Counting Triangles and the Curse of the Last Reducer

Siddharth Suri Sergei Vassilvitskii
Yahoo! Research
11 W. 40th St, 17th Floor
New York, NY 10018, USA
{suri, sergei}@yahoo-inc.com

ABSTRACT

The clustering coefficient of a node in a social network is a fundamental measure that quantifies how tightly-knit the community is around the node. Its computation can be reduced to counting the number of triangles incident on the particular node in the network. In case the graph is too big to fit into memory, this is a non-trivial task, and previous researchers showed how to estimate the clustering coefficient in this scenario. A different avenue of research is to to perform the computation in parallel, spreading it across many machines. In recent years MapReduce has emerged as a de facto programming paradigm for parallel computation on massive data sets. The main focus of this work is to give MapReduce algorithms for counting triangles which we use to compute clustering coefficients.

Our contributions are twofold. First, we describe a sequential triangle counting algorithm and show how to adapt it to the MapReduce setting. This algorithm achieves a factor of 10-100 speed up over the naive approach. Second, we present a new algorithm designed specifically for the MapReduce framework. A key feature of this approach is that it allows for a smooth tradeoff between the memory available on each individual machine and the total memory available to the algorithm, while keeping the total work done constant. Moreover, this algorithm can use any triangle counting algorithm as a black box and distribute the computation across many machines. We validate our algorithms on real world datasets comprising of millions of nodes and over a billion edges. Our results show both algorithms effectively deal with skew in the degree distribution and lead to dramatic speed ups over the naive implementation.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous

General Terms

Algorithms, Experimentation, Theory

Keywords

Clustering Coefficient, MapReduce

1. INTRODUCTION

In recent years there has been an explosion in the amount of data available for social network analysis. It has become relatively common to study social networks consisting of tens of millions of nodes and billions of edges [12, 13]. Since the amount of data to be analyzed has grossly outpaced advances in the memory available on commodity hardware, computer scientists have once again turned to parallel algorithms for data processing. Mapreduce [10] and its open source implementation, Hadoop [19], have emerged as the standard platform for large scale distributed computation. In this paper we will give algorithms for computing one of the fundamental metrics for social networks, the clustering coefficient [18], in the MapReduce framework.

The clustering coefficient measures the degree to which a node's neighbors are themselves neighbors. The field of sociology gives us two theories as to the importance of this measure. The tighter the community, the more likely it is for any given member to have interacted with and know the reputation of any other member. Coleman and Portes [8, 15] argue that if a person were to do something against the social norm, the consequences in a tightly-knit community would be greater because more people would know about the offending action and more people would be able to sanction the individual who committed it. This phenomenon helps foster a higher degree of trust in the community and also helps the formation of positive social norms. Thus if the clustering coefficient of an individual indicated that they were in a tightly-knit community, that individual may be able to better take advantage of the higher levels of trust amongst his/her peers and more positive social norms in his/her community.

Burt [5] contends that individuals may benefit from acting as a bridge between communities. The theory of structural holes argues that having a relationship with two individuals that do not otherwise have a relationship allows the mediator several advantages. The mediator can take ideas from both people and come up with something entirely new. The person in the middle could also take ideas from one of his/her contacts and use them to solve problems that the other is facing. Burt also argues that the effects of this type of information brokerage do not extend past one-hop in either direction from the mediator [6]. Since the clustering coefficient of a node measures the fraction of a nodes neighbors that are also neighbors, it captures the extent to which a node acts as a mediator between his/her friends. Thus, if a person's clustering coefficient indicates that many of their neighbors, are not neighbors themselves, that person might be in a position to take advantage of this type of information brokerage.

1.1 Contributions

We provide two simple and scalable algorithms for counting the number of triangles incident on each node in a graph using MapReduce. As we will see in the next section, given these counts, and the degree of each node, computing the clustering coefficient is straightforward. Previous work dealt with the massive scale of these graphs by resorting to approximate solutions. Tsourakakis et al. [17] used sampling to give an unbiased estimate of the number of triangles in the whole graph. Becchetti et al. [2] adapt the notion of min wise independent permutations [3] to approximately count the number of triangles incident on every node. We show how a judicious partition of the data can be used to achieve comparable improvements in speed *without* resorting to approximations, and allows us to compute exact clustering coefficient for all nodes.

One of the main technical challenges in computing clustering coefficients that we address is dealing with skew. A salient feature of massive social network data is that there are typically several nodes with extraordinarily high degree [1]. In parallel computations this often leads to some machines taking a much longer time than others to complete the same task on a different chunk of the input. Practitioners are all to familiar with the "curse of the last reducer", in which 99% of the computation finishes quickly, but the remaining 1% takes a disproportionately longer amount of time.

For all of the algorithms we propose, we prove worst case bounds on their performance. Specifically, we show our algorithms perform very well under very skewed data distributions. Whereas for the naive algorithm the running time on different partitions of the data may differ by a factor of 20 or more, the algorithms we propose partition the data across the available nodes almost equally, resulting in much more uniform running times.

Since memory is often a limiting factor when performing computation on large datasets, our second algorithm also allows the user to tradeoff the RAM required per machine with the total disk space required to run the algorithm. For example, by increasing the total amount of disk space necessary by a factor of 2 allows us to reduce the maximum memory necessary per machine by a factor of 4, while performing the same amount of work in aggregate. Since RAM is typically a much more expensive resource than hard disk space, this leads to a favorable tradeoff for the end user.

1.2 Related Work

The problem of computing the clustering coefficient, or almost equivalently counting triangles, in a graph has a rich history. The folklore algorithm iterates over all of the nodes and checks whether any two neighbors of a given node v are themselves connected. Although simple, the algorithm does not perform well in graphs with skewed degree distributions as a single high degree node can lead to an $O(n^2)$ running time, even on sparse graphs.

Chiba and Nishizeki [7] studied this problem in detail and introduced an optimal sequential algorithm for this problem. Their procedure, K3, runs in $O(n)$ time on planar graphs, and more generally in $O(m\alpha(G))$ time where $\alpha(\cdot)$ denotes the arboricity of the graph. (Refer to [7] for further details and definitions.) In his thesis, Schank [16], gives a further description and analysis of a number of different algorithms for this problem.

To battle the fact that the graph may be too large to fit into memory Coppersmith and Kumar [9] and Buriol et al. [4] propose streaming algorithms that estimate the total number of triangles with high accuracy all while using limited space. Becchetti et al. [2] solve the harder problem of estimating the number of triangles incident on each node. Their algorithm takes $O(\log n)$ sequential scans over the data, using $O(n)$ memory to give an approximate number of triangles incident on every node.

Perhaps the work most closely related to ours is that of [17]. The authors give a randomized MapReduce procedure for counting the total number of triangles in a graph and prove that the algorithm gives the correct number of triangles in expectation. They also bound the variance and empirically demonstrate the speedups their algorithm achieves over the naive approach. Our algorithms achieve comparable speedups, but there are two key differences between our work and theirs. First, we give an exact algorithm, whereas they give one that gives the correct answer in expectation. Second, and more importantly, our algorithms give the total number of triangles *incident on each node* whereas they give the *total number of triangles in the entire graph.* Having the number of triangles incident on each node is crucial for computing the clustering coefficient of each node, a metric of importance in the study of social networks.

2. PRELIMINARIES

2.1 Notation and Definitions

Let $G = (V, E)$ be an unweighted, undirected simple graph and let $n = |V|$ and $m = |E|$. Denote by $\Gamma(v)$ the set of neighbors of v, i.e. $\Gamma(v) = \{w \in V \mid (v, w) \in E\}$. Let d_v denote the degree of v, $d_v = |\Gamma(v)|$. The clustering coefficient [18] for a node $v \in V$ in an undirected graph is defined by,

$$cc(v) = \frac{|\{(u, w) \in E \mid u \in \Gamma(v) \text{ and } w \in \Gamma(v)\}|}{\binom{d_v}{2}}.$$

The numerator in the above quotient is the number of edges between neighbors of v. The denominator is the number of possible edges between neighbors of v. Thus, the clustering coefficient measures the fraction of a nodes neighbors that are the themselves connected. There is also an equivalent way to view the clustering coefficient. For a pair (u, w) to contribute to the numerator, they must both be connected to v and they must both be connected to each other. Thus $\{u, v, w\}$ must form a triangle in G. A triangle in G is simply a set of three nodes $\{u, v, w\} \subseteq V$ such that $(u, v), (v, w), (u, w) \in E$. Thus, the algorithms we give in this paper to compute the clustering coefficient will do so by counting the number of triangles incident on each node using MapReduce. Computing the degree of each node for the denominator using MapReduce is straightforward, see [11] for details.

2.2 MapReduce Basics

MapReduce has become a *de facto* standard for parallel computation at terabyte and petabyte scales. In this section we give the reader a brief overview of computation in the MapReduce computing paradigm (see [14, 19] for more information).

The input and intermediate data are stored in $\langle key; value\rangle$ pairs and the computation proceeds in rounds. Each round is split into three consecutive phases: map, shuffle and reduce. In the map phase the input is processed one tuple at a time. This allows different tuples to be processed by different machines and creates an opportunity for massive parallelization. Each machine performing the *map* operation, also known as a mapper, emits a sequence of $\langle key; value\rangle$ pairs which are then passed on to the shuffle phase. This is the synchronization step. In this phase, the MapReduce infrastructure collects all of the tuples emitted by the mappers, aggregates the tuples with the same *key* together and sends them to the same physical machine. Finally each key, along with all the values associated with it, are processed together during the reduce phase. Here too, the operations on data with one key are independent of data with a different key and can be processed in parallel by different machines.

Although MapReduce has allowed for computation on massive data sets to be performed on commodity hardware, it is not a silver bullet, and designers of MapReduce algorithms must still be careful not to overstep the bounds of the system. The authors of [11] give a theoretical model of MapReduce computation and describe the following limitations.

- **Machine Memory** The size of the input to a MapReduce computation is typically too large to fit into memory. Therefore, the memory used by a single mapper or a reducer should be *sublinear* in the total size of the input.
- **Total Memory** In addition to being limited in the memory available to a single machine, the total space used by a MapReduce computation must also remain bounded. While some duplication of the data is allowed, the total space used should always be $o(n^2)$ for an input of size n, thus prohibiting exorbitant data duplication [11].
- **Number of Rounds** Finally, since each MapReduce round can involve shuffling terabytes of data across different machines in a data center, the number of rounds taken by a MapReduce algorithm should remain constant, or at worst logarithmic in the input size.

3. SEQUENTIAL ALGORITHMS

We begin by presenting the folklore algorithm for computing the number of triangles in the graph.

3.1 NodeIterator

We call the process of generating paths of length 2 among the neighbors of node v, "pivoting" around v. The algorithm works by pivoting around each node and then checking if there exist an edge that will complete any of the resulting 2-paths to form a triangle. We give the pseudocode below:

We note that each triangle $\{u, v, w\}$ is counted 6 times total (once as $\{u, v, w\}$, $\{u, w, v\}$, $\{v, u, w\}$, $\{v, w, u\}$, $\{w, u, v\}$, and $\{w, v, u\}$). The analysis of the running time is straightforward, as the algorithm runs in time $O(\sum_{v\in V} d_v^2)$. In constant degree graphs, where $d_v = O(1)$ for all v, this is a linear time algorithm. However, even a single high degree node can lead to a quadratic running time. Since such high degree nodes are often found in real world large graphs, this algorithm is not practical for real-world, massive graphs.

Algorithm 1 NodeIterator(V,E)

```
1: $T \leftarrow 0$;
2: for $v \in V$ do
3:    for $u \in \Gamma(v)$ do
4:       for $w \in \Gamma(v)$ do
5:          if $((u, w) \in E)$ then
6:             $T \leftarrow T + 1/2$;
7: return T / 3;
```

3.2 NodeIterator++

To see how to reduce the running time of the above algorithm, note that each triangle is counted six times, twice by pivoting around each node. Moreover, those pivots around high degree nodes generate far more 2-paths and are thus much more expensive than the pivots around the low degree nodes. To improve upon the baseline algorithm Schank [16] proposed that the lowest degree node in each triangle be "responsible" for making sure the triangle gets counted. Let $\succ$ be a total order on all of the vertices, with the property that $v \succ u$ if $d_v > d_u$, with ties broken arbitrarily (but consistently).

Algorithm 2 NodeIterator++(V,E)

```
1: $T \leftarrow 0$;
2: for $v \in V$ do
3:    for $u \in \Gamma(v)$ and $u \succ v$ do
4:       for $w \in \Gamma(v)$ and $w \succ u$ do
5:          if $((u, w) \in E)$ then
6:             $T \leftarrow T + 1$;
7: return T;
```

The algorithm restricts the set of 2-paths generated from v's neighbors, to be only those where both endpoints of the 2-path have degree higher than v. The exact algorithm is presented in Algorithm 2. Although this is a subtle change, it has a very large impact both in theory and in practice.

LEMMA 1 ([16]). *The running time of Algorithm NodeIterator++ is $O(m^{3/2})$.*

We note that this is the best bound possible. Consider the "lollipop graph" consisting of a clique on $\sqrt{n}$ nodes and a path on the remaining nodes. There are $m = \binom{\sqrt{n}}{2} + n - \sqrt{n} = O(n)$ edges in the graph. Furthermore, there are no triangles among any of the nodes in the path, but there is a triangle between any three nodes in the the clique. Thus the graph has $\binom{\sqrt{n}}{3} = \Theta(n^{3/2})$ triangles.

4. MAPREDUCE ALGORITHMS

The algorithms presented in the previous section implicitly assume that the graph data structure fits into memory of a single machine. For massive graphs, this is no longer case and researchers have turned to parallel algorithms to remedy this problem. Although parallel algorithms allow for computation on much larger scales, they are not a panacea, and algorithm design remains crucial to success of algorithms. In particular, space requirements of algorithms do not disappear simply because there are more machines available. For large enough n, $O(n^2)$ memory is still an unreasonable request—for $n \approx 10^8$, an n^2 space algorithm requires approximately 10 petabytes of storage.

In this section we describe the adaptation of sequential algorithms in the previous section to the MapReduce framework.

4.1 NodeIterator in MapReduce

We begin with the NodeIterator algorithm. The implementation of the algorithm is as follows:

- Round 1: Generate the possible length two paths in the graph by pivoting on every node in parallel.
- Round 2: Check which of the length two paths generated in Round 1 can be closed by an edge in the graph and count the triangles accordingly.

As we noted before, in the NodeIterator algorithm, a single high degree node can lead to the generation of $O(n^2)$ 2-paths. Therefore we adapt the NodeIterator++ algorithm to MapReduce. The general approach is the same as above, and we present the exact algorithm in Algorithm 3. Note that in Round 2, in addition to taking the output of Round 1, the algorithm also takes as input the original edge list. The algorithm differentiates between the two different types of input by using a special character "\$" that is assumed not to appear anywhere else in the input.

Algorithm 3 MR-NodeIterator++(V,E)

```
 1: Map 1: Input: $\langle (u,v); \emptyset \rangle$
 2:    if $v \succ u$ then
 3:       emit $\langle u; v \rangle$
 4: Reduce 1: Input $\langle v;\ S \subseteq \Gamma(v) \rangle$
 5:    for $(u,w) : u,w \in S$ do
 6:       emit $\langle v; (u,w) \rangle$
 7: Map 2:
 8:    if Input of type $\langle v; (u,w) \rangle$ then
 9:       emit $\langle (u,w); v \rangle$
10:    if Input of type $\langle (u,v); \emptyset \rangle$ then
11:       emit $\langle (u,v); \$ \rangle$
12: Reduce 2: Input $\langle (u,w); S \subseteq V \cup \{\$\} \rangle$
13:    if $\$ \in S$ then
14:       for $v \in S \cap V$ do
15:          emit $\langle v; 1 \rangle$
```

4.1.1 Analysis

We analyze the total time and space usage of the MapReduce version of the NodeIterator++ algorithm. Correctness and overall running time follow from the sequential case. The next lemma implies that the total memory required per machine is sublinear. Thus, no reducer gets flooded with too much data, even if there is skew in the input.

LEMMA 2. *The input to any reduce instance in the first round has $O(\sqrt{m})$ edges.*

PROOF. For the purposes of the proof, partition the nodes into two subclasses: the high degree nodes, $\mathcal{H} = \{v \in V : d_v \geq \sqrt{m}\}$ and the low degree nodes, $\mathcal{L} = \{c \in V : d_v < \sqrt{m}\}$. Note that the total number of high degree nodes is at most $2m/\sqrt{m} = O(\sqrt{m})$. Therefore a high degree node can have at most $|\mathcal{H}| = O(\sqrt{m})$ high degree neighbors. If $v \in \mathcal{H}$, the reducer which receives v as a key will only receive $u \in \Gamma(v)$ where $d_u \succ d_v$. Thus this reducer will receive at most neighbors $O(\sqrt{m})$ edges. On the other hand, if the key in the reduce instance is a low node, then its neighborhood has size at most $\sqrt{m}$ by definition. □

The sequential analysis also implies that:

LEMMA 3. *The total number of records output at the end of the first reduce instance is $O(m^{3/2})$.*

The bound in the Lemma above is a worst case bound for a graph with m edges. As we show in Section 5, the actual amount of data output as a result of the Reduce 1 is much smaller in practice. Finally, note that the number of records to any reducer in round 2 for a key (u,v) is at most $O(d_u + d_v) = O(n)$. On graphs where this is value is too large to fit into memory, one can use easily distribute the computation by further partitioning the output of Round 2 across r different machines, and sending a copy of the edge to all r of the machines. We omit this technicality for the sake of clarity and refer the interested reader to [11] for exact details.

4.2 Partition Algorithm

In this section we present a different algorithm for counting triangles. The algorithm works by partitioning the graphs into overlapping subsets so that each triangle is present in at least one of the subsets. Given such a partition, we can then use any sequential triangle counting algorithm as a black box on each partition, and then simply combine the results. We prove that the algorithm achieves perfect work efficiency for the triangle counting problem. As the partitions get finer and finer, the total amount of work spent on finding all of the triangles remains at $O(m^{3/2})$; instead there are simply more machines with each doing less work. Overall the algorithm effectively takes any triangle counting algorithm that works on a single machine and distributes the computation without blowing up the total amount of work done.

We begin by providing a high level overview. First, partition the nodes into ρ equal sized groups, $V = V_1 \cup V_2 \cup \ldots \cup V_\rho$ where $\rho > 0$ with $V_i \cap V_j = \emptyset$ for $i \neq j$. Denote by $V_{ijk} = V_i \cup V_j \cup V_k$. Then let $E_{ijk} = \{(u,w) \in E : u,w \in V_{ijk}\}$ be the set of edges between vertices in V_i, V_j and V_k, and let $G_{ijk} = (V_i \cup V_j \cup V_k, E_{ijk})$ be the induced graph on V_{ijk}. Let $\mathcal{G} = \cup_{i<j<k} G_{ijk}$ be the set of graphs. An instance of a graph G_{ijk} contains $3/\rho$ fraction of the vertices of the original graph, and, in expectation a $O(1/\rho^2)$ fraction of the edges. Since each node of a triangle must be in one part of the partition of V, and $\mathcal{G}$ contains all combinations of parts of V, every triangle in graph G appears in at least one graph in the set $\mathcal{G}$. The algorithm performs a weighted count of the number of triangles in each subgraph in $\mathcal{G}$. The weights correct for the fact that a triangle can occur in many subgraphs. The MapReduce code for the partition step is presented in Algorithm 4. Note that we do not restrict the algorithm used in the reducer, we only insist that the triangles be weighted.

4.2.1 Analysis

We begin by proving the correctness of the algorithm

LEMMA 4. *Each triangle gets counted exactly once after weighting.*

PROOF. Consider a triangle (w,x,y) whose vertices lie in $V_{h(w)}, V_{h(x)}$ and $V_{h(z)}$. If the vertices are mapped to distinct partitions, i.e., $h(w) \neq h(x), h(w) \neq h(y)$, and $h(x) \neq h(z)$ then the triangle will appear exactly once. If the vertices are mapped to two distinct partitions, e.g., $h(x) = h(w)$ but $h(x) \neq h(y)$, the triangle will appear in $\rho - 2$ times. For example, suppose that $\rho = 4$, $h(x) = h(y) = 1$ and $h(z) = 3$,

Algorithm 4 MR-GraphPartition(V,E, ρ)

```
1: Map 1: Input: $\langle (u,v); 1\rangle$
2:     {Let $h(\cdot)$ be a universal hash function into $[0, \rho - 1]$}
3:     $i \leftarrow h(u)$
4:     $j \leftarrow h(v)$
5:     for $a \in [0, \rho - 1]$ do
6:         for $b \in [a + 1, \rho - 1]$ do
7:             for $c \in [b + 1, \rho - 1]$ do
8:                 if $\{i, j\} \subseteq \{a, b, c\}$ then
9:                     emit $\langle (a, b, c); (u, v)\rangle$.
10: Reduce 1: Input $\langle (i, j, k); E_{ijk} \subseteq E\rangle$
11:     Count triangles on $G_{ijk}$
12:     for every triangle $(u, v, w)$ do
13:         $z \leftarrow 1$
14:         if $h(u) = h(v) = h(w)$ then
15:             $z \leftarrow \binom{h(u)}{2} + h(u)(\rho - h(u) - 1) + \binom{\rho - h(u) - 1}{2}$
16:         elif $h(u) = h(v) \mid h(v) = h(w) \mid h(u) = h(w)$ then
17:             $z \leftarrow \rho - 2$
18:         Scale triangle $(u, v, w)$ weight by $1/z$
```

then the triangle will appear in graphs $G_{0,1,3}$ and $G_{1,2,3}$. In the case $h(x) = h(y) = h(w)$ one can verify that the triangle appears exactly $\binom{h(x)}{2} + h(x)(\rho - h(x) - 1) + \binom{\rho - h(x) - 1}{2}$ times. The first term represents graphs G_{ijk} with $i < j < h(x)$, the second those with $i < h(x) < k$ and the last those with $h(x) < j < k$. $\square$

We now dive deeper into the role of the tunable parameter, ρ. At a high level, ρ trades off the total space used by the algorithm (as measured at the end of the map phase) with the size of the input to any reduce instance. We quantify this tradeoff below:

LEMMA 5. *For a setting of ρ:*

- *The expected size of the input to any reduce instance is $O(\frac{m}{\rho^2})$.*
- *The expected total space used at the end of the map phase is $O(\rho m)$.*

PROOF. To prove the first bound, fix one subset $G_{ijk} \in \mathcal{G}$. The subset has $|V_{ijk}| = \frac{3n}{\rho}$ vertices in expectation. Consider a random edge $e = (u, v)$ in the graph, there is a $3/\rho$ probability that $u \in V_{ijk}$, and a further $3/\rho$ probability that $v \in V_{ijk}$. Therefore, for a random edge, it is present in G_{ijk} with probability $9/\rho^2$. The first bound follows by linearity of expectation. For the second bound, observe that there are $O(\rho^3)$ different partitions, and thus the total number of edges emitted by the map phase is $O(\rho^3 \cdot \frac{m}{\rho^2}) = O(m\rho)$. $\square$

The previous lemma implies that increasing the total space used by the algorithm by a factor of 2 results in a factor of 4 improvement in terms of the memory requirements of any reducer. This is a favorable tradeoff in practice since RAM (memory requirements of the reducers) is typically much more expensive than disk space (the space required to store the output of the mappers). Finally, we show that our algorithm is work efficient.

THEOREM 1. *For any $\rho \leq \sqrt{m}$ the total amount of work performed by all of the machines is $O(m^{3/2})$.*

	Nodes	Directed Edges	Undirected Edges
Enron	3.7 x 10^4	3.7 x 10^5	3.7 x 10^5
web-BerkStan	6.9 x 10^5	7.6 x 10^6	1.3 x 10^7
as-Skitter	1.7 x 10^6	11 x 10^6	22 x 10^6
LiveJournal	4.8 x 10^6	6.9 x 10^6	8.6 x 10^7
Twitter	4.2 x 10^7	1.5 x 10^9	2.4 x 10^9

Table 1: Statistics about the data sets used for algorithm evaluation.

PROOF. As we noted in section 3 the running time of the best algorithm for counting triangles is $O(m^{3/2})$. Lemma 5 shows that in the MapReduce setting the input is partitioned across ρ^3 machines, each with a $1/\rho$ fraction of the input. Computing the graph partition takes $O(1)$ time per output edge. The total number of edges output is $O(m\rho) = O(m^{3/2})$ for $\rho < \sqrt{m}$. The running time on each reducer is $O\left(\left(\frac{m}{\rho^2}\right)^{3/2}\right) = O\left(\frac{m^{3/2}}{\rho^3}\right)$. Summing up over the $O(\rho^3)$ graph partitions, we recover the $O(m^{3/2})$ total work. $\square$

5. EXPERIMENTS

In this section we give an experimental evaluation of the MapReduce algorithms defined above. All of the algorithms were run on a cluster of 1636 nodes running the Hadoop implementation of MapReduce.

5.1 Data

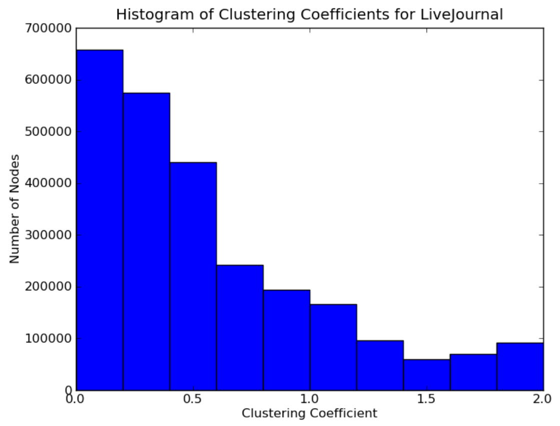
We use several data sets to evaluate our algorithms all of which are publicly available. The Twitter data set is available at [12], and the rest are from the SNAP library[1]. In our experiments we consider each edge of the input to be undirected to make the input graphs even larger, showcasing the scalability of the approach. Thus if an edge (u, v) appears in the input, we also add edge (v, u) if it did not already exist. Table 1 shows how many edges each graph has before and after undirecting the edges.

5.2 Scalability

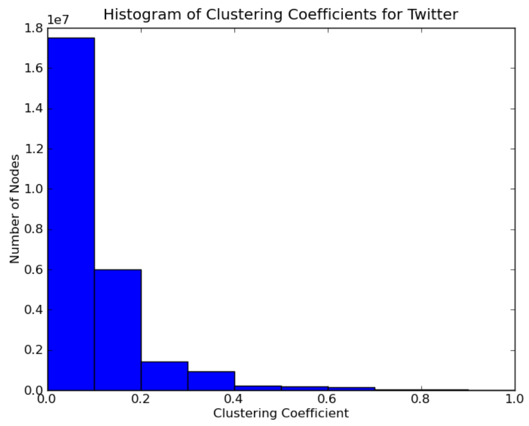
As discussed in Section 3 both the NodeIterator and NodeIterator++ algorithms work by computing length 2-paths in the graph and then checking if there is an edge in the graph that completes the 2-path to form a triangle. If there exists a node of linear degree in the input, the NodeIterator algorithm would output roughly $\Omega(n^2)$ 2-paths, whereas the NodeIterator++ algorithm always checks at most $O(m^{3/2})$ length 2 paths. In order to see if these worst case claims apply in real-world data, Table 2 shows the number of 2-paths generated by both algorithms. In the case of LiveJournal and Enron, NodeIterator++ outputs an order of magnitude fewer 2-paths than NodeIterator. In the case of Twitter, the reduction is almost three orders of magnitude.

As one might expect, the drastic differences in the number of 2-paths generated has a large impact on the running time of the algorithms. In the case of Twitter, the NodeIterator algorithm generates 247 trillion 2-paths. Even if they were encoded using only 10 bytes each, this would amount to 2.5 *petabytes* which is prohibitive to compute with. The NodeIterator++ algorithm only generates 301

[1]`http://snap.stanford.edu`



(a) LiveJournal



(b) Twitter

Figure 1: Figures 1(a) and 1(b) show the distribution of the clustering coefficients for LiveJournal and Twitter.

	NodeItr	NodeItr++	Reduction
Enron	51.13 x 10^6	2.92 x 10^6	17.5
web-BerkStan	56.0 x 10^9	176 x 10^6	318
as-Skitter	32 x 10^9	189 x 10^6	169
LiveJournal	14.5 x 10^9	1.36 x 10^9	10.7
Twitter	246.87 x 10^{12}	3.0 x 10^{11}	820

Table 2: The number of 2-paths generated (and shuffled) by the NodeIterator and the NodeIterator++ algorithms along with the reduction factor.

	NodeIterator	NodeIterator++	GP
web-BerkStan	752	1.77	1.70
as-Skitter	145	1.90	2.08
LiveJournal	59.5	5.33	10.9
Twitter	–	423	483

Table 3: The running times of all algorithms in minutes.

billion 2-paths, which takes about 2 hours. In the case of LiveJournal computing the 2-paths via NodeIterator takes about 50 minutes on average, whereas NodeIterator++ does this in under 2 minutes. To compare the two algorithms, and contrast them against the naive approach, we give the overall running times of all three on various real-world data sets in Table 3.

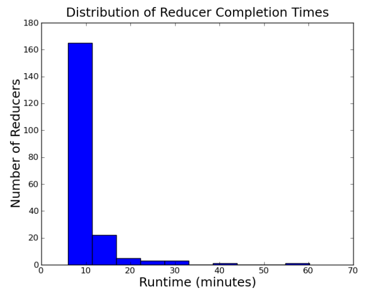
In their work on scaling clustering coefficient Tsourakakis et al [17] give an algorithm that approximates the number of triangles in the entire graph. The authors compare their algorithm to the NodeIterator algorithm and across a wide variety of data sets show that they get speedups ranging from 10 to 100 at approximately 2% cost in accuracy. We show that large speedups can also be obtained without any loss in precision. The algorithm we propose also has the added benefit of computing the number of triangles incident on *each* node, something that is impossible using the Doulion algorithm [17].

Figure 1 shows the distributions of the clustering coefficients in the range (0, 1) for the undirected versions of LiveJournal and Twitter. We removed nodes that have clustering coefficients of exactly 0 or 1 because they tend to be sporadic users of both services that happen to have a few friends that are either completely unconnected or completely connected. Note that there is a larger proportion of nodes with higher clustering coefficients in LiveJournal than in Twitter. Thus it is more likely that two people connected to a given node are also connected in LiveJournal than in Twitter. One possible explanation for this is that people use LiveJournal more for social interactions and such networks often have high clustering [18]. Twitter on the other hand, may be used more for information gathering and dissemination than for the type of social interaction going on in LiveJournal.

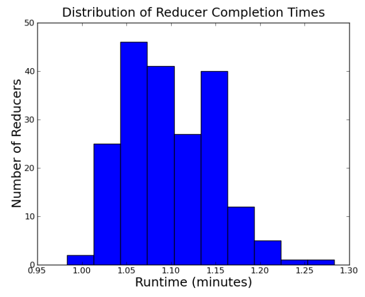
5.3 Data Skew

One problem that is inherently familiar to practitioners working on large data sets is the skew of the data. It is well known, for example, that the degree distribution in many naturally occurring large graphs follows a power law [1], and almost every graph has a handful of nodes with linear degree. This phenomenon manifests itself in computations on large datasets as well. It is rather common to have 99% of the map or reduce tasks finish very quickly, and then spend just as long (if not longer) waiting for the last task to succeed.

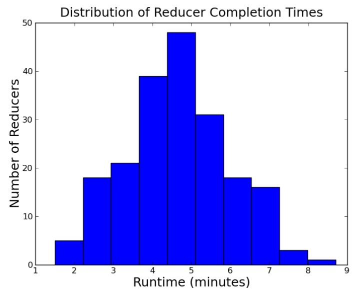
Figures 2(a), 2(b) and 2(c) further explain the differences in wall clock times by showing the distribution of reducer completion times for a typical run of the pivot step of NodeIterator, NodeIterator++ and the main step of GraphPartition. Figure 2(a) illustrates this "curse of the last reducer," by showing a heavy-tailed distribution of reducer completion times for the NodeIterator algorithm. The majority of tasks finish in under 10 minutes, but there are a handful of reducers that take 30 or more minutes and one that takes roughly 60 minutes. The reducers that have the longest completion times received high degree nodes to pivot on. Since for a node of degree d, NodeIterator generates $O(d^2)$ 2-paths, these reducers take a disproportionately long time to run. Figures 2(b) and 2(c) show that the NodeIterator++ and GraphPartition algorithms handle skewed much better.



(a) NodeIterator



(b) NodeIterator++



(c) GraphPartition

Figure 2: Figures 2(a), 2(b) and 2(c) show the distribution of the reducer wall clock times for typical runs of the pivot step (Round 1) for NodeIterator and NodeIterator++ algorithms, and the main round of the GraphPartition algorithm. respectively. All runs in this plot were generated using 200 reducers.

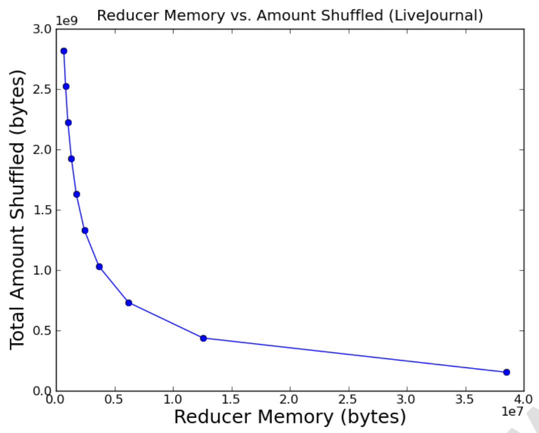


Figure 3: The disk space vs. memory tradeoff for the MR-GraphPartition algorithm as empirically measured on the LiveJournal dataset. The ρ values range from 4 to 31.

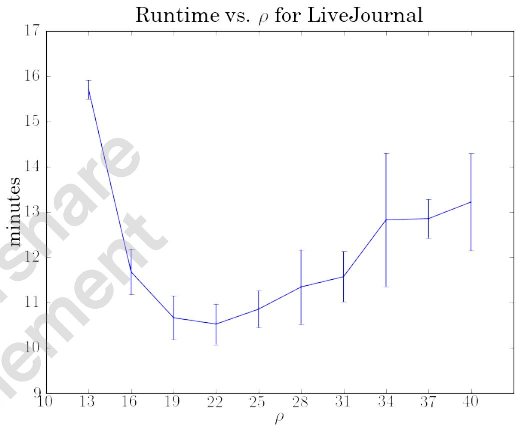


Figure 4: The running time of the MR-GraphPartition algorithm on the LiveJournal dataset. The graph did not fit into memory of a reducer at $\rho < 13$. Error bars indicate 95% confidence intervals.

The distribution of reducer completion times is much more concentrated around the mean.

5.4 Work Efficiency

In Section 4.2.1 we showed that the MR-GraphPartition algorithm allowed the user to trade off between the memory required per machine and the total disk space required to store the intermediate result in the shuffle round. We present this tradeoff for the LiveJournal graph in Figure 3, which plots the two quantities for various values of ρ ranging from 4 to 31. The graph follows almost exactly the tradeoff implied theoretically by Lemma 5. Initially as ρ increases, the total increase in disk space is small but the reduction in memory required per reducer is quite dramatic. As ρ increases further, the decrease in memory is small on an absolute scale, but leads to large and larger increases in disk space required.

In addition to allowing the user to tune the algorithm for the specific parameters of the cluster hardware, the different setting of the parameters changes the overall running time of the job. In Figure 4 we present the running time of the MR-GraphPartition algorithm for different values of ρ. The plot averages five different measurements of ρ. Although the measurements are noisy, since there are many other jobs running on the cluster at the same time, there is a clear decrease in running time as ρ increases initially. We attribute this to the fact that at high ρ the reduce step may need to use more memory than available and page out to disk, which ceases to be a problem as the input size decreases at higher values of ρ. As ρ continues to increase, this benefit disappears and the total running time slowly increases with ρ. This is a combination of two factors. First, for high ρ several reduce tasks are scheduled sequentially on the available machines and are performed serially. Second, full work efficiency is achieved on worst case instances—those where the number of length two paths is $\Theta(m^{3/2})$. Since we are not typically faced with the worst case instances, increasing ρ leads to a slight increase in the total amount of work being done.

5.5 Discussion

In this work we presented two algorithms for efficiently computing clustering coefficients in massive graphs. Our algorithms showcase different aspects of the problem. The first approach, MR-NodeIterator++, is a specialized algorithm designed explicitly for computing the clustering coefficient of every node. The second approach, MR-GraphPartition, describes a more general framework that takes any sequential triangle counting algorithm as a black box and adapts it to the MapReduce setting. In fact, MR-GraphPartition can easily be extended to counting other small subgraphs (for example $K_{2,3}$) in a distributed setting. To count a subgraph H on h nodes, again partition the nodes into ρ equal sized sets; create all possible subsets of size h from the ρ partitions, and distribute the ρ^h subgraphs across different machines.

Both the MR-NodeIterator++ and the MR-GraphPartition algorithms outperform the naive MapReduce implementation for computing the clustering coefficient. The improvement is consistently at least a factor of 10 and often is significantly larger. (For example, it is impossible for us to assess the running time of the naive algorithm on the Twitter graph.) These speedup improvements are comparable to those obtained by [17] and [2], but on a much harder problem. The output of our algorithms gives the *exact* number of triangles incident on every node in the graph. A significant part of the improvement in the running time comes from the fact that the proposed algorithms do a much better job of distributing the computation evenly across the machines, even in the presence of large skew in the input data. This allows them to avoid the "curse of the last reducer," whereby the majority of machines in a distributed job finish quickly, and a single long running job determines the overall running time.

The two algorithms we propose also have different theoretical behavior and exploit different aspects of the MapReduce infrastructure. The MR-NodeIterator++ algorithm uses MapReduce to compute in parallel the intersection between all possible two paths and edges in the graph. Here, the computation itself is trivial, and the infrastructure is used to parallelize this simple approach. The MR-GraphPartition uses the infrastructure to divide the graph into overlapping subgraphs, while ensuring that no triangle is lost. The reducers then execute the sequential algorithm, albeit on a much smaller input.

6. REFERENCES

[1] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[2] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 16–24, New York, NY, USA, 2008. ACM.

[3] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[4] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '06, pages 253–262, New York, NY, USA, 2006. ACM.

[5] Ronald S. Burt. Structural holes and good ideas. *American Journal of Sociology*, 110(2):349–99, September 2004.

[6] Ronald S. Burt. Second-hand brokerage: Evidence on the importance of local structure for managers, bankers, and analysts. *Academy of Management Journal*, 50:119–148, 2007.

[7] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–âĂŞ223, 1985.

[8] James S. Coleman. Social capital in the creation of human capital. *American Journal of Sociology*, 94:S95–S120, 1988.

[9] Don Coppersmith and Ravi Kumar. An improved data stream algorithm for frequency moments. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 151–156, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, pages 137–150, 2004.

[11] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[12] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or news media. In *Proc. 19th International World Wide Web Conference*, 2010.

[13] Jure Leskovec and Eric Horvitz. Planetary-scale views on a large instant-messaging network. In *Proc. 17th International World Wide Web Conference*, 2008.

[14] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Number 7 in Synthesis Lectures on Human Language Technologies. Morgan and Claypool Publishers, April 2010.

[15] Alejandro Portes. Social capital: Its origins and applications in modern sociology. *Annual Review of Sociology*, 24:1–24, 1998.

[16] Thomas Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Universität Karlsruhe (TH), 2007.

[17] Charalampos E. Tsourakakis, U Kang, Gary L. Miller, and Christos Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Knowledge Discovery and Data Mining (KDD)*, 2009.

[18] Duncan Watts and Steve Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 383:440–442, 1998.

[19] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.