CME 323: Distributed Algorithms and Optimization, Spring 2016
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 14, 5/11/2016. Scribed by Victor Storchan and Taman Narayan.

1 Complexity measures for MapReduce

During a MapReduce algorithm, three main stages are involved while computing the complexity. First, all of the map tasks have to finish. Second, all of the pairs which need to be on the same machine need to be shuffled. Finally, reducers take some time to finish. As a consequence, we will provide three complexity time measures corresponding to these three stages:

- **Mappers cost time:** the time the map phase takes. All the mappers have to do their computations and spit out the pairs (this is embarrassingly parallel work, so we can use the same analysis tools that we know for a single processor computation and take the worst one).
- **Shuffle cost time:** number of items the mappers output to be sorted: how many tuples have to be sorted. Note that although the keys are hashable, you can not just assume that the sort time will be $O(n)$ as in practice, a more complex analysis involving the number of machines and the architecture would have to be done.
- **Reducers cost time:** how long it takes for the slowest reducer to finish.

Any of these measures could be the bottleneck of the overall analysis of a MapReduce algorithm.

2 Triangle Counting

For this part, assume that we are given an undirected graph $G(V, E)$ in the edge-list format, i.e. a giant data set of (u, v) pairs, split across machines. Let's make the following assumption: the number of nodes of G fits in memory. For example, $n \simeq 10 \cdot 10^6$. However, the number of edges $m \simeq n^2$ (for a dense graph) does not fit in memory. As a consequence, the node data structure can be thought as a "local array", whereas the edge data structure can be thought of as an "RDD".

Goal: Counting the number of triangles in the graph within a MapReduce implementation.

2.1 Triangle counting on a single machine

First, let us derive a sequential algorithm to count every triangle of our graph G in a sequential way. Let us use the adjacency matrix A of the graph: $A = (a_{ij})$ where

$$a_{ij} = \begin{cases} 1 \text{ if (i,j)} \in \text{E} \\ \\ 0 \text{ otherwise} \end{cases}$$

Then, it is well known that the global coefficient $[A^k]_{ij}$ for $k \in \mathbb{N}$ counts the number of paths of length k in G, starting at i and ending at j. In particular $\forall i \in V$, $[A^3]_{ii}$ counts the number of paths starting at i and ending at i of length 3, i.e. the number of triangles containing i as a vertex. To avoid overcounting, we need to divide $\sum_i [A^3]_{ii}$ by a factor of 6: indeed a single triangle will be counted $2 * 3 = 6$ times. The factor 2 comes from the fact that the triangle (u, v, w) can be traveled as $u \to v \to w \to u$ or $u \to w \to v \to u$.

What is the time complexity? We know that we can compute A^3 in $O(n^\omega)$ with $\omega \simeq 2.373$, but this algorithm cannot be distributed. Likewise with Strassen's algorithm where $\omega \simeq 2.8$, it can not be adapted to a cluster because it would need to send giant submatrices of data across machines. As a general idea, it can be noted that recursive algorithms are not as useful for clusters as they are for the PRAM model.

2.2 Triangle counting on a cluster: the Node Iterator Algorithm

So let us do something simpler for the distributed case. Note that if the graph is sparse, even on a sequential machine you would want to do better than $O(n^{2.373})$. The idea of the Node Iterator Algorithm is the following: given our graph $G(V, E)$, one needs to iterate through each vertex $v \in V$ to find its neighborhood $\Gamma(v)$. Then, for all $u, w \in \Gamma(v)$, check if (u,w) is in E. If it is the case, then we have found a legitimate triangle, and an accumulator variable T can be incremented. To finish we divide T by 6, as stated in **2.1**.

Algorithm 1 *NodeIterator*$(G(V, E))$

```
1: procedure NodeIterator
2:     T ← 0
3:     for v ∈ V do
4:         for u ∈ Γ(V) do
5:             for w ∈ Γ(V) do
6:                 if (u, w) ∈ E then
7:                     T := T + 1/6
       return T
```

Let us consider the time complexity of this algorithm: On a sequential machine, it is at most $O(n^3)$ (if the graph is dense for example, there are at most $O(n^3)$ triangles). But here, we would like to exploit sparsity, i.e. to rewrite this complexity as a function of m. Now let's say that the graph is sparse with m edges. Let's say there is a high degree node v, which means that v is roughly connected to all the other nodes. (v is very popular). Suppose:

$$d(v) > cn \text{ where } c < 1$$

Then, the runtime is going to be $\Omega(n^2)$. Actually, it will take $\Omega(n^2)$ time just for that single node v.

Let's now implement this algorithm in a MapReduce environment.

3 Implementing the Node Iterator Algorithm in MapReduce

The Node Iterator algorithm can be implemented in two MapReduce steps.

3.1 Compute neighborhoods

Recall that we are given an edge list, but that the algorithm requires a list of neighbors for each node in order to iterate over.

Neighborhood collection can be performed in a straightforward MapReduce step, with mappers emitting the first element of the edge (u, v) as the key and the reducers collecting the neighborhoods $\{v_1, v_2, ..., v_n\}$ corresponding to each key u. This leaves us with an **RDD**[(Int, Array[Int])] object.

Note that each list of neighbors is of size at most n, meaning that by assumption it fits in memory as is required for individual elements of RDDs. Note as well that in Spark, we could implement this operation using *groupByKey*.

3.2 Count triangles

The key to accomplishing the counting step in a single MapReduce step is observing that we need the edges to search for *and* the actual edges in the graph both available to the reducer to perform the lookup for whether given triangles exist.

We can do this by having mappers (a) loop over all neighbors u and w for each node $v \in V$ and output key-value pairs [(u, w), "to check"] and (b) traverse the edge list and output the pairs [(v_1, v_2), "present edge"]. That is to say the keys are edges and the values are whether the particular edge is being searched for or represents the actual edges in the graph.

The reducer then has access to, for each edge, a list consisting of zero or more "to check" strings and zero or one "present edge" strings. If there exists a "present edge" string in the list, then the reducer sums the number of "to check" strings it finds, as each represents a discovered triange. Otherwise, that edge either doesn't exist in the graph or wouldn't complete a triangle. Note that the sum, keeping in line with the algorithm, would be divided by six to avoid counting triangles multiple times.

4 Complexity Analysis of MapReduce Algorithm

We focus on the second step of the algorithm and leave an analysis of the first step to the reader.

Recall that analyzing the runtime of a MapReduce operation requires going over each phase: map, shuffle, and reduce.

1. **Map Phase**: The time to process a single neighborhood is $O(n^2)$, with the upper bound coming from a node with $O(n)$ neighbors.

2. **Shuffle Size**: The map phase emits $\Omega(n^2 + m)$ objects in the case of a single dense node with $O(n)$ neighbors. In a dense graph, each node has $O(n)$ neighbors and the shuffle size is $O(n^3 + m)$.

Note that we only consider the shuffle size here, not the cost of the actual shuffle. Partly, this is because we can achieve a linear time sort due to the hashable objects being sorted, albeit with a high constant factor. In addition, the sorting algorithm's asymptotic performance can vary by implementation, while shuffle size is independent of those concerns.

3. **Reduce Phase**: Processing a list of "to check" and "present edge" strings is $O(n)$ in the case that each endpoint of the edge (v_1, v_2) is connected to O(n) other nodes.

For more details, see [1].

5 Improving the Node Iterator Algorithm

Even if the above algorithm were $O(n^2)$ rather than $O(n^3)$, it still would not be good enough to work with large graphs. For example, if $n = 10$ million, $n^2 = 100$ trillion. Ideally, we'd like a dependence on m instead, especially when the graph is sparse.

To get there, we combine two insights. The first is that the driver of high costs is nodes with large neighborhoods, which then have to output all pairs of nodes in the neighborhood. The second is that our current algorithm is substantially overcounting triangles by starting its search from each endpoint of the triangle. For example, given a triangle with endpoints a, b, and c, the algorithm will detect that b and c are in the neighborhood of a, a and c are in the neighborhood of b, and a and b are in the neighborhood of c.

We address each of these shortcomings by pursuing a strategy to only search for a given triangle T in the neighborhood of its least-degree node. To do this, create a total ordering on nodes by degree (i.e. break ties in some way) and remove all nodes from the neighborhood of a node v with a lower degree than v.

By construction, this leaves us with a set of neighborhoods $\Gamma^*(v)$ where each node v is only neighbors with higher-degree nodes. It is now apparent that when searching for triangles by looping over these neighborhoods instead, triangles will only be found from the neighborhood of the least-degree node as we wanted.

It turns out that this approach yields a runtime of $O(m^{3/2})$.

References

[1] S. Suri and S. Vassilvitskii. *Counting Triangles and the Curse of the Last Reducer*. Proceedings of the 20th International Conference on World Wide Web, 2011.