 10–12). This is done using a read followed by a CAS for performance reasons. If the value was updated to the current round number (the CAS returns *true*), then the Update function will return *true* on Line 13, placing the neighbor on the next frontier.

The EDGEMAP returns a vertexSubset containing the vertices on the next frontier, and a VERTEXMAP is applied to the next frontier (Line 26) to update the vertices' Visited bit-vectors with the newly computed bit-vectors from the EDGEMAP that are stored in NextVisited. The VERTEXMAP takes as input the Copy function which simply copies the NextVisited entries into the Visited array (Lines 14–16).

The code terminates when no new vertices are visited in an iteration, which causes EDGEMAP to output an empty frontier. At this point, Ecc[v] stores $d(v, S)$ where S is the set of initially sampled vertices. The second phase of the algorithm finds S', the k vertices with the largest Ecc values breaking ties arbitrarily (we use a parallel integer sort from [42] for this), places them onto an initial frontier, and proceeds in the same manner as in the first phase. The final result gives estimates $\hat{e}(v) = \max(d(v, S), d(v, S'))$ for all v.

The k-BFS implementation takes advantage of shared work among the different BFS's in several ways. First, when a vertex from multiple BFS's whose sources are associated with bits in the same word is on the frontier, only a single word-level operation is needed to visit a shared neighbor that has not yet been visited by any of those BFS's, as opposed to needing multiple operations had the BFS's been run separately. Second, even if the sources of multiple BFS's are not associated with bits in the same word, as long as they are in the same cache line, only one cache miss per neighbor is required to load their data, leading to fewer cache misses overall compared to separate BFS's. Third, compared to doing separate BFS's, in k-BFS vertices are placed on the frontier fewer times, each time doing more work. This leads to fewer edge traversals overall, which is more cache-friendly since each edge traversal typically causes a cache miss.

The work and depth of our k-BFS implementation is bounded by that of the naive implementation, as the worst case corresponds to executing $2k$ separate BFS's. The space usage is $O((1+k/\log n)n)$ as one length-k bit-vector is stored per vertex. To reduce the space usage, we can separate the k BFS's into $c \leq k$ groups and run groups of k/c BFS's at a time. Then, the sources for the second phase of the algorithm are the k/c farthest vertices found in the first phase. This reduces the space to $O((1 + k/(c \log n))n)$ and increases the depth by a factor of c. While k-BFS does not have provable approximation guarantees, we will see that it works very well in practice.

Multiple Components. To handle graphs with more than a single connected component, we first run a connected components algorithm to identify all the components and their sizes. This can theoretically be done within the stated work/depth bounds [7]. Our code uses a connected components algorithm by Slota et al. [44], which we implement in Ligra. We run the algorithm on each of the remaining components. We implemented a version that applies the algorithm to each of the components in parallel, as well as a version that processes the components one at a time, and found the latter version to be faster overall due to lower overheads.

4 More BFS-based Algorithms

In this section, we review two BFS-based algorithms for eccentricity estimation with non-trivial approximation guarantees that require $o(mn)$ work, and describe how to parallelize them.

4.1 RV Algorithm

Roditty and Vassilevska Williams [40] describe an algorithm (which we refer to as ***RV***) for undirected graphs that returns estimates $\hat{e}(v)$ for each vertex v such that $\max(R, (2/3)e(v)) \leq \hat{e}(v) \leq \min(D, (3/2)e(v))$ with high probability. We assume a single component in the graph; otherwise the algorithm can be run separately for each component. We describe the version of RV for unweighted graphs. For a parameter $s = \Theta(\sqrt{n \log n})$, RV picks a random sample S of $\Theta((n/s)\log n) = \Theta(\sqrt{n \log n})$ vertices and computes a BFS for each of these vertices. Following the notation of [40], let $p_S(v)$ be the closest vertex in S to v and $N_s(v)$ be the s closest vertices to v, including v (breaking ties arbitrarily). Let w be the vertex with largest $d(w, p_S(w))$ value. RV computes a BFS for w, obtaining $N_s(w)$, and computes a BFS for each vertex in $N_s(w)$. This gives the exact eccentricity for each vertex in $S \cup N_s(w)$. For every vertex $v \notin S \cup N_s(w)$, define $e'(v) = \max(\max_{q \in S} d(v,q), d(v,w))$ and let $v_t \in N_s(w)$ be the closest vertex to v on the shortest path from v to w. Then $\hat{e}(v) = \max(e'(v), e(v_t))$ if $d(v, v_t) \leq d(v_t, w)$ and $\hat{e}(v) = \max(e'(v), \min_{q \in S} e(q))$ otherwise. We refer the reader to [40] for the proof that the $\hat{e}(v)$ values are within the stated accuracy bounds.

The RV algorithm spends $O(m(n/s)\log n) = O(m\sqrt{n \log n})$ work to generate the BFS's for S. Computing $p_S(v)$ for all v can be done with $O(|S|)$ comparisons per vertex, for a total of $O(n\sqrt{n \log n})$ work. The BFS from w gives $N_s(w)$ as well as v_t for all vertices, and requires $O(m)$ work. Computing the BFS for all vertices in $N_s(w)$ requires $O(m\sqrt{n \log n})$ work. Computing $\hat{e}(v)$ for all v takes $O(|S|)$ comparisons per vertex, for a total of $O(n\sqrt{n \log n})$ work. The overall work is $O(m\sqrt{n \log n})$.

Parallelization. The random sample S can be generated in $O(n)$ work and $O(\log n)$ depth using a parallel filter. Each BFS can be implemented in $O(m)$ work and $O(\min(n, D \log n))$ depth. Finding the $p_S(v)$ values can be done in parallel using prefix sums in $O(n|S|)$ work and $O(\log |S|)$ depth [7]. Finding w is done by computing the maximum distance, again using prefix sums, in $O(n)$ work and $O(\log n)$ depth. Computing all $\hat{e}(v)$ values can be done in parallel, each one taking $O(|S|)$ work and $O(\log |S|)$ depth to compute the maximum and minimum over a set of $|S|$ values. The overall work is $O(m\sqrt{n \log n})$ and depth is $O(\min(n, D \log n))$.

Implementation. We implement the RV algorithm using Ligra. As in k-BFS, we find the connected components of the graph, and run RV for each component. The sample S is generated by picking $(n/s)\log n = \Theta(\sqrt{n \log n})$ vertices at random and placing them into a packed array in parallel. Each vertex in S maintains a distance array to all other vertices, and fills the array using Ligra's parallel BFS (Figure 1), which also gives its own eccentricity estimate. While all parallel BFS's can execute together, we found it more efficient in practice to execute the parallel BFS's one at a time. To find the vertex w that is farthest from S, each vertex $v \in V$ computes $d(v, S)$ in parallel, and a prefix sum is applied to the results to compute the vertex with maximum $d(v, S)$ value.

Then a parallel BFS is executed from w. In addition to computing the distance array for w, we also compute the set $N_s(w)$ and for each v, the closest vertex v_t in $N_s(w)$ on the path from v to w. The BFS is modified (from Figure 1) to fill an array storing $N_s(w)$ each time it visits a new vertex until $N_s(w)$ contains $s = \Theta(\sqrt{n \log n})$ vertices. Furthermore, the distance array is modified to store the closest vertex v_t in $N_s(w)$ as well for each vertex outside of $N_s(w)$, and the Update function of BFS is modified to pass the this information from a frontier vertex to a newly visited neighbor.

Next, a parallel BFS is executed from each vertex in $N_s(w)$ and their exact eccentricities are computed. For each vertex $v \notin S \cup N_s(w)$, we compute $e'(v) = \max(\max_{q \in S} d(v,q), d(v,w))$ in parallel. Then we look up $d(v, v_t)$, and set $\hat{e}(v) = \max(e'(v), e(v_t))$ if $d(v, v_t) \leq d(v_t, w)$ and $\hat{e}(v) = \max(e'(v), \min_{q \in S} e(q))$ otherwise. Note that $\min_{q \in S} e(q)$ only needs to be computed once.

The space usage of the implementation is $\Theta(m\sqrt{n \log n})$ for the distance arrays of the BFS's. As an optimization, we reuse the distance arrays for the BFS's of the vertices in S for the BFS's of the vertices in $N_s(w)$, which requires us to compute $\max_{q \in S} d(v,q)$ for all $v \notin S$ before executing the BFS's from $N_s(w)$.

4.2 CLRSTV Algorithm

Chechik, Larkin, Roditty, Schoenebeck, Tarjan, and Vassilevska Williams describe a eccentricity estimation algorithm for undirected, weighted graphs, which we refer to as ***CLRSTV*** [16]. The algorithm returns estimates $\hat{e}(v)$ for each vertex v such that $(3/5)e(v) \leq \hat{e}(v) \leq e(v)$. CLRSTV is similar to RV, but to obtain the claimed approximation guarantee for the weighted case it differs in four aspects: (1) CLRSTV first applies a transformation to the graph to make it have bounded degree, (2) it sets the parameter s to $\Theta(\sqrt{m \log m})$ and finds the sample S deterministically so that S has a non-empty intersection with $N_s(v)$ for all v (i.e., a hitting set), (3) it runs a single-source shortest paths algorithm not only from $S \cup N_s(w)$ but also from vertices one hop away from $N_s(w)$ (call this set T), and (4) the estimates $\hat{e}(v)$ for $v \notin S \cup N_s(w) \cup T$ are set to $\max_{u \in S \cup N_s(w) \cup T}(\max(d(u,v), e(u) - d(u,v)))$. The overall work of the algorithm is $O((m \log m)^{3/2})$, and the proof of the approximation guarantee is discussed in [16].

The original algorithm in [16] is designed for weighted graphs, but because the large real-world graphs that we could obtain for our experiments are unweighted, we simplify the algorithm to give a slightly weaker approximation guarantee for unweighted graphs. The algorithm can be simplified to have the same structure as RV. In particular, no graph transformation is done, we set $s = \Theta(\sqrt{n \log n})$, and we run BFS's only from the vertices in $S \cup N_s(w)$ instead of from $S \cup N_s(w) \cup T$. Now the only differences from RV are whether we find the set S deterministically or via sampling, and how we compute the $\hat{e}(v)$ values. We choose to compute S using random sampling as in RV because it is simpler and easily parallelizable. The computation of the $\hat{e}(v)$ values uses the formula $\max_{u \in S \cup N_s(w)}(\max(d(u,v), e(u) - d(u,v)))$, and takes $O(|S| + |N_s(w)|) = O(\sqrt{n \log n})$ work. The overall work bound is $O(m\sqrt{n \log n})$. The approximation guarantee of this modified algorithm can be shown to be $(3/5)e(v) - 2 \leq \hat{e}(v) \leq e(v)$ with high probability for undirected, unweighted graphs.[3]

Parallelization and Implementation. Using the same parallelization ideas as we did for RV, we obtain a (modified) CLRSTV algorithm with $O(m\sqrt{n \log n})$ work and $O(\min(n, D \log n))$ depth. Our Ligra implementation of CLRSTV is similar to that of RV, except that the step of computing the $\hat{e}(v)$ values uses a different formula, and the BFS from w does not need to compute the closest vertex v_t in $N_s(w)$ for vertices outside of $N_s(w)$.

5 Counter-based Algorithms

This section describes two algorithms for eccentricity estimation that were originally developed for approximating the *neighborhood function* of a graph, which given a distance h returns the number of pairs of vertices reachable within distance h. The algorithms work by approximating the *individual neighborhood function* for each vertex v, which gives the number of vertices reachable from v within distance h, using probabilistic counters that estimate the number of distinct elements in a multiset. The following descriptions assume an undirected, unweighted graph.

5.1 Flajolet-Martin Counters

The ANF (approximate neighborhood function) algorithm [36] uses Flajolet-Martin (FM) counters [23] to maintain size estimates, where each counter is a bit-vector B of length $L = \Theta(\log n)$. Adding an item works by setting a single bit $b \in \{0, \ldots, L-1\}$ with probability $2^{-(b+1)}$. The ANF algorithm maintains k independent counters per vertex (using multiple counters gives more accurate estimates),

[3]Refer to the proof of Theorem 5.1 in [16]. In the last paragraph of the proof, when analyzing the approximation guarantee for vertex v, consider the distances $d(w,x)$ and $d(x,v)$ instead of $d(w,w')$ and $d(w',v)$, where x is the last vertex in $N_s(w)$ on the path P from w to v, and w' is the vertex after x on P.

which approximate the individual neighborhood function for a given distance h, and proceeds in rounds. Initially $h = 0$, and the counters are initialized to represent a single item (the vertex itself) by setting a single bit per counter with the appropriate probability. Each round computes the individual neighborhood estimates for the current h using the appropriate formula (see [36, 23]), increments h, and updates the counters. To update the counters to represent a distance of $h + 1$, each vertex computes a bitwise-OR of each of its counters with all of its neighbors corresponding counters, all of which represent the neighborhood size at distance h (see [36] for details).

Kang et al. [28] adapt the ANF algorithm for eccentricity estimation by iterating until none of the counters change in a given iteration $\hat{D}$. The estimated diameter of the graph is then $\hat{D}$, and the estimated eccentricity for a vertex v is the largest h such that the neighborhood functions for h and $h + 1$ give the same value. We note that Kang et al. [28] actually compute the *effective eccentricity*, which for each vertex gives an estimate of the smallest value h such that at least 90% of the vertices in the graph are within h hops away. Their algorithm is parallelized using MapReduce.

Implementation. In this paper, we compute estimates of the true eccentricities, and implement the algorithm in Ligra. Since we are interested in estimating the true eccentricities, we do not need to output the individual neighborhood functions for each distance. Instead we only need to keep track of the round in which any of a vertex's counters have last changed, as this is an approximation of the distance to the furthest reachable vertex.

We implement the algorithm in Ligra, and refer to it as ***FM-Ecc***. The implementation maintains two arrays of length n, each entry storing k FM counters associated with a vertex. The size of each counter is set to 32 bits. The counters are initialized using a VERTEXMAP, which uses a function that sets a single bit in each counter with the appropriate probability. All vertices are placed onto the initial frontier. An EDGEMAP is used to perform an AtomicOR of the counters of the frontier vertices with their neighbors' counters, and vertices whose counters have changed in a given round will be active in the following round. The logic of the rest of the implementation is the same as the first phase of k-BFS (Figure 2).

As all vertices can be active in each round and the number of rounds is bounded by D, the overall work of the algorithm is $O(kmD)$ and depth is $O(D \log n)$ (each application of EDGEMAP can be shown to take $O(\log n)$ depth). The space usage is $O(kn)$ as k counters are stored per vertex; as in k-BFS, this can be reduced to $O(kn/c)$ for $c \leq k$, with the depth increasing by a factor of c.

5.2 LogLog Counters

Recently, Boldi et al. [10] describe HyperANF, an improved algorithm for approximating the neighborhood function. The high-level idea is similar to ANF, but instead of using the FM counters, they use the more recent HyperLogLog counters [24]. The HyperLogLog counters take less space than the FM counters, requiring $\Theta(\log \log n)$ bits as opposed to $\Theta(\log n)$ bits. Thus they are able to fit more than one counter in a single word and use broadword programming to update multiple counters with a single operation. They use HyperANF to estimate the effective diameter of graphs as well as other distance statistics. The implementation in [10] uses parallelism by splitting the vertices among processors, keeps track of modified counters, and processes only vertices with modified counters when this set is small enough.

Implementation. As in ANF, the HyperANF algorithm can be modified to not output the individual neighborhood functions when approximating true eccentricities. To perform a fair comparison with other methods, we implement this algorithm in Ligra, and refer to it as ***LogLog-Ecc***. Ligra automatically performs the optimization of processing only vertices with modified counters when this set is small enough, as discussed in Section 2. The high-level structure of the algorithm is similar to FM-Ecc, except for how counters are initialized and combined. We implement an atomic version (using CAS) of the broadword programming procedure to perform a bitwise-OR of multiple counters within a single word described in [10]. The implementation uses 64-bit words and fits 10 HyperLogLog counters per word.

Input Graph	Num. Vertices	Num. Edges[†]	Diameter	Avg. Ecc.
com-Youtube	1,157,828	2,987,624	24	14.59
as-skitter	1,696,415	11,095,298	31	21.2
roadNet-CA	1,971,281	2,766,607	865	662.1
wiki-Talk	2,394,385	4,659,565	11	7.49
soc-LJ	4,847,571	42,851,237	20	12.82
cit-Patents	6,009,555	16,518,947	26	11.14
com-LJ	4,036,538	34,681,189	21	13.47
com-Orkut	3,072,627	117,185,083	10	7.1
nlpkkt240	27,993,601	373,239,376	242*	211.4*
Twitter	41,652,231	1,202,513,046	23*	15.2*
com-Friendster	124,836,180	1,806,607,135	37*	11.76*
Yahoo	1,413,511,391	6,434,561,035	2919*	770.9*
randLocal (synthetic)	10,000,000	49,100,524	12	11
3D-grid (synthetic)	9,938,375	29,815,125	321	321

Table 1: Graph inputs. [†]Number of unique undirected edges. *Numbers are based on estimates using k-BFS.

The work of the algorithm is $O((1 + k \log \log n / \log n)mD)$ as $O(\log n / \log \log n)$ counters fit in each word, and the depth is $O(D \log n)$. The space complexity is $O((1 + k \log \log n / \log n)n)$, and again can be reduced to $O((1 + k \log \log n/(c \log n))n)$ by increasing the depth by a factor of c.

6 Experiments

This section experimentally compares the performance and accuracy of our parallel implementations of approximate and exact algorithms for graph eccentricity on undirected, unweighted graphs.

6.1 Setup

Experimental Setup. The experiments are performed on a 40-core (with two-way hyper-threading) machine with 4×2.4GHz Intel 10-core E7-8870 Xeon processors (with a 1066MHz bus and 30MB L3 cache), and 256GB of main memory. All implementations are written using Ligra, and compiled with Cilk Plus for parallelism. Cilk Plus automatically assigns work to available processors using a work-stealing scheduler. Using the scheduler, an implementation with work $\mathcal{W}$ and depth $\mathcal{D}$ using p processors has an expected running time of $\mathcal{W}/p + O(\mathcal{D})$ [8]. The code is compiled with `g++` version 4.8.0 (which supports Cilk Plus) with the `-O2` flag. The results reported are based on a median over multiple trials.

Input Graphs. We use a set of undirected, unweighted real-world and synthetic graphs, whose size, diameter, and average eccentricity are shown in Table 1. The first 8 graphs are real-world graphs obtained from the Stanford Network Analysis Project (`http://snap.stanford.edu/data`). ***nlpkkt240*** is a graph obtained from `http://www.cise.ufl.edu/research/sparse/matrices/`. ***Twitter*** is a graph of the Twitter network [29]. ***Yahoo*** is a web graph obtained from `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`. The two synthetic graphs are generated using graph generators from the Problem Based Benchmark Suite [42]. ***randLocal*** is a random graph where every vertex has five edges to neighbors chosen with probability proportional to the difference in the neighbor's ID value from the vertex's ID. ***3D-grid*** is a grid graph in 3-dimensional space where every vertex has six edges, each connecting it to its 2 neighbors in each dimension. We symmetrized the graphs for the experiments, and removed all self and duplicate edges. The number of edges reported in Table 1 is the number of undirected edges, although our implementations store each edge in both directions. We note that all of the graphs fit in main memory, and the Yahoo graph is the largest graph considered in previous work on eccentricity estimation.

Accuracy Measures. We use two metrics to measure the accuracy of the algorithms. The first is the ***average relative error***, defined to be $(1/n)\sum_{v\in V}|\hat{e}(v)-e(v)|/e(v)$ (if $e(v)=0$, our implementations always compute $\hat{e}(v)=0$, so in this case we define the contribution of v to the average relative error to be 0). The second measure is the ***correctness ratio***, which is the number of vertices with a correct eccentricity estimate divided by n. While both measures should be minimized, we believe that for many applications, the average relative error more closely indicates the usefulness of the approximation, i.e. estimates "close" to the true value are still likely to be useful whereas estimates "far" from the true value are unlikely to be useful. We note that all of the algorithms, except for RV and the simple 2-approximation algorithm, never overestimate the eccentricity of a vertex.

6.2 Performance and Accuracy

Accuracy versus Running Time. We ran the implementations of k-BFS, FM-Ecc, and LogLog-Ecc with varying values of k. To demonstrate the importance of the second phase of k-BFS, we also experimented with a variant of k-BFS that only performs a single phase of BFS's, which we refer to as ***k-BFS-1Phase***. All of the bit-vectors and counters fit in memory for the values of k used. Each implementation was run multiple times for different values of k using all 40 cores with hyper-threading and plots of the average relative error versus running time are shown in Figure 3 for the subset of graphs in Table 1 that we were able to compute the true eccentricities of.

Overall, k-BFS achieves significantly (up to orders of magnitude) lower error for a given time budget than the other three implementations on real-world graphs. The detailed accuracy and running time data for k-BFS is shown in Table 2, and we see that it is able to achieve less than 10^{-4} (0.01%) error for all of the real-world graphs. In contrast, using about the same amount of time as k-BFS for the largest k we tried, the other three implementations achieve errors ranging from 0.1% to 59%, as shown in Table 3. Overall, k-BFS also outperforms the other three implementations in terms of correctness ratio. On the real-world graphs, k-BFS achieves a correctness ratio of at least 96% and at least 99.9% in most cases (see Table 2), while the other three implementations achieve much lower correctness ratios overall (see Table 3).

Intuitively, k-BFS works very well in practice because the first phase identifies many of the "periphery" vertices (vertices on the edge of the component). Since the eccentricity of a vertex is measured by its distance to the component's periphery, the second phase uses these periphery nodes to generate very accurate eccentricity estimates for most of the vertices. For the real-world graphs, k-BFS is much more accurate for a fixed running time than k-BFS-1Phase due to having the second phase.

For the two synthetic graphs (randLocal and 3D-grid), the curves in Figure 3 for k-BFS and k-BFS-1Phase are similar, with k-BFS-1Phase being slightly faster for a given accuracy, and both k-BFS and k-BFS-1Phase outperform FM-Ecc and LogLog-Ecc. k-BFS-1Phase is slightly better than k-BFS here because there are no periphery vertices in these graphs, and so the second phase of k-BFS does not provide much improvement in accuracy.

Overall, k-BFS-1Phase is more accurate for a given running time than FM-Ecc and LogLog-Ecc. FM-Ecc and LogLog-Ecc have similar curves, with LogLog-Ecc being more efficient as it fits multiple counters per word.

Comparison to 2-Approximation Algorithm. As a baseline, we also compare with the 2-approximation algorithm described in Section 2 that runs a BFS from an arbitrary vertex in each component in the graph, and we refer to it as ***Simple-Approx***. Our implementation first finds the connected components of the graph, and then picks a random vertex in each component to run a parallel BFS from. The running time and accuracy for Simple-Approx is shown in Table 3. Not surprisingly, Simple-Approx is faster than all of the other implementations, as it performs just a single BFS per component. *Observe that for the real-world graphs, k-BFS achieves up to orders of magnitude lower average relative errors than Simple-Approx.* However, k-BFS is 2.2–15.8x slower as it performs more BFS's.

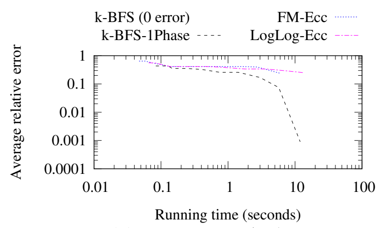
On the two synthetic graphs, Simple-Approx performs extremely well because the range of eccentricities of the vertices is small (in fact, all vertices have the same eccentricity in 3D-grid, and 99.8% have the same eccentricity in randLocal), and so using the eccentricity of a random vertex as the estimate for the remaining vertices gives high accuracy. Simple-Approx is more accurate than k-BFS-1Phase ($k=2^{15}$) on all of the graphs except com-Youtube and roadNet-CA, and more accurate than FM-Ecc (512 counters) and LogLog-Ecc (5120 counters) on all but the roadNet-CA graph. Observe that the average relative error of Simple-Approx is much lower than the worst-case 2-approximation. Overall, Simple-Approx is reasonably accurate, and can be used over k-BFS if some accuracy can be sacrificed in exchange for lower running time.

Comparison to RV and CLRSTV. We experiment with RV and CLRSTV, which have non-trivial theoretical guarantees on the estimates produced. Due to the high running time and space usage of RV and CLRSTV, we were only able to run experiments on four of the graphs. The parallel running time, average relative error, and correctness ratio for the inputs are shown in Table 3. RV and CLRSTV achieve reasonably low average relative errors, with CLRSTV achieving better accuracy overall but with a slightly higher running time. The relative error of the implementations is much lower than the theoretical worst case discussed in Section 4.

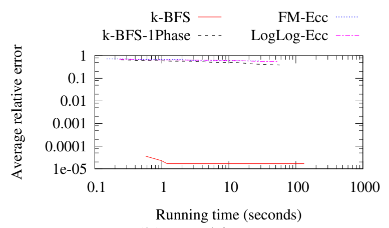
Both implementations are more than two orders of magnitude slower than k-BFS for a similar accuracy (for com-Youtube, as-skitter, and wiki-Talk, refer to the running time of k-BFS for $k=2^6$ in Table 2, and for roadNet-CA refer to the running time for $k=2^{10}$). The performance agrees with the work bounds for k-BFS, RV, and CLRSTV. In particular, k-BFS requires $O(km)$ work, whereas RV and CLRSTV require $O(m\sqrt{n\log n})$ work (with large constant factors). For k-BFS to achieve a comparable accuracy to RV and CLRSTV, usually $k \ll \sqrt{n\log n}$. *We conclude that k-BFS is orders of magnitude faster than RV and CLRSTV in practice while obtaining similar accuracy.* Note that RV and CLRSTV can achieve much higher accuracy than k-BFS-1Phase, FM-Ecc, LogLog-Ecc, and Simple-Approx (see Table 3).

Comparison to Exact Algorithms. We study the performance of two parallel algorithms that compute exact vertex eccentricities—the TK algorithm discussed in Section 2.2 (run on each component in the graph), and one which runs k-BFS-1Phase for each set of n/k vertices in the graph, which we refer to as ***k-BFS-Exact***. Parallel running times are shown in Table 3 for the three graphs on which the algorithms finished in a reasonable amount of time.

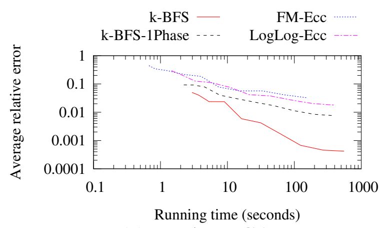
Observe that TK is much faster than k-BFS-Exact for two of the inputs as the number of BFS's is much lower than n. However, for as-skitter, TK is slower than k-BFS-Exact because the number of BFS iterations is only reduced to about $0.75n$, and TK has additional overheads compared to k-BFS-Exact. *Due to their quadratic work complexities, both TK and k-BFS-Exact are orders of magnitude slower than k-BFS.* Compared to RV and CLRSTV, TK is sometimes faster and sometimes slower. The running time of TK highly varies, as the reduction in number of BFS's performed is strongly related to the graph structure.



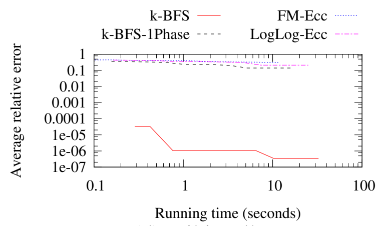
(a) com-Youtube*



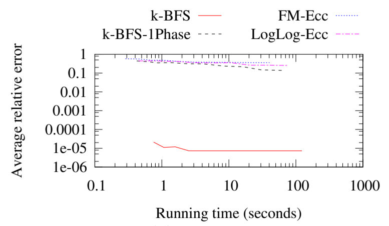
(b) as-skitter



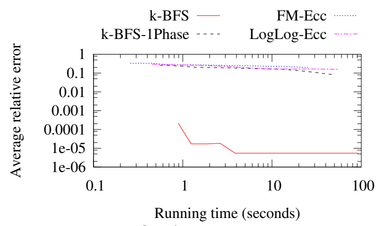
(c) roadNet-CA



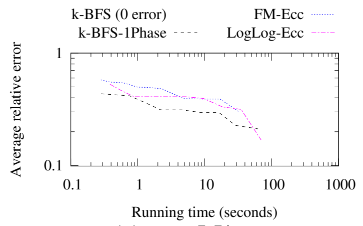
(d) wiki-Talk



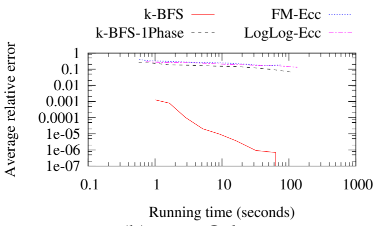
(e) soc-LJ



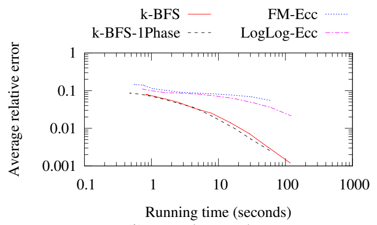
(f) cit-Patents



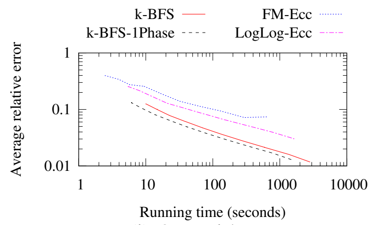
(g) com-LJ*



(h) com-Orkut



(i) randLocal



(j) 3D-grid

Figure 3: Parallel (40 cores with hyper-threading) running time versus average relative error for k-BFS, k-BFS-1Phase, FM-Ecc, and LogLog-Ecc (log-log scale). *k-BFS achieves an error of 0 for all data points for this graph.

k	2^6			2^7			2^8			2^9			2^{10}			2^{12}			2^{14}		
	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR
com-Youtube	0.146	0	1	0.202	0	1	0.266	0	1	0.439	0	1	0.74	0	1	2.81	0	1	11	0	1
as-skitter	0.578	10^{-5}	0.999	0.984	10^{-5}	0.999	1.2	10^{-5}	0.999	2.01	10^{-5}	0.999	3.94	10^{-5}	0.999	16.3	10^{-5}	0.999	64.3	10^{-5}	0.999
roadNet-CA	2.89	0.05	0.09	3.72	0.04	0.09	5.33	0.02	0.21	9	0.02	0.21	16.3	0.005	0.39	60.8	0.002	0.85	270	10^{-4}	0.964
wiki-Talk	0.288	10^{-5}	0.999	0.427	10^{-5}	0.999	0.765	10^{-6}	0.999	1.18	10^{-6}	0.999	1.82	10^{-6}	0.999	6.51	10^{-6}	0.999	17.1	10^{-7}	0.999
soc-LJ	0.76	10^{-5}	0.999	1.07	10^{-5}	0.999	1.58	10^{-5}	0.999	2.53	10^{-5}	0.999	4.5	10^{-5}	0.999	15.9	10^{-5}	0.999	57.1	10^{-5}	0.999
cit-Patents	0.894	10^{-4}	0.997	1.24	10^{-5}	0.999	1.86	10^{-5}	0.999	2.63	10^{-5}	0.999	3.85	10^{-5}	0.999	13.3	10^{-5}	0.999	46.9	10^{-5}	0.999
com-LJ	0.526	0	1	0.874	0	1	1.42	0	1	2.6	0	1	4.17	0	1	16.7	0	1	56.6	0	1
com-Orkut	1.01	0.001	0.99	1.64	0.001	0.994	2.85	10^{-4}	0.999	5.16	10^{-5}	0.999	9.19	10^{-5}	0.999	32.1	10^{-6}	0.999	114	0	1
nlpkkt240	30.5	–	–	46.4	–	–	80.1	–	–	142	–	–	289	–	–	1140	–	–	4070	–	–
Twitter	200	–	–	238	–	–	409	–	–	518	–	–	781	–	–	2690	–	–	5990	–	–
com-Friendster	85.1	–	–	117	–	–	198	–	–	251	–	–	367	–	–	1120	–	–	–	–	–
Yahoo	655	–	–	1060	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
randLocal	0.829	0.078	0.143	1.13	0.07	0.228	1.58	0.061	0.334	2.42	0.05	0.45	4.18	0.035	0.616	15.2	0.014	0.846	55.2	0.003	0.965
3D-grid	9.67	0.123	10^{-6}	14	0.098	10^{-4}	21.4	0.078	10^{-4}	36.4	0.061	0.001	66.8	0.047	0.001	264	0.028	0.005	1340	0.016	0.019

Table 2: Running time (seconds) on 40 cores with hyper-threading (RT), average relative error (ARE), and correctness ratio (CR) versus k for k-BFS.

	k-BFS-1Phase ($k = 2^{15}$)			FM-Ecc (512 counters)			LogLog-Ecc (5120 counters)			Simple-Approx			RV			CLRSTV			TK	k-BFS-Exact
	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR	RT	ARE	CR	RT	RT
com-Youtube	11.9	0.001	0.986	6.02	0.239	0.02	13.8	0.25	0.02	0.033	0.042	0.473	81.1	0.001	0.988	87.1	0.001	0.988	46.5	644
as-skitter	55.8	0.39	0.001	28.6	0.591	0.001	53	0.561	0.001	0.038	0.025	0.527	136	0.002	0.956	139	0	1	15000	2240
roadNet-CA	399	0.008	0.036	154	0.033	0.007	390	0.018	0.01	0.309	0.187	0.009	3110	0.019	0.015	2950	0.008	0.08	–	–
wiki-Talk	16.1	0.14	0.02	11.5	0.305	0.002	24.9	0.211	0.002	0.036	0.063	0.5	142	0.003	0.976	152	0.003	0.979	168	3220
soc-LJ	62.4	0.14	0.002	40.1	0.357	0.001	75.9	0.254	0.001	0.077	0.039	0.54	–	–	–	–	–	–	–	–
cit-Patents	47	0.084	0.38	26.2	0.197	0.374	56.5	0.159	0.374	0.408	0.032	0.572	–	–	–	–	–	–	–	–
com-LJ	62.2	0.212	0.01	33.5	0.298	0.01	69.3	0.17	0.01	0.071	0.046	0.447	–	–	–	–	–	–	–	–
com-Orkut	107	0.067	0.52	77.2	0.184	0.006	132	0.132	0.085	0.064	0.02	0.852	–	–	–	–	–	–	–	–
randLocal	64.4	0.002	0.975	59.6	0.055	0.395	121	0.022	0.763	0.149	10^{-4}	0.998	–	–	–	–	–	–	–	–
3D-grid	1610	0.013	0.026	676	0.074	0.001	1620	0.03	0.003	0.509	0	1	–	–	–	–	–	–	–	–

Table 3: Running time (seconds) on 40 cores with hyper-threading (RT), average relative error (ARE), and correctness ratio (CR) for k-BFS-1Phase using $k = 2^{15}$, FM-Ecc using 512 counters per vertex, LogLog-Ecc using 5120 counters per vertex, Simple-Approx, RV, CLRSTV, TK, and k-BFS-Exact.

	k-BFS ($k = 2^6$)	FM-Ecc (1 counter)	LogLog-Ecc (10 counters)
com-Youtube	20.93	23.29	26.39
as-skitter	15.6	20.17	18.5
roadNet-CA	19.45	17.72	24.86
wiki-Talk	14.43	14.24	15.5
soc-LJ	36.47	47.28	37.53
cit-Patents	29.43	34.61	36.63
com-LJ	36.12	34.71	35.77
com-Orkut	38.45	41.25	36.03

Table 4: Self-relative parallel speedup on 40 cores with two-way hyper-threading.

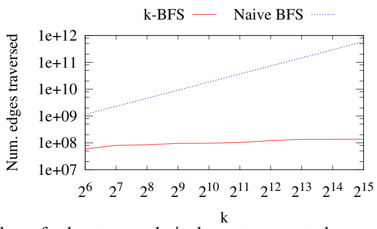


Figure 5: Number of edge traversals in largest connected component of the com-Youtube graph as a function of k (log-log scale).

We note that Takes and Kosters [46] also describe a pruning strategy that removes all degree-1 vertices from the graph to reduce the number of BFS's needed to compute eccentricities. While we did not implement this strategy in TK, it can be applied to all of the implementations in this paper to possibly improve their performance.

Parallelism. The parallel speedups of k-BFS, FM-Ecc, and LogLog-Ecc on 40-cores with hyper-threading for a subset of the graphs are shown in Table 4. We observe that the implementations achieve reasonably good speedup, ranging from 14x to 47x on 40 cores with two-way hyper-threading. The parallel speedup tends to be better for larger graphs, as there is more work to offset the overheads of parallelism. Figure 4 plots the self-relative parallel speedup versus thread count for all of the implementations on several input graphs, showing improvements in speedups as the thread count increases.

Sharing work in k-BFS. Observe from Table 2 that the parallel running time of k-BFS increases as we increase k, but usually sublinearly with k. This is because k-BFS takes advantage of shared work among the different BFS's, as described in Section 3. As an illustration of one of the benefits of k-BFS, Figure 5 plots the number of edge traversals of k-BFS in the largest component as a function of k for the com-Youtube graph, compared to the number of edge traversals required by running k separate BFS's in each phase of the algorithm (naive BFS). We see that k-BFS does fewer edge traversals, with the difference being larger for larger k. This improves cache performance since each edge traversal typically corresponds to a cache miss. The same trend was observed for the other inputs as well.

6.3 Eccentricity Plots

Due to the efficiency and high accuracy of k-BFS, we are able to quickly generate eccentricity distributions for some of the largest real-world graphs studied in the literature. Figure 6 plots the eccentricity distributions generated using k-BFS with $k = 2^6$ for our three largest graphs—Twitter, com-Friendster, and Yahoo. Generating the distribution for the largest graph, Yahoo, took only 11 minutes.

For Twitter and com-Friendster, we observe that the diameter is relatively small (23 and 37, respectively, as estimated by k-BFS), while the diameter for Yahoo is large (estimated by k-BFS to be 2919). There is a single peak in the plot for the Twitter graph at an eccentricity of 15, which is close to its average eccentricity of 15.2. For com-Friendster, there are two peaks, the first at 0 caused by the many disconnected vertices (almost half of the vertices in the graph are singletons), and the second at 22 (about 29% of the vertices have this value). There are two obvious peaks in the eccentricity plot for the Yahoo graph at 0 (almost half of the vertices are disconnected) and 1552 (about 30% of the vertices have this value), and another flatter peak at around 800–1100. k-BFS estimates the average eccentricity of the Yahoo graph to be 770.9, and excluding the singleton vertices, the average eccentricity is about 1513. The high average eccentricity of the vertices is due to the long “whiskers” [28] present in the graph. We note that Kang et al. [28] have previously studied the *effective* eccentricities (the distance at which 90% of the vertices can be reached) of the Yahoo graph using MapReduce.

7 Related Work

For a connected, undirected graph with diameter D, Aingworth et al. [1] describe an algorithm to generate an estimate $\hat{D}$ such that $(2/3)D \leq \hat{D} \leq D$ in $O(m\sqrt{n \log n} + n^2 \log n)$ work. Their algorithm extends to graphs with non-negative edge weights, generating an estimate $\hat{D}$ such that $(2/3)D - w_{max} \leq \hat{D} \leq D$, where w_{max} is the maximum edge weight. Roditty and Vassilevska Williams [40] improve the work of the diameter estimation algorithm of [1] to $O(m\sqrt{n \log n})$ with high probability. They also show that the algorithm can be used for eccentricity estimation, which we described in Section 4.1. The diameter approximation of weighted graphs was improved by Chechik et al. [16], who present two algorithms that generate estimates $\hat{D}$ such that $(2/3)D \leq \hat{D} \leq D$, with the first algorithm requiring $O(m^{3/2}\sqrt{\log n})$ work and the second algorithm requiring $O(mn^{2/3} \log^{5/3} n)$ work. They also describe how to generate an additive n^{ϵ}-approximation to the diameter in $O(n^{1-\epsilon}(m + n \log n))$ work. Finally, they present an algorithm for eccentricity estimation, which we described in Section 4.2. There have been several other papers on approximating the graph diameter or radius [17, 18, 37, 9, 6] as well as on their exact computation [45, 11, 31, 21, 20]. Diameter estimation has also been studied in the external-memory setting [34, 2, 3] and parallel/distributed setting [28, 10, 27, 14]. Leskovec et al. study the evolution of the diameter of real-world graphs over time [30].

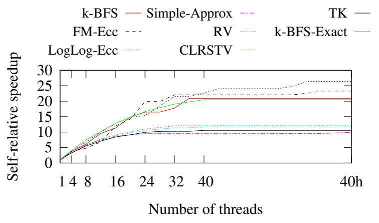
Almeida et al. [4] describe a distributed algorithm for exact eccentricity computation, which essentially does a BFS from each vertex, hence requiring $O(nm)$ work. Cardoso et al. [13] and Garin et al. [25] describe a distributed algorithm for eccentricity estimation using probabilistic counters, where combining counters uses the minimum operator. The algorithm is equivalent to executing one phase of BFS's from multiple sources and for each vertex, using the maximum distance from a source as its eccentricity estimate.

Related to our k-BFS implementation, Then et al. [47] recently describe an algorithm for executing multiple BFS's using bit-level optimizations. Their algorithm, however, is optimized for the case where BFS's are executed from a large fraction of the vertices in the graph. In contrast, our algorithm is optimized for the case where only a small set of vertices execute BFS's. Another difference is that their algorithm obtains parallelism across several multiple-BFS instances, whereas k-BFS has parallelism within a single multiple-BFS instance. This is advantageous when BFS's are executed from a small number of sources, as in eccentricity estimation.

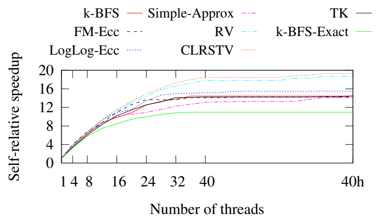
8 Conclusion

We have presented a comprehensive study of parallel algorithms for eccentricity estimation on large-scale undirected real-world graphs. Our study shows that k-BFS achieves high accuracy, efficiency, and parallelism, and we believe that the implementation will be useful in the analysis of large-scale networks. An interesting direction for future work is to prove approximation guarantees for k-BFS (or variants of it). We are also interested in evaluating eccentricity estimation algorithms for directed and/or weighted graphs.

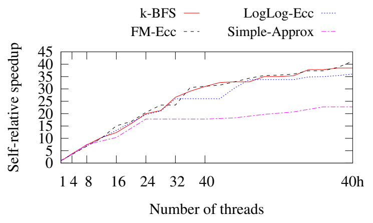
Acknowledgments. This work is partially supported by the National Science Foundation under grant CCF-1314590, and by the Intel Labs Academic Research Office for the Parallel Algorithms



(a) com-Youtube

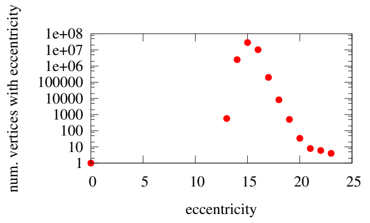


(b) wiki-Talk

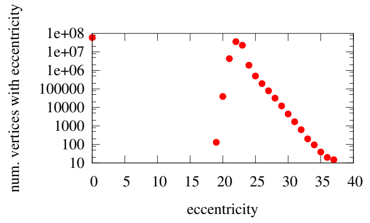


(c) com-Orkut*

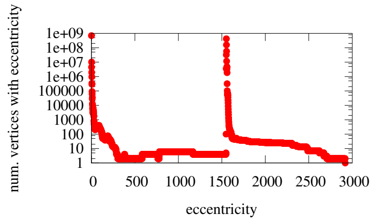
Figure 4: Self-relative parallel speedup versus thread count for k-BFS ($k = 2^6$), FM-Ecc (1 counter) and LogLog-Ecc (10 counters), Simple-Approx, RV, CLRSTV, TK, and k-BFS-Exact in log-log scale. "40h" corresponds to 40 cores with two-way hyper-threading. *We do not include RV, CLRSTV, TK, and k-BFS-Exact for com-Orkut because we were unable to generate the plots in a reasonable amount of time.



(a) Twitter



(b) com-Friendster



(c) Yahoo

Figure 6: Eccentricity distributions for several large input graphs using k-BFS with $k = 2^6$ (y-axis is in log scale).

for Non-Numeric Computing Program. Thanks to Guy Blelloch, Laxman Dhulipala, and Dan Larkin for helpful discussions.

9 References

[1] D. Aingworth et al. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 1999.

[2] D. Ajwani et al. I/O-efficient approximation of graph diameters by parallel cluster growing—a first experimental study. In *ARCS*, 2012.

[3] D. Ajwani et al. I/O-efficient hierarchical diameter approximation. In *ESA*. 2012.

[4] P. S. Almeida et al. Fast distributed computation of distances in networks. In *CDC*, 2012.

[5] S. Beamer et al. Direction-optimizing breadth-first search. In *SC*, 2012.

[6] P. Berman and S. P. Kasiviswanathan. Faster approximation of distances in graphs. In *WADS*. 2007.

[7] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 1996.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 1999.

[9] K. Boitmanis et al. Fast and simple approximation of the diameter and radius of a graph. In *WEA*. 2006.

[10] P. Boldi et al. HyperANF: Approximating the neighbourhood function of very large graphs on a budget. In *WWW*, 2011.

[11] M. Borassi et al. On the solvability of the six degrees of Kevin Bacon game—A faster graph diameter and radius computation method. In *Fun with Algorithms*, 2014.

[12] P. Bose et al. Network farthest-point diagrams and their application to feed-link network extension. *J. Computational Geometry*, 2013.

[13] J. S. Cardoso et al. Probabilistic estimation of network size and diameter. In *LADC*, 2009.

[14] M. Ceccarello et al. Space and time efficient parallel graph decomposition, clustering and diameter approximation. In *SPAA*, 2015.

[15] T. M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. *ACM Trans. Algorithms*, 2012.

[16] S. Chechik et al. Better approximation algorithms for the graph diameter. In *SODA*, 2014.

[17] D. G. Corneil et al. Diameter determination on restricted graph families. *Discrete Applied Mathematics*, 2001.

[18] D. G. Corneil et al. On the power of BFS to determine a graphs diameter. In *LATIN*, 2002.

[19] P. Crescenzi et al. A comparison of three algorithms for approximating the distance distribution in real-world graphs. In *TAPAS*. 2011.

[20] P. Crescenzi et al. On computing the diameter of real-world directed (weighted) graphs. In *SEA*. 2012.

[21] P. Crescenzi et al. On computing the diameter of real-world undirected graphs. *Theoretical Computer Science*, 2013.

[22] G. Dal, W. Kosters, and F. Takes. Fast diameter computation of large sparse graphs using GPUs. In *PDP*, 2014.

[23] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 1985.

[24] P. Flajolet et al. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms*, 2007.

[25] F. Garin et al. Distributed estimation of diameter, radius and eccentricities in anonymous networks. In *NecSys*, 2012.

[26] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981.

[27] S. Holzer et al. Brief announcement: Distributed $3/2$-approximation of the diameter. In *DISC*, 2014.

[28] U. Kang et al. HADI: Mining radii of large graphs. In *TKDD*, 2011.

[29] H. Kwak et al. What is Twitter, a social network or a news media? In *WWW*, 2010.

[30] J. Leskovec et al. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD*, 2005.

[31] C. Magnien et al. Fast computation of empirically tight bounds for the diameter of massive graphs. *J. Exp. Algorithmics*, 2009.

[32] D. Magoni and J. J. Pansiot. Analysis of the autonomous system network topology. In *SIGCOMM*, 2001.

[33] D. Magoni and J. J. Pansiot. Analysis and comparison of internet topology generators. In *Networking*, 2002.

[34] U. Meyer. On trade-offs in external-memory diameter-approximation. In *SWAT*. 2008.

[35] M. Mneimneh and K. Sakallah. Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution. In *SAT*. 2004.

[36] C. R. Palmer et al. ANF: a fast and scalable tool for data mining in massive graphs. In *KDD*, 2002.

[37] M. Parnas and D. Ron. Testing the diameter of graphs. *Random Struct. Algorithms*, 20(2), 2002.

[38] G. Pavlopoulos et al. Using graph theory to analyze biological networks. *BioData Min*, 2011.

[39] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 1989.

[40] L. Roditty and V. Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *STOC*, 2013.

[41] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[42] J. Shun et al. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012.

[43] J. Shun et al. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*, 2015.

[44] G. M. Slota et al. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IPDPS*, 2014.

[45] F. W. Takes and W. A. Kosters. Determining the diameter of small world networks. In *CIKM*, 2011.

[46] F. W. Takes and W. A. Kosters. Computing the eccentricity distribution of large graphs. *Algorithms*, 2013.

[47] M. Then et al. The more the merrier: Efficient multi-source graph traversal. In *PVLDB*, 2014.

[48] C. E. Tsourakakis. Large scale graph mining with MapReduce: Diameter estimation and eccentricity plots of massive graphs with mining applications. In *Social Network Mining, Analysis and Research Trends*, 2012.

[49] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 1998.

[50] R. Williams. Faster all-pairs shortest paths via circuit complexity. In *STOC*, 2014.

[51] U. Zwick. Exact and approximate distances in graphs—a survey. In *ESA*. 2001.