Clustering on Clusters: Massively Parallel Algorithms for Clustering Graphs and Vectors

Grigory Yaroslavtsev

http://grigory.us
Clustering on Clusters: Overview

• Algorithm design for massively parallel computing
  – Blog: http://grigory.us/blog/mapreduce-model/

• MPC algorithms for graphs
  – Connectivity
  – Correlation clustering

• MPC algorithms for vectors
  – K-means
  – Single-linkage clustering

• Open problems and directions
# Clustering on Clusters: Overview

<table>
<thead>
<tr>
<th></th>
<th>Graphs</th>
<th>Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td>Connectivity</td>
<td>K-means</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td>Correlation Clustering</td>
<td>Single-Linkage Clustering</td>
</tr>
</tbody>
</table>
Cluster Computation (a la BSP)

- **Input**: size $n$ (e.g. $n = \text{billions of edges in a graph}$)
- $M$ Machines, $S$ Space (RAM) each
  - Constant overhead in RAM: $M \cdot S = O(n)$
  - $S = n^{1-\epsilon}$, e.g. $\epsilon = 0.1$ or $\epsilon = 0.5$ ($M = S = O(\sqrt{n})$)
- **Output**: solution to a problem (often size $O(n)$)
  - Doesn’t fit in local RAM ($S \ll n$)

**Diagram**

- **Input**: size $n$ ⇒
- **Load Balancer**
- $M$ machines
- **$S$ space**
- ⇒ **Output**
Cluster Computation (a la BSP)

- Computation/Communication in $R$ rounds:
  - Every machine performs a **near-linear time** computation $\Rightarrow$ Total user time $O(S^{1+o(1)} R)$
  - Every machine **sends/receives at most $S$ bits** of information $\Rightarrow$ Total communication $O(nR)$.

**Goal:** Minimize $R$.

**Ideally:** $R = \text{constant}$. 

$O(S^{1+o(1)})$ time 

$\leq S$ bits 

$M$ machines 

$S$ space
MapReduce-style computations

What I won’t discuss today

• PRAMs (shared memory, multiple processors) (see e.g. [Karloff, Suri, Vassilvitskii ‘10])
  – Computing XOR requires $\tilde{\Omega}(\log n)$ rounds in CRCW PRAM
  – Can be done in $O(\log_s n)$ rounds of MapReduce

• Pregel-style systems, Distributed Hash Tables (see e.g. Ashish Goel’s class notes and papers)

• Lower-level implementation details (see e.g. Rajaraman-Leskovec-Ullman book)
Models of parallel computation

• Bulk-Synchronous Parallel Model (BSP) [Valiant,90]
  **Pro:** Most general, generalizes all other models
  **Con:** Many parameters, hard to design algorithms

• Massive Parallel Computation [Feldman-Muthukrishnan-Sidiropoulos-Stein-Svitkina’07, Karloff-Suri-Vassilvitski’10, Goodrich-Sitchinava-Zhang’11, …, Beame, Koutris, Suciu’13]
  **Pros:**
  • Inspired by *modern* systems (Hadoop, MapReduce, Dryad, Spark, Giraph, …)
  • Few parameters, *simple* to design algorithms
  • *New algorithmic ideas*, robust to the exact model specification
  • *# Rounds* is an information-theoretic measure => can prove unconditional results
  **Con:** sometimes not enough to model more complex behavior
Business perspective

- Pricings:
  - https://cloud.google.com/pricing/
  - https://aws.amazon.com/pricing/
- Linear with **space** and **time** usage
  - 100 machines: 5K $/year
  - 10000 machines: 0.5M $/year
- You pay a **lot more** for using provided algorithms
Part 1: Clustering Graphs

- Applications:
  - Community detection
  - Fake account detection
  - Deduplication
  - Storage localization
  - ...

Problem 1: Connectivity

• **Input**: n edges of a graph (arbitrarily partitioned between machines)

• **Output**: is the graph connected? (or # of connected components)

• **Question**: how many rounds does it take?

1. \(O(1)\)
2. \(O(\log^\alpha n)\)
3. \(O(n^\alpha)\)
4. \(O(2^\alpha n)\)
5. Impossible
Algorithm for Connectivity

• Version of **Boruvka’s algorithm**:  
  – All vertices assigned to different components  
  – Repeat $O(\log n)$ times:  
    • Each component chooses a neighboring component  
    • All pairs of chosen components get merged

• **How to avoid chaining?**

• If the graph of components is bipartite and only one side gets to choose then no chaining

• **Randomly** assign components to the sides
Algorithm for Connectivity: Setup

Data: \( n \) edges of an undirected graph.

Notation:
- \( \pi(v) \equiv \) unique id of \( v \)
- \( \Gamma(S) \equiv \) set of neighbors of a subset of vertices \( S \).

Labels:
- Algorithm assigns a label \( \ell(v) \) to each \( v \).
- \( L_v \equiv \) the set of vertices with the label \( \ell(v) \) (invariant: subset of the connected component containing \( v \)).

Active vertices:
- Some vertices will be called active (exactly one per \( L_v \)).
Algorithm for Connectivity

- Mark every vertex as **active** and let $\ell(v) = \pi(v)$.
- For phases $i = 1, 2, \ldots, O(\log n)$ do:
  - Call each active vertex a **leader** with probability $1/2$. If $v$ is a leader, mark all vertices in $L_v$ as leaders.
  - For every active non-leader vertex $w$, find the smallest leader (by $\pi$) vertex $w^*$ in $\Gamma(L_w)$.
  - Mark $w$ **passive**, relabel each vertex with label $w$ by $w^*$.
- **Output**: set of connected components based on $\ell$. 
Algorithm for Connectivity: Analysis

• If \( \ell(u) = \ell(v) \) then \( u \) and \( v \) are in the same CC.
• **Claim:** Unique labels with high probability after \( O(\log N) \) phases.
• For every CC # active vertices reduces by a constant factor in every phase.
  – Half of the active vertices declared as non-leaders.
  – Fix an active **non-leader** vertex \( v \).
  – If at least two different labels in the CC of \( v \) then there is an edge \((v', u)\) such that \( \ell(v) = \ell(v') \) and \( \ell(v') \neq \ell(u) \).
  – \( u \) marked as a **leader** with probability \( 1/2 \) \( \Rightarrow \) half of the active non-leader vertices will change their label.
  – Overall, expect \( 1/4 \) of labels to disappear.
  – After \( O(\log N) \) phases # of active labels in every connected component will drop to one with high probability.
Algorithm for Connectivity: Implementation Details

- Distributed data structure of size $O(|V|)$ to maintain labels, ids, leader/non-leader status, etc.
  - $O(1)$ rounds per stage to update the data structure
- Edges stored locally with all auxiliary info
  - Between stages: use distributed data structure to update local info on edges
- For every active non-leader vertex $w$, find the smallest leader (w.r.t $\pi$) vertex $w^* \in \Gamma(L_w)$
  - Each (non-leader, leader) edge sends an update to the distributed data structure
- Much faster with Distributed Hash Table Service (DHT) [Kiveris, Lattanzi, Mirrokni, Rastogi, Vassilvitskii’14]
Algorithms for Graphs

• **Dense graphs** vs. sparse graphs
  
  – **Dense**: $S \gg |V|$
    
    • Linear sketching: one round, see [McGregor’14]
    • “Filtering” (Output fits on a single machine) [Karloff, Suri, Vassilvitskii, SODA’10; Ene, Im, Moseley, KDD’11; Lattanzi, Moseley, Suri, Vassilvitskii, SPAA’11; Suri, Vassilvitskii, WWW’11]...
  
  – **Sparse**: $S \ll |V|$ (or $S \ll$ solution size)
    
    Sparse graph problems appear hard (**Big open question**: connectivity in $o(\log n)$ rounds?)
Problem 2: Correlation Clustering

- Inspired by machine learning at
- Practice: [Cohen, McCallum ‘01, Cohen, Richman ‘02]
- Theory: [Blum, Bansal, Chawla ’04]
Correlation Clustering: Example

- **Minimize** # of **incorrectly** classified pairs:
  
  # Covered non-edges + # Non-covered edges

4 incorrectly classified = 1 covered non-edge + 3 non-covered edges
Approximating Correlation Clustering

• **Minimize # of incorrectly classified pairs**
  – $\approx 20000$-approximation [Blum, Bansal, Chawla’04]
  – [Demaine, Emmanuel, Fiat, Immorlica’04],[Charikar, Guruswami, Wirth’05], [Ailon, Charikar, Newman’05][Williamson, van Zuylen’07], [Ailon, Liberty’08],…
  – $\approx 2$-approximation [Chawla, Makarychev, Schramm, Y. ’15]

• **Maximize # of correctly classified pairs**
  – $(1 - \epsilon)$-approximation [Blum, Bansal, Chawla’04]
Correlation Clustering

One of the most successful clustering methods:

• Only uses **qualitative information** about similarities

• **# of clusters unspecified** (selected to best fit data)

• Applications: document/image **deduplication** (data from crowds or black-box machine learning)

• **NP-hard** [Bansal, Blum, Chawla ‘04], admits simple approximation algorithms with good provable guarantees
Correlation Clustering

More:

• **Survey** [Wirth]

• **KDD’14** tutorial: “Correlation Clustering: From Theory to Practice” [Bonchi, Garcia-Soriano, Liberty]

• **Wikipedia** article:
  http://en.wikipedia.org/wiki/Correlation_clustering
Data-Based Randomized Pivoting

3-approximation (expected) [Ailon, Charikar, Newman]

Algorithm:
• Pick a random pivot vertex $v$
• Make a cluster $v \cup N(v)$, where $N(v)$ is the set of neighbors of $v$
• Remove the cluster from the graph and repeat
Data-Based Randomized Pivoting

• Pick a random pivot vertex $p$
• Make a cluster $p \cup N(p)$, where $N(p)$ is the set of neighbors of $p$
• Remove the cluster from the graph and repeat

8 incorrectly classified = 2 covered non-edges + 6 non-covered edges
Parallel Pivot Algorithm

- $(3 + \epsilon)$-approx. in $O(\log^2 n / \epsilon)$ rounds [Chierichetti, Dalvi, Kumar, KDD’14]

- Algorithm: while the graph is not empty
  - $D =$ current maximum degree
  - Activate each node independently with prob. $\epsilon/D$
  - Deactivate nodes connected to other active nodes
  - The remaining nodes are pivots
  - Create cluster around each pivot as before
  - Remove the clusters
Parallel Pivot Algorithm: Analysis

- **Fact**: Halves max degree after \( \frac{1}{\epsilon} \log n \) rounds
  \( \Rightarrow \) terminates in \( O \left( \frac{\log^2 n}{\epsilon} \right) \) rounds

- **Fact**: Activation process induces close to uniform marginal distribution of the pivots
  \( \Rightarrow \) analysis similar to regular pivot gives \((3 + \epsilon)\)-approximation
Part 2: Clustering Vectors

• Input: $v_1, \ldots, v_n \in \mathbb{R}^d$
  - Feature vectors in ML, word embeddings in NLP, etc.
  - (Implicit) weighted graph of pairwise distances

• Applications:
  - Same as before + Data visualization
Problem 3: K-means

- **Input:** \(v_1, ..., v_n \in \mathbb{R}^d\)
- **Find** \(k\) centers \(c_1, ..., c_k\)
- **Minimize sum of squared distance to the closest center:**
  \[
  \min_{j=1}^{k} \sum_{i=1}^{n} ||v_i - c_j||_2^2
  \]
- \(||v_i - c_j||_2^2 = \sum_{t=1}^{d} (v_{it} - c_{jt})^2\)
- **NP-hard**
K-means++ [Arthur,Vassilvitskii’07]

- \( C = \{c_1, \ldots, c_t\} \) (collection of centers)
- \( d^2(v, C) = \min_{j=1}^{k} ||v - c_j||_2^2 \)

K-means++ algorithm (gives \( O(\log k) \)-approximation):

- Pick \( c_1 \) uniformly at random from the data
- Pick centers \( c_2 \ldots, c_k \) sequentially from the distribution where point \( v \) has probability
  \[
  \frac{d^2(v, C)}{\sum_{i=1}^{n} d^2(v_i, C)}
  \]
K-means∥ [Bahmani et al. ‘12]

- Pick $C = c_1$ uniformly at random from data
- Initial cost: $\psi = \sum_{i=1}^{n} d^2(v_i, c_1)$
- Do $O(\log \psi)$ times:
  - Add $O(k)$ centers from the distribution where point $v$ has probability
    
    $$\frac{d^2(v, C)}{\sum_{i=1}^{n} d^2(v_i, C)}$$
  
- Solve k-means for these $O(k \log \psi)$ points locally

- **Thm.** If final step gives $\alpha$-approximation
  $\Rightarrow O(\alpha)$-approximation overall
Problem 4: Single Linkage Clustering

- [Zahn’71] Clustering via Minimum Spanning Tree: $k$ clusters: remove $k - 1$ longest edges from MST
- Maximizes minimum intercluster distance

[Kleinberg, Tardos]
Large geometric graphs

• Graph algorithms: **Dense graphs** vs. sparse graphs
  – **Dense**: $S \gg |V|$. 
  – **Sparse**: $S \ll |V|$. 

• **Our setting:**
  – Dense graphs, sparsely represented: $O(n)$ space
  – Output doesn’t fit on one machine ($S \ll n$)

• **Today:** $(1 + \epsilon)$-approximate MST [Andoni, Onak, Nikolov, Y.]
  – $d = 2$ (easy to generalize)
  – $R = \log_S n = O(1)$ rounds ($S = n^{\Omega(1)}$)
$O(\log n)$-MST in $R = O(\log n)$ rounds

- Assume points have integer coordinates $[0, \ldots, \Delta]$, where $\Delta = O(n^2)$.

Impose an $O(\log n)$-depth quadtree

Bottom-up: For each cell in the quadtree
- compute optimum MSTs in subcells
- Use only one representative from each cell on the next level

Wrong representative: $O(1)$-approximation per level
\( \epsilon L \)-nets

- **\( \epsilon L \)-net for a cell C with side length \( L \):**
  Collection \( S \) of vertices in C, every vertex is at distance \( \leq \epsilon L \) from some vertex in \( S \). (Fact: Can efficiently compute \( \epsilon \)-net of size \( O \left( \frac{1}{\epsilon^2} \right) \))

  Bottom-up: For each cell in the quadtree
  - Compute optimum MSTs in subcells
  - Use \( \epsilon L \)-net from each cell on the next level

- **Idea:** Pay only \( O(\epsilon L) \) for an edge cut by cell with side \( L \)
- Randomly shift the quadtree:
  \( \Pr[\text{cut edge of length } \ell \text{ by } L] + \ell / L \) - charge errors
  \( O(1) \)-approximation per level
Randomly shifted quadtree

- Top cell shifted by a random vector in $[0, L]^2$

Impose a **randomly shifted** quadtree (top cell length $2\Delta$)

Bottom-up: For each cell in the quadtree
  - Compute optimum MSTs in subcells
  - Use $\epsilon L$-net from each cell on the next level

Pay 5 instead of 4

$\Pr[\text{Bad Cut}] = \Omega(1)$
\( (1 + \epsilon) \)-MST in \( \mathbb{R} = O(\log n) \) rounds

- **Idea:** Only use short edges inside the cells

  Impose a *randomly shifted* quadtree (top cell length \( \frac{2\Delta}{\epsilon} \))

  Bottom-up: For each node (cell) in the quadtree
  
  – compute optimum Minimum Spanning Forests in subcells, using edges of length \( \leq \epsilon L \)
  – Use only \( \epsilon^2 L \)-net from each cell on the next level

\[
L = \Omega\left(\frac{1}{\epsilon}\right)
\]

\[
\text{Pr}[\text{Bad Cut}] = O(\epsilon)
\]
$(1 + \epsilon)$-MST in $\mathbb{R} = O(1)$ rounds

- $O(\log n)$ rounds $\Rightarrow$ $O(\log_s n) = O(1)$ rounds
  - Flatten the tree: $(\sqrt{M} \times \sqrt{M})$-grids instead of $(2 \times 2)$ grids at each level.

$$\sqrt{M} = n^{\Omega(1)}$$

Impose a **randomly shifted** $(\sqrt{M} \times \sqrt{M})$-tree

Bottom-up: For each node (cell) in the tree
- compute optimum MSTs in subcells via edges of length $\leq \epsilon L$
- Use only $\epsilon^2 L$-net from each cell on the next level
Theorem: Let $l =$ # levels in a random tree $P$

$$\mathbb{E}_P[\text{ALG}] \leq (1 + O(\epsilon ld))\text{OPT}$$

Proof (sketch):

• $\Delta_P(u, v) =$ cell length, which first partitions $(u, v)$

• New weights: $w_P(u, v) = \|u - v\|_2 + \epsilon \Delta_P(u, v)$

$$\|u - v\|_2 \leq \mathbb{E}_P[w_P(u, v)] \leq (1 + O(\epsilon ld))\|u - v\|_2$$

• Our algorithm implements Kruskal for weights $w_P$
(1 + \(\varepsilon\))-MST:

- “Load balancing”: partition the tree into parts of the same size
- **Almost linear time locally**: Approximate Nearest Neighbor data structure \([\text{Indyk’99}]\)
  
  - Dependence on dimension \(d\) (size of \(\varepsilon\)-net is \(O\left(\frac{d^d}{\varepsilon}\right)\))
  
  - Generalizes to bounded **doubling dimension**
Thanks! Questions?

- Slides will be available on http://grigory.us
- More about algorithms for massive data: http://grigory.us/blog/
- More in the classes I teach: