STREAMER: a Distributed Framework for Incremental Closeness Centrality Computation

A. Erdem Sarıyüce $^{1,2}$, Erik Saule $^4$, Kamer Kaya $^1$, Ümit V. Çatalyürek $^{1,3}$

$^1$ Department of Biomedical Informatics
$^2$ Department of Computer Science & Engineering
$^3$ Department of Electrical & Computer Engineering

The Ohio State University
$^4$ Department of Computer Science
University of North Carolina Charlotte

IEEE Cluster 2013, Indianapolis, IN
Massive Graphs are everywhere

- Facebook has a billion users and a trillion connections
- Twitter has more than 200 million users
Large(r) Networks and Centrality

- Who is more important in a network? Who controls the flow between nodes?
  - Centrality metrics answer these questions
  - Closeness Centrality (CC) is an intriguing metric

- How to handle changes?
  - Incremental algorithms are good but not enough in practice
  - Parallelism is essential
Closeness Centrality

- Let $G=(V, E)$ be a graph with vertex set $V$ and edge set $E$
  - Farness (far) of a vertex is the sum of shortest distances to each vertex
    \[
    \text{far}[u] = \sum_{v \in V, d_G(u,v) \neq \infty} d_G(u, v)
    \]
  - Closeness centrality (cc) of a vertex:
    \[
    \text{cc}[u] = \frac{1}{\text{far}[u]}
    \]
- Best algorithm: All-pairs shortest paths
  - $O(|V| \cdot |E|)$ complexity for unweighted networks
- For large and dynamic networks
  - From scratch computation is infeasible
  - Faster solutions are essential
CC Algorithm

Algorithm 1: CC: Basic centrality computation

Data: $G = (V, E)$
Output: $cc[.]$

1. for each $s \in V$ do
   
   ▶ SSSP$(G, s)$ with centrality computation
   
   $Q \leftarrow$ empty queue
   
   $d[v] \leftarrow \infty, \forall v \in V \setminus \{s\}$
   
   $Q.push(s), d[s] \leftarrow 0$
   
   $far[s] \leftarrow 0$

   while $Q$ is not empty do

   $v \leftarrow Q.pop()$

   for all $w \in \Gamma_G(v)$ do

   if $d[w] = \infty$ then

   $Q.push(w)$

   $d[w] \leftarrow d[v] + 1$

   $far[s] \leftarrow far[s] + d[w]$

   \[ cc[s] = \frac{1}{far[s]} \]

2. return $cc[.]$
Incremental Closeness Centrality

- Computing cc values from scratch after each edge change is very costly
  - Incremental algorithms are used to handle changes
  - Main idea is to reduce number of SSSPs to be executed

- Three filtering techniques are proposed
  - Filtering with level differences
  - Filtering with biconnected components
  - Filtering with identical vertices

- Details can be found at
Filtering with level differences

• Upon edge insertion, breadth-first search tree of each vertex will change. Three possibilities:

  Case 1
  0———s
  1———
  2———
  3———u
  4———v

  Case 2
  0———s
  1———u
  2———v
  3———
  4———

  Case 3
  0———s
  1———u
  2———w
  3———v
  4———

• Case 1 and 2 will not change cc of s!
  • No need to apply SSSP from them

• Just Case 3
  • BFSs are executed from u and v and level diff is checked
Filtering with biconnected components

- What if the graph have articulation points?

- Change in A can change cc of any vertex in A and B

- Computing the change for $u$ is **enough** for finding changes for any vertex $v$ in B (constant factor is added)
Filtering with identical vertices

• Two types of identical vertices:
  • Type I: $u$ and $v$ are identical vertices if $N(u) = N(v)$, i.e., their neighbor lists are same
  • Type II: $u$ and $v$ are identical vertices if $\{u\} \cup N(u) = \{v\} \cup N(v)$, i.e., they are also connected

• If $u$ and $v$ are identical vertices, their cc are the same
  • Same breadth-first search trees!
Is it enough?

| name              | $|V|$   | $|E|$   | Time (in sec.) |
|-------------------|------|-------|----------|
| web-NotreDame     | 325K | 1,090K| 53.0     |
| amazon0601        | 403K | 2,443K| 298.1    |
| web-Google        | 875K | 4,322K| 824.4    |

- Too slow for real-time processing
- The problem is mostly parallel and graphs are relatively small.
  - Source-level parallelism can be used to fill up a cluster
DataCutter

• Component-based middleware tool
  • Supports filter-stream programming
  • Implements the computation as a set of components (filters) that exchange data through logical streams (unidirectional data flows)

• Layout is a filter ontology
  • Describes the set of tasks, streams and the connections
  • All replicable
STREAMER Framework

Sends the updates on the graph to everyone

Filters the work using level difference, BCD and identical vertices
- List of vertices needing SSSP update are sent to ComputeCC
- # of updates are sent to Aggregator

Stores the farness values of all vertices and does adjustments to identical vertices and biconnected components

**Fig. 6.** STREAMER: a Distributed Framework for Incremental Closeness Centrality Computation
Multicore architecture and NUMA effects

- Preparator makes the actual graph
- Pointers are shared between Executors
Experiments

• Dataset

| Name               | |V|   | |E|   | % of computation saved |
|--------------------|-----------------|-----|-----|-----------------------|
| web-NotreDame      | 325,729         | 1,090,008 | 97.5 |
| amazon0601         | 403,394         | 2,443,308 | 92.3 |
| web-Google         | 916,428         | 4,321,958 | 94.4 |
| soc-pokec          | 1,632,804       | 30,622,464| 93.9 |

• 64 node cluster

  • Each with dual Intel Xeon E5520 Quad-Core processor
  • 8MB L3 cache per processor
  • 48GB main memory, 20Gbps Infiniband Connection
  • Compiled with GCC 4.5.2 with -O3 flag
Performance Results

(a) amazon0601

456x speedup

(b) web-NotreDame

316x speedup

linear scaling

sublinear scaling (will be explained)

(c) web-Google

497x speedup

(d) soc-pokec

IEEE Cluster'13

STREAMER: a Distributed Framework for Incremental Closeness Centrality Computation
The performance of Streamer with 31 worker nodes and different node-level configurations normalized to 1 thread case (performance on soc-pokec is normalized to 8 threads, 1 graph/thread). The last column is the advantage of Shared Memory awareness (ratio of columns 5 and 3).

<table>
<thead>
<tr>
<th>Name</th>
<th>4 threads</th>
<th>8 threads, 1 graph per thread</th>
<th>Shared Mem. awareness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>thread</td>
<td>node</td>
</tr>
<tr>
<td>web-NotreDame</td>
<td>3.69</td>
<td>6.46</td>
<td>7.13</td>
</tr>
<tr>
<td>amazon0601</td>
<td>3.26</td>
<td>6.75</td>
<td>6.81</td>
</tr>
<tr>
<td>web-Google</td>
<td>3.69</td>
<td>7.77</td>
<td>7.55</td>
</tr>
<tr>
<td>soc-pokec</td>
<td>-</td>
<td>1.00</td>
<td>0.92</td>
</tr>
</tbody>
</table>

- Exploiting multiple cores and properly taking the shared-memory aspect brings significant improvement.
Sublinear scaling case, 3 nodes

Runtime is dominated by processing updates

Walltime (in seconds)

(a) 3 worker nodes

Update emitted
Update processed
SE start

Runtime is dominated by processing updates

i.e., # of jobs submitted

Update emitted
Update processed
SE start

(a) 3 worker nodes

Stream Event

Update emitted
Update processed
SE start

Walltime (in seconds)
Sublinear scaling case, 15 nodes

(b) 15 worker nodes

Sublinear scaling is due to the insufficient number of emitted updates!
StreamingMaster becomes the main bottleneck, i.e., not fast enough to send updates to workers.

(c) 63 worker nodes

Streaming Event

Walltime (in seconds)

Update

Update emitted

Update processed

SE start

Plateau 1

Plateau 2

Plateau 3

Pipelined parallelism

(c) 63 worker nodes
Conclusion

• STREAMER, a distributed-memory framework, proves to be an effective solution for fast and exact incremental closeness centrality computation
  • Exploits replicated and pipelined parallelism
  • Scales well
  • Reaches speedup of 497 with 64 nodes and 8 cores/node

• Future Work
  • StreamingMaster and Aggregator can be replicated and work can be partitioned
  • Biconnected Decomposition, main part of Aggregator, can be parallelized as well
Thanks

• For more information
  • Email umit@bmi.osu.edu
  • Visit http://bmi.osu.edu/~umit or http://bmi.osu.edu/hpc

• Acknowledgement of Support