Outline

• Introduction

• Network Analysis

• Static Parallel Algorithms
  – Graph Traversal / Graph500
  – Social Networking Algorithms
  – Graph Libraries
  – (many backup slides available...)

• Dynamic Parallel Algorithms

• GraphCT & STINGER
Outline of Static Parallel Algorithms

• Rules of thumb for massive graphs
  – Advice, not requirements...

• Basic graph algorithms
  – Breadth-first search (BFS) (Graph500)
  – Single-Source Shortest Path (\(\Delta\)-stepping algorithm) (new Graph500)

• Social network analysis
  – Betweenness Centrality (SSCA#2)
  – Community Detection (DIMACS mix challenge winner)

• Choose: SSSP, BC, or community detection...

• Backup
  – Spanning Tree (ST), Connected Components (CC)
  – Minimum Spanning Tree (MST), Minimum Spanning Forest (MSF)
  – Biconnected Components
  – Seed Set Expansion
  – K-Betweenness Centrality
What to avoid in algorithms...

- “We order the vertices (or edges) by...” unless followed by bisecting searches.
- “We look at a region of size *more than two steps...” Many target massive graphs have diameter of around 20. More than two steps swallows much of the graph.
- “Our algorithm requires *more than Õ(|E|/#)...” Massive means you hit asymptotic bounds, and |E| is plenty of work.
- “For each vertex, we *do something sequential...” The few high-degree vertices will be large bottlenecks.

Rules of thumb may be broken... with reasons.
What to avoid in *implementation*...

- Scattered memory accesses through traditional sparse matrix representations like CSR. Use your cache lines.

  32b idx  32b idx  ...  32b idx  32b idx  64b weight  64b weight  ...

  64b weight  64b weight  ...

  Also can reduce register pressure.

- Using too much memory, which is a painful trade-off with parallelism. Think Fortran and workspace...

- Synchronizing too often. There will be work imbalance; try to use the imbalance to reduce “hot-spotting” on locks or cache lines.

Rules of thumb may be broken... with reasons.
ALGORITHMS: GRAPH TRAVERSAL
Graph traversal (BFS) problem definition

Input:

Output:

Memory requirements (minimum # of data items, uncompressed):
• Sparse graph representation: m+n
• Stack of visited vertices: n
• Distance array: n
Parallel BFS Strategies

1. Expand current frontier (level-synchronous approach, suited for low diameter graphs)

   - O(D) parallel steps
   - Adjacencies of all vertices in current frontier are visited in parallel

2. Stitch multiple concurrent traversals (Ullman-Yannakakis approach, suited for high-diameter graphs)

   - path-limited searches from “super vertices”
   - APSP between “super vertices”
Expand current frontier (**level-synchronous** approach, suited for **low diameter** graphs)

- Best-performing Graph500 implementations follow this approach.
  - Distributed memory, 2-D: Painful implementation. [Buluç & Madduri]
  - Shared memory: Simple, an afternoon's effort.

- **New idea** from Scott Beamer at UCB:
  - Once you've covered half the graph, stop expanding forward. Instead, parallelize over the unincluded vertices and look back. [Beamer, Asanović, Patterson]

- **O(D)** parallel steps
- Adjacencies of all vertices in current frontier are visited in parallel
Parallel Single-source Shortest Paths (SSSP) algorithms

- No known PRAM algorithm that runs in sub-linear time and $O(m+n\log n)$ work
- Parallel priority queues: relaxed heaps [DGST88], [BTZ98]
- Ullman-Yannakakis randomized approach [UY90]
- Meyer et al. $\Delta$ - stepping algorithm [MS03]
- Distributed memory implementations based on graph partitioning
- Heuristics for load balancing and termination detection

Δ - stepping algorithm [MS03]

- **Label-correcting** algorithm: Can relax edges from unsettled vertices also
- Δ - stepping: “approximate bucket implementation of Dijkstra’s algorithm”
  - Δ: bucket width
- Vertices are ordered using buckets representing priority range of size Δ
  - Avoids heroic data structures.
- Each bucket may be processed in parallel
Δ - stepping algorithm: illustration

\[ Δ = 0.1 \text{ (say)} \]

One parallel **phase**

**while** (bucket is non-empty)

i) Inspect light edges

ii) Construct a set of “requests” (R)

iii) Clear the current bucket

iv) Remember deleted vertices (S)

v) Relax request pairs in R

Relax heavy request pairs (from S)

Go on to the next bucket

\[ \Delta = 0.1 \text{ (say)} \]

\[ 0 1 2 3 4 5 6 \]

\[ \infty \infty \infty \infty \infty \infty \infty \]

**D** array

\[ 0 1 2 3 4 5 6 \]

**Buckets**
Δ - stepping algorithm: illustration

One parallel **phase**

while (bucket is non-empty)

i) Inspect light edges

ii) Construct a set of “requests” (R)

iii) Clear the current bucket

iv) Remember deleted vertices (S)

v) Relax request pairs in R

Relax heavy request pairs (from S)

Go on to the next bucket

*Initialization:*

Insert s into bucket, $d(s) = 0$
$\Delta$ - stepping algorithm: illustration

One parallel phase

while (bucket is non-empty)

i) Inspect light edges

ii) Construct a set of “requests” ($R$)

iii) Clear the current bucket

iv) Remember deleted vertices ($S$)

v) Relax request pairs in $R$

Relax heavy request pairs (from $S$)

Go on to the next bucket

$\mathbf{d}$ array

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Buckets

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$R$

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S$

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Parallel Programming for Graph Analysis
\[\Delta - \text{stepping algorithm: illustration}\]

One parallel **phase**

\[\textbf{while} \text{ (bucket is non-empty)}\]

i) Inspect light edges

ii) Construct a set of “requests” (R)

iii) Clear the current bucket

iv) Remember deleted vertices (S)

v) Relax request pairs in R

Go on to the next bucket

---

<table>
<thead>
<tr>
<th>d array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0    1   2   3   4   5   6</td>
</tr>
<tr>
<td>0   ∞    ∞   ∞   ∞   ∞    ∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Δ - stepping algorithm: illustration

One parallel phase
while (bucket is non-empty)
  i) Inspect light edges
  ii) Construct a set of “requests” (R)
  iii) Clear the current bucket
  iv) Remember deleted vertices (S)
  v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket
Δ - stepping algorithm: illustration

One parallel **phase**

while (bucket is non-empty)

i) Inspect light edges

ii) Construct a set of “requests” (R)

iii) Clear the current bucket

iv) Remember deleted vertices (S)

v) Relax request pairs in R

Relax heavy request pairs (from S)

Go on to the next bucket
Δ - stepping algorithm: illustration

One parallel phase
while (bucket is non-empty)
   i) Inspect light edges
   ii) Construct a set of “requests” (R)
   iii) Clear the current bucket
   iv) Remember deleted vertices (S)
   v) Relax request pairs in R
Relax heavy request pairs (from S)
Go on to the next bucket
**Δ - stepping algorithm: illustration**

One parallel **phase**

**while** (bucket is non-empty)

i) Inspect light edges

ii) Construct a set of “requests” (R)

iii) Clear the current bucket

iv) Remember deleted vertices (S)

v) Relax request pairs in R

Relax heavy request pairs (from S)

Go on to the next bucket
Δ - stepping algorithm: illustration

One parallel phase
while (bucket is non-empty)
  i) Inspect light edges
  ii) Construct a set of “requests” (R)
  iii) Clear the current bucket
  iv) Remember deleted vertices (S)
  v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

- d array
  0 1 2 3 4 5 6
  0 .03 .01 .06 .16 .29 .62

- Buckets
  1 4
  2 5
  6 6

- R

- S
  0 2 1 3
No. of phases (machine-independent performance count)
Last non-empty bucket (machine-independent performance count)

Fewer buckets, more parallelism
<table>
<thead>
<tr>
<th>Problem</th>
<th>Graph</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multithreaded BFS</td>
<td>Random graph, 256M vertices, 4B edges</td>
<td>0.84 s, 5.1 B TEPS</td>
<td>Optimized for the Graph500 generator</td>
</tr>
<tr>
<td>[BAP11]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>External Memory BFS</td>
<td>Random graph, 256M vertices, 1B edges</td>
<td><strong>8.9 hrs</strong> (3.2 GHz Xeon)</td>
<td>State-of-the-art external memory BFS</td>
</tr>
<tr>
<td>[ADM06]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>External memory: NAND flash (LLNL &amp; FusionIO)</td>
<td>Random graph, 68.7B vertices, 1.1T edges</td>
<td>0.5 hrs, 609M TEPS (64 nodes)</td>
<td>Custom Graph500 submission</td>
</tr>
<tr>
<td>[PGA10]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multithreaded SSSP</td>
<td>Random graph, 256M vertices, 1B edges</td>
<td><strong>11.96 sec</strong> (40p) MTA-2</td>
<td>Works well for all low-diameter graph families</td>
</tr>
<tr>
<td>[MBBC06]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel Dijkstra</td>
<td>Random graph, 240M vertices, 1.2B edges</td>
<td><strong>180 sec</strong>, 96p 2.0GHz cluster</td>
<td>Best known distributed-memory SSSP implementation for large-scale graphs</td>
</tr>
<tr>
<td>[EBGL06]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ALGORITHMS:
SOCIAL NETWORK ANALYSIS (CENTRALITY)
Finding “central” entities is a key graph analytics routine

- **Centrality**: Quantitative measure to capture the importance of a vertex/edge in a graph.
  - Application-specific: can be based on degree, paths, flows, eigenvectors, ...

**Intelligence**
- **Problem**: Unraveling terrorist networks.

**Bioinformatics**
- **Problem**: Identifying drug target proteins, metabolic pathways.

**US power transmission grid**
- **Problem**: Contingency analysis

**Online Social networks**
- **Problem**: Discover emergent communities, identify influential people.

---


Image Source: http://www.orgnet.com/hijackers.html

Image Source: http://www.orgnet.com/hijackers.html

Parallel Programming for Graph Analysis
Centrality in Massive Social Network Analysis

- **Centrality metrics**: Quantitative measures to capture the importance of person in a social network
  - Betweenness is a global index related to shortest paths that traverse through the person
  - Can be used for community detection as well
- Identifying *central* nodes in large complex networks is the key metric in a number of applications:
  - Biological networks, protein-protein interactions
  - Sexual networks and AIDS
  - Identifying key actors in terrorist networks
  - Organizational behavior
  - Supply chain management
  - Transportation networks
- Current Social Network Analysis (SNA) packages handle 1,000’s of entities, our techniques handle BILLIONS (6+ orders of magnitude larger data sets)
Betweenness Centrality (BC)

- Key metric in social network analysis
  [Freeman ’77, Goh ’02, Newman ’03, Brandes ’01]

\[ BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \]

- \( \sigma_{st} \): Number of shortest paths between vertices \( s \) and \( t \)
- \( \sigma_{st}(v) \): Number of shortest paths between vertices \( s \) and \( t \) passing through \( v \)

- Exact BC is compute-intensive
BC Algorithms

- Brandes [2001] proposed a faster sequential algorithm for BC on sparse graphs
  - $O(mn + n^2 \log n)$ time and $O(n)$ space for weighted graphs
  - $O(mn)$ time for unweighted graphs
- We designed and implemented the first parallel algorithm:
  - [Bader, Madduri; ICPP 2006]

- Approximating Betweenness Centrality
  - [Bader Kintali Madduri Mihail 2007]
  - Novel approximation algorithm for determining the betweenness of a specific vertex or edge in a graph
  - Adaptive in the number of samples
  - Empirical result: At least 20X speedup over exact BC

Graph: 4K vertices and 32K edges,
System: Sun Fire T2000 (Niagara 1)
An undirected graph of 1.54 million vertices (movie actors) and 78 million edges. An edge corresponds to a link between two actors, if they have acted together in a movie.
Fine-grained Parallel BC Algorithm

• Consider an undirected, unweighted graph
• High-level idea: Level-synchronous parallel Breadth-First Search augmented to compute centrality scores
• Exact BC computation
  - \( n \) source vertices (iterations)
  - Each iteration:
    • traversal and path counting
    • dependency accumulation

\[
\delta(v) = \sum_{w \in P(v)} \frac{\sigma(v)}{\sigma(w)}(1 + \delta(w))
\]
Illustration of Parallel BC (pBC-Old)

1. **Traversal step**: visit adjacent vertices, update distance and path counts.

![Graph Diagram](image)
1. **Traversal step**: visit adjacent vertices, update distance and path counts.
Step 1 Illustration

1. **Traversal step**: visit adjacent vertices, update distance and path counts.

**Level-synchronous approach**: The adjacencies of all vertices in the current frontier can be visited in parallel.
1. **Traversal step**: at the end, we have all reachable vertices, their corresponding predecessor multisets, and D values.

**Level-synchronous approach**: The adjacencies of all vertices in the current frontier can be visited in parallel.
Step 1 pBC-Old pseudo-code

for all vertices $u$ at level $d$ \textit{in parallel do}

for all adjacencies $v$ of $u$ \textit{in parallel do}

$dv = D[v]$;

if ($dv < 0$) \hspace{1em} // $v$ is visited for the first time

$vis = \text{fetch\_and\_add}(&\text{Visited}[v], 1)$;

if ($vis == 0$) \hspace{1em} // $v$ is added to a stack only once

$D[v] = d+1$;

$pS[count++] = v$; \hspace{1em} // Add $v$ to local thread stack

fetch\_and\_add($&\sigma[v], \sigma[u]$);

fetch\_and\_add($&\text{Pcount}[v], 1$); \hspace{1em} // Add $u$ to predecessor list of $v$

if ($dv == d + 1$)

fetch\_and\_add($&\sigma[v], \sigma[u]$);

fetch\_and\_add($&\text{Pcount}[v], 1$); \hspace{1em} // Add $u$ to predecessor list of $v$
Step 1 analysis

• Exploit concurrency in exploration of current frontier and visiting adjacencies, as the graph diameter is low: \(O(\log n)\) or \(O(1)\).

• Potential performance bottlenecks: atomic updates to predecessor multisets, atomic increments of path counts.

• Major improvement: Data structure change to eliminate storage of “predecessor” multisets. We store successor edges along shortest paths instead.
  - simplifies the accumulation step
  - Eliminates two atomic operations in traversal step
  - cache-friendly!
pBC-LockFree change in data representation

source vertex

P: Predecessor multiset

Succ: Successor multiset

Parallel Programming for Graph Analysis
Step 1 pBC-LockFree Locality Analysis

for all vertices \( u \) at level \( d \) in parallel do

for all adjacencies \( v \) of \( u \) do

\[ dv = D[v]; \]

if (\( dv < 0 \))

\[ \text{vis} = \text{fetch\_and\_add}(&\text{Visited}[v], 1); \]

if (\( \text{vis} == 0 \))

\[ D[v] = d+1; \]

\[ pS[\text{count}++] = v; \]

\[ \text{fetch\_and\_add}(&\text{sigma}[v], \text{sigma}[u]); \]

\[ \text{Scount}[u]++; \]

if (\( dv == d + 1 \))

\[ \text{fetch\_and\_add}(&\text{sigma}[v], \text{sigma}[u]); \]

\[ \text{Scount}[u]++; \]

All the vertices are in a contiguous block (stack)

All the adjacencies of a vertex are stored compactly (graph rep.)

Non-contiguous memory access

Non-contiguous memory access

Non-contiguous memory access

Indicates store to \( S[u] \)

Store \( D[v] \), \( \text{Visited}[v] \), \( \text{sigma}[v] \) contiguously for better cache locality.
2. Accumulation step: Pop vertices from stack, update dependence scores.

\[ \delta(v) = \sum_{w \in P(v)} \frac{\sigma(v)}{\sigma(w)} (1 + \delta(w)) \]

- **S**: stack of visited vertices
- **Delta**: Dependency score
- **P**: Predecessor multiset

Source vertex: 0
2. Accumulation step: Can also be done in a level-synchronous manner.

\[ \delta(v) = \sum_{w \in P(v)} \frac{\sigma(v)}{\sigma(w)}(1 + \delta(w)) \]

- S: stack of visited vertices
- Delta: Dependency score
- P: Predecessor multiset

Source vertex

Parallel Programming for Graph Analysis
Step 2 pBC-Old pseudo-code

for level $d = \text{GraphDiameter}$ to 2 do

for all vertices $w$ at level $d$ in parallel do

for all $v$ in $P[w]$ do

    acquire_lock(v);

    $\delta[v] = \delta[v] + (1 + \delta[w]) \times \sigma(v)/\sigma(w)$;

    release_lock(v);

    $\delta[v] \rightarrow BC[v]$ = $\delta[v]$ \\

    $\delta(v) = \sum_{v \in P(w)}\frac{\sigma(v)}{\sigma(w)}(1 + \delta(w))$
Step 2 pBC-LockFree

for level $d = \text{GraphDiameter}-2$ to 1 do

for all vertices $v$ at level $d$ in parallel do

for all $w$ in $S[v]$ in parallel do reduction($\delta$)

$$\delta_{\text{sum}}_v = \delta[v] + (1 + \delta[w]) \times \sigma[v]/\sigma[w]$$

$\text{BC}[v] = \delta[v] = \delta_{\text{sum}}_v$;
New parallel BC algorithm works well for massive “small-world” networks

• Low graph diameter.
  - Key source of concurrency in graph traversal.
• Skewed (“power law”) degree distribution of the number of neighbors.
  - Inner loop easier to parallelize after elimination of successor multisets. Preprocess for balanced partitioning of work among processors/threads.
  - High-degree vertices can be processed in parallel, separately.
• Dynamic network abstractions, from diverse data sources; massive networks (billions of entities).
  - Data representations and structures are space-efficient, support edge attributes, and fast parallel insertions and deletions.
Performance Results: Experimental Setup

**Cray XMT**

- **Latency tolerance** by massive multithreading
  - Hardware support for 128 threads on each processor
  - Globally hashed address space
  - No data cache
  - Single cycle context switch
  - Multiple outstanding memory requests
- **Support for fine-grained, word-level synchronization**
- **16 x 500 MHz processors, 128 GB RAM**

**DARPA HPCS SSCA#2 Graph Analysis benchmark**

- Representative of graph-theoretic computations in real-world networks.
  - [http://www.graphanalysis.org](http://www.graphanalysis.org)
- **Approximate** betweenness centrality is a key kernel.
- Synthetic R-MAT networks generated based on Kronecker products.
- Performance measure: Traversed edges per second (TEPS) rate.

\[
\text{BC TEPS rate} = \frac{7n \cdot 2^{K^{4 \text{Approx}}}}{t} \quad \text{edges/second}
\]

**IMDb actors network**

- Real-world social network constructed from IMDb data.
- Undirected network: 1.54 million vertices (actors) and 78 million edges (edge → two actors co-starring in a movie).
Performance compared to previous algorithm

- SSCA#2 network, SCALE 24 (16.77 million vertices and 134.21 million edges.)

![Speedup comparison graph]

- Speedup of 2.3 over previous approach.
• Synthetic network with 16.77 million vertices and 134.21 million edges (SCALE 24), K4Approx = 8.

Speedup of 10.43 on 16 processors.
Cray XMT Performance vs. Problem size

- SSCA#2 networks, \( n = 2^{\text{SCALE}} \) and \( m = 8n \).

Sufficient concurrency on 16 processors for problem instances with SCALE > 24.
Community Identification

- Implicit communities in large-scale networks are of interest in many cases.
  - WWW
  - Social networks
  - Biological networks

- Formulated as a graph clustering problem.
  - Informally, identify/extract “dense” subgraphs.

- Several different objective functions exist.
  - Metrics based on intra-cluster vs. inter-cluster edges, community sizes, number of communities, overlap ...

- Highly studied research problem
  - 100s of papers yearly in CS, Social Sciences, Physics, Comp. Biology, Applied Math journals and conferences.
Related Work: Partitioning Algorithms from Scientific Computing

• Theoretical and empirical evidence: existing techniques perform poorly on small-world networks

• [Mihail, Papadimitriou ’02] Spectral properties of power-law graphs are skewed in favor of high-degree vertices


• [Abou-Rjeili, Karypis ’06] Multilevel partitioning heuristics give large edge-cut for small-world networks, new coarsening schemes necessary
Modularity: A popular optimization metric

• Measure based on *optimizing inter-cluster density over intra-cluster sparsity*.

• For a weighted, directed network with vertices partitioned into non-overlapping clusters, modularity is defined as

\[
Q = \frac{1}{2w} \sum_{i \in V} \sum_{j \in V} \left( w_{ij} - \frac{w^\text{out}_i w^\text{in}_j}{2w} \right) \delta(C_i, C_j)
\]

\[
w^\text{out}_i = \sum_j w_{ij}, \quad w^\text{in}_j = \sum_i w_{ij}, \quad 2w = \sum_i \sum_j w_{ij}
\]

\[
\delta(C_i, C_j) = 1 \text{ if } C_i = C_j,
\]

\[
0 \text{ otherwise.}
\]

• If a particular clustering has no more intra-cluster edges than would be expected by random chance, \(Q=0\). Values greater than 0.3 typically indicate community structure.

• Maximizing modularity is *NP-complete*. 

For an unweighted and undirected network, modularity is given by

$$Q = \frac{1}{2m} \sum_{i \in V} \sum_{j \in V} \left( e_{ij} - \frac{d_i d_j}{2m} \right) \delta(C_i, C_j)$$

where

$$e_{ij} = 1 \text{ if } \langle i, j \rangle \in E$$

$$\delta(C_i, C_j) = 1 \text{ if } C_i = C_j,$$

$$0 \text{ otherwise.}$$

and in terms of clusters/modules, it is equivalently

$$Q = \sum_s \left( \frac{m_s}{m} - \left( \sum_{C_v = s} \frac{d_v}{2m} \right)^2 \right)$$

Resolution limit: Modules will not be found, optimizing modularity, if

$$m_s < \sqrt{\frac{m}{2}} - 1$$
Many approaches

• No single “right” community detection algorithm exists. Community structure analysis should be user-driven and application-specific.

• Approaches fall into categories:
  – **Divisive**: Repeatedly split the graph (*e.g.* spectral).
  – **Agglomerative**: Grow communities by merging vertices.
  – Other... Machine learning, mathematical programming, *etc.*

• Recent comparison: 10th DIMACS Impl. Challenge
  [http://www.cc.gatech.edu/dimacs10/](http://www.cc.gatech.edu/dimacs10/)
  Results very mixed, still open.
Divisive Clustering, Parallelization

- **Top-down** approach: Start with entire network as one community, recursively split the graph to yield smaller modules.

- **Two popular methods:**
  - **Edge-betweenness** based: iteratively remove high-centrality edges.
    
    \[
    BC(e) = \sum_{s,t \in V} \frac{\sigma_{st}(e)}{\sigma_{st}}
    \]

    • Centrality computation is the compute-intensive step, parallelize it.
  
  - **Spectral:** apply recursive spectral bisection on the “modularity matrix” \( B \), whose elements are defined as \( B_{ij} = A_{ij} - d_i d_j / 2m \).

    Modularity can be expressed in terms of \( B \) as:

    \[
    Q = \frac{1}{4m} s^T Bs
    \]

    • Parallelize the eigenvalue computation step (dominated by sparse matrix-vector products).
Agglomerative Clustering, Parallelization

- **Bottom-up** approach: Start with $|V|$ singleton communities, iteratively merge pairs to form larger communities.
  - What measure to minimize/maximize? **modularity**
  - How do we order merges? **priority queue** or **matching**

- Parallelization: perform **multiple “independent” merges** simultaneously.
Scalable Agglomeration

- Compute a score per edge (e.g. modularity change).
- Use a large-weight matching instead of a queue.
  - Greedily compute a matching within half the best.
  - Only approximating the result...
  - [http://www.cc.gatech.edu/~jriedy/community-detection/](http://www.cc.gatech.edu/~jriedy/community-detection/)
    (See papers in PPAM11, DIMACS challenge, MTAAP12.)
- Merge matched edges, producing a smaller community graph.
- Repeat until... Many choices.
  - Edge scores drop precipitously.
  - Enough of the graph is clustered.
- Differences from queuing implementations: maintains some balance, can contract too quickly at end.
Scalable Agglomeration: Data structures

- An array of \( (i, j; w) \) weighted edge pairs, each \( i, j \) stored only once and packed, uses \( 3|E| \) space.
- An array to store self-edges, \( d(i) = w, |V| \).
- A temporary floating-point array for scores, \( |E| \).
- A additional temporary arrays using \( 4|V| + 2|E| \) to store degrees, matching choices, offsets...

- Relatively simple, packed data.
- Weights count number of agglomerated vertices or edges.
- Scoring methods (modularity, conductance) need only vertex-local counts.
- Storing an undirected graph in a symmetric manner reduces memory usage drastically and works with our simple matcher.
Scalable Agglomeration: Data structures

• An array of \((i, j; w)\) weighted edge pairs, each \(i, j\) stored only once and packed, uses \(3|E|\) space

• An array to store self-edges, \(d(i) = w, |V|\)

• A temporary floating-point array for scores, \(|E|\)

• A additional temporary arrays using \(4|V| + 2|E|\) to store degrees, matching choices, offsets...

• Keep edge list in buckets by first stored vertex \((i)\).
  – Like CSR, but non-contiguous. Built without a prefix-sum (atomic fetch & add). Less synchronous.

• \textit{Hash} order of stored vertex indices... Breaks up high-degree edge lists.
Scalable Agglomeration: Routines

- Three core routines, similar to multi-level partitioning:
  - Scoring edges, trivial.
  - Computing a matching, greedy and quick.
  - Contracting the community graph, expensive.

- Repeat, stopping when no edge improves the metric enough, enough edges are in the clusters, ... Application-specific.

- Scoring: Compute the change in the optimization quantity if the edge is contracted.
  - Depends on the metric. Just algebra.
  - Note that we ignore conflicts...
Scalable Agglomeration: Matching

• Cheat #1: Impose a total order, (score, least vertex, larger vertex). Ensures greedy algorithm will converge correctly and not deadlock.

• Until done:
  – For each unmatched vertex, find the best unmatched neighbor. Remember, only storing around half the neighbors in each vertex's bucket...
  – Try to claim the best match neighbor. (locking/full-empty)
  – If that succeeded, try to claim self. (locking/full-empty)
  – If neither worked, remain in unmatched array.

• Technically, not $O(|E|)$... Variant of Hoepman's.
Scalable Agglomeration: Contraction

- For each edge, relabel endpoints, re-order for hashing.
- Rough bucketing:
  - Count / histogram by the first index $i$ in the edge.
    - (atomic int-fetch-add)
  - Prefix-sum for offsets (for now)
  - Copy $j$ and weight into temporary buckets.
  - Within each, sort & uniq. (*rule of thumb...*)
  - Copy back out. Asynchronous and not ordered by $i$ (no prefix-sum).
Scalable Agglomeration: Where does time go?

- Results from 10th DIMACS Impl. Challenge

Graph name

**Primitive**
- score
- match
- contract
- other

Parallel Programming for Graph Analysis
Scalable Agglomeration: Performance

- Results from 10th DIMACS Impl. Challenge

Parallel Programming for Graph Analysis
Scalable Agglomeration: Performance

- Results from 10th DIMACS Impl. Challenge
Graph Software: Current Status

Home computers

Plethora of solutions, motivated by social network analysis and computational biology research problems. Cannot handle massive data.
Representative software: Cytoscape, igraph

Commodity clusters

Implementations of Bulk-synchronous algorithms; MapReduce-based approaches. Performance a concern. Likely not generic enough to process queries on dynamic networks.
Boost Graph library, CGM-lib

Accelerators

Impressive performance on synthetic network instances/simple problems. Applicability to complex informatics problems unclear. e.g., recent BFS performance studies

Multicore Servers

SNAP: C + threads
Can process networks with billions of vertices and edges, on high-end multicore servers.
Fastest cache-based multicore implementations of several algorithms.

Massively multithreaded Systems (Cray XMT)

MTGL: Multithreaded graph library based on the “visitor” design pattern.
C++ with XMT pragmas
Can also run on multicore systems.
Sequential Graph Packages

- LEDA
- JUNG
- MATLAB / GNU Octave
- GNU R packages
- igraph
- Cytoscape
- Neo4j
- Boost Graph Library (will discuss in parallel)
LEDA

• Commercial C++ class for data types and algorithms
• Sequential programming
• graph datatype stores a static graph in an efficient representation
• Per-vertex partitions & priority queues
  – Useful for some algorithms
• Provides simple queries, i.e. Is_Acyclic()
• Used in Europe, not as much in USA

http://www.algorithmic-solutions.info/leda_guide/Graphs.html
Java Universal Network/Graph Framework (JUNG)

- Implements common algorithms:
  - Clustering
  - Statistical Analysis
  - Centrality & PageRank

- Interactive Visualization

- Limited by the heap size of the Java VM

http://jung.sourceforge.net/
MATLAB / GNU Octave

• Universal across all engineering disciplines
• Has sparse matrices
• Graph Theory toolbox calculates simple functions:
  – Euler tour
  – Max Flow, Min Cut
  – Minimum Spanning Tree
• Others in sparse matrix support (etree).
• Several orders of magnitude slower than a native desktop implementation
• Good for prototyping, but does not scale up
  – See KDT for scaling...
- Statistical package of S+ heritage.
- 16+ graph packages
- All small, sequential, use different data structures, *etc.*
- Many related publications, so useful for small-scale work.
- Often very focused metrics that do not scale presently.
- (Good items to mine for future work.)
igraph

• C library for directed & undirected graphs
• R package, Python & Ruby extensions
• Functions for generating & manipulating synthetic and real data
• Structural analyses including betweenness centrality, PageRank, and k-cores
• Limited by the size of main memory
• Contains many file format readers & writers.

http://igraph.sourceforge.net/
Cytoscape

- Open source bioinformatics tool for complex network analysis & visualization
- Runs in Java
- Can filter edges & find active pathways
- Limited to about 100,000 vertices

http://www.cytoscape.org/
Neo4j

• A commercial graph database
  – (A growing category:
    http://en.wikipedia.org/wiki/Graph_database)

• Runs under Java using transactions

• Stores semi-structured data as nodes, relationships, and properties

• Queries given as a “traverser”

• Can traverse 100,000 edges in 10 seconds
  – 5+ GB dataset on a 2 x 2.4 GHz server with 8 GB RAM

http://neo4j.org/
Parallel Graph Frameworks

- **SNAP**: Georgia Tech, Bader/Madduri
- **Parallel Boost Graph Library**: Indiana, Lumsdaine
- **MultiThreaded Graph Library (MTGL)**: Sandia, Berry
- **KDT**: UCSB, LBNL, others. Gilbert, Buluc,...
- **Giraph, GoldenOrb**: Hadoop-based BSP
- **GraphCT**: Georgia Tech, Ediger, Riedy, Jiang, Bader
- **STINGER**: Georgia Tech, Bader, Riedy, Ediger, Jiang
SNAP: Small-world Network Analysis and Partitioning

- Parallel framework for small-world network analysis
- Often 10-100x faster than existing approaches
- Can process graphs with billions of vertices and edges.
  - Shared memory
- Open-source
- [Bader/Madduri]

Image Source: visualcomplexity.com

SNAP parallel framework

Interaction data

Exploratory Network Analysis

Advanced Graph Analysis Queries
partitioning, subgraph isomorphism ...

Graph metrics and Preprocessing routines

Graph kernels
BFS, MST, connected components ...

Graph representation
formats, data structures

snap-graph.sourceforge.net
Multithreaded Graph Library (MTGL)

- Under development at Sandia National Labs
- Primitives for “visiting” a vertex
  - Get data about the vertex
  - Retrieve a list of all adjacencies
- Abstract connector to graph representation
- Tailored for Cray XMT, but portable to multicore using Qthreads
- Programmer must still understand code that is generated in order to get good performance on the XMT

https://software.sandia.gov/trac/mtgl
Parallel Boost Graph Library

- C++ library for parallel & \textit{distributed} graph computations
- Provides similar data structures and algorithms as sequential Boost Graph Library
- Developed by Indiana University in 2005
- Scales up to 100 processors for some algorithms on ideal graphs
- In active development: light-weight active messages for hybrid parallelism.

http://osl.iu.edu/research/pbgl/
Giraph, GoldenOrb, ...

• Once upon a time, Google mentioned Pregel.

• Now:
  – Giraph: http://incubator.apache.org/giraph/
  – GoldenOrb: http://goldenorbos.org/

• Vertex parallelism. Phased communication.

• Textual data...

• Free software... No known comparisons.
Knowledge Discovery Toolbox
http://kdt.sourceforge.net/

- Aimed at domain experts who know their problem well but don’t know how to program a supercomputer
- Easy-to-use Python interface
- Runs on a laptop as well as a cluster with 10,000 processors
- A collaboration among UCSB, UCB, and Lawrence Berkeley Lab
- Open source software, released under New BSD license
- v0.1 released March 2011; v0.2 expected March 2012

A general graph library with operations based on linear algebraic primitives
GraphCT

- Developed at Georgia Tech for the Cray XMT
- Low-level primitives to high-level analytic kernels
- Common graph data structure
- Develop custom reports by mixing and matching functions
- Create subgraphs for more in-depth analysis
- Kernels are tuned to maximize scaling and performance (up to 128 processors) on the Cray XMT

Load the Graph Data

Find Connected Components

Run k-Betweenness Centrality on the largest component

http://www.cc.gatech.edu/~bader/code.html
STINGER

• Enhanced representation developed for dynamic graphs developed in consultation with David A. Bader, Johnathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C. Poulos.

• Design goals:
  - Be useful for the entire “large graph” community
  - Portable semantics and high-level optimizations across multiple platforms & frameworks (XMT C, MTGL, etc.)
  - Permit good performance: No single structure is optimal for all.
  - Assume globally addressable memory access
  - Support multiple, parallel readers and a single writer

• Operations:
  - Insert/update & delete both vertices & edges
  - Aging-off: Remove old edges (by timestamp)
  - Serialization to support checkpointing, etc.

http://www.cc.gatech.edu/stinger/
From laptop...

- Sequential Program (changing, but memory...)
- Often Java-based
- Up to 100,000 vertices
- Minutes or Hours

...to supercomputer

- Massively Parallel
- C/C++ with extensions
- 10+ Billion vertices
- Minutes or Hours