CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Databricks and Stanford.

Lecture 3, Complexity Measures for Clusters, 4/6/2015.
Scribed by Jim Cai, Augustus Hong, Charles Zhang.

3 Complexity Analysis of Algorithms in a distributed setting

- **Single Threaded Setting:** Traditionally, we do complexity analysis of algorithms assuming that we are working with single-thread process, in which case (once you account for memory considerations):

$$\text{computation time} \propto \# \text{ operations}$$

- **Distributed Setting:** In the distributed setting, we can no longer assume that runtime is proportional to the number of operations. If a single thread's runtime is S, in a distributed setting, the *best case scenario* for the runtime of the same task on K machines is $(\frac{S}{K})$. This happens when the problem is "embarrassingly parallel" with little communication overhead, each machine solves its own local subproblem without needing to communicate much.

The main difference between single-thread and distributed computation is that you have to wait for the last core/machine to finish, so consequently you want to spread out work as equally as possible).

3.1 Example: Triangle Counting

Problem: Given a graph (V, E), count the number of triangles. Let $m = |E|$ (the number of edges) and $n = |V|$ (the number of vertices). A triangle is a triple (u, v, w) such that (u, v), (v, w), and $(u, w) \in E$.

Solution 1: Given an adjacency matrix A, calculate $A^2 + A$ and sum the off-diagonals. You can do this as fast as you can multiply 2 matrices (i.e., $O(n^3)$ in the naive case and, currently, $O(n^{2.37..})$ in the non-naive case).

This solution works for arbitrary graphs and is good when the graph is dense. Most graphs, however, are sparse (i.e. m is $O(n)$) and we can do better than this solution in these cases. Ideally, we would like to find an algorithm with a complexity that is a function of m (nicely, cheap w.r.t m). In the next section, we discuss Node Iterator, which is one such algorithm.

3.1.1 Node Iterator Algorithm

Here is a simple solution based on iterating through each node: given a graph $G = (V, E)$, iterate through each node $v \in V$ and construct $\Gamma(v)$, the neighborhood of v. Then for each pair of nodes, $u, w \in \Gamma(v)$ check if $(u, w) \in E$. If so, then we have found a triangle. Note that this will count each triangle 6 times, twice for each node.

Algorithm 1 NodeIterator(V,E)

T $\leftarrow$ 0
for $v \in V$ **do**
  for $u \in \Gamma(v)$ **do**
    for $w \in \Gamma(v)$ **do**
      if $(u, w) \in E$ **then**
        T $\leftarrow$ T+1/2
return T/3

Let $\succ$ be a total order on all vertices such that $u \succ v$ if and only if $\deg(u) > \deg(v)$, with ties broken arbitrarily but consistently. How ties are broken is not particularly important as long as the ordering remains consistent between runs. We can use this ordering to reduce the amount of over-counting and therefore work done by NodeIterator.

Define the modified neighborhood of v, $\Gamma^*(v)$, as $\{u \in \Gamma(v) \mid u \succ v\}$. Roughly speaking, this is the set of neighbors of v with higher degree than v, and we can use these modified neighborhoods when counting triangles so that only the lowest degree node in each triangle 'counts' the triangle. Then, instead of counting each triangle 6 times we only count each triangle twice.

Algorithm 2 NodeIterator++(V,E)

T $\leftarrow$ 0
for $v \in V$ **do**
  for $u \in \Gamma^*(v)$ **do**
    for $w \in \Gamma^*(v)$ **do**
      if (u,w)$\in$ E **then**
        T $\leftarrow$ T+$\frac{1}{2}$
return T

If we fix a arbitrary number $t > 0$, the complexity of this algorithm is upper bounded by

$$\sum_{v \in V} \binom{\deg(v)}{2} = \sum_{v \in V, \deg(v) \geq t} \binom{\deg(v)}{2} + \sum_{v \in V, \deg(v) < t} \binom{\deg(v)}{2}.$$

The expansion seperates the summation for high degree nodes ($\deg(v) \geq t$) and low degree nodes. By the handshake lemma, we know that the number of high degree nodes is at most $\frac{2m}{t}$ (there are m edges each contributing to 2 degrees). Therefore, the first part of the sum is at most $(\frac{2m}{t})^3$ (# high degree nodes choose 3)

For the low degree nodes (i.e. nodes v such that $\deg(v) < t$) we have that

$$\sum_{v \in V, \deg(v) < t} \binom{\deg(v)}{2} \leq \sum_{v \in V, \deg(v) < t} \deg(v)^2 \leq t \left(\sum_{v \in V, \deg(v) \leq t} \deg(v) \right) \leq 2mt.$$

This implies that the total work done by NodeIterator++ is upper bounded by $(\frac{2m}{t})^3 + 2mt$. We can choose $t = \sqrt{m}$ to minimize this, giving $4m^{3/2} + 2m^{3/2} = O(m^{3/2})$ runtime for NodeIterator++. For sparse matrices, $O(m^{\frac{3}{2}}) < O(n^{2.373...})$, so we have achieved our goal of improving the matrix multiplication algorithm by exploiting graph sparsity.

A reasonable question to ask is whether or not we lost some rigor or sharpness by changing the algorithm in this way. We can construct an example such that there is a clique $K_{\sqrt{n}}$ and $n - \sqrt{n}$ nodes that are connected in a line (see figure 3.1.1). In this example, the number of triangles is $\Theta(\binom{\sqrt{n}}{3}) = \Theta(n^{\frac{3}{2}})$ and $m = \Theta(\binom{\sqrt{n}}{2}) = O(n)$. Thus, our algorithm matches up with this bound. One important concept that this example illustrates is that there is a noticeable difference in complexity between listing and counting triangles. In order to list the triangles in a clique, there is $O(n^3)$ work as there are just that many triangles. However, if we were just counting, we could do it faster by just calculating $A^2 + A$, which is just $O(n^{2.37})$ amount of work.

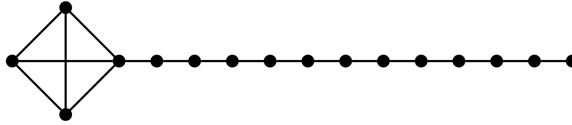

Figure 1: An example of a lollipop graph

3.2 Cluster/Distributed Complexity Measures

In any distributed setting (and, especially, in a MapReduce framework), there are three things that we analyze to determine complexity:

- **Number of iterations** (you want as few maps, join-bys, etc. as possible)
- **Shuffle size after combining** (number of tuples output by mapper machine)
- **Time and memory for reducers to finish**

3.3 Distributed Node Iterator

In an industrial setting, the edges arrive and leave one at a time (think social graphs), then the input will be given to us as a bunch of edges, e.g., the input will be $\{(u_1, v_1), (u_2, v_2), \ldots, (u_m, v_m)\}$

Then, to run node iterator in a distributed setting, we can decompose the algorithm into the following 3 MapReduce jobs:

1. Compute Degrees (combiner is good)
2. Construct Modified Neighborhoods and run node iterator (can't use combiner)
3. Check triangle closure by joining with edges (combiner use depends on join implementation)

3.3.1 (1. Compute Degrees)

We can analyze the work done in this step. In the mapper:

for each (u, v), **emit (u, "1"), (v, "1")**.

and then, in the reducer, we sum up the number of records each key has (the key is determined by the first element of the tuple, i.e. u). Since the summation is associative and commutative, we can use a combiner without any issue and the shuffle size is $2m$. At the machine-level, we can dish out the edges locally and there will be perfect scaling.

3.3.2 (2. Node Iterator)

We usually assume n fits into memory but m cannot, so the results from part 1 (an array with the degrees counts that is of size n) can be broadcast to and stored by each machine.

Then, we have $map((u, v))$:

if u$\prec$v: emit(u,v)
else emit(v,u)

and we have *reduce*(v, $\Gamma^*(v)$)

for (u,w) $\in \Gamma^*(v)$: emit((u, w), v)

We need to emit all the candidate edges because there's no way to check against the real edges since we cannot keep them in local memory. Shuffle size is $O(m)$ because we are emitting each edge. Memory constraints are fine because we are only keeping degree counts.

For each reducer we are interested in the upper bound of edges it could receive.

For low degree nodes, $\Gamma^*(v) \leq \sqrt{m}$

For high degree nodes, $\Gamma^*(v) \leq \frac{2m}{t} = 2\sqrt{m}$. This is because we are using Γ^* so the only neighbors of high degree nodes are other high degree nodes

Some assorted notes:

- If you tried to run this without modified neighborhoods (i.e., the original node iterator algorithm), you would be in $O(n^2)$ complexity and this suddenly becomes an intractable problem.
- There is a paper about this on the DAO website ("the curse of the last reducer").
- This algorithm is not perfect either as the work may not be evenly distributed.
- Combiners are run before any network activity on the mapper node on the mapper output. Utilize them to reduce reducer operations and shuffle size.