Streaming Community Detection for Partitioning Parallel Filesystems

Tim Shaffer
A (well-meaning) user tried to run a bioinformatics pipeline to analyze a batch of genomic data.
Shared filesystem performance became degraded, with other users unable to access the filesystem.
That user got a strongly worded email and had to stop their analyses.
Certain program behaviors produce **large bursts** of metadata I/O activity (e.g. library search).

These behaviors can occur at the same time across **multiple workers** (e.g. startup, new analysis phase).

Techniques such as worker-side caching and pre-staging can help, but how do we know which parts of the FS would benefit?
Analysis pipelines have many components, show data-dependent and non-deterministic behavior.

Each run makes a large number of filesystem accesses (millions).

We need to use `strace`-type events of numerous analyses to profile filesystem interactions.
29204  open("/etc/passwd", O_RDONLY|O_CLOEXEC) = 3
In this graph, nodes are filesystem entries (inodes).

Edge weights indicate the number of times the access pattern

\[ \text{inode } A \rightarrow \text{inode } B \]

occurred over all runs.

Amenable to streaming updates
Groups of filesystem entries frequently accessed together are visible as communities in the execution graph.

**Hierarchical community detection** allows us to identify good shards/partitions for manual distribution.

Streaming algorithm exists for this...
Sequential algorithm: Girvan–Newman

Progressively removes edges from graph

The remaining components are the communities.

Uses edge betweenness: the number of shortest paths between pairs of nodes that run along an edge
Pseudocode of algorithm (courtesy of Wikipedia)

1. For each edge $E$ in $G$, compute the betweenness of $E$.
2. Remove the edge with highest betweenness from $G$.
3. Recalculate betweenness for edges affected by the removal.
4. Repeat Steps 2 and 3 until no edges remain.

Results in a dendrogram showing successively finer clusters
Computing edge centrality is expensive, must be (partially) computed after each edge removal.

Sequential algorithm (Girvan–Newman) runs in $O(VE^2)$ or $O(V^3)$ in a sparse graph.

STINGER supports streaming updates and parallel agglomerative clustering.
## Data Sets

<table>
<thead>
<tr>
<th></th>
<th>Events</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>47</td>
<td>46</td>
<td>45</td>
</tr>
<tr>
<td>bash</td>
<td>5,499</td>
<td>840</td>
<td>1,569</td>
</tr>
<tr>
<td>MAKER</td>
<td>1,813,544</td>
<td>24897</td>
<td>129,153</td>
</tr>
</tbody>
</table>
(Poorly done) Visualization of MAKER Graph
(Poorly done) Visualization of MAKER Graph
NetworkX includes an implementation of Girvan–Newman!

Straightforward Python implementation parses the `strace` logs, constructs the graph, and invokes Girvan–Newman.

Dendrogram (arbitrarily) limited to $k=5$ levels deep.
Performance Results

<table>
<thead>
<tr>
<th></th>
<th>Girvan–Newman</th>
<th>1st Betweenness Centrality</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>0.43 s</td>
<td>0.41 s</td>
</tr>
<tr>
<td>bash</td>
<td>290 s</td>
<td>10. s</td>
</tr>
<tr>
<td>MAKER</td>
<td>?? ( &gt; 45 min)</td>
<td></td>
</tr>
</tbody>
</table>

Results are (roughly) consistent with $O(VE^2)$ running time.
Enhanced Implementation

STINGER allows for streaming and parallel operations.

C rather than Python.

Heuristics to reduce the size of the graph might help.
http://www.yandell-lab.org/software/maker.html

https://en.wikipedia.org/wiki/Girvan%E2%80%93Newman_algorithm

http://www.stingergraph.com/
