Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation

U Kang Brendan Meeder Christos Faloutsos

Carnegie Mellon University, School of Computer Science
{ukang,bmeeder,christos}@cs.cmu.edu

Abstract. Given a graph with billions of nodes and edges, how can we find patterns and anomalies? Are there nodes that participate in too many or too few triangles? Are there close-knit near-cliques? These questions are expensive to answer unless we have the first several eigenvalues and eigenvectors of the graph adjacency matrix. However, eigensolvers suffer from subtle problems (e.g., convergence) for large sparse matrices, let alone for billion-scale ones.
We address this problem with the proposed HEIGEN algorithm, which we carefully design to be accurate, efficient, and able to run on the highly scalable MAPREDUCE (HADOOP) environment. This enables HEIGEN to handle matrices more than *1000*$\times$ larger than those which can be analyzed by existing algorithms. We implement HEIGEN and run it on the M45 cluster, one of the top 50 supercomputers in the world. We report important discoveries about near-cliques and triangles on several real-world graphs, including a snapshot of the Twitter social network (*38Gb*, 2 billion edges) and the "YahooWeb" dataset, one of the largest publicly available graphs (*120Gb*, 1.4 billion nodes, 6.6 billion edges).

1 Introduction

Graphs with billions of edges, or *billion-scale* graphs, are becoming common; Facebook boasts about 0.5 billion active users, who-calls-whom networks can reach similar sizes in large countries, and web crawls can easily reach billions of nodes. Given a billion-scale graph, how can we find near-cliques, the count of triangles, and related graph properties? As we discuss later, triangle counting and related expensive operations can be computed quickly, provided we have the first several eigenvalues and eigenvectors. In general, spectral analysis is a fundamental tool not only for graph mining, but also for other areas of data mining. Eigenvalues and eigenvectors are at the heart of numerous algorithms such as triangle counting, singular value decomposition (SVD), spectral clustering, and tensor analysis [10]. In spite of their importance, existing eigensolvers do not scale well. As described in Section 6, the maximum order and size of input matrices feasible for these solvers is million-scale.

In this paper, we discover patterns on near-cliques and triangles, on several real-world graphs including a Twitter dataset (*38Gb*, over 2 billion edges) and the "YahooWeb" dataset, one of the largest publicly available graphs (*120Gb*, 1.4 billion nodes, 6.6 billion edges). To enable discoveries, we propose HEIGEN, an eigensolver for billion-scale, sparse symmetric matrices built on the top of HADOOP, an open-source MAPREDUCE framework. Our contributions are the following:

1. **Effectiveness:** With HEIGEN we analyze billion-scale real-world graphs and report discoveries, including a high triangle vs. degree ratio for adult sites and web pages that participate in billions of triangles.
2. **Careful Design:** We choose among several serial algorithms and selectively parallelize operations for better efficiency.
3. **Scalability:** We use the HADOOP platform for its excellent scalability and implement several optimizations for HEIGEN, such as cache-based multiplications and skewness exploitation. This results in linear scalability in the number of edges, the same accuracy as standard eigensolvers for small matrices, and more than a 76× performance improvement over a naive implementation.

Due to our focus on scalability, HEIGEN can handle sparse symmetric matrices with *billions of nodes and edges*, surpassing the capability of previous eigensolvers (e.g. [20] [16]) by more than *1,000* ×. Note that HEIGEN is different from Google's PageRank algorithm since HEIGEN computes top k eigenvectors while PageRank computes only the first eigenvector. Designing top k eigensolver is much difficult and subtle than designing the first eigensolver, as we will see in Section 4. With this powerful tool we are able to study several billion-scale graphs, and we report fascinating patterns on the near-cliques and triangle distributions in Section 2.

The HEIGEN algorithm (implemented in HADOOP) is available at `http://www.cs.cmu.edu/~ukang/HEIGEN`. The rest of the paper presents the discoveries in real-world networks, design decisions and details of our method, experimental results, and conclusions.

2 Discoveries

In this section, we show discoveries on billion-scale graphs using HEIGEN. We focus on the structural properties of networks: spotting near-cliques and finding triangles. The graphs we used in this and Section 5 are described in Table 1. [1]

Name	Nodes	Edges	Description
YahooWeb	1,413 M	6,636 M	WWW pages in 2002
Twitter	62.5 M	2,780 M	who follows whom in 2009/11
LinkedIn	7.5 M	58 M	person-person in 2006
Kronecker	59 K ∼ 177 K	282 M ∼ 1,977 M	synthetic graph
Epinions	75 K	508 K	who trusts whom

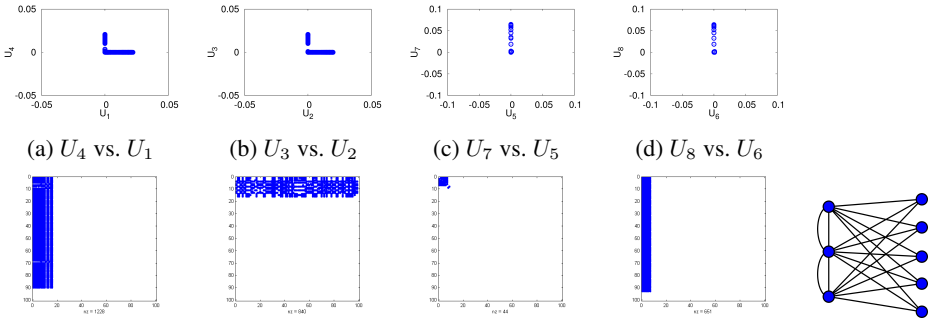
Table 1. Order and size of networks.

2.1 Spotting Near-Cliques

In a large, sparse network, how can we find tightly connected nodes, such as those in near-cliques or bipartite cores? Surprisingly, eigenvectors can be used for this purpose [14]. Given an adjacency matrix W and its SVD $W = U\Sigma V^T$, an *EE-plot* is

[1] YahooWeb, LinkedIn: released under NDA.
Twitter: http://www.twitter.com/
Kronecker: http://www.cs.cmu.edu/∼ukang/dataset
Epinions: not public data.

defined to be the scatter plot of the vectors U_i and U_j for any i and j. EE-plots of some real-world graphs contain clear separate lines (or 'spokes'), and the nodes with the largest values in each spoke are separated from the other nodes by forming near-cliques or bipartite cores. Figures 1 shows several EE-plots and spyplots (i.e., adjacency matrix of induced subgraph) of the top 100 nodes in top eigenvectors of YahooWeb graph.



(a) U_4 vs. U_1 (b) U_3 vs. U_2 (c) U_7 vs. U_5 (d) U_8 vs. U_6

(e) U_1 spoke (f) U_2 spoke (g) U_3 spoke (h) U_4 spoke (i) Structure of bi-clique

Fig. 1. EE-plots and spyplots from YahooWeb. **(a)-(d)**: EE-plots showing the values of nodes in the ith eigenvector vs. in the jth eigenvector. Notice the clear 'spokes' in top eigenvectors signify the existence of a strongly related group of nodes in near-cliques or bi-cliques as depicted in (i). **(e)-(h)**: Spyplots of the top 100 largest nodes from each eigenvector. Notice that we see a near clique in U_3, and bi-cliques in U_1, U_2, and U_4. **(i)**: The structure of 'bi-clique' in (e), (f), and (h).

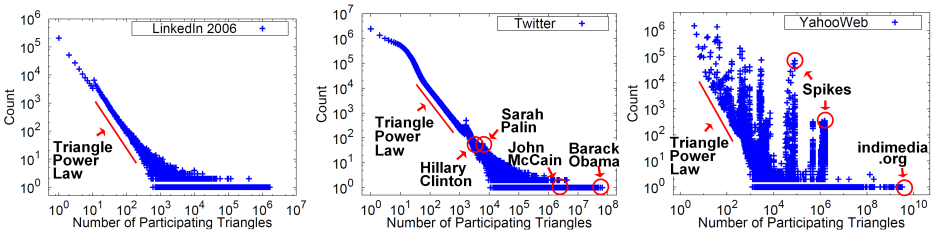
In Figure 1 (a) - (d), we observe clear 'spokes,' or outstanding nodes, in the top eigenvectors. Moreover, the top 100 nodes with largest values in U_1, U_2, and U_4 form a 'bi-clique', shown in (e), (f), and (h), which is defined to be the combination of a clique and a bipartite core as depicted in Figure 1 (i). Another observation is that the top seven nodes shown in Figure 1 (g) belong to `indymedia.org` which is the site with the maximum number of triangles in Figure 2.

Observation 1 (Eigenspokes) *EE-plots of YahooWeb show clear spokes. Additionally, the extreme nodes in the spokes belong to cliques or bi-cliques.*

2.2 Triangle Counting

Given a particular node in a graph, how are its neighbors connected? Do they form stars? Cliques? The above questions about the community structure of networks can be answered by studying triangles (three nodes connected to each other). However, directly counting triangles in graphs with billions of nodes and edges is prohibitively expensive [19]. Fortunately, we can approximate triangle counts with high accuracy using HEIGEN by exploiting its connection to eigenvalues [18]. In a nutshell, the total number of triangles in a graph is related to the sum of cubes of eigenvalues, and the first few eigenvalues provide extremely good approximations. A slightly more elaborate analysis approximates the number of triangles in which a node participates, using the cubes of the first few eigenvalues and the corresponding eigenvectors.

Using the top k eigenvalues computed with HEIGEN, we analyze the distribution of triangle counts of real graphs including the Linkedin, Twitter social, and YahooWeb



(a) LinkedIn (58M edges) (b) Twitter (2.8B edges) (c) YahooWeb (6.6B edges)

Fig. 2. The distribution of the number of participating triangles of real graphs. In general, they obey the "triangle power-law." Moreover, well-known U.S. politicians participate in many triangles, demonstrating that their followers are well-connected. In the YahooWeb graph, we observe several anomalous spikes which possibly come from cliques.

graphs in Figure 2. We first observe that there exists several nodes with extremely large triangle counts. In Figure 2 (b), Barack Obama is the person with the fifth largest number of participating triangles, and has many more than other U.S. politicians. In Figure 2 (c), the web page `lists.indymedia.org` contains the largest number of triangles; this page is a list of mailing lists which apparently point to each other.

We also observe regularities in triangle distributions and note that the beginning part of the distributions follows a power-law.

Observation 2 (Triangle power law) *The beginning part of the triangle count distribution of real graphs follows a power-law.*

In the YahooWeb graph in Figure 2 (c), we observe many spikes. One possible reason of the spikes is that they come from cliques: a k-clique generates k nodes with $\binom{k-1}{2}$ triangles.

Observation 3 (Spikes in triangle distribution) *In the Web graph, there exist several spikes which possibly come from cliques.*

The rightmost spike in Figure 2 (c) contains 125 web pages that each have about 1 million triangles in their neighborhoods. They all belong to the news site `ucimc.org`, and are connected to a tightly coupled group of pages.

Triangle counts exhibit even more interesting patterns when combined with the degree information as shown in the degree-triangle plot of Figure 3. We see that celebrities have high degree and mildly connected followers, while accounts for adult sites have many fewer, but extremely well connected, followers. Degree-triangle plots can be used to spot and eliminate harmful accounts such as those of adult advertisers and spammers.

Observation 4 (Anomalous Triangles vs. Degree Ratio) *In Twitter, anomalous accounts have very high triangles vs. degree ratio compared to other regular accounts.*

All of the above observations need a fast, scalable eigensolver. This is exactly what HEIGEN does, and we describe our proposed design next.

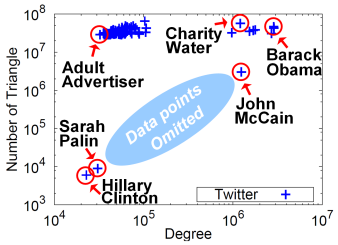


Fig. 3. The degree vs. participating triangles of some 'celebrities' (rest: omitted, for clarity) in Twitter accounts. Also shown are accounts of adult sites which have smaller degree, but belong to an abnormally large number of triangles (= many, well connected followers - probably, 'robots').

3 Background - Sequential Algorithms

In the next two sections, we describe our method of computing eigenvalues and eigenvectors of billion-scale graphs. We first describe sequential algorithms to find eigenvalues and eigenvectors of matrices. We limit our attention to symmetric matrices due to the computational difficulties; even the best methods for non-symmetric eigensolver require much more computation than symmetric eigensolvers. We list the alternatives for computing the eigenvalues of symmetric matrix and the reasoning behind our choice.

- **Power method**: the simplest and most famous method for computing the topmost eigenvalue. However, it can not find the top k eigenvalues.
- **Simultaneous iteration (or QR)**: an extension of the Power method to find top k eigenvalues. It requires large matrix-matrix multiplications that are prohibitively expensive for billion-scale graphs.
- **Lanczos-NO(No Orthogonalization)**: the basic Lanczos algorithm [5] which approximates the top k eigenvalues in the subspace composed of intermediate vectors from the Power method. The problem is that while computing the eigenvalues, they can 'jump' up to larger eigenvalues, thereby outputting spurious eigenvalues.

Although all of the above algorithms are not suitable for calculations on billion-scale graphs using MAPREDUCE, we present a tractable, MAPREDUCE-based algorithm for computing the top k eigenvectors and eigenvalues in the next section.

4 Proposed Method

In this section we describe HEIGEN, a parallel algorithm for computing the top k eigenvalues and eigenvectors of symmetric matrices in MAPREDUCE.

4.1 Summary of the Contributions

Efficient top k eigensolvers for billion-scale graphs require careful algorithmic considerations. The main challenge is to carefully design algorithms that work well on distributed systems and exploit the inherent structure of data, including block structure and skewness, in order to be efficient. We summarize the algorithmic contributions here and describe each in detail in later sections.

1. **Careful Algorithm Choice:** We carefully choose a sequential eigensolver algorithm that is efficient for MAPREDUCE and gives accurate results.
2. **Selective Parallelization:** We group operations into expensive and inexpensive ones based on input sizes. Expensive operations are done in parallel for scalability, while inexpensive operations are performed faster on a single machine.
3. **Blocking:** We reduce the running time by decreasing the input data size and the amount of network traffic among machines.
4. **Exploiting Skewness:** We decrease the running time by exploiting skewness of data.

4.2 Careful Algorithm Choice

In Section 3, we considered three algorithms that are not tractable for analyzing billion-scale graphs with MAPREDUCE. Fortunately, there is an algorithm suitable for such a purpose. Lanczos-SO (Selective Orthogonalization) improves on the Lanczos-NO by selectively reorthogonalizing vectors instead of performing full reorthogonalizations.

Algorithm 1: Lanczos -SO(Selective Orthogonalization)

Input: Matrix $A^{n \times n}$, random n-vector b, maximum number of steps m, error threshold ϵ
Output: Top k eigenvalues $\lambda[1..k]$, eigenvectors $Y^{n \times k}$

$\beta_0 \leftarrow 0$, $v_0 \leftarrow 0$, $v_1 \leftarrow b/||b||$;
for $i = 1..m$ **do**
  $v \leftarrow Av_i$; // Find a new basis vector
  $\alpha_i \leftarrow v_i^T v$;
  $v \leftarrow v - \beta_{i-1} v_{i-1} - \alpha_i v_i$; // Orthogonalize against two previous basis vectors
  $\beta_i \leftarrow ||v||$;
  $T_i \leftarrow$ (build tri-diagonal matrix from α and β);
  $QDQ^T \leftarrow EIG(T_i)$; // Eigen decomposition of T_i
  for $j = 1..i$ **do**
    if $\beta_i |Q[i, j]| \leq \sqrt{\epsilon} ||T_i||$ **then**
      $r \leftarrow V_i Q[:, j]$;
      $v \leftarrow v - (r^T v) r$; // Selectively orthogonalize
    end if
  end for
  if (v was selectively orthogonalized) **then**
    $\beta_i \leftarrow ||v||$; // Recompute normalization constant β_i
  end if
  if $\beta_i = 0$ **then**
    break for loop;
  end if
  $v_{i+1} \leftarrow v/\beta_i$;
end for
$T \leftarrow$ (build tri-diagonal matrix from α and β);
$QDQ^T \leftarrow EIG(T)$; // Eigen decomposition of T
$\lambda[1..k] \leftarrow$ top k diagonal elements of D; // Compute eigenvalues
$Y \leftarrow V_m Q_k$; // Compute eigenvectors. Q_k is the columns of Q corresponding to λ

The main idea of Lanczos-SO is as follows: We start with a random initial basis vector b which comprises a rank-1 subspace. For each iteration, a new basis vector

is computed by multiplying the input matrix with the previous basis vector. The new basis vector is then orthogonalized against the last two basis vectors and is added to the previous rank-$(m-1)$ subspace, forming a rank-m subspace. Let m be the number of the current iteration, Q_m be the $n \times m$ matrix whose ith column is the ith basis vector, and A be the matrix for which we want to compute eigenvalues. We also define $T_m = Q_m^* A Q_m$ to be a $m \times m$ matrix. Then, the eigenvalues of T_m are good approximations of the eigenvalues of A . Furthermore, multiplying Q_m by the eigenvectors of T_m gives good approximations of the eigenvectors of A. We refer to [17] for further details.

If we used exact arithmetic, the newly computed basis vector would be orthogonal to all previous basis vectors. However, rounding errors from floating-point calculations compound and result in the loss of orthogonality. This is the cause of the spurious eigenvalues in Lanczos-NO. Orthogonality can be recovered once the new basis vector is fully re-orthogonalized to all previous vectors. However, doing this becomes expensive as it requires $O(m^2)$ re-orthogonalizations, where m is the number of iterations. A better approach uses a quick test (line 10 of Algorithm 1) to selectively choose vectors that need to be re-orthogonalized to the new basis [6]. This selective-reorthogonalization idea is shown in Algorithm 1.

The Lanczos-SO has all the properties that we need: it finds the top k largest eigenvalues and eigenvectors, it produces no spurious eigenvalues, and its most expensive operation, a matrix-vector multiplication, is tractable in MAPREDUCE. Therefore, we choose Lanczos-SO as our choice of the sequential algorithm for parallelization.

4.3 Selective Parallelization

Among many sub-operations in Algorithm 1, which operations should we parallelize? A naive approach is to parallelize all the operations; however, some operations run more quickly on a single machine rather than on multiple machines in parallel. The reason is that the overhead incurred by using MAPREDUCE exceeds gains made by parallelizing the task; simple tasks where the input data is very small complete faster on a single machine. Thus, we divide the sub-operations into two groups: those to be parallelized and those to be run in a single machine. Table 2 summarizes our choice for each sub-operation. Note that the last two operations in the table can be done with a single-machine standard eigensolver since the input matrices are tiny; they have m rows and columns, where m is the number of iterations.

4.4 Blocking

Minimizing the volume of information sent between nodes is important to designing efficient distributed algorithms. In HEIGEN, we decrease the amount of network traffic by using the block-based operations. Normally, one would put each edge "(source, destination)" in one line; HADOOP treats each line as a data element for its 'map()' function. Instead, we propose to divide the adjacency matrix into blocks (and, of course, the corresponding vectors also into blocks), and put the edges of each block on a single line, and compress the source- and destination-ids. This makes the map() function a bit more complicated to process blocks, but it saves significant transfer time of data over the network. We use these edge-blocks and the vector-blocks for many parallel operations in Table 2, including matrix-vector multiplication, vector update, vector dot product, vector scale, and vector L2 norm. Performing operations on blocks is faster than doing

Operation	Description	Input	P?
$y \leftarrow y + ax$	vector update	Large	Yes
$\gamma \leftarrow x^T x$	vector dot product	Large	Yes
$y \leftarrow \alpha y$	vector scale	Large	Yes
$\|\|y\|\|$	vector L2 norm	Large	Yes
$y \leftarrow M^{n \times n} x$	large matrix-large,dense vector multiplication	Large	Yes
$y \leftarrow M_s^{n \times m} x_s$	large matrix-small vector multiplication ($n \gg m$)	Large	Yes
$A_s \leftarrow M_s^{n \times m} N_s^{m \times k}$	large matrix-small matrix multiplication ($n \gg m > k$)	Large	Yes
$\|\|T\|\|$	matrix L2 norm which is the largest singular value of the matrix	Tiny	No
$EIG(T)$	symmetric eigen decomposition to output QDQ^T	Tiny	No

Table 2. Parallelization Choices. The last column (**P**) indicates whether the operation is parallelized in HEIGEN. Some operations are better to be run in parallel since the input size is very large, while others are better in a single machine since the input size is small and the overhead of parallel execution overshadows its decreased running time.

so on individual elements since both the input size and the key space decrease. This reduces the network traffic and sorting time in the MAPREDUCE Shuffle stage. As we will see in Section 5, the blocking decreases the running time by more than $4\times$.

4.5 Exploit Skewness: Matrix-Vector Multiplication

HEIGEN uses an adaptive method for sub-operations based on the size of the data. In this section, we describe how HEIGEN implements different matrix-vector multiplication algorithms by exploiting the skewness pattern of the data. There are two matrix-vector multiplication operations in Algorithm 1: the one with a large vector (line 3) and the other with a small vector (line 11).

The first matrix-vector operation multiplies a matrix with a large and dense vector, and thus it requires a two-stage standard MAPREDUCE algorithm by Kang et al. [9]. In the first stage, matrix elements and vector elements are joined and multiplied to make partial results which are added together to get the result vector in the second stage.

The other matrix-vector operation, however, multiplies with a small vector. HEIGEN uses the fact that the small vector can fit in a machine's main memory, and distributes the small vector to all the mappers using the distributed cache functionality of HADOOP. The advantage of the small vector being available in mappers is that joining edge elements and vector elements can be done inside the mapper, and thus the first stage of the standard two-stage matrix-vector multiplication can be omitted. In this one-stage algorithm the mapper joins matrix elements and vector elements to make partial results, and the reducer adds up the partial results. The pseudo code of this algorithm, which we call CBMV(Cache-Based Matrix-Vector multiplication), is shown in Algorithm 2. We want to emphasize that this operation cannot be performed when the vector is large, as is the case in the first matrix-vector multiplication. The CBMV is faster than the standard method by $57\times$ as described in Section 5.

4.6 Exploiting Skewness: Matrix-Matrix Multiplication

Skewness can also be exploited to efficiently perform matrix-matrix multiplication (line 26 of Algorithm 1). In general, matrix-matrix multiplication is very expensive. A standard, yet naive, way of multiplying two matrices A and B in MAPREDUCE is to multiply $A[:, i]$ and $B[i, :]$ for each column i of A and sum the resulting matrices. This

Algorithm 2: CBMV(Cache-Based Matrix-Vector Multiplication) for HEIGEN

Input: Matrix $M = \{(id_{src}, (id_{dst}, mval))\}$, Vector $x = \{(id, vval)\}$
Output: Result vector y

```
Stage1-Map(key k, value v, Vector x) // Multiply matrix elements and the vector x
id_src ← k;
(id_dst, mval) ← v;
Output(id_src, (mval × x[id_dst])); // Multiply and output partial results

Stage1-Reduce(key k, values V[]) // Sum up partial results
sum ← 0;
for v ∈ V do
    sum ← sum + v;
end for
Output(k, sum);
```

algorithm, which we call MM(direct Matrix-Matrix multiplication), is very inefficient since it generates huge matrices and sums them up many times. Fortunately, when one of the matrices is very small, we can utilize the skewness to make an efficient MAPREDUCE algorithm. This is exactly the case in HEIGEN; the first matrix is very large, and the second is very small. The main idea is to distribute the second matrix by the distributed cache functionality in HADOOP, and multiply each element of the first matrix with the corresponding rows of the second matrix. We call the resulting algorithm Cache-Based Matrix-Matrix multiplication, or CBMM. There are other alternatives to matrix-matrix multiplication: one can decompose the second matrix into column vectors and iteratively multiply the first matrix with each of these vectors. We call the algorithms, introduced in Section 4.5, Iterative matrix-vector multiplications (IMV) and Cache-based iterative matrix-vector multiplications (CBMV). The difference between CBMV and IMV is that CBMV uses cache-based operations while IMV does not. As we will see in Section 5, the best method, CBMM, is faster than naive methods by 76×.

4.7 Analysis

We analyze the time and the space complexity of HEIGEN. In the lemmas below, m is the number of iterations, $|V|$ and $|E|$ are the number of nodes and edges, and M is the number of machines.

Lemma 1 (Time Complexity). HEIGEN *takes* $O(m\frac{|V|+|E|}{M}log\frac{|V|+|E|}{M})$ *time.*

Proof. (Sketch) The running time of one iteration of HEIGEN is dominated by the matrix-large vector multiplication whose running time is $O(m\frac{|V|+|E|}{M}log\frac{|V|+|E|}{M})$. □

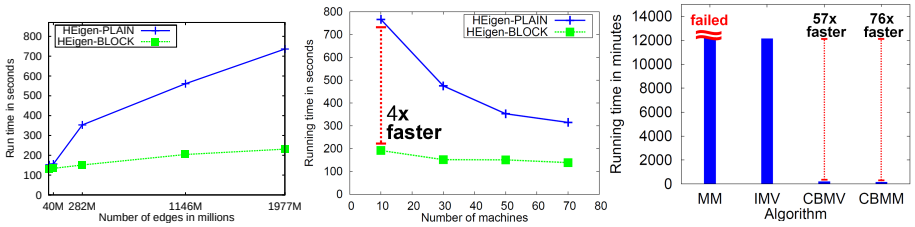
Lemma 2 (Space Complexity). HEIGEN *requires* $O(|V| + |E|)$ *space.*

Proof. (Sketch) The maximum storage is required at the intermediate output of the two-stage matrix-vector multiplication where $O(|V| + |E|)$ space is needed. □

5 Performance

In this section, we present experimental results to answer the following questions:

– **Scalability:** How well does HEIGEN scale up?



(a) Time vs. # of edges (b) Time vs. # of machines (c) Time vs. algorithms

Fig. 4. (**a**) Running time vs. number of edges in 1 iteration of HEIGEN with 50 machines. Notice the near-linear running time proportional to the edges size. (**b**) Running time vs. number of machines in 1 iteration of HEIGEN. The running time decreases as number of machines increase. (**c**) Comparison of running time between different skewed matrix-matrix and matrix-vector multiplications. For matrix-matrix multiplication, our proposed CBMM outperforms naive methods by at least 76×. The slowest matrix-matrix multiplication algorithm(MM) even didn't finish and the job failed due to excessive data. For matrix-vector multiplication, our proposed CBMV is faster than the naive method by 57×.

– **Optimizations:** Which of our proposed methods give the best performance?

We perform experiments in the Yahoo! M45 HADOOP cluster with total 480 hosts, 1.5 petabytes of storage, and 3.5 terabytes of memory. We use HADOOP 0.20.1. The scalability experiments are performed using synthetic Kronecker graphs [12] since realistic graphs of any size can be easily generated.

5.1 Scalability

Figure 4(a,b) shows the scalability of HEIGEN-BLOCK, an implementation of HEIGEN that uses blocking, and HEIGEN-PLAIN, an implementation which does not. Notice that the running time is near-linear in the number of edges and machines. We also note that HEIGEN-BLOCK performs up to 4× faster when compared to HEIGEN-PLAIN.

5.2 Optimizations

Figure 4(c) shows the comparison of running time of the skewed matrix-matrix multiplication and the matrix-vector multiplication algorithms. We used 100 machines for YahooWeb data. For matrix-matrix multiplications, the best method is our proposed CBMM which is 76× faster than repeated naive matrix-vector multiplications (IMV). The slowest MM algorithm didn't even finish, and failed due to heavy amounts of data. For matrix-vector multiplications, our proposed CBMV is faster than the naive method(IMV) by 48×.

6 Related Works

The related works form two groups, large-scale eigensolvers and MAPREDUCE/HADOOP.

Large-scale Eigensolvers: There are many parallel eigensolvers for large matrices: the work by Zhao et al. [21], HPEC [7], PLANO [20], PREPACK [15], SCALABLE [4], PLAYBACK [3] are several examples. All of them are based on MPI with message passing, which has difficulty in dealing with billion-scale graphs. The maximum order of matrices analyzed with these tools is less than 1 million [20] [16], which

is far from web-scale data. Very recently(March 2010), the Mahout project [2] provides SVD on top of HADOOP. Due to insufficient documentation, we were not able to find the input format and run a head-to-head comparison. But, reading the source code, we discovered that Mahout suffers from two major issues: (a) it assumes that the vector (b, with n=O(billion) entries) fits in the memory of a single machine, and (b) it implements the full re-orthogonalization which is inefficient.

MapReduce and Hadoop: MAPREDUCE is a parallel programming framework for processing web-scale data. MAPREDUCE has two major advantages: (a) it handles data distribution, replication, and load balancing automatically, and furthermore (b) it uses familiar concepts from functional programming. The programmer needs to provide only the *map* and the *reduce* functions. The general framework is as follows [11]: The *map* stage processes input and outputs (key, value) pairs. The *shuffling* stage sorts the map output and distributes them to reducers. Finally, the *reduce* stage processes the values with the same key and outputs the final result. HADOOP [1] is the open source implementation of MAPREDUCE. It also provides a distributed file system (HDFS) and data processing tools such as PIG [13] and Hive . Due to its extreme scalability and ease of use, HADOOP is widely used for large scale data mining [9, 8] .

7 Conclusion

In this paper we discovered patterns in real-world, billion-scale graphs. This was possible by using HEIGEN, our proposed eigensolver for the spectral analysis of very large-scale graphs. The main contributions are the following:

- **Effectiveness:** We analyze spectral properties of real world graphs, including Twitter and one of the largest public Web graphs. We report patterns that can be used for anomaly detection and find tightly-knit communities.
- **Careful Design:** We carefully design HEIGEN to selectively parallelize operations based on how they are most effectively performed.
- **Scalability:** We implement and evaluate a billion-scale eigensolver. Experimentation shows that HEIGEN is accurate and scales linearly with the number of edges.

Future research directions include extending the analysis and the algorithms for multi-dimensional matrices, or tensors [10].

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants No. IIS-0705359, IIS0808661, IIS-0910453, and CCF-1019104, by the Defense Threat Reduction Agency under contract No. HDTRA1-10-1-0120, and by the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053. This work is also partially supported by an IBM Faculty Award, and the Gordon and Betty Moore Foundation, in the eScience project. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or

the U.S. Government or other funding parties. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. Brendan Meeder is also supported by a NSF Graduate Research Fellowship and funding from the Fine Foundation, Sloan Foundation, and Microsoft.

References

[1] Hadoop information. http://hadoop.apache.org/.
[2] Mahout information. http://lucene.apache.org/mahout/.
[3] P. Alpatov, G. Baker, C. Edward, J. Gunnels, G. Morrow, J. Overfelt, R. van de Gejin, and Y.-J. Wu. Plapack: Parallel linear algebra package - design overview. *SC97*, 1997.
[4] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, and I. Dhillon. Scalapack users's guide. *SIAM*, 1997.
[5] L. C. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bur. Stand.*, 1950.
[6] J. W. Demmel. Applied numerical linear algebra. *SIAM*, 1997.
[7] M. R. Guarracino, F. Perla, and P. Zanetti. A parallel block lanczos algorithm and its implementation for the evaluation of some eigenvalues of large sparse symmetric matrices on multicomputers. *Int. J. Appl. Math. Comput. Sci.*, 2006.
[8] U. Kang, D. H. Chau, and C. Faloutsos. Mining large graphs: Algorithms, inference, and discoveries. *IEEE International Conference on Data Engineering*, 2011.
[9] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM*, 2009.
[10] T. G. Kolda and J. Sun. Scalable tensor decompsitions for multi-aspect data mining. *ICDM*, 2008.
[11] R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70:1–30, 2008.
[12] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *PKDD*, 2005.
[13] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, 2008.
[14] B. A. Prakash, M. Seshadri, A. Sridharan, S. Machiraju, and C. Faloutsos. Eigenspokes: Surprising patterns and community structure in large graphs. *PAKDD*, 2010.
[15] J. L. R.B., S. D.C., and Y. C. Arpack user's guide: Solution of large-scale eigenvalue problems with implicitly restarted arnoldi methods. *SIAM*, 1998.
[16] Y. Song, W. Chen, H. Bai, C. Lin, and E. Chang. Parallel spectral clustering. In *ECML*, 2008.
[17] L. N. Trefethen and D. B. III. Numerical linear algebra. *SIAM*, 1997.
[18] C. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *ICDM*, 2008.
[19] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. *KDD*, 2009.
[20] K. Wu and H. Simon. A parallel lanczos method for symmetric generalized eigenvalue problems. *Computing and Visualization in Science*, 1999.
[21] Y. Zhao, X. Chi, and Q. Cheng. An implementation of parallel eigenvalue computation using dual-level hybrid parallelism. *Lecture Notes in Computer Science*, 2007.