Radius Plots for Mining Tera-byte Scale Graphs: Algorithms, Patterns, and Observations

U Kang, SCS, CMU
Charalampos E. Tsourakakis, SCS, CMU
Ana Paula Appel*, CSD, USP at São Carlos
Christos Faloutsos, SCS, CMU
Jure Leskovec†, CSD, Stanford

Abstract

Given large, multi-million node graphs (e.g., FaceBook, web-crawls, etc.), how do they evolve over time? How are they connected? What are the central nodes and the outliers of the graphs? We show that the Radius Plot (pdf of node radii) can answer these questions. However, computing the Radius Plot is prohibitively expensive for graphs reaching the planetary scale.

There are two major contributions in this paper: (a) We propose HADI (HAdoop DIameter and radii estimator), a carefully designed and fine-tuned algorithm to compute the diameter of massive graphs, that runs on the top of the HADOOP /MAPREDUCE system, with excellent scale-up on the number of available machines (b) We run HADI on several real world datasets including YahooWeb (*6B edges, 1/8 of a Terabyte*), one of the largest public graphs ever analyzed.

Thanks to HADI, we report fascinating patterns on large networks, like the surprisingly small effective diameter, the multi-modal/bi-modal shape of the Radius Plot, and its palindrome motion over time.

1 Introduction

How do real, Terabyte-scale graphs look like? Is it true that the nodes with the highest degree are the most central ones, i.e., have the smallest radius? How do we compute the diameter and node radii in graphs of such size?

Graphs appear in numerous settings, such as social networks (FaceBook, LinkedIn), computer network intrusion logs, who-calls-whom phone networks, search engine click-streams (term-URL bipartite graphs), and many more. The contributions of this paper are the following:

1. *Design:* We propose HADI, a scalable algorithm, to compute the radii and diameter of network. As shown in Figure 1, our method is *7.6*$\times$ faster than the naive version.
2. *Optimization and Experimentation:* We carefully fine-tune our algorithm, and we tested it on one of the largest public web graph ever analyzed, with several *billions* of nodes and edges, spanning 1/8 of a Terabyte.
3. *Observations:* Thanks to HADI, we find interesting patterns and observations, like the "Multi-modal and Bi-modal" pattern, and the surprisingly small effective diameter of the Web. For example, see the Multi-modal pattern in the radius plot of Figure 1, which also shows the effective diameter and the center node of the Web('google.com').

The HADI algorithm (implemented in HADOOP) and several datasets are available at `http://www.cs.cmu.edu/~ukang/HADI`. The rest of the paper is organized as follows: Section 2 defines related terms and a sequential algorithm for the Radius Plot. Section 3 describes large scale algorithms for the Radius Plot, and Section 4 analyzes the complexity of the algorithms and provides a possible extension. In Section 5 we present timing results, and in Section 6 we observe interesting patterns. After describing backgrounds in Section 7, we conclude in Section 8.

2 Preliminaries; Sequential Radii Calculation

2.1 Definitions In this section, we define several terms related to the radius and the diameter. Recall that, for a node v in a graph G, the *radius* $r(v)$ of v is the distance between v and a reachable node farthest away from v. The *diameter* $d(G)$ of a graph G is the maximum radius of nodes $v \in G$. That is, $d(G) = \max_v r(v)$.

Since the radius and the diameter are susceptible to outliers (e.g., long chains), we follow the literature and define the *effective* radius and diameter as follows.

DEFINITION 1. (EFFECTIVE RADIUS) *For a node v in a graph G, the effective radius $r_{eff}(v)$ of v is the 90th-percentile of all the distances from v.*

*Work performed while visiting CMU.

†Work performed while at CMU.

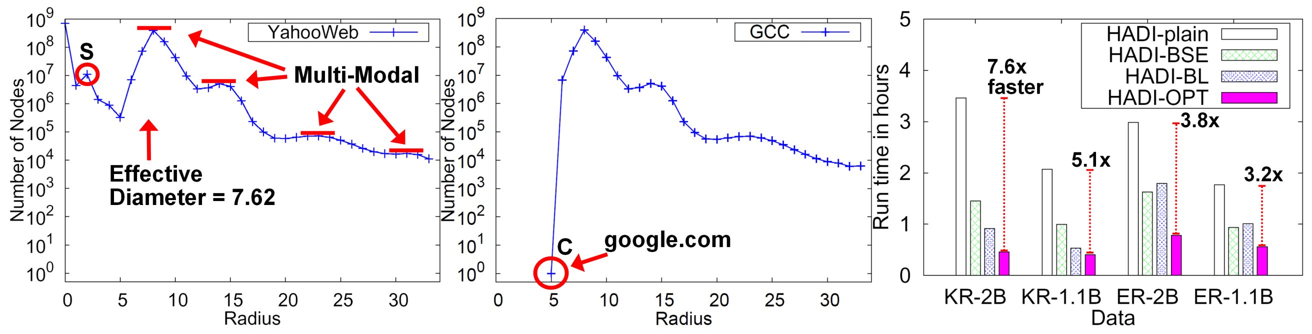


Figure 1: **(Left)** Radius Plot(Count versus Radius) of the YahooWeb graph. Notice the effective diameter is surprisingly small. Also notice the peak(marked 'S') at radius 2, due to star-structured disconnected components.
(Middle) Radius Plot of GCC(Giant Connected Component) of YahooWeb graph. The *only* node with radius 5 (marked 'C') is `google.com`.
(Right) Running time of HADI with/without optimizations for Kronecker and Erdős-Rényi graphs with billions edges. Run on the M45 HADOOP cluster, using 90 machines for 3 iterations. HADI-OPT is up to **7.6**× faster than HADI-plain.

Symbol	Definition
G	a graph
n	number of nodes in a graph
m	number of edges in a graph
d	diameter of a graph
h	number of hops
$N(h)$	number of node-pairs reachable in $\leq h$ hops (neighborhood function)
$N(h,i)$	number of neighbors of node i reachable in $\leq h$ hops
$b(h,i)$	Flajolet-Martin bitstring for node i at h hops.
$\hat{b}(h,i)$	Partial Flajolet-Martin bitstring for node i at h hops

Table 1: Table of symbols

DEFINITION 2. (EFFECTIVE DIAMETER) *The effective diameter $d_{eff}(G)$ of a graph G is the minimum number of hops in which 90% of all connected pairs of nodes can reach each other.*

We will use the following three Radius-based Plots:

1. **Static Radius Plot** (or just "Radius Plot") of graph G shows the distribution (count) of the effective radius of nodes at a specific time, as shown in Figure 1.
2. **Temporal Radius Plot** shows the distributions of effective radius of nodes at several times(see Figure 9 for an example).
3. **Radius-Degree Plot** shows the scatter-plot of the effective radius $r_{eff}(v)$ versus the degree d_v for each node v, as shown in Figure 8.

Table 1 lists the symbols used in this paper.

2.2 Computing Radius and Diameter To generate the Radius Plot, we need to calculate the effective radius of every node. In addition, the effective diameter is useful for tracking the evolution of networks. Therefore, we describe our algorithm for computing the effective radius and the effective diameter of a graph. As described in Section 7, existing algorithms do not scale well. To handle graphs with billions of nodes and edges, we use the following two main ideas:

1. We use an approximation rather than an exact algorithm.
2. We design a parallel algorithm for HADOOP /MAPREDUCE (the algorithm can also run in a parallel RDBMS).

To approximate the effective radius and the effective diameter, we use the Flajolet-Martin algorithm [17][29] for counting the number of distinct elements in a multiset. While many other applicable algorithms exist (e.g., [6], [10], [18]), we choose the Flajolet-Martin algorithm because it gives an unbiased estimate, as well as a tight $O(\log n)$ bound for the space complexity [3].

The main idea is that we maintain K Flajolet-Martin (FM) bitstrings $b(h,i)$ for each node i and current hop number h. $b(h,i)$ encodes the number of nodes reachable from node i within h hops, and can be used to estimate radii and diameter as shown below. The bitstrings $b(h,i)$ are iteratively updated until the bitstrings of all nodes stabilize. At the h-th iteration, each node receives the bitstrings of its neighboring nodes, and updates its own bitstrings $b(h-1,i)$ handed over from the previous iteration:

$$b(h,i) = b(h-1,i) \text{ BIT-OR } \{b(h-1,j) | (i,j) \in E\} \quad (2.1)$$

where "BIT-OR" denotes bitwise OR. After h iterations, a node i has K bitstrings that encode the *neighborhood function* $N(h, i)$, that is, the number of nodes within h hops from the node i. $N(h, i)$ is estimated from the K bitstrings by

$$N(h, i) = \frac{1}{0.77351} 2^{\frac{1}{K} \sum_{l=1}^{K} b_l(i)} \quad (2.2)$$

where $b_l(i)$ is the position of leftmost '0' bit of the l^{th} bitstring of node i. The iterations continue until the bitstrings of all nodes stabilize, which is a necessary condition that the current iteration number h exceeds the diameter $d(G)$. After the iterations finish at h_{max}, we can calculate the effective radius for every node and the diameter of the graph, as follows:

- $r_{eff}(i)$ is the smallest h such that $N(h, i) \geq 0.9 \cdot N(h_{max}, i)$.
- $d_{eff}(G)$ is the smallest h such that $N(h) = \sum_i N(h, i) \geq 0.9 \cdot N(h_{max})$.

Algorithm 1 shows the summary of the algorithm described above.

Algorithm 1 Computing Radii and Diameter

```
Input: Input graph G and integers $MaxIter$ and $K$
Output: $r_{eff}(i)$ of every node $i$, and $d_{eff}(G)$
1: for $i = 1$ to $n$ do
2:   $b(0, i) \leftarrow$ NewFMBitstring($n$);
3: end for
4: for $h = 1$ to $MaxIter$ do
5:   $Changed \leftarrow 0$;
6:   for $i = 1$ to $n$ do
7:     for $l = 1$ to $K$ do
8:       $b_l(h, i) \leftarrow b_l(h-1, i)$BIT-OR$\{b_l(h-1, j) | \forall j$ adjacent from $i\}$;
9:     end for
10:    if $\exists l$ s.t. $b_l(h, i) \neq b_l(h-1, i)$ then
11:      increase $Changed$ by 1;
12:    end if
13:  end for
14:  $N(h) \leftarrow \sum_i N(h, i)$;
15:  if $Changed$ equals to 0 then
16:    $h_{max} \leftarrow h$, and break for loop;
17:  end if
18: end for
19: for $i = 1$ to $n$ do {estimate eff. radii}
20:   $r_{eff}(i) \leftarrow$ smallest $h'$ where $N(h', i) \geq 0.9 \cdot N(h_{max}, i)$;
21: end for
22: $d_{eff}(G) \leftarrow$ smallest $h'$ where $N(h') \geq 0.9 \cdot N(h_{max})$;
```

The parameter K is typically set to 32[17], and $MaxIter$ is set to 256 since real graphs have relatively small effective diameter. The NewFMBitstring() function in line 2 generates K FM bitstrings [17]. The effective radius $r_{eff}(i)$ is determined at line 20, and the effective diameter $d_{eff}(G)$ is determined at line 22.

Algorithm 1 runs in $O(dm)$ time, since the algorithm iterates at most d times with each iteration running in $O(m)$ time. By using approximation, Algorithm 1 runs faster than previous approaches (see Section 7 for discussion). However, Algorithm 1 is a sequential algorithm and requires $O(n \log n)$ space and thus can not handle extremely large graphs (more than billions of nodes and edges) which can not be fit into a single machine. In the next sections we present efficient parallel algorithms.

3 Proposed Method

In the next two sections we describe HADI, a parallel radius and diameter estimation algorithm. As mentioned in Section 2, HADI can run on the top of both a MAPREDUCE system and a parallel SQL DBMS. In the following, we first describe the general idea behind HADI and show the algorithm for MAPREDUCE. The algorithm for parallel SQL DBMS is sketched in Section 4.

3.1 HADI Overview HADI follows the flow of Algorithm 1; that is, it uses the FM bitstrings and iteratively updates them using the bitstrings of its neighbors. The most expensive operation in Algorithm 1 is line 8 where bitstrings of each node are updated. Therefore, HADI focuses on the efficient implementation of the operation using MAPREDUCE framework.

It is important to notice that HADI is a disk-based algorithm; indeed, memory-based algorithm is not possible for Tera- and Peta-byte scale data. HADI saves two kinds of information to a distributed file system (such as HDFS (Hadoop Distributed File System) in the case of HADOOP):

- **Edge** has a format of ($srcid$, $dstid$).
- **Bitstrings** has a format of ($nodeid$, $bitstring_1$, ..., $bitstring_K$).

Combining the bitstrings of each node with those of its neighbors is very expensive operation which needs several optimization to scale up near-linearly. In the following sections we will describe three HADI algorithms in a progressive way. That is we first describe HADI-naive, to give the big picture and explain why it such a naive implementation should not be used in practice, then the HADI-plain, and finally HADI-optimized, the proposed method that should be used in practice. We use HADOOP to describe the MAPREDUCE version of HADI.

3.2 HADI-naive in MAPREDUCE HADI-naive is inefficient, but we present it for ease of explanation.

Data The edge file is saved as a sparse adjacency matrix in HDFS. Each line of the file contains a nonzero element of the adjacency matrix of the graph, in the format of

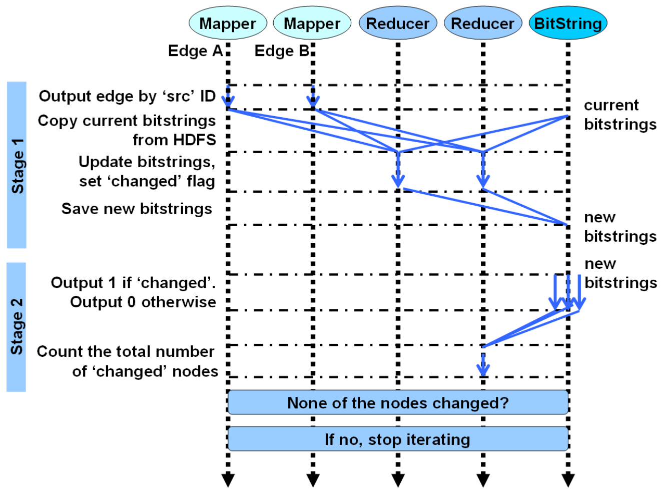

Mapper
Mapper
Reducer
Reducer
BitString
Edge A
Edge B
Stage 1
Output edge by 'src' ID
Copy current bitstrings from HDFS
Update bitstrings, set 'changed' flag
Save new bitstrings
current bitstrings
new bitstrings
Stage 2
Output 1 if 'changed'. Output 0 otherwise
new bitstrings
Count the total number of 'changed' nodes
None of the nodes changed?
If no, stop iterating

Figure 2: One iteration of HADI-naive. First stage: Bitstrings of all nodes are sent to every reducer. Second stage: sums up the count of changed nodes. The multiple arrows at the beginning of Stage 2 mean that there may be many machines containing bitstrings.

($srcid$, $dstid$). Also, the bitstrings of each node are saved in a file in the format of ($nodeid$, $flag$, $bitstring_1$, ..., $bitstring_K$). The $flag$ records information about the status of the nodes(e.g., 'Changed' flag to check whether one of the bitstrings changed or not). Notice that we *don't know* the physical distribution of the data in HDFS.

Main Program Flow The main idea of HADI-naive is to use the bitstrings file as a logical "cache" to machines which contain edge files. The bitstring update operation in Equation (2.1) requires that the machine which updates the bitstrings of node i should have access to (a) all edges adjacent from i, and (b) all bitstrings of the adjacent nodes. To meet the requirement (a), it is needed to reorganize the edge file such that edges with a same source id are grouped together. That can be done by using an Identity mapper which outputs the given input edges in ($srcid$, $dstid$) format. The most simple yet naive way to meet the requirement (b) is sending the bitstrings to every machine which receives the reorganized edge file.

Thus, HADI-naive iterates over two-stages of MAPREDUCE. The first stage updates the bitstrings of each node and sets the 'Changed' flag if at least one of the bitstrings of the node is different from the previous bitstring. The second stage counts the number of changed nodes and stops iterations when the bitstrings stabilized, as illustrated in the swim-lane diagram of Figure 2.

Although conceptually simple and clear, HADI-naive is unnecessarily expensive, because it ships all the bitstrings to all reducers. Thus, we propose HADI-plain and additional optimizations, which we explain next.

3.3 HADI-plain in MAPREDUCE

HADI-plain improves HADI-naive by *copying only the necessary bitstrings to each reducer*. The details are next:

Data As in HADI-naive, the edges are saved in the format of ($srcid$, $dstid$), and bitstrings are saved in the format of ($nodeid$, $flag$, $bitstring_1$, ..., $bitstring_K$) in files over HDFS. The initial bitstrings generation, which corresponds to line 1-3 of Algorithm 1, can be performed in completely parallel way. The $flag$ of each node records the following information:

- **Effective Radii** and Hop Numbers to calculate the effective radius.
- **Changed** flag to indicate whether at least a bitstring has been changed or not.

Main Program Flow As mentioned in the beginning, HADI-plain copies only the necessary bitstrings to each reducer. The main idea is to replicate bitstrings of node j exactly x times where x is the in-degree of node j. The replicated bitstrings of node j is called the *partial bitstring* and represented by $\hat{b}(h,j)$. The replicated $\hat{b}(h,j)$'s are used to update $b(h,i)$, the bitstring of node i where (i,j) is an edge in the graph. HADI-plain iteratively runs three-stage MAPREDUCE jobs until all bitstrings of all nodes stop changing. Algorithm 2, 3, 4 shows HADI-plain. We use h for the current iteration number, starting from h=1. Output(a,b) means to output a pair of data with the key a and the value b.

Stage 1 We generate (key, value) pairs, where the key is a node id i and the value is the partial bitstrings $\hat{b}(h,j)$'s where j ranges over all the neighbors adjacent from node i. To generate such pairs, the bitstrings of node j are grouped together with edges whose $dstid$ is j. Notice that at the very first iteration, bitstrings of nodes do not exist; they have to be generated on the fly, and we use the *Bitstring Creation Command* for that. Notice also that line 22 of Algorithm 2 is used to propagate the bitstrings of one's own node. These bitstrings are compared to the newly updated bitstrings at Stage 2 to check convergence.

Stage 2 Bitstrings of node i are updated by combining partial bitstrings of itself and nodes adjacent from i. For the purpose, the mapper is the Identity mapper (output the input without any modification). The reducer combines them, generates new bitstrings, and sets $flag$ by recording (a) whether at least a bitstring changed or not, and (b) the current iteration number h and the neighborhood value $N(h,i)$ (line 9). This h and $N(h,i)$ are used to calculate the effective radius of nodes after all bitstrings converge, i.e., don't change. Notice that only the last neighborhood $N(h_{last},i)$ and other neighborhoods $N(h',i)$ that satisfy $N(h',i) \geq 0.9 \cdot N(h_{last},i)$ need to be saved to calculate the effective radius. The output of Stage 2 is fed into the input of Stage 1 at the next iteration.

Stage 3 We calculate the number of changed nodes and

Algorithm 2 HADI Stage 1

```
Input: Edge data $E = \{i, j)\}$,
    Current bitstring $B = \{(i, b(h-1, i))\}$ or
    Bitstring Creation Command $BC = \{(i, cmd)\}$
Output: Partial bitstring $B' = \{(i, b(h-1, j))\}$
1: Stage1-Map(key $k$, value $v$);
2: if $(k, v)$ is of type B or BC then
3:     Output($k, v$);
4: else if $(k, v)$ is of type E then
5:     Output($v, k$);
6: end if
7:
8: Stage1-Reduce(key $k$, values $V$[]);
9: SRC ← [];
10: for $v \in V$ do
11:     if $(k, v)$ is of type BC then
12:         $\hat{b}(h-1, k)$ ←NewFMBitstring();
13:     else if $(k, v)$ is of type B then
14:         $\hat{b}(h-1, k) \leftarrow v$;
15:     else if $(k, v)$ is of type E then
16:         Add $v$ to $SRC$;
17:     end if
18: end for
19: for $src \in SRC$ do
20:     Output($src, \hat{b}(h-1, k)$);
21: end for
22: Output($k, \hat{b}(h-1, k)$);
```

Algorithm 3 HADI Stage 2

```
Input: Partial bitstring $B = \{(i, \hat{b}(h-1, j)\}$
Output: Full bitstring $B = \{(i, b(h, i)\}$
1: Stage2-Map(key $k$, value $v$); // Identity Mapper
2: Output($k, v$);
3:
4: Stage2-Reduce(key $k$, values $V$[]);
5: $b(h, k) \leftarrow 0$;
6: for $v \in V$ do
7:     $b(h, k) \leftarrow b(h, k)$ BIT-OR $v$;
8: end for
9: Update $flag$ of $b(h, k)$;
10: Output($k, b(h, k)$);
```

sum up the neighborhood value of all nodes to calculate $N(h)$. We use only two unique keys(key_for_changed and key_for_neighborhood), which correspond to the two calculated values. The analysis of line 2 can be done by checking the $flag$ field and using Equation (2.2) in Section 2.

When all bitstrings of all nodes converged, a MAPREDUCE job to finalize the effective radius and diameter is performed and the program finishes. Compared to HADI-naive, the advantage of HADI-plain is clear: bitstrings and edges are evenly distributed over machines so that the algorithm can handle as much data as possible, given sufficiently many machines.

Algorithm 4 HADI Stage 3

```
Input: Full bitstring $B = \{(i, b(h, i))\}$
Output: Number of changed nodes, Neighborhood $N(h)$
1: Stage3-Map(key $k$, value $v$);
2: Analyze $v$ to get ($changed$, $N(h, i)$);
3: Output(key_for_changed,$changed$);
4: Output(key_for_neighborhood, $N(h, i)$);
5:
6: Stage3-Reduce(key $k$, values $V$[]);
7: $Changed \leftarrow 0$;
8: $N(h) \leftarrow 0$;
9: for $v \in V$ do
10:     if k is key_for_changed then
11:         $Changed \leftarrow Changed + v$;
12:     else if k is key_for_neighborhood then
13:         $N(h) \leftarrow N(h) + v$;
14:     end if
15: end for
16: Output(key_for_changed,$Changed$);
17: Output(key_for_neighborhood, $N(h)$);
```

3.4 HADI-optimized in MAPREDUCE HADI-optimized further improves HADI-plain. It uses two orthogonal ideas: "block operation" and "bit shuffle encoding". Both try to address some subtle performance issues. Specifically, HADOOP has the following two major bottlenecks:

- Materialization: at the end of each map/reduce stage, the output is written to the disk, and it is also read at the beginning of next reduce/map stage.
- Sorting: at the *Shuffle* stage, data is sent to each reducer and sorted before they are handed over to the *Reduce* stage.

HADI-optimized addresses these two issues.

Block Operation Our first optimization is the block encoding of the edges and the bitstrings. The main idea is to group w by w sub-matrix into a super-element in the adjacency matrix E, and group w bitstrings into a super-bitstring. Now, HADI-plain is performed on these super-elements and super-bitstrings, instead of the original edges and bitstrings. Of course, appropriate decoding and encoding is necessary at each stage. Figure 3 shows an example of converting data to block.

By this block operation, the performance of HADI-plain changes as follows:

- *Input size* decreases in general, since we can use fewer bits to index elements inside a block.
- *Sorting* time decreases, since the number of elements to sort decreases.
- *Network traffic* decreases since the result of matching a super-element and a super-bitstring is a bitstring which

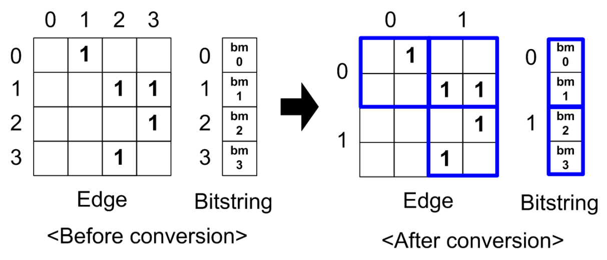


Figure 3: Converting the original edge and bitstring to blocks. The 4-by-4 edge and length-4 bitstring are converted to 2-by-2 super-elements and length-2 super-bitstrings. Notice the lower-left super-element of the edge is not produced since there is no nonzero element inside it.

can be at maximum $block\_width$ times smaller than that of HADI-plain.

- *Map and Reduce functions* takes more time, since the block must be decoded to be processed, and be encoded back to block format.

For reasonable-size blocks, the performance gains (smaller input size, faster sorting time, less network traffic) outweigh the delays (more time to perform the map and reduce function). Also notice that the number of edge blocks depends on the community structure of the graph: if the adjacency matrix is nicely clustered, we will have fewer blocks. See Section 5, where we show results from block-structured graphs ('Kronecker graphs' [24]) and from random graphs ('Erdős-Rényi graphs' [15]).

Bit Shuffle Encoding In our effort to decrease the input size, we propose an encoding scheme that can compress the bitstrings. Recall that in HADI-plain, we use K (e.g., 32, 64) bitstrings for each node, to increase the accuracy of our estimator. Since HADI requires $K \cdot ((n+m)\log n)$ space, the amount of data increases when K is large. For example, the YahooWeb graph in Section 6 spans 120 GBytes (with 1.4 billion nodes, 6.6 billion edges). However the required disk space for just the bitstrings is $32 \cdot (1.4B + 6.6B) \cdot 8$ byte = 2 Tera bytes (assuming 8 byte for each bitstring), which is more than 16 times larger than the input graph.

The main idea of Bit Shuffle Encoding is to carefully reorder the bits of the bitstrings of each node, and then use run length encoding. By construction, the leftmost part of each bitstring is almost full of one's, and the rest is almost full of zeros. Specifically, we make the reordered bit strings to contain long sequences of 1's and 0's: we get all the first bits from all K bitstrings, then get the second bits, and so on. As a result we get a single bit-sequence of length $K * |bitstring|$, where most of the first bits are '1's, and most of the last bits are '0's. Then we encode only the length of each bit sequence, achieving good space savings (and, eventually, time savings, through fewer I/Os).

4 Analysis and Discussion

In this section, we analyze the time/space complexity of HADI and its possible implementation at RDMBS.

4.1 Time and Space Analysis We analyze the algorithm complexity of HADI with M machines for a graph G with n nodes and m edges with diameter d. We are interested in the time complexity, as well as the space complexity.

LEMMA 4.1. (TIME COMPLEXITY OF HADI) *HADI takes* $O(\frac{d(n+m)}{M} log \frac{n+m}{M})$ *time.*

Proof. (Sketch) The Shuffle steps after Stage1 takes $O(\frac{n+m}{M} \log \frac{n+m}{M})$ time which dominates the time complexity. □

Notice that the time complexity of HADI is less than previous approaches in Section 7($O(n^2 + nm)$, at best). Similarly, for space we have:

LEMMA 4.2. (SPACE COMPLEXITY OF HADI) *HADI requires* $O((n+m)\log n)$ *space.*

Proof. (Sketch) The maximum space $k \cdot ((n+m)\log n)$ is required at the output of `Stage1`-Reduce. Since k is a constant, the space complexity is $O((n+m)\log n)$. □

4.2 HADI in parallel DBMSs Using relational database management systems (RDBMS) for graph mining is a promising research direction, especially given the findings of [31]. We mention that HADI can be implemented on top of an Object-Relational DBMS (parallel or serial): it needs repeated joins of the edge file with the appropriate file of bitstrings, and a user-defined function for bit-OR-ing. See [20] for details.

5 Scalability of HADI

In this section, we perform experiments to answer the following questions:

- Q1: How fast is HADI?
- Q2: How does it scale up with the graph size and the number of machines?
- Q3: How do the optimizations help performance?

5.1 Experimental Setup We use both real and synthetic graphs in Table 2 for our experiments and analysis in Section 5 and 6, with the following details.

- YahooWeb: web pages and their hypertext links indexed by Yahoo! Altavista search engine in 2002.
- Patents: U.S. patents, citing each other (from 1975 to 1999).
- LinkedIn: people connected to other people (from 2003 to 2006).

Graph	Nodes	Edges	File	Desc.
YahooWeb	1.4 B	6.6 B	116G	page-page
LinkedIn	7.5 M	58 M	1G	person-person
Patents	6 M	16 M	264M	patent-patent
Kronecker	177 K	1,977 M	25G	synthetic
	120 K	1,145M	13.9G	
	59 K	282 M	3.3G	
Erdős-Rényi	177 K	1,977 M	25G	random $G_{n,p}$
	120 K	1,145 M	13.9G	
	59 K	282 M	3.3G	

Table 2: Datasets. B: Billion, M: Million, K: Thousand, G: Gigabytes

- Kronecker: Synthetic Kronecker graphs [24] using a chain of length two as the seed graph.

For the performance experiments, we use synthetic Kronecker and Erdős-Rényi graphs. The reason of this choice is that we can generate any size of these two types of graphs, and Kronecker graph mirror several real-world graph characteristics, including small and constant diameters, power-law degree distributions, etc. The number of nodes and edges of Erdős-Rényi graphs have been set to the same values of the corresponding Kronecker graphs. The main difference of Kronecker compared to Erdős-Rényi graphs is the emergence of a block-wise structure of the adjacency matrix, from its construction [24]. We will see how this characteristic affects in the running time of our block-optimization in the next sections.

HADI runs on *M45*, one of the fifty most powerful supercomputers in the world. M45 has 480 hosts (each with 2 quad-core Intel Xeon 1.86 GHz, running RHEL5), with 3Tb aggregate RAM, and over 1.5 Peta-byte disk size.

Finally, we use the following notations to indicate different optimizations of HADI:

- HADI-BSE: HADI-plain with bit shuffle encoding.
- HADI-BL: HADI-plain with block operation.
- HADI-OPT: HADI-plain with bit shuffle encoding and block operation.

5.2 Running Time and Scale-up Figure 4 gives the wall-clock time of HADI-OPT versus the number of edges in the graph. Each curve corresponds to a different number of machines used (from 10 to 90). HADI has excellent scalability, with its running time being linear on the number of edges. The rest of the HADI versions (HADI-plain, HADI-BL, and HADI-BSE), were slower, but had a similar, linear trend, and they are omitted to avoid clutter.

Figure 5 gives the throughput $1/T_M$ of HADI-OPT. We also tried HADI with one machine; however it didn't complete, since the machine would take so long that it would often fail in the meanwhile. For this reason, we do not report the typical scale-up score $s = T_1/T_M$ (ratio of time with 1 machine, over time with M machine), and instead we report just the inverse of T_M. HADI scales up near-linearly with the number of machines M, close to the ideal scale-up.

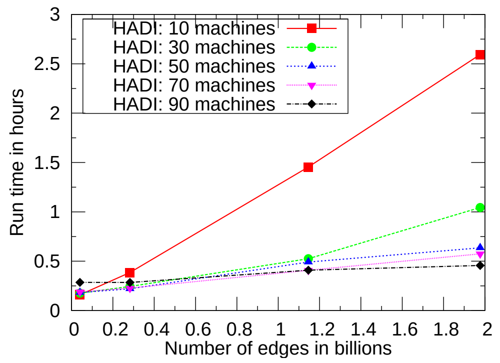


Figure 4: Running time versus number of edges with HADI-OPT on Kronecker graphs for three iterations. Notice the excellent scalability: linear on the graph size (number of edges).

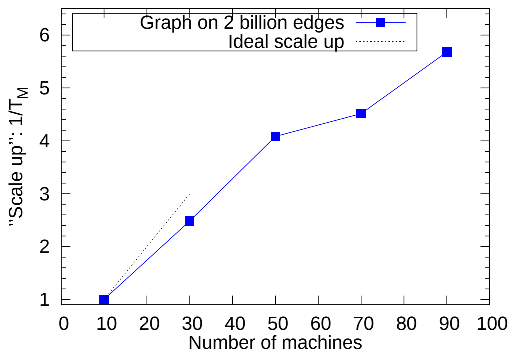


Figure 5: "Scale-up" (throughput $1/T_M$) versus number of machines M, for the Kronecker graph (2B edges). Notice the near-linear growth in the beginning, close to the ideal(dotted line).

5.3 Effect of Optimizations Among the optimizations that we mentioned earlier, which one helps the most, and by how much? Figure 1 plots the running time of different graphs versus different HADI optimizations. For the Kronecker graphs, we see that block operation is more efficient than bit shuffle encoding. Here, HADI-OPT achieves **7.6**× better performance than HADI-plain. For the Erdős-Rényi graphs, however, we see that block operations do not help more than bit shuffle encoding, because the adjacency matrix has no block structure, as Kronecker graphs do. Also notice that HADI-BLK and HADI-OPT run faster on Kronecker graphs than on Erdős-Rényi graphs of the same size. Again, the reason is that Kronecker graphs have fewer nonzero blocks (i.e., "communities") by their construction, and the "block" operation yields more savings.

6 HADI At Work

HADI reveals new patterns in massive graphs which we present in this section.

6.1 Static Patterns

Diameter What is the effective diameter of the Web? Barabasi et al. [2] conjectured that it is around 19 for the 1.4 billion-node Web, and Broder et al. [7] reported 6.83 by sampling from $\approx$ 200 million-nodes Web. What should be the diameter, for a significantly larger crawl of the web, with billions of nodes? Figure 1 gives the surprising answer:

OBSERVATION 1. (SMALL WEB) *The effective diameter of the YahooWeb graph (year: 2002) is surprisingly small ($\approx 7 \sim 8$).*

Shape of Distribution The next question is, how are the radii distributed in real networks? Is it Poisson? Lognormal? Figure 1 gives the surprising answer: multimodal! In other relatively small networks, however, have bi-modal structures. As shown in the Radius Plot of Patent and LinkedIn network in Figure 6, they have a peak at zero, a dip at a small radius value (9, and 4, respectively) and another peak very close to the dip. Other small networks (including IMDB), had similar bi-modal behavior but we omitted here for brevity. Given the prevalence of bi-modal shape, our conjecture is that the multi-modal shape of YahooWeb is possibly due to a mixture of relatively smaller sub-graphs, which got loosely connected recently.

OBSERVATION 2. (MULTI-MODAL AND BI-MODAL) *The Radius distribution of the Web graph has a multi-modal structure. Many smaller networks have bi-modal structures.*

About the bi-modal structures, a natural question to ask is what are the common properties of the nodes that belong to the first peak; similarly, for the nodes in the first dip, and the same for the nodes of the second peak. After investigation, the former are nodes that belong to the disconnected components ('DC's); nodes in the dip are usually core nodes in the giant connected component (GCC), and the nodes at the second peak are the vast majority of well connected nodes in the GCC. Figure 7 exactly shows the radii distribution for the nodes of the GCC (in red), and the nodes of the few largest remaining components. Notice that the first peak disappeared, exactly because it consists of nodes from the DCs (Disconnected Components), that we omitted here.

In Figure 6, '*outsiders*' are nodes in the disconnected components, and responsible for the first peak and the negative slope to the dip. '*Core*' are the central nodes from the giant connected component. '*Whiskers*' [26] are the nodes connected to the GCC with long paths(resembling a whisker), and are the reasons of the second negative slope.

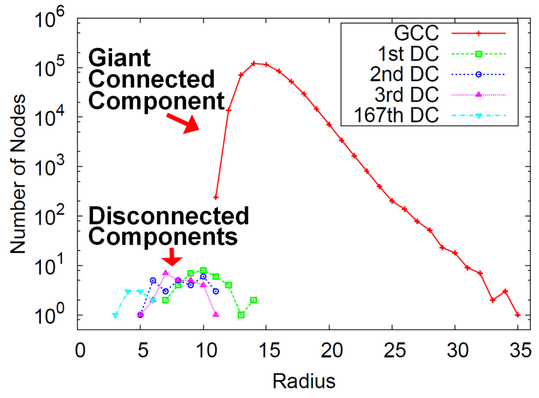


Figure 7: Radius plot (Count versus radius) for several connected components of the Patent data in 1985. In red: the distribution for the GCC (Giant Connected Component); rest colors: several DC (Disconnected Component)s. Notice that the first peak from Figure 6(a) disappeared.

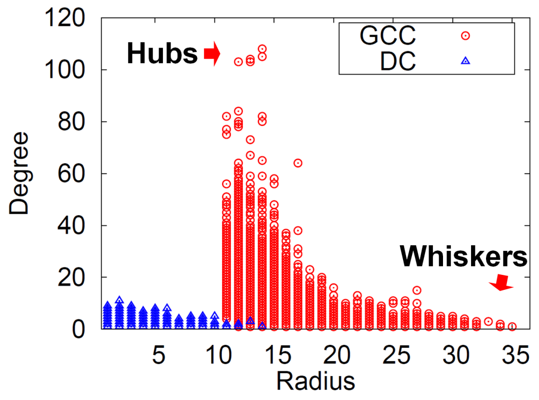
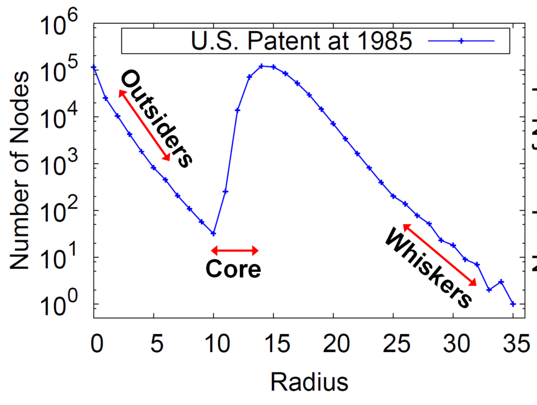


Figure 8: Radius-Degree plot of Patent at 1985. Notice that the hubs are not necessarily the nodes with smallest radius within GCC, and whiskers have small degree.

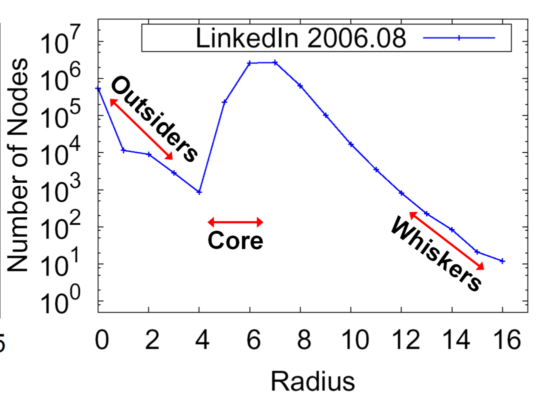
Radius plot of GCC Figure 1(b) shows a striking pattern: all nodes of the GCC of the YahooWeb graph have radius 6 or more, except for 1 (only!). Inspection shows that this is `google.com`. We were surprised, because we would expect a few more popular nodes to be in the same situation (eg., Yahoo, eBay, Amazon).

"Whisker" nodes The next question is, what can we say about the connectivity of the core nodes, and the *whisker* nodes? For example, is it true that the highest degree nodes are the most central ones (i.e. minimum radius)? The answer is given by the "Radius-Degree" plot in Figure 8: This is a scatter-plot, with one dot for every node, plotting the degree of the node versus its radius. We also color-coded the nodes of the GCC (in red), while the rest are in blue.

OBSERVATION 3. (HIGH DEGREE NODES) *The highest degree nodes (a) belong to the GCC but (b) are* not *necessarily the ones with the smallest radius.*



(a) U.S. Patent



(b) LinkedIn

Figure 6: Static Radius Plot(Count versus Radius) of U.S. Patent and LinkedIn. Notice the bi-modal structure with ‘outsiders’, ‘core’, and ‘whiskers’.

The next observation is that *whisker* nodes have small degree, that is, they belong to chains (as opposed to more complicated shapes)

6.2 Temporal Patterns Here we study how the radius distribution changes over time. We know that the diameter of a graph typically grows with time, spikes at the ‘gelling point’, and then shrinks [28],[25]. Indeed, this holds for our datasets (plots omitted for brevity).

The question is, how does the radius distribution change over time? Does it still have the bi-modal pattern? Do the peaks and slopes change over time? We show the answer in Figure 9 and Observation 4.

OBSERVATION 4. (EXPANSION-CONTRACTION) *The radius distribution expands to the right until it reaches the gelling point. Then, it contracts to the left.*

Another striking observation is that the two decreasing segments seem to be well fit by a line, in log-lin axis, thus indicating an exponential decay.

OBSERVATION 5. (EXPONENTIAL DECAYS) *The decreasing segments of several, real radius plots seem to decay exponentially, that is*

$$count(r) \propto \exp(-cr) \qquad (6.3)$$

for every time tick after *the gelling point. $count(r)$ is the number of nodes with radius r, and c is a constant.*

For the Patents dataset, the correlation coefficient was excellent, (typically, -0.98 or better).

7 Background

We briefly present related works on algorithms for radius and diameter computation, as well as on large graph mining.

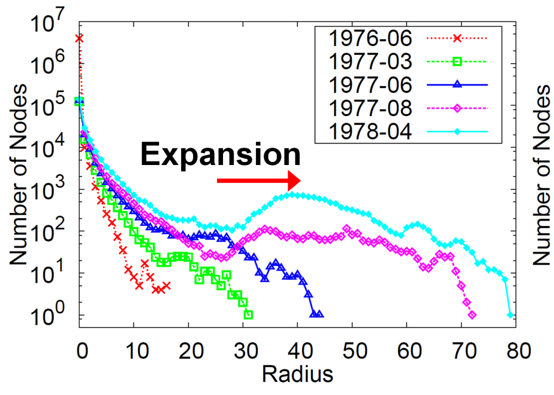
Computing Radius and Diameter The typical algorithms to compute the radius and the diameter of a graph include Breadth First Search (BFS) and Floyd’s algorithm ([11]). Both approaches are prohibitively slow for large graphs, requiring $O(n^2 + nm)$ and $O(n^3)$ time, where n and m are the number of nodes and edges, respectively. For the same reason, related BFS or all-pair shortest-path based algorithms like [16], [4], [27], [34] can not handle large graphs.

A sampling approach starts BFS from a subset of nodes, typically chosen at random as in [7]. Despite its practicality, this approach has no obvious solution for choosing the representative sample for BFS.

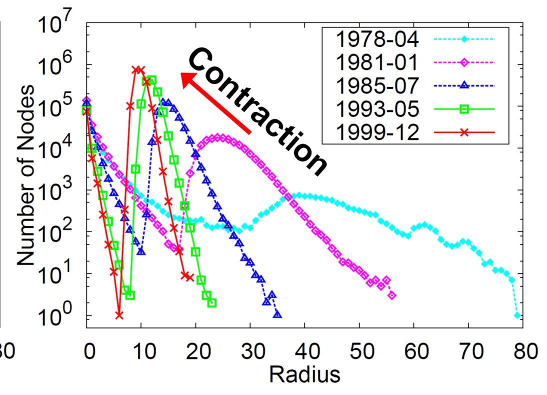
Large Graph Mining There are numerous papers on large graph mining and indexing, mining subgraphs([22], [39], ADI[37], gSpan[38]), graph clustering([33], Graclus [13], METIS [21]), partitioning([12], [9], [14]), tensors([23]), triangle counting([5], [35], [36]), minimum cut([1]), to name a few. However, none of the above computes the diameter of the graph or radii of the nodes.

Large scale data processing using scalable and parallel algorithms has attracted increasing attention due to the needs to process web-scale data. Due to the volume of the data, platforms for this type of processing choose “shared-nothing” architecture. Two promising platforms for such large scale data analysis are (a) MAPREDUCE and (b) parallel RDBMS.

The MAPREDUCE programming framework processes huge amounts of data in a massively parallel way, using thousands or millions commodity machines. It has advantages of (a) fault-tolerance, (b) familiar concepts from functional programming, and (c) low cost of building the cluster. HADOOP, the open source version of MAPREDUCE, is a very promising tool for massive parallel graph mining applications, (e.g., cross-associations [30], connected components [20]). Other advanced MAPREDUCE-like systems include [19], [8], and [32].



(a) Patent-Expansion



(b) Patent-Contraction

Figure 9: Radius distribution over time. "Expansion": the radius distribution moves to the right until the gelling point. "Contraction": the radius distribution moves to the left after the gelling point.

Parallel RDBMS systems, including Vertica and Aster Data, are based on traditional database systems and provide high performance using distributed processing and query optimization. They have strength in processing structured data. For detailed comparison of these two systems, see [31]. Again, none of the above articles shows how to use such platforms to efficiently compute the diameter of a graph.

8 Conclusions

Our main goal is to develop an open-source package to mine Giga-byte, Tera-byte and eventually Peta-byte networks. We designed HADI, an algorithm for computing radii and diameter of Tera-byte scale graphs, and analyzed large networks to observe important patterns. The contributions of this paper are the following:

- *Design:* We developed HADI, a scalable MAPREDUCE algorithm for diameter and radius estimation, on massive graphs.
- *Optimization:* Careful fine-tunings on HADI, leading to up to *7.6*× faster computation, linear scalability on the size of the graph (number of edges) and near-linear speed-up on the number of machines. The experiments ran on the M45 HADOOP cluster of Yahoo, one of the 50 largest supercomputers in the world.
- *Observations:* Thanks to HADI, we could study the diameter and radii distribution of one of the largest public web graphs ever analyzed (over 6 *billion* edges); we also observed the "Small Web" phenomenon, multi-modal/bi-modal radius distributions, and palindrome motions of radius distributions over time in real networks.

Future work includes algorithms for additional graph mining tasks like computing eigenvalues, and outlier detection, for graphs that span Tera- and Peta-bytes.

Acknowledgments

This work was partially funded by the National Science Foundation under Grants No. IIS-0705359, IIS-0808661, CAPES (PDEE project number 3960-07-2), CNPq, Fapesp and under the auspices of the U.S. Dept. of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. We would like to thank YAHOO! for the web graph and access to the M45, and Adriano A. Paterlini for feedback. The opinions expressed are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] C. C. Aggarwal, Y. Xie, and P. S. Yu. Gconnect: A connectivity index for massive disk-resident graphs. *PVLDB*, 2009.

[2] R. Albert, H. Jeong, and A.-L. Barabasi. Diameter of the world wide web. *Nature*, (401):130–131, 1999.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments, 1996.

[4] D. A. Bader and K. Madduri. A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. *Parallel Comput.*, 2008.

[5] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*, 2008.

[6] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. SIGMOD, 2007.

[7] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks 33*, 2000.

[8] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy

and efficient parallel processing of massive data sets. *VLDB*, 2008.

[9] D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *KDD*, 2004.

[10] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. PODS, 2000.

[11] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[12] S. Daruru, N. M. Marin, M. Walker, and J. Ghosh. Pervasive parallelism in data mining: dataflow solution to co-clustering large and sparse netflix data. In *KDD*, 2009.

[13] I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE TPAMT*, 2007.

[14] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *KDD*, 2003.

[15] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 1959.

[16] J.-A. Ferrez, K. Fukuda, and T. Liebling. Parallel computation of the diameter of a graph. In *HPCSA*, 1998.

[17] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 1985.

[18] M. N. Garofalakis and P. B. Gibbon. Approximate query processing: Taming the terabytes. *VLDB*, 2001.

[19] R. L. Grossman and Y. Gu. Data mining using high performance data clouds: experimental studies using sector and sphere. *KDD*, 2008.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM*, 2009.

[21] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[22] Y. Ke, J. Cheng, and J. X. Yu. Top-k correlative graph mining. *SDM*, 2009.

[23] T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, 2008.

[24] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *PKDD*, pages 133–145, 2005.

[25] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, 2005.

[26] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW '08*, 2008.

[27] J. Ma and S. Ma. Efficient parallel algorithms for some graph theory problems. *JCST*, 1993.

[28] M. Mcglohon, L. Akoglu, and C. Faloutsos. Weighted graphs and disconnected components: patterns and a generator. *KDD*, 2008.

[29] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. Anf: a fast and scalable tool for data mining in massive graphs. *KDD*, pages 81–90, 2002.

[30] S. Papadimitriou and J. Sun. Disco: Distributed co-clustering with map-reduce. *ICDM*, 2008.

[31] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. SIGMOD, June 2009.

[32] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 2005.

[33] V. Satuluri and S. Parthasarathy. Scalable graph clustering using stochastic flows: applications to community discovery. *KDD*, 2009.

[34] B. P. Sinha, B. B. Bhattacharya, S. Ghose, and P. K. Srimani. A parallel algorithm to compute the shortest paths and diameter of a graph and its vlsi implementation. *IEEE Trans. Comput.*, 1986.

[35] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. *KDD*, 2009.

[36] C. E. Tsourakakis, M. N. Kolountzakis, and G. L. Miller. Approximate triangle counting. *CoRR*, abs/0904.3761, 2009.

[37] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. *KDD*, 2004.

[38] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *ICDM*, 2002.

[39] C. H. You, L. B. Holder, and D. J. Cook. Learning patterns in the dynamics of biological networks. In *KDD*, 2009.