Scalable Overlapping Community Detection

Ismail El-Helw ¹  Rutger Hofman ¹  Wenzhe Li ²
Sungjin Ahn ³  Max Welling ⁴  Henri Bal ¹

¹Vrije Universiteit Amsterdam ²University of Southern California
³University of Montreal ⁴University of Amsterdam

May 27, 2016
My Background
An Introduction

- Studied programming environments for distributed systems (Orca, Ibis, JavaGAT, Glasswing, ....)
- Studied many applications (astronomy, bioinformatics, climate modeling, digital forensics; game tree search, model checking, semantic web, multimedia analysis)
- Success stories
  - Solved game of Awari (2003, selected in Pickover’s math book)
  - Multimedia content analysis with Ibis on a world-wide scale (CCGrid SCALE challenge 2008)
  - Calculating the Closure of 100 Billion Semantic Web Triples (CCGrid SCALE challenge 2010)
- This work: trying to do overlapping community detection on Friendster data set (1.8 billion edges)
Network Communities
An Introduction

- Modeled as a graph of vertices and edges
- Densely connected group of vertices
- Sparsely connected to rest of graph
- Graph partitioning / clustering
- What do you mean by overlapping?
Network Communities
An Example
Network Communities
Can we understand sub-community structure
Target Graphs

<table>
<thead>
<tr>
<th>Name</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>#Ground-truth communities</th>
</tr>
</thead>
<tbody>
<tr>
<td>com-LiveJournal</td>
<td>3,997,962</td>
<td>34,681,189</td>
<td>287,512</td>
</tr>
<tr>
<td>com-Friendster</td>
<td>65,608,366</td>
<td>1,806,067,135</td>
<td>957,154</td>
</tr>
<tr>
<td>com-Orkut</td>
<td>3,072,441</td>
<td>117,185,083</td>
<td>6,288,363</td>
</tr>
<tr>
<td>com-Youtube</td>
<td>1,134,890</td>
<td>2,987,624</td>
<td>8,385</td>
</tr>
<tr>
<td>com-DBLP</td>
<td>317,080</td>
<td>1,049,866</td>
<td>13,477</td>
</tr>
<tr>
<td>com-Amazon</td>
<td>334,863</td>
<td>925,872</td>
<td>75,149</td>
</tr>
<tr>
<td>ca-HepPh</td>
<td>9,877</td>
<td>25,998</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Summary of SNAP graph data sets used for evaluation.
Algorithm 1 Sequential version of SG-MCMC for a-MMSB

1: Initialize $\pi, \beta, \phi, \theta$
2: while sampling do
3: Sample mini-batch of vertex pairs $E_n$ from $E$ or $E$
4: for each vertex in $E_n$ do
5: Sample mini-batch of vertices $V_n$ from $V$
6: Update $\phi_a$ using Eq. 5
7: Obtain $\pi_a$ from $\phi_a^*$
8: end for
9: for $k = 1, \ldots, K$ do
10: Update $\theta_k$ using Eq. 3
11: Obtain $\beta_k$ from $\theta_k^*$
12: end for
13: end while

<table>
<thead>
<tr>
<th>symbol</th>
<th>type</th>
<th>size</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>0/1/2/3/4</td>
<td>$K$</td>
<td>set of communities</td>
</tr>
<tr>
<td>$V$</td>
<td>{vertex}</td>
<td>$N$</td>
<td>vertices in the graph</td>
</tr>
<tr>
<td>$E$</td>
<td>{edge}</td>
<td></td>
<td>linked edges in the graph</td>
</tr>
<tr>
<td>$E_h$</td>
<td>{edge}</td>
<td></td>
<td>$V \times V$: linked and nonlinked edges</td>
</tr>
<tr>
<td>$E_n$</td>
<td>{edge}</td>
<td></td>
<td>held-out subset of the graph</td>
</tr>
<tr>
<td>$M$</td>
<td></td>
<td></td>
<td>number of vertices in $E_n$</td>
</tr>
<tr>
<td>$V_n$</td>
<td>{vertex}</td>
<td></td>
<td>sampled neighbor set for a vertex in $E_n$</td>
</tr>
<tr>
<td>$\theta$</td>
<td>float vector</td>
<td>$K \times 2$</td>
<td>reparameterization of $\beta$ $\beta[k] = \theta[k][1] / \sum_j \theta[k][j]$</td>
</tr>
<tr>
<td>$\beta$</td>
<td>float vector</td>
<td>$K$</td>
<td>community strength</td>
</tr>
<tr>
<td>$\phi$</td>
<td>float vector</td>
<td>$N \times K$</td>
<td>reparameterization of $\pi$ $\pi[i][k] = \phi[i][k] / \sum_j \phi[i][j]$</td>
</tr>
<tr>
<td>$\pi$</td>
<td>float vector</td>
<td>$N \times K$</td>
<td>$\pi[i][k]$ is probability that vertex $i$ is in community $k$</td>
</tr>
</tbody>
</table>
Talk Structure

- Single-node acceleration
  - to solve: speed issues
  - outside scope of this talk: GPU?
- Distributed cluster implementation
  - to solve: problem size limits
From Python to sequential C++
Python (numpy) implementation

- Start out with Python (numpy) implementation
- Python data structures (sets etc) not numpy-accelerated
- Stepwise port to C++
  - literal translation, maintaining Python idioms
  - transform into efficient C++
    e.g. $x^y$ (where $y \in \{0, 1\}) \rightarrow (y == 0 ? 1 : x)$
  - switch to float32 $\Leftarrow \Leftarrow \Leftarrow \Leftarrow \Leftarrow \Leftarrow \Leftarrow$ baseline
  - multithreaded
Sequential times on one Intel Xeon E5-2630v3 node
From Python to C++

Graph=ca-HepPh, $K=1024$, $|\text{minibatch}|=32$, $|\text{neighbor sample}|=32$

<table>
<thead>
<tr>
<th></th>
<th>Python</th>
<th>float64 (baseline)</th>
<th>float32 baseline</th>
<th>multi-threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms/iteration</td>
<td>9,634</td>
<td>95.8</td>
<td>10.4</td>
<td>6.1</td>
</tr>
</tbody>
</table>

This problem too small for multithreaded efficiency!
(max. achievable: $\lesssim 11 \times$ speedup wrt. baseline)
Cluster Implementation
The Challenges

- Data distribution
- No data locality
- Synchronization & data dependencies
Each cluster node hosts a subset of the program’s state using RDMA Distributed Key-Value Store
Each cluster node hosts a subset of the program’s state using RDMA Distributed Key-Value Store

Use MPI for collective operations & synchronization
Each cluster node hosts a subset of the program’s state using RDMA Distributed Key-Value Store
Use MPI for collective operations & synchronization
Use OpenMP for multi-core parallelism
Each cluster node hosts a subset of the program’s state using RDMA Distributed Key-Value Store

- Use MPI for collective operations & synchronization
- Use OpenMP for multi-core parallelism
- Use pipelining to overlap remote data loads with computations
Cluster Implementation: Distributed Data Storage
RDMA Distributed Key-Value Store

Distributed Key-Value Store:
- static layout: Key = graph vertex $i$, Value = $\phi[i][K]$

Data accesses in disjoint phases:
- scattered read phase
- write phase without conflicts

RDMA (Remote Direct Memory Access):
- transfer from/to remote memory through network card
  without host involvement

Primitive RDMA reads/writes without data synchronization!
Experimental Setup
VU DAS5 Cluster

- 65-node DAS-5 cluster
  see: H. Bal et al., A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term, IEEE Computer, May 2016
- Each node:
  - Dual 8-core Intel Xeon E5-2630v3 2.4GHz CPU
  - 64GB of memory
- 4TB Storage
- FDR Infiniband
#communities K scales with number of nodes.
Cluster Implementation
Strong Scalability

com-Friendster \(K=1024\) \(|m\text{-batch}|=16384\) \(|\text{neighb}|=32\) \(\text{iterations}=2048\)

(a) iteration time vs. cluster size  (b) speedup w.r.t. 8 nodes
Performance of DKV store vs. \textit{qperf} and MPI roundtrip.
Execution time vs. $K$ on 64 nodes, with and without pipelining optimizations.
1TB/40-core machine (SURFsara HPC Cloud) vs. DAS5

(a) HPC Cloud (16 and 40 cores) vs. 1 DASS node (16 cores) using com-DBLP

(b) HPC Cloud (40 cores) vs. 64 DASS nodes (64×16 cores) using com-Friendster

(a) 1 DAS5 node
16 cores

(b) cluster of 64 DAS5 nodes
64 × 16 cores
MCMC Algorithm Convergence

(a) com-Friendster | 12K communities on 64+1 nodes

(b) com-LiveJournal | 192K communities on 64+1 nodes

(c) com-Orkut | 256K communities on 64+1 nodes

(d) com-Youtube | 8,385 communities on 12+1 nodes

(e) com-DBLP | 13,477 communities on 20+1 nodes

(f) com-Amazon | 75,149 communities on 20+1 nodes
Conclusions

- Overlapping community detection: existing Python SG-MCMC program
- Transform into efficient C++
- Distributed implementation:
  - handle very large graphs/#communities
  - distribute data storage
  - RDMA primitives for remote memory access
  - further "straightforward" parallel optimizations (MPI, OpenMP, pipelining)
- First to solve problems at this scale (Friendster)