Algorithms for Database Systems Seminar – Spring 2014

PEGASUS: A Peta-Scale Graph Mining System

Written By

Hany Abdelrahman

Supervised by:

Prof. Michael Böhlen

1) Introduction:

The large volume of available data and the growth of social networks has led to graphs of magnificent size. Typical graph algorithms assume that the data fits into a single machine. Nowadays, there are many graphs that spans multiple Tera-byte and Peta-bytes of data.

Pegasus begins with an intuition that many graph mining algorithms can be represented as a generalized matrix by vector product operation. It assumes that the algorithm to be implemented can be reduced to a matrix by vector product. Examples of algorithms which can be reduced to this form are PageRank, diameter estimation, connected components of a graph.

2) Proposed Method:

GIM-V is a generalization of the normal matrix-vector multiplication. Suppose we have an *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:

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

In GIM-V The matrix-vector multiplication can be represented by three operations:

1- `combine2`($m_{i,j}$, v_j): combine $m_{i,j}$ and v_j

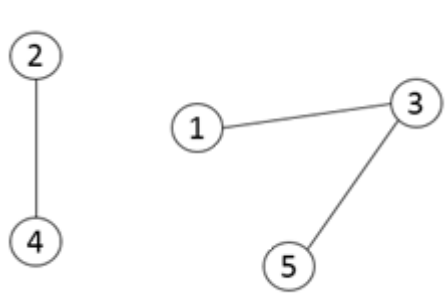
2- $\texttt{combineAll}_{\mathbf{i}}(x_1, .., x_n)$: combine all the results from `combine2` for element *i.*

3- `assign`(v_i, v_{new}) : decide how to update v_i with v_{new} .

In GIM-V the operator $\times_{\mathbf{G}}$ is defined using the three operations defined above. The word iterative in the name of GIM-V algorithm denotes that the operator $\times_{\mathbf{G}}$ is applied iteratively until a specific convergence criteria is met. Also, by customizing the methods `combine2`, `combineAll` and `assign`, we can obtain different useful algorithms as described below in more details.

2.1) GIM-V and Connected Components:

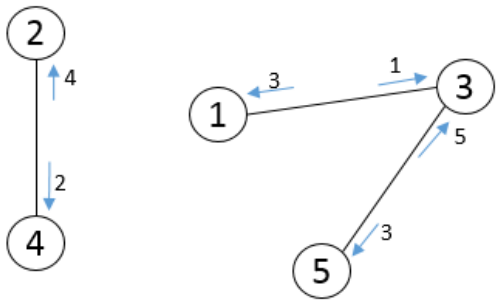
An iterative algorithm based on GIM-V called HCC is proposed. First we show an example of how one can compute connected components of a graph by a simple passing of messages among nodes.



$$\begin{pmatrix}1\\2\\3\\4\\5\end{pmatrix} \quad C^0$$

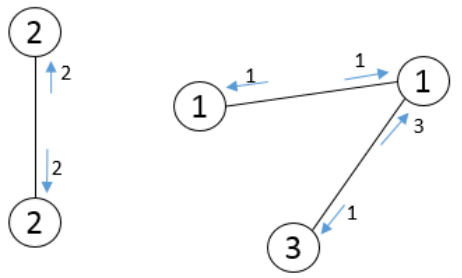
Fig. 1, Vector C represnts the component that every node belongs to

The algorithm proceeds as follows: We maintain a vector C^h which represents the component of every node in the graph at iteration h. At every iteration every node in the graph sends its node id to all the neighbors, then every node will update its component id with the minimum from all received messages as shown in figure x:



$$\begin{pmatrix}1\\2\\1\\2\\3\end{pmatrix} \quad C^1$$

Fig. 2, every node in the graph receives the component id of all its neighbor and then updates its component id to the minimum of all received messages and its current value.



$$\begin{pmatrix}1\\2\\1\\2\\1\end{pmatrix} \quad C^2$$

Fig. 3, as seen the vector C after the second iteration holds the correct component id of every node.

The above algorithm can be represented in GIM-V as follows:

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

Where M is the adjacency matrix, the operator $\times_G$ is implemented as:

1-combine2($m_{i,j}$, v_j) = $m_{i,j} * v_j$

2-combineAll$_i$ (x_1, . . . , x_n) = MIN({$x_j \neq 0 \mid j = 1 \ldots n$})

3-assign(v_i, v_{new}) = MIN(v_i, v_{new})

The algorithm is repeated until the ids converge. The algorithm terminates in a maximum d iterations where d is the diameter of the graph. This is easily observed as the minimum node id task at most d hops to propagate to the furthest vertex in the same connected component.

2.2) GIM-V and PageRank:

PageRank is a popular algorithm used by search engines to calculate relative importance of web pages. 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 and *U* is a matrix with all elements set to $\frac{1}{n}$.

It is clear from the above equation that the vector *p* is an eigenvector of the matrix $(cE^T + (1-c)U)$. To calculate the eigenvector *p*, the power method is used. The power method multiplies an initial vector with the matrix until convergence is met. The current PageRank vector p^{cur} is initialized and every element is set to $\frac{1}{n}$. The next PageRank vector p^{next} is calculated by

$p^{next} = (cE^T + (1-c)U)\ p^{cur}$.

The multiplication is repeated until *p* converges.

The implementation of PageRank in GIM-V is very straightforward. The matrix M is constructed by normalizing the columns of E^T such that every column of *M* sum to 1. First we rewrite the PageRank equation as follows:

$p^{next} = (cE^T + (1\text{-}c)U)\, p^{cur}$

$$p_i = \sum_{j=1}^{n} c * m_{i,j} * p_j + \frac{1-c}{n} \sum_{j=1}^{n} p_j$$

In page rank we require that all elements in the vector p should sum to 1. So the equation can be rewritten as:

$$p_i = \sum_{j=1}^{n} c * m_{i,j} * p_j + \frac{1-c}{n}$$

Then the algorithm proceeds as follows: $p^{next} = M \times_G p^{cur}$

where the three operations are defined as:

1- $\texttt{combine2}(m_{i,j}, v_j) = c * m_{i,j} * v_j$

2- $\texttt{combineAll}_i\ (x_1, \ldots, x_n) = \sum_{j=1}^{n} x_j + \frac{1-c}{n}$

3- $\texttt{assign}(v_i, v_{new}) = v_{new}$

3 - Algorithms for GIM-V:

The following section presents how the GIM-V algorithm can be executed in a parallelized fashion. First, a naïve Hadoop algorithm is introduced. Then more advanced optimizations are presented.

3.1 - Naïve GIM-V implementation (*GIM-V BASE*):

How can we implement a matrix by vector multiplication in map reduce? The figure below shows an example of matrix by vector multiplication in a map reduce environment using three machines

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 2 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$$

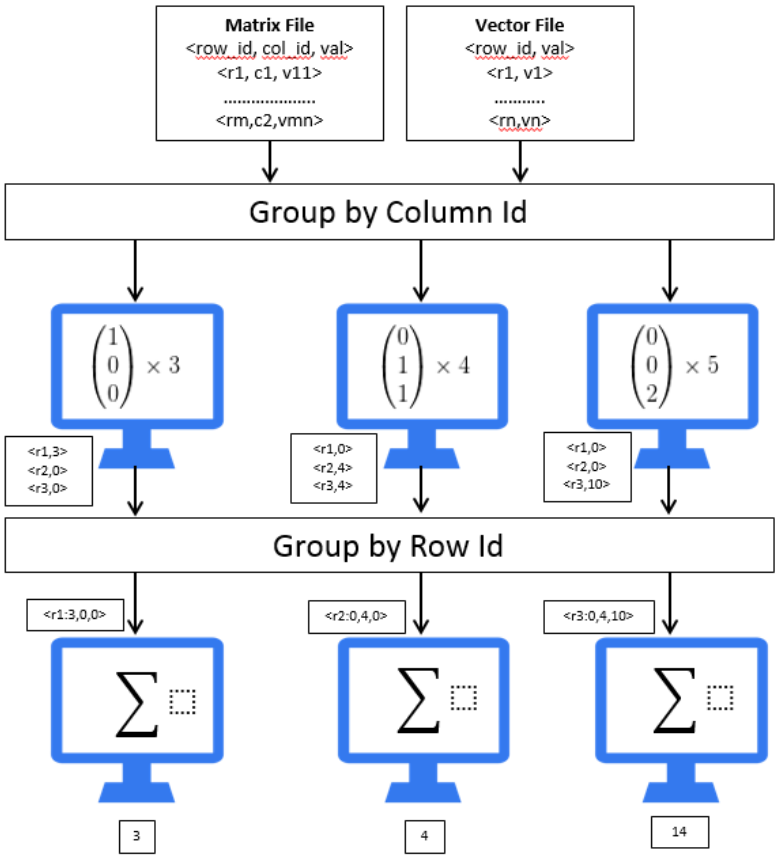


Fig. 4, distribution of work among machines during GIM-V execution

The above figure can be easily mapped to a two stage map reduce program as shown below:

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 == "self" 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
```

3.2 - GIM-V BL: Block Manipulation:

GIM-V BL is a faster algorithm which is based on block manipulation. The idea is to group elements of the input matrix into blocks of size b by b. Elements of the input vectors are also divided into blocks of length b. The format of matrix block with k nonzero elements is $(row_{block}, col_{block}, row_{elem1}, col_{elem1}, mval_{elem1}, ..., row_{elem-k}, col_{elem-k}, mval_{elem-k})$. Only blocks with at least one non-zero element are saved to disk. There are two improvements from this representation:

1- Compression of the data: In the naïve implementation it takes 4 * 2 bytes to save each <row,col> pair. In the above implementation it takes approximately 2*log b bits to represent an element. This compression results in speed up as fewer I/O operations are needed.
2- The bottleneck of the naïve implementation is the grouping stage which is implemented by sorting. So we can achieve better speedup by reducing the number of elements sorted. This is illustrated in figure 5. In the naïve implementation 36 elements have to be sorted, in the block representation 9 elements only have to be sorted.

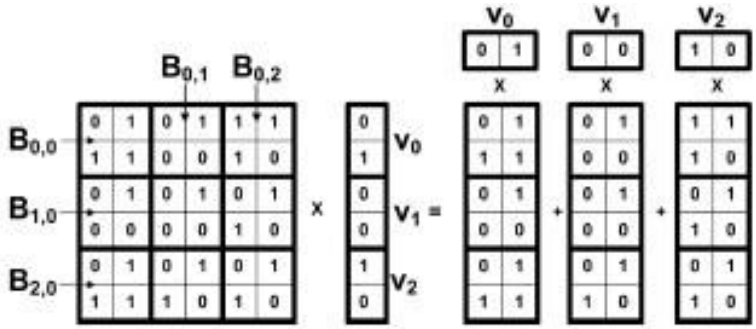


Figure 5 GIM-V BL using 2 x 2 blocks. $B_{i,j}$ represents a matrix block, and v_i represents a vector block. The matrix and vector are joined block-wise, not element-wise.

3.3 - *GIM-V* CL: *Clustered Edges*:

Another improvement which can be done is clustering edges. Clustering is a pre-processing step which can be ran only once on the data file and reused in the future. This can be used to reduce the number of used blocks. As in figure 6, the number of blocks can be reduced from nine to three by clustering.

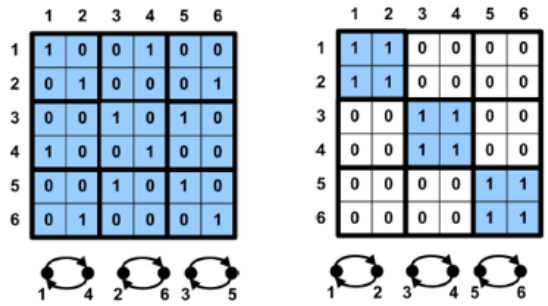


Fig. 6 Non-clustered (left) vs. clustered (right). The graph can be represented by three clustering. By numbering the nodes in every cluster sequentially we get the clustered graph.

One algorithm which can used for clustering using map-reduce is described in [2]

3.4 - *GIM-V* **DI:** *Diagonal Block Iteration***:**

One more possible improvement is to reduce the number of iterations required to converge. In HCC, it is possible to reduce the number of iterations by multiplying diagonal matrix blocks and the corresponding vector blocks more than once in a single iteration. Recall that in HCC multiplying a matrix and a vector corresponds to passing node ids to one step neighbors. By doing this more than once, the node id can be passed to neighbors more than one step away. This improvement is illustrated in Figure 7. The number of iterations required with m nodes and block size b is at most 2*[m/b] – 1. The proof is as follows, the furthest block is at most [m/b] – 1 steps away. It takes two iterations to pass node id from one block to the other (one iteration for propagating the node id in its own block, another iteration to move to the adjacent block). Finally, when the node id reaches the furthest possible block, it takes one more iteration to propagate in the last block. So it takes 2*[m/b] – 1 iterations.

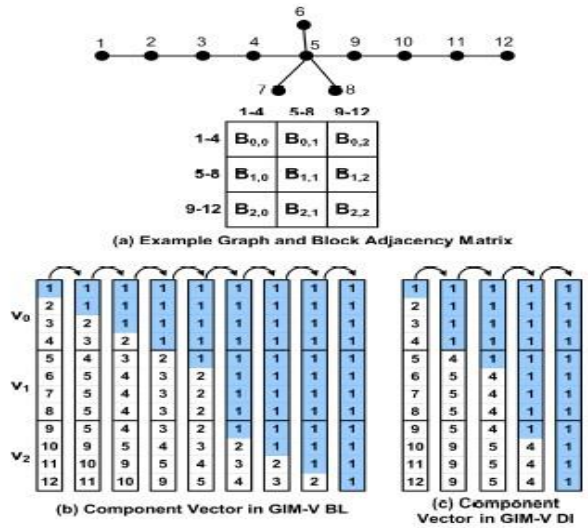


Figure 7 Propagation of component id(=1) when block width is 4. Each element in the adjacency matrix of (a) represents a 4 by 4 block; each column in (b) and (c) represents the vector after each iteration. GIM-V DI finishes in 4 iterations while GIM-V BL requires 8 iterations.

3.5 - Analysis:

The time complexity of *GIM-V*: An iteration of GIM-V takes $O(\frac{V+E}{M}\log(\frac{V+E}{M}))$ where M is the number of machines. Every machine receives $O(\frac{V+E}{M})$. The running time is dominated by the sorting time in the shuffle stage which is $O(\frac{V+E}{M}\log(\frac{V+E}{M}))$.

The space complexity of *GIM-V* is $(V + E)$ We note that the maximum storage required is at the output of Stage1 mappers which requires $O(V + E)$

4 - Performance and Scalability:

In this section the following two questions are answered:

1) How does GIM-V scale up?
2) What is the relative performance gain of the proposed optimizations (block multiplication, clustered edges, diagonal block iteration) ?

Synthetic Kronecker graphs has been used in the mentioned experiments as they can be generated with any size and are among the most realistic graphs among synthetic graphs.

4.1 - Results

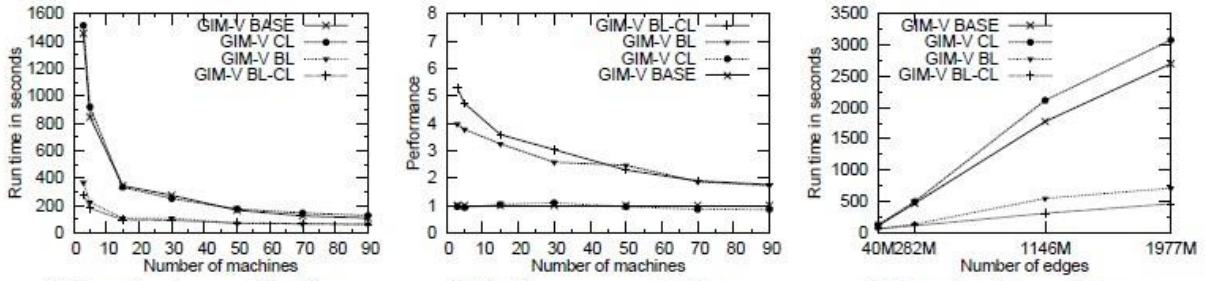
First, the relationship between performance and number of machines is investigated. Figure 8 shows the running time of PageRank on a Kronecker graph of 282 million edges and block size 32.

Figure 8(a) shows that the running time for all methods decreases as we add more machines. Note that clustered edges (GIM-V CL) doesn't help performance unless it is combined with block encoding. When combined with block encoding (GIM-V BL-CL), it shows the best performance.

Figure 8(b) shows that the performance of every method relative to GIM-V BASE decreases as the number of machines increases. With 3 machines, 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.

Figure 8(c) shows how the methods performs when the input size changes. As shown, all methods scales linearly with the number of edges.

Finally, Figure 9 shows the performance of GIM-V DI for HCC in Graphs with long chain. For this experiment a new graph has been constructed by adding a chain of length 15 to the previous Kronecker graph. As shown in Figure 9, GIM-V DI finished in 6 iterations while GIM-V BL-CL finished in 18 iterations. The running time of both methods for the first 6 iterations is almost the same. Therefore, we can conclude that the diagonal block iteration method decreases the number of iterations used without affecting the running time of every iteration.



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

Figure 8 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.

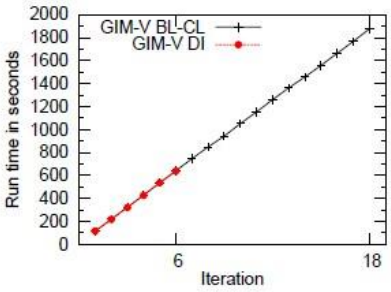


Figure 9. Comparison of GIM-V DI and GIM-V BL-CL for HCC. GIM-V DI finished in 6 iterations while GIM-V BL-CL finished in 18 iterations due to long chains.

5- Conclusions:

A graph mining framework based on Hadoop architecture has been introduced. The main contributions are summarized as:

1- The framework can be useful for many algorithms like PageRank, Random walk with Restart and connected components.
2- Several optimizations to the naïve GIM-V implementation has been introduced and the relative performance has been analyzed on both synthetic and real data sets.
3- The framework requires that the algorithm has to be written in the form of a matrix by vector multiplication. This is a serious limitation as this step is not trivial.

4- Pegasus doesn't provide much support for spectral analysis algorithms (SVD, dimensionality reduction …). For example, in page rank the power method was used which converges slowly. This has been addressed in some future work by the authors in [3].

References:

[1] U Kung, Charalampos E. Tsourakakis, and Christos Faloutsos, Pegasus: A Peta-Scale Graph Mining System - Implementation and Observations. Proc. Intl. Conf. Data Mining, 2009, 229-238.
[2] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce." ICDM, 2008
[3] U Kang, C Faloutsos – "Big Graph Mining: algorithms and discoveries", ACM SIG KDD Explorations Newsletter, 2013