PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations

U Kang
SCS, Carnegie Mellon University
ukang@cs.cmu.edu

Charalampos E. Tsourakakis
SCS, Carnegie Mellon University
ctsourak@cs.cmu.edu

Christos Faloutsos
SCS, Carnegie Mellon University
christos@cs.cmu.edu

***Abstract*—In this paper, we describe PEGASUS, an open source Peta Graph Mining library which performs typical graph mining tasks such as computing the diameter of the graph, computing the radius of each node and finding the connected components. As the size of graphs reaches several Giga-, Tera- or Peta-bytes, the necessity for such a library grows too. To the best of our knowledge, PEGASUS is the first such library, implemented on the top of the HADOOP platform, the open source version of MAPREDUCE.**

Many graph mining operations (PageRank, spectral clustering, diameter estimation, connected components etc.) are essentially a repeated matrix-vector multiplication. In this paper we describe a very important primitive for PEGASUS, called GIM-V (Generalized Iterated Matrix-Vector multiplication). GIM-V is highly optimized, achieving (a) good scale-up on the number of available machines (b) linear running time on the number of edges, and (c) more than *5 times* faster performance over the non-optimized version of GIM-V.

Our experiments ran on M45, one of the top 50 supercomputers in the world. We report our findings on several real graphs, including one of the largest publicly available Web Graphs, thanks to Yahoo!, with $\approx$ 6,7 billion edges.

***Keywords*-PEGASUS; graph mining; hadoop**

I. INTRODUCTION

Graphs are ubiquitous: computer networks, social networks, mobile call networks, the World Wide Web [1], protein regulation networks to name a few.

The large volume of available data, the low cost of storage and the stunning success of online social networks and web2.0 applications all lead to graphs of unprecedented size. Typical graph mining algorithms silently assume that the graph fits in the memory of a typical workstation, or at least on a single disk; the above graphs violate these assumptions, spanning multiple Giga-bytes, and heading to Tera- and Peta-bytes of data.

A promising tool is parallelism, and specifically MAPREDUCE [2] and its open source version, HADOOP. Based on HADOOP, here we describe PEGASUS, a graph mining package for handling graphs with *billions* of nodes and edges. The PEGASUS code and several dataset are at **http://www.cs.cmu.edu/∼pegasus**. The contributions are the following:

1) Unification of seemingly different graph mining tasks, via a generalization of matrix-vector multiplication (GIM-V).
2) The careful implementation of GIM-V, with several optimizations, and several graph mining operations (PageRank, Random Walk with Restart(RWR), diameter estimation, and connected components). Moreover, the method is linear on the number of edges, and scales up well with the number of available machines.
3) Performance analysis, pinpointing the most successful combination of optimizations, which lead to up to *5 times* better speed than naive implementation.
4) Analysis of large, real graphs, including one of the largest publicly available graph that was ever analyzed, Yahoo's web graph.

The rest of the paper is organized as follows. Section II presents the related work. Section III describes our framework and explains several graph mining algorithms. Section IV discusses optimizations that allow us to achieve significantly faster performance in practice. In Section V we present timing results and Section VI our findings in real world, large scale graphs. We conclude in Section VII.

II. BACKGROUND AND RELATED WORK

The related work forms two groups, graph mining, and HADOOP.

Large-Scale Graph Mining.: There are a huge number of graph mining algorithms, computing communities (eg., [3], DENGRAPH [4], METIS [5]), subgraph discovery(e.g., GraphSig [6], [7], [8], [9], gPrune [10], gApprox [11], gSpan [12], Subdue [13], HSIGRAM/VSIGRAM [14], ADI [15], CSV [16]), finding important nodes (e.g., PageRank [17] and HITS [18]), computing the number of triangles [19], [20], computing the diameter [21], topic detection [22], attack detection [23], with too-many-to-list alternatives for each of the above tasks. Most of the previous algorithms do not scale, at least directly, to several millions and billions of nodes and edges.

For connected components, there are several algorithms, using Breadth-First Search, Depth-First-Search, "propagation" ([24], [25], [26]), or "contraction" [27] . These works rely on a shared memory model which limits their ability to handle large, disk-resident graphs.

MapReduce and Hadoop.: MAPREDUCE is a programming framework [2] [28] for processing huge amounts of unstructured data in a massively parallel way. MAPREDUCE has two major advantages: (a) the programmer is oblivious

of the details of the data distribution, replication, load balancing etc. and furthermore (b) the programming concept is familiar, i.e., the concept of functional programming. Briefly, the programmer needs to provide only two functions, a *map* and a *reduce*. The typical framework is as follows [29]: (a) the *map* stage sequentially passes over the input file and outputs (key, value) pairs; (b) the *shuffling* stage groups of all values by key, (c) the *reduce* stage processes the values with the same key and outputs the final result.

HADOOP is the open source implementation of MAPREDUCE. HADOOP provides the Distributed File System (HDFS) [30] and PIG, a high level language for data analysis [31]. Due to its power, simplicity and the fact that building a small cluster is relatively cheap, HADOOP is a very promising tool for large scale graph mining applications, something already reflected in academia, see [32]. In addition to PIG, there are several high-level language and environments for advanced MAPREDUCE-like systems, including SCOPE [33], Sawzall [34], and Sphere [35].

III. PROPOSED METHOD

How can we quickly find connected components, diameter, PageRank, node proximities of very large graphs fast? We show that, even if they seem unrelated, eventually we can unify them using the `GIM-V` primitive, standing for Generalized Iterative Matrix-Vector multiplication, which we describe in the next.

A. Main Idea

`GIM-V`, or 'Generalized Iterative Matrix-Vector multiplication' is a generalization of normal matrix-vector multiplication. Suppose we have a n by n matrix M and a vector v of size n. Let $m_{i,j}$ denote the (i, j)-th element of M. Then the usual matrix-vector multiplication is

$$M \times v = v' \text{ where } v'_i = \sum_{j=1}^{n} m_{i,j} v_j.$$

There are three operations in the previous formula, which, if customized separately, will give a surprising number of useful graph mining algorithms:

1) `combine2`: multiply $m_{i,j}$ and v_j.
2) `combineAll`: sum n multiplication results for node i.
3) `assign`: overwrite previous value of v_i with new result to make v'_i.

In `GIM-V`, let's define the operator $\times_G$, where the three operations can be defined arbitrarily. Formally, we have:

$v' = M \times_G v$
where v'_i = `assign`(v_i,`combineAll`$_i(\{x_j \mid j = 1..n$, and $x_j =$`combine2`$(m_{i,j}, v_j)\})$).

The functions `combine2`(), `combineAll`(), and `assign`() have the following signatures (generalizing the product, sum and assignment, respectively, that the traditional matrix-vector multiplication requires):

1) `combine2`$(m_{i,j}, v_j)$: combine $m_{i,j}$ and v_j.
2) `combineAll`$_i(x_1, ..., x_n)$: combine all the results from `combine2`() for node i.
3) `assign`(v_i, v_{new}) : decide how to update v_i with v_{new}.

The 'Iterative' in the name of `GIM-V` denotes that we apply the $\times_G$ operation until an algorithm-specific convergence criterion is met. As we will see in a moment, by customizing these operations, we can obtain different, useful algorithms including PageRank, Random Walk with Restart, connected components, and diameter estimation. But first we want to highlight the strong connection of `GIM-V` with SQL: When `combineAll`$_i$() and `assign`() can be implemented by user defined functions, the operator $\times_G$ can be expressed concisely in terms of SQL. This viewpoint is important when we implement `GIM-V` in large scale parallel processing platforms, including HADOOP, if they can be customized to support several SQL primitives including JOIN and GROUP BY. Suppose we have an `edge` table `E(sid, did, val)` and a `vector` table `V(id, val)`, corresponding to a matrix and a vector, respectively. Then, $\times_G$ corresponds to the following SQL statement - we assume that we have (built-in or user-defined) functions `combineAll`$_i$() and `combine2`()) and we also assume that the resulting table/vector will be fed into the `assign`() function (omitted, for clarity):

```
SELECT E.sid, combineAll_{E.sid}(combine2(E.val,V.val))
  FROM E, V
  WHERE E.did=V.id
  GROUP BY E.sid
```

In the following sections we show how we can customize `GIM-V`, to handle important graph mining operations including PageRank, Random Walk with Restart, diameter estimation, and connected components.

B. GIM-V and PageRank

Our first application of `GIM-V` is PageRank, a famous algorithm that was used by Google to calculate relative importance of web pages [17]. The PageRank vector p of n web pages satisfies the following eigenvector equation:

$$p = (cE^T + (1 - c)U)p$$

where c is a damping factor (usually set to 0.85), E is the row-normalized adjacency matrix (source, destination), and U is a matrix with all elements set to $1/n$.

To calculate the eigenvector p we can use the power method, which multiplies an initial vector with the matrix, several times. We initialize the current PageRank vector p^{cur} and set all its elements to $1/n$. Then the next PageRank p^{next} is calculated by $p^{next} = (cE^T + (1 - c)U)p^{cur}$. We continue to do the multiplication until p converges.

PageRank is a direct application of `GIM-V`. In this view, we first construct a matrix M by column-normalize E^T such that every column of M sum to 1. Then the next

PageRank is calculated by $p^{next} = M \times_G p^{cur}$ where the three operations are defined as follows:

1) `combine2`$(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$
2) `combineAll`$_i(x_1, ..., x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j$
3) `assign`$(v_i, v_{new}) = v_{new}$

C. GIM-V and Random Walk with Restart

Random Walk with Restart(RWR) is an algorithm to measure the proximity of nodes in graph [36]. In RWR, the proximity vector r_k from node k satisfies the equation:

$$r_k = cMr_k + (1-c)e_k$$

where e_k is a n-vector whose k^{th} element is 1, and every other elements are 0. c is a restart probability parameter which is typically set to 0.85 [36]. M is a column-normalized and transposed adjacency matrix, as in Section III-B. In GIM-V, RWR is formulated by $r_k^{next} = M \times_G r_k^{cur}$ where the three operations are defined as follows ($I(x)$ is 1 if x is true, and 0 otherwise.):

1) `combine2`$(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$
2) `combineAll`$_i(x_1, ..., x_n) = (1-c)I(i \neq k) + \sum_{j=1}^{n} x_j$
3) `assign`$(v_i, v_{new}) = v_{new}$

D. GIM-V and Diameter Estimation

HADI [21] is an algorithm to estimate the diameter and radius of large graphs. The diameter of a graph is the maximum of the length of the shortest path between every pair of nodes. The radius of a node v_i is the number of hops that we need to reach the farthest-away node from v_i. The main idea of HADI is as follows. For each node v_i in the graph, we maintain the number of neighbors reachable from v_i within h hops. As h increases, the number of neighbors increases until h reaches it maximum value. The diameter is h where the number of neighbors within $h+1$ does not increase for every node. For further details and optimizations, see [21].

The main operation of HADI is updating the number of neighbors as h increases. Specifically, the number of neighbors within hop h reachable from node v_i is encoded in a probabilistic bitstring b_i^h which is updated as follows:

b_i^{h+1} = b_i^h BITWISE-OR $\{b_k^h \mid (i,k) \in E\}$

In GIM-V, the bitstring update of HADI is represented by

$$b^{h+1} = M \times_G b^h$$

where M is an adjacency matrix, b^{h+1} is a vector of length n which is updated by
b_i^{h+1} =`assign`(b_i^h,`combineAll`$_i$($\{x_j \mid j = 1..n$, and x_j =`combine2`$(m_{i,j}, b_j^h)\}$)),
and the three operations are defined as follows:

1) `combine2`$(m_{i,j}, v_j) = m_{i,j} \times v_j$.
2) `combineAll`$_i(x_1, ..., x_n)$ = BITWISE-OR$\{x_j \mid j = 1..n\}$
3) `assign`(v_i, v_{new}) = BITWISE-OR(v_i, v_{new}).

The $\times_G$ operation is run iteratively until the bitstring for all the nodes do not change.

E. GIM-V and Connected Components

We propose HCC, a new algorithm for finding connected components in large graphs. Like HADI, HCC is an application of GIM-V with custom functions. The main idea is as follows. For every node v_i in the graph, we maintain a component id c_i^h which is the minimum node id within h hops from v_i. Initially, c_i^h of v_i is set to its own node id: that is, $c_i^0 = i$. For each iteration, each node sends its current c_i^h to its neighbors. Then c_i^{h+1}, component id of v_i at the next step, is set to the minimum value among its current component id and the received component ids from its neighbors. The crucial observation is that this communication between neighbors can be formulated in GIM-V as follows:

$$c^{h+1} = M \times_G c^h$$

where M is an adjacency matrix, c^{h+1} is a vector of length n which is updated by
c_i^{h+1} =`assign`(c_i^h,`combineAll`$_i$($\{x_j \mid j = 1..n$, and x_j =`combine2`$(m_{i,j}, c_j^h)\}$)),
and the three operations are defined as follows:

1) `combine2`$(m_{i,j}, v_j) = m_{i,j} \times v_j$.
2) `combineAll`$_i(x_1, ..., x_n)$ = MIN$\{x_j \mid j = 1..n\}$
3) `assign`(v_i, v_{new}) = MIN(v_i, v_{new}).

By repeating this process, component ids of nodes in a component are set to the minimum node id of the component. We iteratively do the multiplication until component ids converge. The upper bound of the number of iterations in HCC are determined by the following theorem.

Theorem 1 (Upper bound of iterations in HCC*):* HCC requires maximum d iterations where d is the diameter of the graph.

Proof: The minimum node id is propagated to its neighbors at most d times. ■

Since the diameter of real graphs are relatively small, HCC completes after small number of iterations.

IV. FAST ALGORITHMS FOR GIM-V

How can we parallelize the algorithm presented in the previous section? In this section, we first describe naive HADOOP algorithms for GIM-V. After that we propose several faster methods for GIM-V.

A. GIM-V BASE: Naive Multiplication

GIM-V BASE is a two-stage algorithm whose pseudo code is in Algorithm 1 and 2. The inputs are an edge file and a vector file. Each line of the edge file contains one $(id_{src}, id_{dst}, mval)$ which corresponds to a non-zero cell in the adjacency matrix M. Similarly, each line of the vector file contains one $(id, vval)$ which corresponds to an element in the vector V. `Stage1` performs `combine2` operation by combining columns of matrix(id_{dst} of M) with rows of vector(id of V). The output of `Stage1` are (key, value) pairs where key is the source node id of the

Algorithm 1: GIM-V BASE Stage 1.

Input : Matrix $M = \{(id_{src}, (id_{dst}, mval))\}$,
Vector $V = \{(id, vval)\}$
Output: Partial vector
$V' = \{(id_{src}, \texttt{combine2}(mval, vval)\}$

```
1  Stage1-Map(Key k, Value v);
2  begin
3      if (k, v) is of type V then
4          Output(k, v);                          // (k: id, v: vval)
5      else if (k, v) is of type M then
6          (id_dst, mval) ← v;
7          Output(id_dst, (k, mval));             // (k: id_src)
8  end
9  Stage1-Reduce(Key k, Value v[1..m]);
10 begin
11     saved_kv ←[ ];
12     saved_v ←[ ];
13     foreach v ∈ v[1..m] do
14         if (k, v) is of type V then
15             saved_v ← v;
16             Output(k, ("self", saved_v));
17         else if (k, v) is of type M then
18             Add v to saved_kv      // (v: (id_src, mval))
19     end
20     foreach (id'_src, mval') ∈ saved_kv do
21         Output(id'_src, ("others",combine2(mval', saved_v)));
22     end
23 end
```

Algorithm 2: GIM-V BASE Stage 2.

Input : Partial vector $V' = \{(id_{src}, vval')\}$
Output: Result Vector $V = \{(id_{src}, vval)\}$

```
1  Stage2-Map(Key k, Value v);
2  begin
3      Output(k, v);
4  end
5  Stage2-Reduce(Key k, Value v[1..m]);
6  begin
7      others_v ←[ ];
8      self_v ←[ ];
9      foreach v ∈ v[1..m] do
10         (tag, v') ← v;
11         if tag == "same" then
12             self_v ← v';
13         else if tag == "others" then
14             Add v' to others_v;
15     end
16     Output(k,assign(self_v,combineAll_k(others_v)));
17 end
```

matrix(id_{src} of M) and the value is the partially combined result(combine2($mval, vval$)). This output of Stage1 becomes the input of Stage2. Stage2 combines all partial results from Stage1 and assigns the new vector to the old vector. The combineAll$_i$() and assign() operations are done in line 16 of Stage2, where the "self" and "others" tags in line 16 and line 21 of Stage1 are used to make v_i and v_{new} of GIM-V, respectively.

This two-stage algorithm is run iteratively until application-specific convergence criterion is met. In Algorithm 1 and 2, Output(k, v) means to output data with the key k and the value v.

B. GIM-V *BL: Block Multiplication*

GIM-V BL is a fast algorithm for GIM-V which is based on block multiplication. The main idea is to group elements of the input matrix into blocks or submatrices of size b by b. Also we group elements of input vectors into blocks of length b. Here the grouping means we put all the elements in a group into one line of input file. Each block contains only non-zero elements of the matrix or vector. The format of a matrix block with k nonzero elements is ($row_{block}, col_{block}, row_{elem_1}, col_{elem_1}, mval_{elem_1}, ...,$ $row_{elem_k}, col_{elem_k}, mval_{elem_k}$). Similarly, the format of a vector block with k nonzero elements is ($id_{block}, id_{elem_1}, vval_{elem_1}, ..., id_{elem_k}, vval_{elem_k}$). Only blocks with at least one nonzero elements are saved to disk. This block encoding forces nearby edges in the adjacency matrix to be closely located; it is different from HADOOP's default behavior which do not guarantee co-locating them. After grouping, GIM-V is performed on blocks, not on individual elements. GIM-V BL is illustrated in Figure 1.

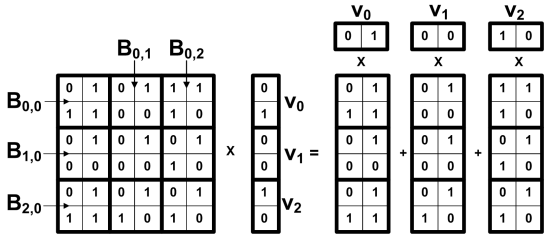


Figure 1. GIM-V BL using 2 x 2 blocks. $B_{i,j}$ represents a matrix block, and v_i represents a vector block. The matrix and vector are joined block-wise, not element-wise.

In our experiment at Section V, GIM-V BL is more than 5 times faster than GIM-V BASE. There are two main reasons for this speed-up.

- **Sorting Time** Block encoding decrease the number of items to sort in the shuffling stage of HADOOP. We observed that the main bottleneck of programs in HADOOP is its shuffling stage where network transfer, sorting, and disk I/O happens. By encoding to blocks of width b, the number of lines in the matrix and the vector file decreases to $1/b^2$ and $1/b$ times of their original size, respectively for full matrices and vectors.

- **Compression** The size of the data decreases significantly by converting edges and vectors to block format. The reason is that in GIM-V BASE we need 4×2 bytes to save each (srcid, dstid) pair since we need 4 bytes to save a node id using Integer. However in GIM-V BL we can specify each *block* using a block row id and a block column id with two 4-byte Integers, and refer to elements inside the block using $2 \times logb$ bits. This is possible because we can use log$b$ bits to refer to a row or column inside a block. By this block method we decreased the edge file size(e.g., more than 50% for YahooWeb graph in Section V).

C. GIM-V *CL: Clustered Edges*

When we use block multiplication, another advantage is that we can benefit from clustered edges. As can be seen from Figure 2, we can use smaller number of blocks if input edge files are clustered. Clustered edges can be built if we can use heuristics in data preprocessing stage so that edges are clustered, or by co-clustering (e.g., see [32]). The preprocessing for edge clustering need to be done only once; however, they can be used by every iteration of various application of GIM-V. So we have two variants of GIM-V: GIM-V CL, which is GIM-V BASE with clustered edges, and GIM-V BL-CL, which is GIM-V BL with clustered edges. Be aware that clustered edges is only useful when combined with block encoding. If every element is treated separately, then clustered edges don't help anything for the performance of GIM-V.

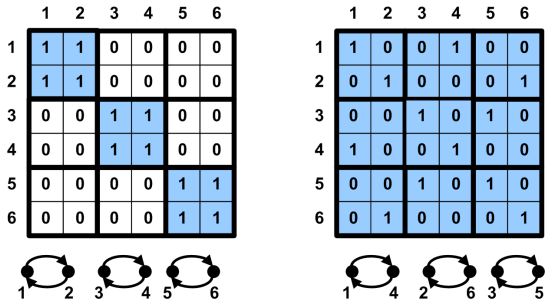


Figure 2. Clustered vs. non-clustered graphs with same topology. The edges are grouped into 2 by 2 blocks. The left graph uses only 3 blocks while the right graph uses 9 blocks.

D. GIM-V *DI: Diagonal Block Iteration*

As mentioned in Section IV-B, the main bottleneck of GIM-V is its shuffling and disk I/O steps. Since GIM-V iteratively runs Algorithm 1 and 2, and each Stage requires disk IO and shuffling, we could decrease running time if we decrease the number of iterations.

In HCC, it is possible to decrease the number of iterations. The main idea is to multiply diagonal matrix blocks and corresponding vector blocks as much as possible in one iteration. Remember that multiplying a matrix and a vector corresponds to passing node ids to one step neighbors in HCC. By multiplying diagonal blocks and vectors until the contents of the vectors do not change in one iteration, we can pass node ids to neighbors located more than one step away. This is illustrated in Figure 3.

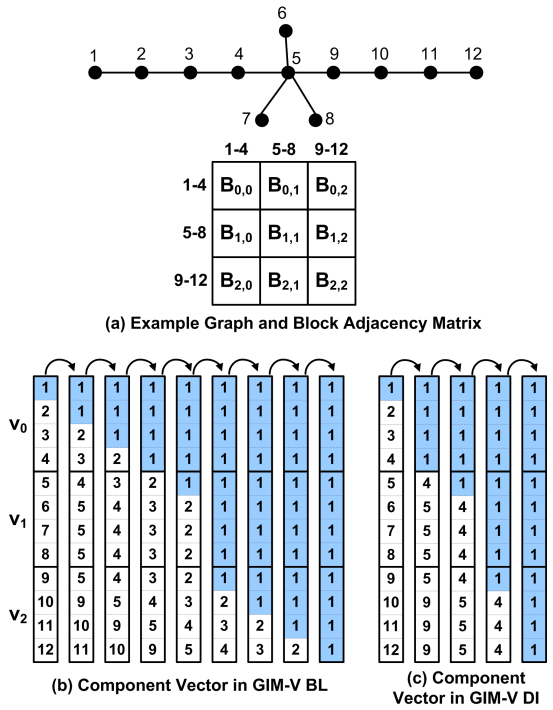


Figure 3. Propagation of component id(=1) when block width is 4. Each element in the adjacency matrix of (a) represents a 4 by 4 block; each column in (b) and (c) represents the vector after each iteration. GIM-V DL finishes in 4 iterations while GIM-V BL requires 8 iterations.

We see that in Figure 3 (c) we multiply $B_{i,i}$ with v_i several times until v_i do not change in one iteration. For example in the first iteration v_0 changed from {1,2,3,4} to {1,1,1,1} since it is multiplied to $B_{0,0}$ four times. GIM-V DI is especially useful in graphs with long chains.

The upper bound of the number of iterations in HCC DI with chain graphs are determined by the following theorem.

Theorem 2 (Upper bound of iterations in HCC *DI):* In a chain graph with length m, it takes maximum $2 * \lceil m/b \rceil - 1$ iterations in HCC DI with block size b.

Proof: The worst case happens when the minimum node id is in the beginning of the chain. It requires 2 iterations(one for propagating the minimum node id inside the block, another for passing it to the next block) for the minimum node id to move to an adjacent block. Since the farthest block is $\lceil m/b \rceil - 1$ steps away, we need $2 * (\lceil m/b \rceil - 1)$ iterations. When the minimum node id reached the farthest away block, GIM-V DI requires one more iteration to propagate the minimum node id inside the last block. Therefore, we need $2 * (\lceil m/b \rceil - 1) + 1 = 2 * \lceil m/b \rceil - 1$ iterations. ■

E. Analysis

We analyze the time and space complexity of GIM-V. In the theorems below, M is the number of machines.

Theorem 3 (Time Complexity of GIM-V*):* One iteration of GIM-V takes $O(\frac{V+E}{M} log \frac{V+E}{M})$ time.

Proof: Assuming uniformity, mappers and reducers of Stage1 and Stage2 receives $O(\frac{V+E}{M})$ records per machine. The running time is dominated by the sorting time for $\frac{V+E}{M}$ records, which is $O(\frac{V+E}{M} log \frac{V+E}{M})$. ■

Theorem 4 (Space Complexity of GIM-V*):* GIM-V requires $O(V + E)$ space.

Proof: We assume the value of the elements of the input vector v is constant. Then the theorem is proved by noticing that the maximum storage is required at the output of Stage1 mappers which requires $O(V + E)$ space up to a constant. ■

V. Performance and Scalability

We do experiments to answer following questions:

- Q1 How does GIM-V scale up?
- Q2 Which of the proposed optimizations(block multiplication, clustered edges, and diagonal block iteration) gives the highest performance gains?

The graphs we used in our experiments at Section V and VI are described in Table I [1] .

Name	Nodes	Edges	Description
YahooWeb	1,413 M	6,636 M	WWW pages in 2002
LinkedIn	7.5 M	58 M	person-person in 2006
	4.4 M	27 M	person-person in 2005
	1.6 M	6.8 M	person-person in 2004
	85 K	230 K	person-person in 2003
Wikipedia	3.5 M	42 M	doc-doc in 2007/02
	3 M	35 M	doc-doc in 2006/09
	1.6 M	18.5 M	doc-doc in 2005/11
Kronecker	177 K	1,977 M	synthetic
	120 K	1,145 M	synthetic
	59 K	282 M	synthetic
	19 K	40 M	synthetic
DBLP	471 K	112 K	document-document
flickr	404 K	2.1 M	person-person
Epinions	75 K	508 K	who trusts whom

Table I
Order and size of networks.

We run PEGASUS in M45 HADOOP cluster by Yahoo! and our own cluster composed of 9 machines. M45 is one of the top 50 supercomputers in the world with 1.5 Pb total storage and 3.5 Tb memory. For the performance and scalability experiments, we used synthetic Kronecker graphs [37] since we can generate them with any size, and they are one of the most realistic graphs among synthetic graphs.

[1]Wikipedia: http://www.cise.ufl.edu/research/sparse/matrices/
Kronecker, DBLP: http://author's website/PEGASUS/
YahooWeb, LinkedIn: released under NDA.
flickr, Epinions, patent: not public data.

A. Results

We first show how the performance of our method changes as we add more machines. Figure 4 shows the running time and performance of GIM-V for PageRank with Kronecker graph of 282 million edges, and size 32 blocks if necessary.

In Figure 4 (a), for all of the methods the running time decreases as we add more machines. Note that clustered edges(GIM-V CL) didn't help performance unless it is combined with block encoding. When it is combined, however, it showed the best performance (GIM-V BL-CL).

In Figure 4 (b), we see that the relative performance of each method compared to GIM-V BASE method decreases as number of machines increases. With 3 machines (minimum number of machines which HADOOP distributed mode supports), the fastest method(GIM-V BL-CL) ran 5.27 times faster than GIM-V BASE. With 90 machines, GIM-V BL-CL ran 2.93 times faster than GIM-V BASE. This is expected since there are fixed component(JVM load time, disk I/O, network communication) which can not be optimized even if we add more machines.

Next we show how the performance of our methods changes as the input size grows. Figure 4 (c) shows the running time of GIM-V with different number of edges under 10 machines. As we can see, all of the methods scales linearly with the number of edges.

Finally, we compare the performance of GIM-V DI and GIM-V BL-CL for HCC in graphs with long chains. For this experiment we made a new graph whose diameter is 17, by adding a length 15 chain to the 282 million Kronecker graph which has diameter 2. As we see in Figure 5, GIM-V DI finished in 6 iteration while GIM-V BL-CL finished in 18 iteration. The running time of both methods for the first 6 iterations are nearly same. Therefore, the diagonal block iteration method decrease the number of iterations while not affecting the running time of each iteration much.

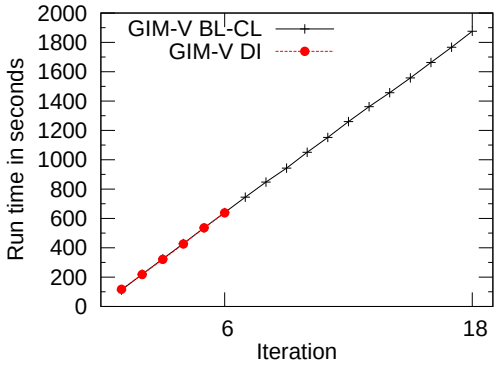
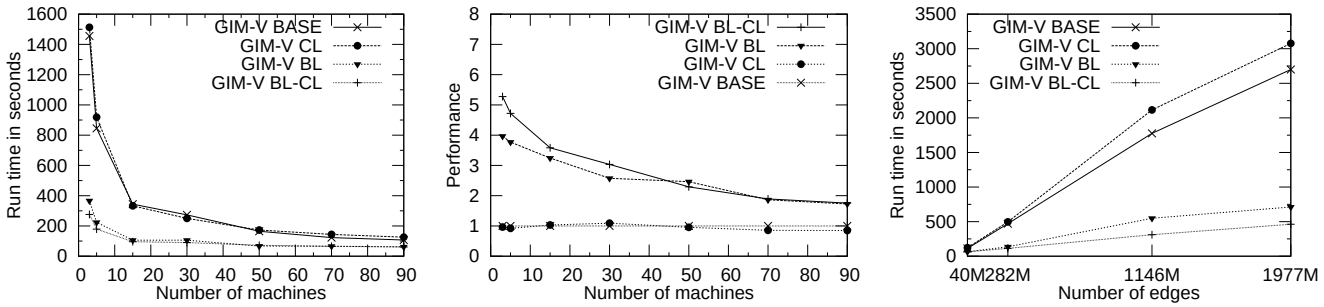


Figure 5. Comparison of GIM-V DI and GIM-V BL-CL for HCC. GIM-V DI finishes in 6 iterations while GIM-V BL-CL finishes in 18 iterations due to long chains.



(a) Running time vs. Machines (b) Performance vs. Machines (c) Running time vs. Edges

Figure 4. Scalability and Performance of GIM-V. (a) Running time decreases quickly as more machines are added. (b) The performance(=$1/running\ time$) of 'BL-CL' wins more than 5x (for n=3 machines) over the 'BASE'. (c) Every version of `GIM-V` shows linear scalability.

VI. `GIM-V` At Work

In this section we use PEGASUS for mining very large graphs. We analyze connected components, diameter, and PageRank of large real world graphs. We show that PEGASUS can be useful for finding patterns, outliers, and interesting observations.

A. Connected Components of Real Networks

We used the LinkedIn social network and Wikipedia page-linking-to-page network, along with the YahooWeb graph for connected component analysis. Figure 6 show the evolution of connected components of LinkedIn and Wikipedia data. Figure 7 show the distribution of connected components in the YahooWeb graph. We have following observations.

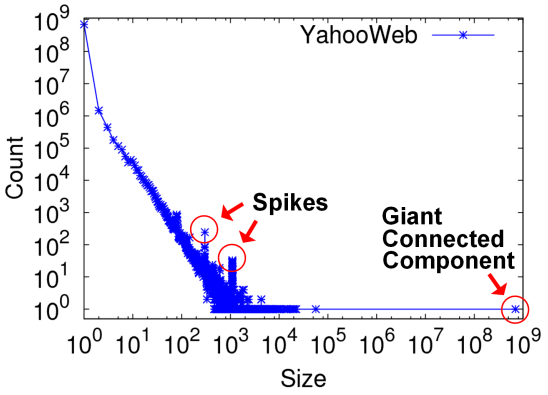


Figure 7. Connected Components of YahooWeb. Notice the two anomalous spikes which are far from the constant-slope tail.

Power Law Tails in Connected Components Distributions We observed power law relation of count and size of small connected components in Figure 6(a),(b) and Figure 7. This reflects that the connected components in real networks are formed by processes similar to Chinese Restaurant Process and Yule distribution [38].

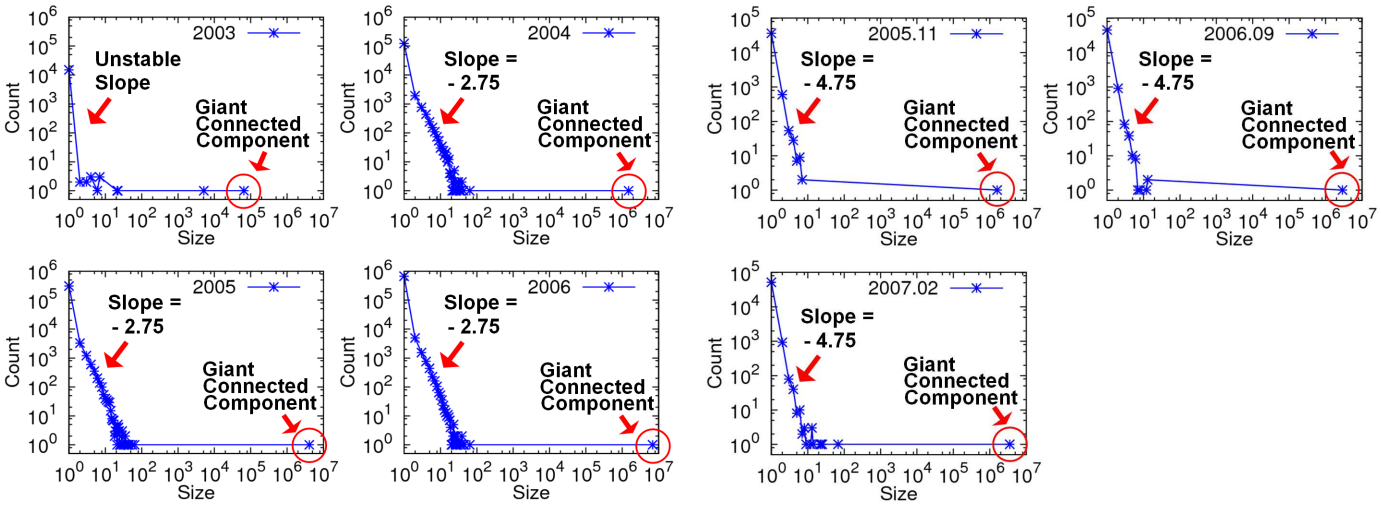
Stable Connected Components After Gelling Point In Figure 6(a), the distribution of connected components remain stable after a 'gelling' point[39] at year 2003.We can see that the slope of tail distribution do not change after year 2003. We observed the same phenomenon in Wikipedia graph in Figure 6 (b). The graph show stable tail slopes from the beginning, since the network were already mature in year 2005.

Absorbed Connected Components and Dunbar's number In Figure 6(a), we find two large connected components in year 2003. However it became merged in year 2004. The giant connected component keeps growing, while the second and the third largest connected components do not grow beyond size 100 until they are absorbed to the giant connected component in Figure 6 (a) and (b). This agrees with the observation[39] that the size of the second/third connected components remains constant or oscillates. Lastly, the maximum connected component size except the giant connected component in the LinkedIn graph agrees well with Dunbar's number[40], which says that the maximum community size in social networks is roughly 150.

Anomalous Connected Components In Figure 7, we found two outstanding spikes. In the first spike at size 300, more than half of the components have exactly the same structure and they were made from a domain selling company where each component represents a domain to be sold. The spike happened because the company *replicated* sites using the same template, and injected the disconnected components into WWW network. In the second spike at size 1101, more than 80 % of the components are porn sites disconnected from the giant connected component. By looking at the distribution plot of connected components, we could find interesting communities with special purposes which are disconnected from the rest of the Internet.

B. PageRanks of Real Networks

We analyzed PageRank of YahooWeb graph with PEGASUS. Figure 8 shows the distribution of PageRank of the graph. We observed that the PageRank follows a power law distribution with exponent 1.97, which is very close to the exponent 1.98 of the in-degree distribution of the same graph. Pandurangan et. al.[41] observed that the two exponent are same for 100,000 pages in Brown University



(a) Connected Components of LinkedIn (b) Connected Components of Wikipedia

Figure 6. The evolution of connected components. (a) The giant connected component grows for each year. However, the second largest connected component do not grow above Dunbar's number($\approx$ 150) and the slope of the tail remains constant after the gelling point at year 2003. (b) As in LinkedIn, notice the growth of giant connected component and the constant slope for tails.

domain. Our result is that the same observation holds true for *10,000 times* larger network with 1.4 *billion* pages snapshot of the Internet.

The top 3 highest PageRank sites at year 2002 are www.careerbank.com, access.adobe.com, and top100.rambler.ru. As expected, they have huge in-degrees (from $\approx$70K to $\approx$70M).

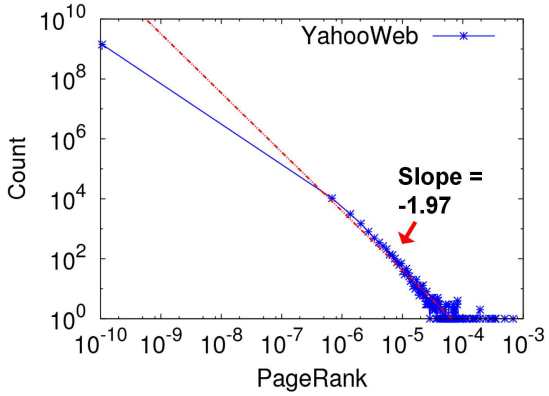


Figure 8. PageRank distribution of YahooWeb. The distribution follows power law with exponent 1.97.

C. Diameter of Real Network

We analyzed the diameter and radius of real networks with PEGASUS. Figure 9 shows the radius plot of real networks. We have following observations:

Small Diameter For all the graphs in Figure 9, the average diameter was less than 6.09. This means that the real world graphs are well connected.

Constant Diameter over Time For LinkedIn graph, the average diameter was in the range of 5.28 and 6.09. For Wikipedia graph, the average diameter was in the range of 4.76 and 4.99. Note that the diameter do not monotonically increase as network grows: they remain constant or shrinks over time.

Bimodal Structure of Radius Plot For every plot, we observe bimodal shape which reflects the structure of these real graphs. The graphs have one giant connected component where majority of nodes belong to, and many smaller connected components whose size follows power law. Therefore, the first mode is at radius zero which comes from one-node components; second mode(e.g., at radius 6 in Epinion) comes from the giant connected component.

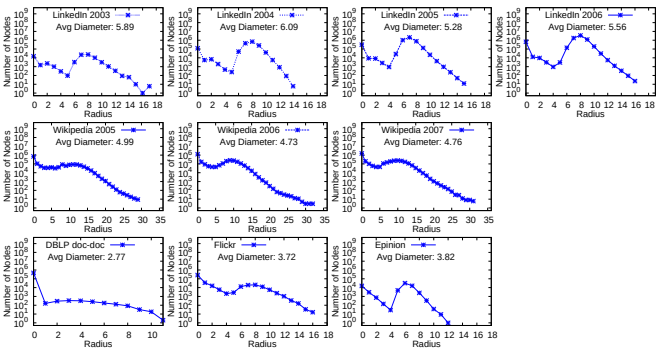


Figure 9. Radius of real graphs.
X axis: radius. Y axis: number of nodes.
(Row 1) LinkedIn from 2003 to 2006.
(Row 2) Wikipedia from 2005 to 2007.
(Row 3) DBLP, flickr, Epinion.

VII. Conclusions

In this paper we proposed PEGASUS, a graph mining package for very large graphs using the HADOOP architecture. The main contributions are followings:

- We identified the common, underlying primitive of several graph mining operations, and we showed that it is a generalized form of a matrix-vector multiplication. We call this operation Generalized Iterative Matrix-Vector multiplication and showed that it includes the diameter estimation, the PageRank estimation, RWR calculation, and finding connected-components, as special cases.
- Given its importance, we proposed several optimizations (block-multiplication, diagonal block iteration etc) and reported the winning combination, which achieves *5 times* faster performance to the naive implementation.
- We implemented PEGASUS and ran it on M45, one of the 50 largest supercomputers in the world (3.5 Tb memory, 1.5Pb disk storage). Using PEGASUS and our optimized Generalized Iterative Matrix-Vector multiplication variants, we analyzed real world graphs to reveal important patterns including power law tails, stability of connected components, and anomalous components. Our largest graph, "YahooWeb", spanned 120Gb, and is one of the largest publicly available graph that was ever studied.

Other open source libraries such as HAMA (Hadoop Matrix Algebra) [42] can benefit significantly from PEGASUS. One major research direction is to add to PEGASUS an eigensolver, which will compute the top k eigenvectors and eigenvalues of a matrix. Another directions includes tensor analysis on HADOOP ([43]), and inferences of graphical models in large scale.

Acknowledgment

The authors would like to thank YAHOO! for providing us with the web graph and access to the M45.

This material is based upon work supported by the National Science Foundation under Grants No. IIS-0705359 IIS0808661 and under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-404625), subcontracts B579447, B580840.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, or other funding parties.

References

[1] 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.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *OSDI*, 2004.

[3] J. Chen, O. R. Zaiane, and R. Goebel, "Detecting communities in social networks using max-min modularity," *SDM*, 2009.

[4] T. Falkowski, A. Barth, and M. Spiliopoulou, "Dengraph: A density-based community detection algorithm," *Web Intelligence*, 2007.

[5] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning for irregular graphs," *SIAM Review*, vol. 41, no. 2, 1999.

[6] S. Ranu and A. K. Singh, "Graphsig: A scalable approach to mining significant subgraphs in large graph databases," *ICDE*, 2009.

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

[8] P. Hintsanen and H. Toivonen, "Finding reliable subgraphs from large probabilistic graphs," *PKDD*, 2008.

[9] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," *ICDE*, 2008.

[10] F. Zhu, X. Yan, J. Han, and P. S. Yu, "gprune: A constraint pushing framework for graph pattern mining," *PAKDD*, 2007.

[11] C. Chen, X. Yan, F. Zhu, and J. Han, "gapprox: Mining frequent approximate patterns from a massive network," *ICDM*, 2007.

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

[13] N. S. Ketkar, L. B. Holder, and D. J. Cook, "Subdue: Compression-based frequent pattern discovery in graph data," *OSDM*, August 2005.

[14] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph," *SIAM Data Mining Conference*, 2004.

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

[16] N. Wang, S. Parthasarathy, K.-L. Tan, and A. K. H. Tung, "Csv: Visualizing and mining cohesive subgraph," *SIGMOD*, 2008.

[17] S. Brin and L. Page, "The anatomy of a large-scale hypertextual (web) search engine." in *WWW*, 1998.

[18] J. Kleinberg, "Authoritative sources in a hyperlinked environment," in *Proc. 9th ACM-SIAM SODA*, 1998.

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

[20] C. E. Tsourakakis, M. N. Kolountzakis, and G. L. Miller, "Approximate triangle counting," Apr 2009. [Online]. Available: http://arxiv.org/abs/0904.3761

[21] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec, "Hadi: Fast diameter estimation and mining in massive graphs with hadoop," *CMU-ML-08-117*, 2008.

[22] T. Qian, J. Srivastava, Z. Peng, and P. C. Sheu, "Simultaneouly finding fundamental articles and new topics

using a community tracking method," *PAKDD*, 2009.
[23] N. Shrivastava, A. Majumder, and R. Rastogi, "Mining (social) network graphs to detect random link attacks," *ICDE*, 2008.
[24] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, pp. 57–67, 1982.
[25] B. Awerbuch and Y. Shiloach, "New connectivity and msf algorithms for ultracomputer and pram," *ICPP*, 1983.
[26] D. Hirschberg, A. Chandra, and D. Sarwate, "Computing connected components on parallel computers," *Communications of the ACM*, vol. 22, no. 8, pp. 461–464, 1979.
[27] J. Greiner, "A comparison of parallel algorithms for connected components," *Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures*, June 1994.
[28] G. Aggarwal, M. Data, S. Rajagopalan, and M. Ruhl, "On the streaming model augmented with a sorting primitive," *Proceedings of FOCS*, 2004.
[29] R. Lämmel, "Google's mapreduce programming model – revisited," *Science of Computer Programming*, vol. 70, pp. 1–30, 2008.
[30] "Hadoop information," http://hadoop.apache.org/.
[31] 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, pp. 1099–1110.
[32] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce," *ICDM*, 2008.
[33] 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.
[34] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming Journal*, 2005.
[35] R. L. Grossman and Y. Gu, "Data mining using high performance data clouds: experimental studies using sector and sphere," *KDD*, 2008.
[36] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu, "Automatic multimedia cross-modal correlation discovery," *ACM SIGKDD*, Aug. 2004.
[37] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," *PKDD*, 2005.
[38] M. E. J. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary Physics*, no. 46, pp. 323–351, 2005.
[39] M. Mcglohon, L. Akoglu, and C. Faloutsos, "Weighted graphs and disconnected components: patterns and a generator," *KDD*, pp. 524–532, 2008.
[40] R. Dunbar, "Grooming, gossip, and the evolution of language," *Harvard Univ Press*, October 1998.
[41] G. Pandurangan, P. Raghavan, and E. Upfal, "Using pagerank to characterize web structure," *COCOON*, August 2002.
[42] "Hama website," http://incubator.apache.org/hama/.
[43] T. G. Kolda and J. Sun, "Scalable tensor decompsitions for multi-aspect data mining," *ICDM*, 2008.