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

Julian Shun
Graph Eccentricities

- The **eccentricity** of a vertex $v$ is the distance to furthest reachable vertex from $v$

  \[
  \begin{align*}
  \text{ecc}(a) &= 4 \\
  \text{ecc}(b) &= 3 \\
  \text{ecc}(c) &= 2 \\
  \text{ecc}(d) &= 3 \\
  \text{ecc}(e) &= 4 \\
  \text{ecc}(f) &= 3 \\
  \text{ecc}(g) &= 4 
  \end{align*}
  \]

- The **diameter** of a graph is the maximum eccentricity value
- Extends to directed and/or weighted graphs
Applications

Routing networks

Protein networks

Social networks

Web graphs

• Core-periphery structure
How to compute eccentricities?

- All pairs shortest paths (APSP)
  - At least quadratic work
  - Would take over 2670 years for Yahoo! Web graph (1.4B vertices, 6.4B edges)
- Takes and Kosters \([\text{Algorithms 2013}]\) algorithm
  - Quadratic work in the worst case
- Approximation algorithms
  - Orders of magnitude faster
  - Can process Yahoo! Web graph in minutes
- Parallelism

Experiments done on one thread of a 40-core (2.4 GHz) Intel Nehalem machine with 256 GB memory

wiki-Talk (\(|V| = 2.4\text{M}, |E| = 4.7\text{M}|\)

Sequential Running Time
(Hours)

<table>
<thead>
<tr>
<th></th>
<th>APSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>wiki-Talk ((</td>
<td>V</td>
</tr>
</tbody>
</table>

4 seconds, < 0.0001 average relative error
This work

- First comprehensive comparison of parallel implementations of eccentricity algorithms on large (undirected/unweighted) real-world graphs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>APSP</td>
<td>Exact</td>
<td>$O(</td>
</tr>
<tr>
<td>Takes-Kosters [Algorithms 2013]</td>
<td>Exact</td>
<td>$O(</td>
</tr>
<tr>
<td>Single BFS</td>
<td>2-approx</td>
<td>$O(</td>
</tr>
<tr>
<td>Chechik et al. [SODA 2014]</td>
<td>1.66-approx</td>
<td>$O(</td>
</tr>
<tr>
<td>Roditty-Vassilevska Williams [STOC 2013]</td>
<td>1.5-approx</td>
<td>$O(</td>
</tr>
<tr>
<td>ANF/HADI [Palmer et al. KDD ‘02, Kang et al. TKDD ‘10]</td>
<td></td>
<td>$O(k</td>
</tr>
<tr>
<td>HyperANF [Boldi et al. WWW 2011]</td>
<td></td>
<td>$O(k (\log \log</td>
</tr>
<tr>
<td>k-BFS [this work]</td>
<td></td>
<td>$O(k</td>
</tr>
</tbody>
</table>

$|V| = \#$ vertices $|E| = \#$ edges $k = \#$ probabilistic counters/BFS's $D =$ diameter

- Simple shared-memory implementations in the Ligra graph processing framework [Shun and Blelloch PPoPP 2013]
- k-BFS as a parallel primitive for fast, scalable, and accurate eccentricity estimation
Simple 2-approximation

- **Step 1:** Perform breadth-first search (BFS) from arbitrary vertex \( v \) to compute eccentricity
- **Step 2:** Assign all vertices \( \text{ecc}(v) \)
- **Guarantee:** \( \frac{\text{ecc}(w)}{2} \leq \text{e\text{c}c}(w) \leq 2 \text{ecc}(w) \) for all \( w \)
- BFS complexity is \( O(|E|) \) and easily parallelizable with parallel time proportional to diameter

This algorithm is fast but its estimates are useless!
Stronger provable approx algorithms

• RV: [Roditty-Vassilevska Williams STOC 2013]
  - $(2/3) \text{ecc}(v) \leq \text{e\c c}(v) \leq (3/2) \text{ecc}(v)$
  - Complexity: $O(|E| (|V| \log |V|)^{1/2})$

• CLRSTV: [Chechik, Larkin, Roditty, Schoenebeck, Tarjan, and Vassilevska Williams SODA 2014]
  - $(3/5) \text{ecc}(v) \leq \text{e\c c}(v) \leq \text{ecc}(v)$
  - Complexity: $O(|E| (|V| \log |V|)^{1/2})$

• We provide the first empirical evaluation of these algorithms
  - Shared-memory parallel implementations in Ligra
How do they perform in practice?

- Average relative error = \[ \frac{1}{|V|} \sum_{v \in V} \left| \frac{\hat{e}(v) - e(v)}{e(v)} \right| \]

Experiments done on a 40-core Intel Nehalem machine with 256 GB memory

RV/CLRSTV accurate but not scalable
Multiple BFS’s

- Run multiple BFS’s from a sample of random vertices and use distance from furthest sample as eccentricity:

\[ \text{e\text{cc}(v)} = \max(d(v, s_1), d(v, s_2), d(v, s_3), d(v, s_4)) \text{ for all } v \]

- Does not work that well in practice

- Double-sweep: find furthest vertices from sample of random vertices, then run BFS’s from the set of furthest vertices (used before for diameter estimation [Corneil et al. 2001, Magnien et al. 2009])
k-BFS Implementation (second sweep)

• **k BFS sources**
• Each vertex \( v \) stores estimate \( e\hat{c}(v) = 0 \)
• Run parallel BFS from each source, updating each encountered vertex \( v \) at distance \( d \) to \( \max(e\hat{c}(v), d) \)

• **Observation:** There is shared work among the different BFS’s
  • Vertices updated multiple times
  • Vertices placed on frontiers multiple times \( \rightarrow \) edges traversed multiple times
**k-BFS Implementation**

- **Observation:** There is shared work among the different BFS’s

  - \( \text{dist}(s_1,v) = d \)
  - \( \text{dist}(s_2,w) = d \)
  - \( \text{dist}(s_3,w) = d \)

  - \( \text{dist}(s_1,x) = d+1 \)
  - \( \text{dist}(s_2,x) = d+1 \)
  - \( \text{dist}(s_3,x) = d+1 \)
  - \( \text{dist}(s_1,y) = d+1 \)
  - \( \text{dist}(s_2,y) = d+1 \)
  - \( \text{dist}(s_3,y) = d+1 \)

- **Goal:** Reduce redundant computation on vertices
- **Goal:** Reduce the number of times each visited vertex is placed onto the frontier
k-BFS Implementation

• Run all k BFS’s simultaneously
• Take advantage of bit-level parallelism to store “visited” information

Bitwise-OR of bit-vectors when visiting neighbor

Unique bit set for each source

000000000…000000000
000000000…000000000
000000000…000000000

... k/64 words

010000000…000000000
000000000…000000000
... 000000000…000000000
... 000000000…000000000

s1

v

s2
k-BFS Implementation

- **Initial frontier** = \{s_1, s_2, \ldots, s_k\}
- d = 0
- **While** frontier not empty:  
  - nextFrontier = {}
  - d = d+1
  - For each vertex v in frontier:
    - For each neighbor ngh:
      - Do bitwise-OR of v’s words with ngh’s words and store in ngh
      - If any of ngh’s words changed:
        - ecc(ngh) = max(ecc(ngh), d) and place ngh on nextFrontier if not there
    - frontier = nextFrontier

//Advance all BFS’s by 1 level

//pass “visited” information

![Graph showing num. edges traversed vs k](image)

<table>
<thead>
<tr>
<th>k</th>
<th>num. edges traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^6</td>
<td></td>
</tr>
<tr>
<td>2^7</td>
<td></td>
</tr>
<tr>
<td>2^8</td>
<td></td>
</tr>
<tr>
<td>2^9</td>
<td></td>
</tr>
<tr>
<td>2^10</td>
<td></td>
</tr>
<tr>
<td>2^11</td>
<td></td>
</tr>
<tr>
<td>2^12</td>
<td></td>
</tr>
<tr>
<td>2^13</td>
<td></td>
</tr>
<tr>
<td>2^14</td>
<td></td>
</tr>
<tr>
<td>2^15</td>
<td></td>
</tr>
</tbody>
</table>

- k-BFS and Naive BFS comparison
  - com-Youtube
  - k-BFS
  - Naive BFS
Parallel k-BFS

- Initial **frontier** = \{s_1, s_2, \ldots, s_k\}
- d = 0
- While **frontier** not empty: //Advance all BFS’s by 1 level
  - **nextFrontier** = {}
  - d = d+1
  - For each vertex v in **frontier**:
    - For each neighbor ngh:
      - Do bitwise-OR of v’s words with ngh’s words and store in ngh
      - If any of ngh’s words changed:
        - \( ec(ngh) = \max(ec(ngh), d) \) and place ngh on **nextFrontier** if not there
    - **frontier** = **nextFrontier**

- Ligra framework that we use takes care of most details
  - User only specifies function to apply on each edge traversed
  - Performs “direction-optimizing” BFS [Beamer ‘12] automatically
k-BFS Performance (varying k)

wiki-Talk (|V| = 2.4M, |E| = 4.7M)

Experiments done on a 40-core Intel Nehalem machine with 256 GB memory

k = \{2^6, 2^7, 2^8, \ldots, 2^{15}\}

k-BFS (k = 64) 0.288 sec < 10^{-4}
k-BFS Performance

wiki-Talk (|V| = 2.4M, |E| = 4.7M)

Average Relative Error vs. Running time (seconds)

- k-BFS
- k-BFS-1Phase

Experiments done on a 40-core Intel Nehalem machine with 256 GB memory

cit-Patents (|V| = 6M, |E| = 16.5M)

Average Relative Error vs. Running time (seconds)

- k-BFS
- k-BFS-1Phase

soc-LJ (|V| = 4.85M, |E| = 42.9M)

Average Relative Error vs. Running time (seconds)

- k-BFS
- k-BFS-1Phase

com-Orkut (|V| = 3M, |E| = 117.2M)

Average Relative Error vs. Running time (seconds)

- k-BFS
- k-BFS-1Phase
Probabilistic Counter-based Algorithms

• Assign k probabilistic counters per vertex
• Bitwise-OR counters with all of neighbors’ counters until all counters stabilize
• Eccentricity estimate for a vertex is the last round in which any of its counters changed
• ANF/HADI algorithm [Palmer et al. ’03, Kang et al. ‘10] used Flajolet-Martin counters
• HyperANF algorithm [Boldi et al. ’11] use more space-efficient HyperLogLog counters [Flajolet et al. ‘08]
• Shared-memory implementations of variants of ANF/HADI and HyperANF in Ligra
k-BFS outperforms ANF/HADI and HyperANF

- **wiki-Talk (|V| = 2.4M, |E| = 4.7M)**
- **cit-Patents (|V| = 6M, |E| = 16.5M)**
- **soc-LJ (|V| = 4.85M, |E| = 42.9M)**
- **com-Orkut (|V| = 3M, |E| = 117.2M)**
Performance on Synthetic Graphs

**randLocal** ($|V| = 10M, |E| = 49M)$

- k-BFS-1Phase
- ANF/HADI
- HyperANF
- k-BFS

**3D-grid** ($|V| = 10M, |E| = 30M)$

- k-BFS-1Phase
- ANF/HADI
- HyperANF
- k-BFS
Parallel Scalability

Experiments done on a 40-core Intel Nehalem machine with 256 GB memory

- k-BFS achieves 14—38x speedup on 40 cores (higher for larger graphs)
Scaling to Large Graphs

Experiments done on a 40-core Intel Nehalem machine with 256 GB memory

Size

|V| = 4.2 x 10^7
|E| = 1.2 x 10^9

|V| = 1.2 x 10^8
|E| = 1.8 x 10^9

|V| = 1.4 x 10^9
|E| = 6.4 x 10^9

k-BFS (k=64) Time

3.5 minutes

1.5 minutes

11 minutes
Conclusion

- **Comprehensive evaluation** of shared-memory parallel eccentricity estimation algorithms
- k-BFS is orders of magnitude more **accurate** for a fixed running time than other estimation algorithms
- k-BFS is **scalable** to largest publicly-available real-world graphs studied in the literature

**Future work**
- Extensions to directed, weighted graphs
- Theoretical bounds for (variants of) k-BFS
Thank you!

Code: [http://github.com/jshun/ligra](http://github.com/jshun/ligra)