

# Efficient Large-Scale Graph Analysis in MapReduce

Karthik Kambatla, Giorgos Kollias, Ananth Grama

*Department of Computer Science, Purdue University  
West Lafayette, IN, USA 47906*

---

## Abstract

Large-scale graph-structured datasets have become increasingly prominent — with web applications serving millions of users — prompting effective, efficient large-scale graph analysis. Earlier, graph analysis was mostly used in the context of relatively smaller graphs mostly on single processors or at most small clusters. These existing techniques do not scale to large-scale graphs due to the large working sets involved. On the other hand, recent research has studied scalable solutions for data analysis resulting in several distributed execution environments to exploit the independent data parallelism exhibited by these applications. The earlier attempts at realizing graph analysis using these scalable distributed execution environments are inefficient due to the workload distributions they adopt.

In this paper, we study *iterative matrix-vector product* in *MapReduce*, which is the key underlying operation in several graph algorithms. By studying the effect of workload distribution on resource utilization and other communication overheads, we propose an efficient implementation of *mat-vec* in *MapReduce*. We evaluate the proposed *mat-vec* implementation in the context of finding topographical similarity between nodes in large-scale graphs. By using the proposed *mat-vec* in conjunction with a novel, scalable decomposition technique, we compute graph similarity in billion-node graphs.

---

## 1. Introduction

Graphs are ubiquitous; graph structured datasets are used in diverse domains to efficiently capture the relations between various data items. Modern applications — web applications, social networks, complex scientific computations, *etc.* — often store the exceedingly large amounts of data they operate on as graph data structures yielding graphs of unprecedented sizes. For instance, Facebook social network contains 500 million nodes (users) with an average degree (number of friends) of 130<sup>1</sup>. Effective

---

*Email addresses:* [kkambat1@cs.purdue.edu](mailto:kkambat1@cs.purdue.edu) (Karthik Kambatla),  
[gkollias@cs.purdue.edu](mailto:gkollias@cs.purdue.edu) (Giorgos Kollias), [ayg@cs.purdue.edu](mailto:ayg@cs.purdue.edu) (Ananth Grama)

<sup>1</sup>Facebook Statistics. <http://www.facebook.com/press/info.php?statistics>

analyses of these graph datasets can lead to significant insights into application performance and possible optimizations in terms of data placement and processing.

Most of the existing graph analysis techniques are proposed for relatively smaller graphs and do not scale to web-scale graphs mostly due to the large working sets involved. Traditional graph analysis techniques assume the graphs to be small enough for the working sets to fit in memory. The working sets of large-scale graphs do not fit in memory on medium-sized clusters (10's of machines) and hence the existing graph analysis techniques do not apply. Also, the data in these large-scale datasets often comes from multiple sources and existing algorithms do not address this distributed nature of data.

Distributed execution engines offer highly desirable features like scalability and fault-tolerance and are being extensively used in data analytics. Data analytics is the analysis of large-scale data where data items can be analyzed independently. *MapReduce* [7], in particular, allows expressing the functionality in *map* and *reduce* functions abstracting away the intricacies of programming complex distributed systems. Though significant progress has been made in data analytics, the proposed approaches do not readily apply to graph analysis; graph analysis involves the interaction across multiple data points while data analytics adopt an independent view of the data points. *MapReduce* promises scalability and fault-tolerance, but *MapReduce* adaptations of graph algorithms are in relative infancy. *Pegasus* [13] proposes a graph-mining system using *MapReduce* exploiting fine-grained parallelism through 2-D block partitioning of the graph's adjacency matrix. However, such fine-grained parallelism does not yield optimal workload distribution in *MapReduce*-based clusters with long latencies and high bandwidths.

In this paper, we study alternate workload distributions and relevant optimizations for iterative computations consisting of pure matrix-vector multiplication (*mat-vec*) steps. *mat-vec* is the key primitive operation in several graph algorithms. Hence, our approach can easily be adapted to a broad selection of graph-based applications leading to significant performance improvements. For evaluation purposes of the proposed *mat-vec* implementations, we focus on the problem of computing topological similarity across and within graphs. In particular, by using a scalable decomposition technique implemented as optimized iterations over *mat-vec* steps, we compute similarity on a billion-node graph on just 32 Amazon EC2 machines. To the best of our knowledge, this is the first successful attempt to compute similarity on graphs of such scale.

The specific contributions of our work are —

1. A coarse-grained parallel *mat-vec* using 1-D block partitioning, further optimized by accounting for all local edges locally. Our coarse-grained adaptations demonstrate an order of magnitude performance improvements over the *Pegasus* system.
2. An optimizing restructuring of coarse-grained *mat-vec* iterations into a two-level scheme (inner/outer iterations).
3. A scalable decomposition technique to compute graph similarity that is conducive to *MapReduce*.
4. Demonstration of the proposed technique and implementation designs to compute similarity on a billion-node graph (with 20 billion edges).

## 2. Background and Related Work

### 2.1. Graph Analysis and Graph Mining

Several algorithms (and their parallel adaptations) have been proposed to solve a variety of graph problems — graph partitioning [14, 1], classification [15] and clustering (*eg.*, k-means), contraction [22], detecting network motifs [28], node ranking [17, 26, 4], etc. Many graph mining algorithms have been proposed for pattern mining (by approximation [5] and constraint pushing [31], on substructures [29], for compression-based datasets [16] and semistructured data [27]), apriori-based graph mining [30], frequent subgraph discovery [19, 9], etc. Unfortunately, most of these algorithms fail to scale either in graph sizes (to billions of nodes) or in the number of machines (to datacenters). The main reason behind the limited scalability is the requirement for the working sets to fit into memory or at least on one disk. Web-scale graphs run into terabytes and petabytes much higher than the memory that even datacenters can offer. *Pegasus* [13] proposes using *MapReduce* to achieve scalability; however, we identify that the *MapReduce* adaptations in *Pegasus* are highly sub-optimal. This work is the first successful attempt at efficient large-scale graph analysis and we demonstrate it in the context of graph similarity.

#### 2.1.1. Graph Similarity

Applications often require measuring the similarity of objects naturally participating in interrelation structures as captured in graph abstractions; *eg.*, in the analysis of complex networks of biological data, Web or general hypertext structures. In the context of web-scale graphs, similarity of graphs can be used in tracking the growth of the application data and identifying the system level enhancements in storing and accessing such data for performance improvements.

Graph similarity is traditionally computed through (1) iterative diffusion-based methods, or (2) exhaustive enumeration of topologically common subgraphs in the vicinity of a node. The enumeration approach, although straightforward, is computationally demanding since the number of topologically distinct subgraphs — called *graphlets* in [24] — grows exponentially with the number of nodes, severely limiting the applicability of these methods to graphs up to a few thousand nodes.

Similarity is recursively encoded in diffusion-based approaches – “Two nodes are similar if their neighbors are similar”. Melnik et al. [20] use *similarity flooding* (as the diffusion process) to match elements of data schemas or instances; while *SimRank* [11] introduces a general algorithm for capturing the aforementioned recursive nature of similarity. Blondel *et.al.* [3] emphasize the connection of similarity computation with *HITS* ranking [17] and they apply their method for the automatic extraction of synonyms from a dictionary of word definitions. *IsoRank* [26] is essentially *PageRank* [4] over the product graph of the two networks under comparison (namely Protein-Protein Interaction networks - PPI) and therefore it opts for the inclusion of pre-existing knowledge about the node similarity and the variation of the contribution of the networks themselves in the diffusion process.

The diffusion-based methods are typically implemented as iterations of triple-matrix products. The matrices are effectively the adjacency matrices of the two graphs, and the matrix of similarity scores of all tentative node pairs; when iterations converge, the

best matching of nodes is extracted. However the presence of triple-matrix-product severely suppresses potential advantages from the parallelization of these methods. To remedy this, our decomposition technique, *NSD*, transforms the triple-matrix-product into a series of sparse *mat-vecs* also reducing the final matching extraction phase into a trivial mapping of two sorted vectors. These advances permit the parallelization of the diffusion-based similarity methods and also lead quite naturally, in the presence of very large graphs, to the idea of porting *NSD* to a fault-tolerant platform like *MapReduce* and analyzing its performance.

## 2.2. Hadoop Background

Dean and Ghemawat proposed *MapReduce* [7] to facilitate development of highly-scalable, fault-tolerant, large-scale distributed applications. The *MapReduce* runtime system divests programmers of low-level details of scheduling, load balancing, and fault tolerance. The *map phase* of a *MapReduce job* takes as input a list of key-value pairs,  $\langle key, value \rangle : list$ , and applies a programmer-specified (*map*) *function*, independently, on each pair in the list. The output of the *map phase* is a list of keys and their associated value lists –  $\langle key, value : list \rangle : list$ , referred to as intermediate data. The *reduce phase* of the *MapReduce job* takes this intermediate data as input, and applies another programmer-specified (*reduce*) *function* on the *map* output to generate a list of final values.

In the open source implementation of MapReduce, *Hadoop*<sup>2</sup>, *map* and *reduce phases* are split into multiple *tasks* operating on parts of the input, with each *task* potentially executing on multiple nodes. When a node fails, the corresponding *tasks* are re-executed on another node. The output of *map tasks* is sorted and stored locally. Each *reduce task* pulls its input from multiple *map tasks*, this transfer *phase* is often referred to as the *shuffle phase*. The *reduce tasks* wait for the *map phase* to complete before applying the *reduce functions* to ensure all values corresponding to each key are available.

A number of research efforts have targeted improving both the systems and applications aspects of MapReduce. The MapReduce programming model has been validated on diverse application domains like data-analytics and data-mining. Pig [21] offers a high-level SQL like language for easier analysis of large-scale data, while HadoopDB [2] builds a distributed database by using *Hadoop* for (storage and analysis) distribution over single-node databases. Dryad [10] supports a more general data-flow model expressed as acyclic graphs. To extend the applicability of *MapReduce*, MapReduce Online [6] supports online aggregation and continuous queries, while [12] extends *MapReduce* to support asynchronous algorithms efficiently through relaxed synchronization semantics.

---

<sup>2</sup>Apache Hadoop. <http://hadoop.apache.org/>

### 3. Mat-Vec design and implementation

In this section, we study the effects of workload distribution on *mat-vec* in *MapReduce* to propose optimized versions of both single and multiple<sup>3</sup> *mat-vecs*. To arrive at the optimal implementations, we analyze the performance trade-offs of multiple implementations through execution on smaller datasets. In these experiments, we monitor the resource utilization on each machine, and the duration of *MapReduce* stages to propose improvements to the naive technique. We stress out the fact although these experiments have been performed in the context of computing graph similarity, the proposed improvements are general and apply to several other sparse graph algorithms.

Table 1: Testbeds, Software

<b>Amazon EC2 Instances</b>	4 64 bit EC2 Compute Units 7.5 GB RAM, 850 GB storage
<b>Software</b>	Hadoop 0.20.2, Java 1.6
<b>Map Capacity</b>	16 map tasks; 4 per node
<b>Reduce Capacity</b>	16 reduce tasks; 4 per node
<b>Heap space</b>	512 MB per <i>map/reduce</i> task

Our experiments were conducted on Amazon EC2 nodes to mimic a typical cloud setup. Table 1 outlines the corresponding setup details. Our analyses mostly use 4 EC2 large instances; each instance running 4 map tasks and 4 reduce tasks. For scalability testing, we use 4, 8, 16 and 32 instances. Given the scalable nature of *MapReduce* and cloud setups, we believe the results translate to computing similarity on bigger graphs on larger clusters.

**Input Graphs.** The input graphs used in the analyses are randomly generated using *MapReduce*. Generating the graphs in the EC2 cluster is economical, avoiding the need for moving large datasets to the cluster. Each *map* generates the adjacency lists for nodes within a particular range, with an average out-degree of 20. The subgraph to be generated by a *map* is determined by the input parameters — *start* and *subgraph-size*. The parameter *border-edge-fraction* determines the fraction of edges to nodes outside the subgraph. An input graph with  $2^a$  nodes and a subgraph size of  $2^b$  is labeled as graph-*a-b*.

#### 3.1. Optimizing the single Matrix-Vector Product

Expressing *mat-vec* in *MapReduce* entails making design choices like the matrix/graph representation and the functionality of *map* and *reduce* functions. Here, we examine some of these design choices to provide an optimized *mat-vec*, and also compares against the Pegasus system<sup>4</sup>.

<sup>3</sup>These are the iteration steps, also referred as iterative *mat-vecs* in the 4

<sup>4</sup>Pegasus. An open-source, scalable graph-mining system. <http://www.cs.cmu.edu/pegasus>

### 3.1.1. Pegasus– 2-D partitioning

Pegasus focuses on reducing the number of iterations required for an iterative *mat-vec* to converge. However, there is scope for significant improvement in the single *mat-vec* implementation. The main reason for the sub-optimal performance is the use of 2-D partitioning adjacency matrix in Pegasus. Pegasus *mat-vec* uses an edge list representation of the graph/matrix to allow block multiplication, further enabling optimizations like clustered edges and diagonal block iterations. These optimizations prove to be very effective in lowering the number of iterations to converge on iterative *mat-vecs*. However, the edge list representation leads to the *mat-vec* requiring two *MapReduce* jobs. With each *MapReduce* job requiring a global synchronization before and after a reduce operation, the associated I/O overheads result in significant overhead. [12] shows that the execution time of most sparse graph algorithms in *MapReduce* is proportional to the number of jobs due to the huge overheads involved in global synchronization.

The Pegasus *mat-vec* implementation used in our experiments corresponds to the naive *mat-vec* in *PageRank* implementation. Pegasus implements *PageRank* as a series of *mat-vecs*; the number of executed iterations is bounded by a specified maximum number of iterations and a threshold on the maximum difference in ranks between two iterations. Our experiments set the threshold to 0 and maximum iterations to 1. We notice that, for a single *mat-vec*, their naive implementation significantly faster than their blocked version, though the blocked version significantly reduces the number of iterations if we are interested in convergence. The Pegasus implementation scales sub-linearly with the graph size on a given number of machines, and linearly with the number of machines.

### 3.1.2. Naive Mat-Vec

Our naive implementation adopts a 1-D partitioning of the adjacency matrix — each *map* or *reduce* operates on one row of the adjacency matrix. Each *map* (per node) emits the node’s rank to each out-going neighbor, and each *reduce* accumulates all such ranks from its in-coming neighbors to compute its own rank. In the context of the adjacency matrix, each *map* reads a row of the matrix and multiplies it by the corresponding vector element, and each *reduce* aggregates the elements corresponding to one element in the output vector. Unlike Pegasus, our implementation requires a single *MapReduce* job immediately improving performance. However, it does not allow the optimizations proposed in Pegasus to decrease the number of iterations.

Figure 1 shows the resource utilization (cpu, disk, and network) for naive *mat-vec* on the input graph graph-25-19. The *map* and *reduce* phases are clearly distinguishable — the *map* phase has significantly high CPU utilization, and very little disk and network utilizations, while the *reduce* phase has low CPU utilization and high disk and network utilizations. Even though both *map* and *reduce* phases operate on roughly the same amount of data (the graph), the *map* phase takes significant proportion of the time, two-thirds in this case, under high CPU utilization.

To determine the cause for such unexpected behavior, we compute the fraction of time spent in each stage of *MapReduce*— *map*, *shuffle*, *reduce*; we split the *map* phase further into *map-compute*, *map-disk-spill* and *map-merge*. Figure 2 shows the duration of these micro-stages as percentages of the total execution time — *map*, *spill*, *merge*

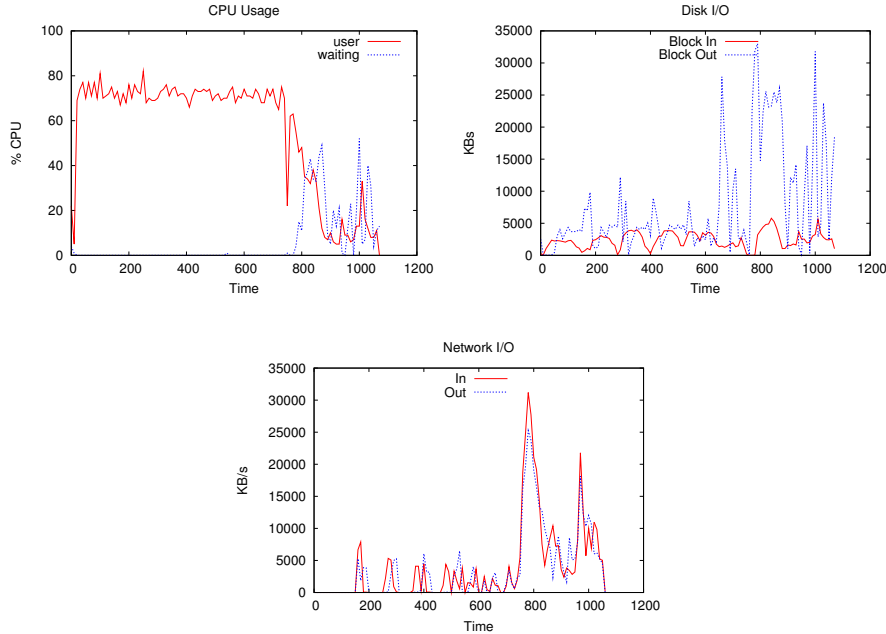


Figure 1: Resource Utilization for naive *mat-vec* on graph-25-19

correspond to the *map-compute*, *map-disk-spill*, and *map-merge* stages respectively. The stages overlap and hence the percentages do not add up to 100%. Interestingly, disk spill of the *map* outputs to the local disk and *map-merge* (merging of the sorted files on disk) dominate *map* execution. The disk spill and local sort overheads are more pronounced in jobs with larger intermediate data. Though the *shuffle* fraction is significantly high, most of it is due to the waiting for *map* tasks to finish. Once the *map* tasks finish, *shuffle* moves the data from *map* tasks to *reduce* tasks. This transfer of data over the network benefits from lesser intermediate data.

The *map-disk-spill*, *map-merge*, and *shuffle* overheads in *MapReduce* can be addressed by

- reducing the intermediate data generated at the application level
- *MapReduce* implementations with efficient disk spill and local sort.

### 3.1.3. Partitioned Mat-Vec

The partitioned *mat-vec* adopts a 1-D block partitioning of the adjacency matrix to reduce the intermediate data. The partitioned *mat-vec* takes a graph partition as input (*i.e.*, few adjacency lists), as opposed to an adjacency list in the naive implementation. The graph partition can be as big as the subgraphs in the input, because each subgraph is written to a different file (*i.e.*,  $partition-size \leq subgraph-size$ ). In the *map* phase, all nodes partially accumulate the ranks from the in-coming neighbors that

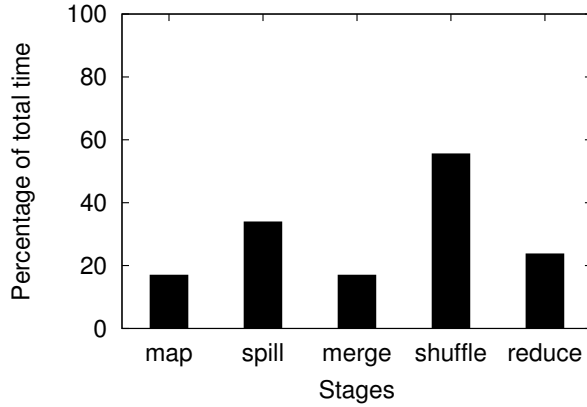


Figure 2: *Hadoop* stages: percentage split-up of times. Percentages do not sum up to 1 because of overlap.

are in the partition, and the *map* output corresponds to only those edges that connect nodes across partitions. The nodes with all their in-coming neighbors in the partition end up computing their final ranks in the map phase itself. The size of the intermediate data depends mostly on the *partition-size*. The optimal *partition-size* depends on (i) physical resources and (ii) graph structure. The memory available forces a hard limit on the size of the partitioned that can be operated on locally. For instance, a heap size of 512 MB used in our experiments holds partitions of size  $2^{19}$  nodes. Figure 3a shows the variation of execution times with *partition-sizes* on an input graph, graph-23-19 with a 20% *border-edge-fraction*. Larger partitions yield better performance. In addition to the physical resources, graph structure also effects performance. For graphs with many small strongly-connected components, smaller partitions allow exploiting the parallelism available.

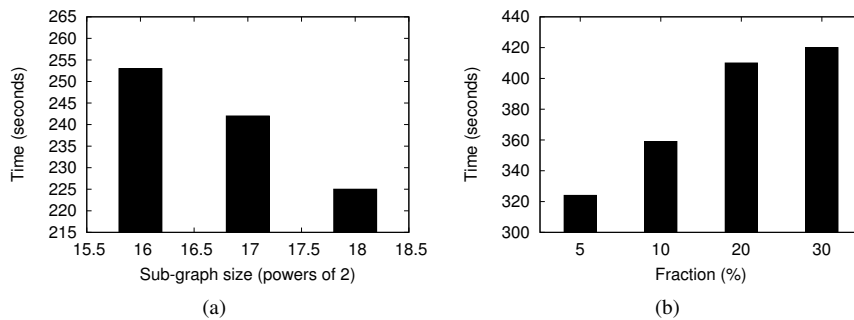


Figure 3: Partitioned *mat-vec* performance: (a) Dependence on subgraph sizes on graph-23-19, border-edge-fraction = 0.2; (b) Dependence on ratio of edges on graph-24-18, partition-size = subgraph-size =  $2^{18}$

Partitioned *mat-vec* works very well in practice as most practical large graphs fol-

low a power-law like distribution, with very few edges cutting across strongly connected components. The performance of partitioned *mat-vec* can be augmented by optimally partitioning the graph. Most graph datasets are inherently well partitioned; the crawlers used to collect such data induce locality as they crawl neighborhoods before crawling remote sites. Also, tools like Metis can be used for partitioning graph datasets. For our analyses, we generate a random graph with a pre-specified fraction of edges to nodes in other subgraphs. Figure 3b shows the variation in execution times with the ratio of edges across subgraphs. As expected, execution time increases with more edges connecting nodes across subgraphs.

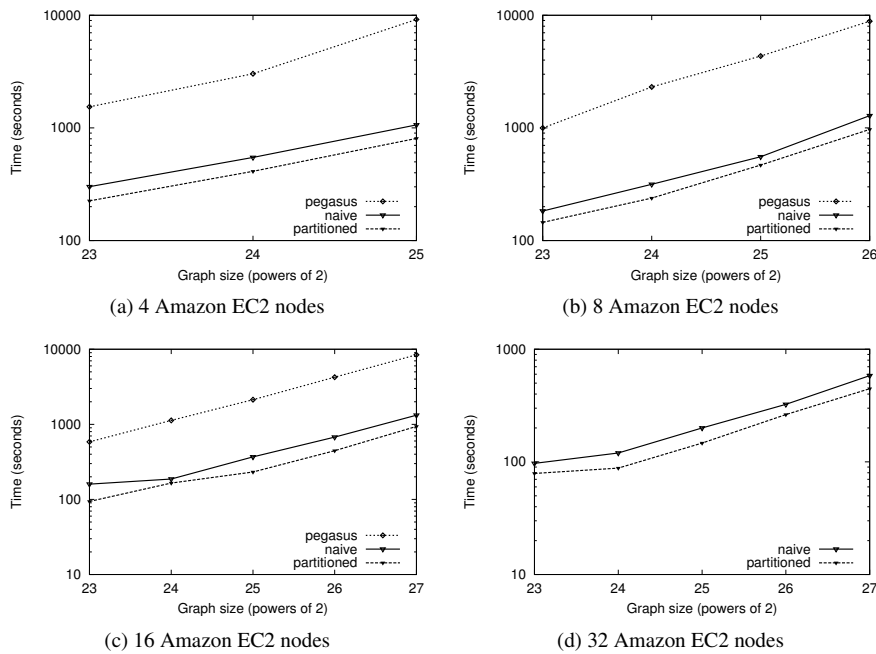


Figure 4: *Mat-Vec* performance: Pegasus Naive vs Naive vs Partitioned; subgraph-size =  $2^{19}$ , partition-size =  $2^{18}$ , fraction of border edges = 20%

Figure 4 compares the naive and partitioned *mat-vec* implementations, against the Pegasus implementation for graphs of different sizes. The results are very consistent across different cluster sizes. The partitioned *mat-vec* consistently performs better across different graphs — the naive *mat-vec* takes 1.3x longer for the chosen *partition-size* and *border-edge-fraction*. Larger *partition-sizes* and smaller *border-edge-fractions* yield further gains, while a *partition-size* of 1 boils down to the naive case.

In comparison to Pegasus, both our implementations perform an order of magnitude better than Pegasus *mat-vecs*. The Pegasus *mat-vecs* were taking considerably longer than our implementations, hence we did not run it on 32 machines. The main reasons behind the improved performance are —

- Our implementations span a single *MapReduce* job due to the 1-D partitioning of

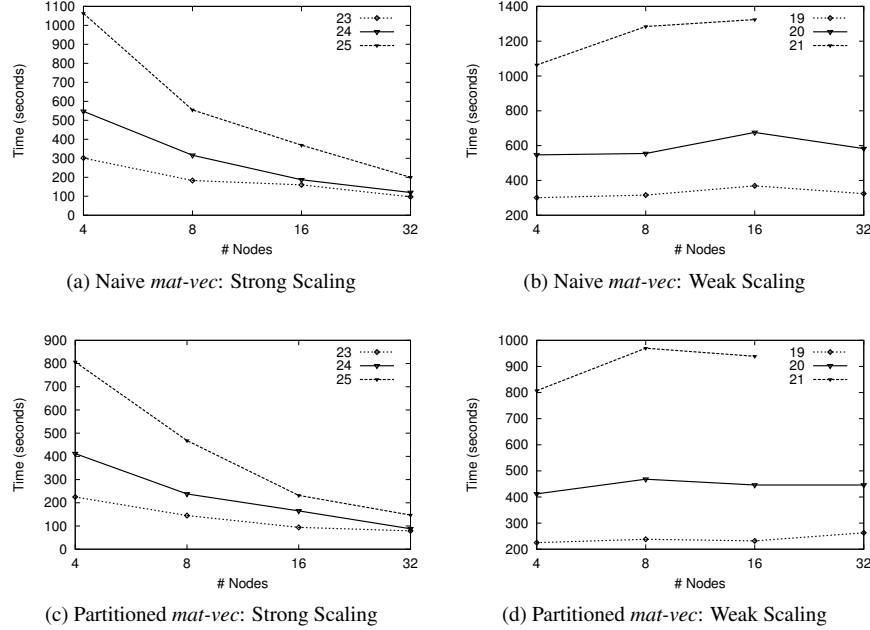


Figure 5: *Mat-Vec* Scalability: Naive and Partitioned; subgraph-size =  $2^{19}$ , partition-size =  $2^{18}$ , fraction of border edges = 20%

the matrix, while the `Pegasus` implementations require two *MapReduce* jobs.

- **Implementation Detail:** Our use of integers for graph nodes and floats for node ranks, instead of longs and doubles respectively, reduces the I/O by almost a factor of 2.

**Scalability.** In Figure 4, for each cluster size, the execution times increase linearly with graph sizes demonstrating scalability in input sizes. Figure 5 shows the strong and weak scalability of the proposed naive and partitioned *mat-vec* implementations. Strong scaling is the variation of execution times with number of computing nodes for fixed graph size, while weak scaling is the variation of execution times with number of computing nodes for fixed per-core graph size. We use graphs of sizes  $2^{23}$ ,  $2^{24}$  and  $2^{25}$  for strong scaling, and  $2^{19}$ ,  $2^{20}$  and  $2^{21}$  for strong scaling. Both naive and partitioned *mat-vecs* exhibit satisfactory strong and weak scaling properties.

### 3.1.4. No-Reduce Matrix-Vector Product

In both the naive and partitioned *mat-vecs*, the implementation has two phases — all nodes disseminate their ranks in the first phase, and the disseminated ranks are accumulated in the second phase. The communication cost is  $O(\text{edges})$  in the naive case, and the  $O(\text{edges across subgraphs})$  in the partitioned case. Both implementations operate on input graphs represented as adjacency lists listing out-going edges, requiring

two phases and the inter-phase communication of  $O(\text{edges})$ . The transpose of such input graph yields a graph representation of adjacency lists listing in-coming edges. *Mat-Vec* on the transformed graph (theoretically) requires only one phase — computing the new rank of a node involves looking up the ranks of the in-coming nodes listed in the adjacency list. Such a modified implementation requires communicating the ranks of nodes after every iteration, reducing the communication cost to  $O(\text{nodes})$ .

For smaller datasets, the node ranks (initial or from the previous iteration) can be cached in memory; in *Hadoop*, one can use *DistributedCache* to replicate the ranks which can then be fetched during *map* setup. However, such an approach does not scale with the number of nodes due to memory constraints. Disk-resident hash tables like memcached<sup>5</sup>, or distributed key-value stores like HBase<sup>6</sup> or Cassandra<sup>7</sup> can be used to address large graphs. However, our implementations suffer from a significantly high look-up latency, which negates any performance gain achieved by avoiding inter-phase communication, rendering the approach practically infeasible.

### 3.2. Optimizing *Mat-Vec* iterations: The Inner/Outer case

The goal of iterative *mat-vecs* is to find an ordering (by rank) of the graph nodes. We are interested only in the final ordering of the nodes, and not their exact rank values. The relaxed problem allows asynchronous iterations of *mat-vecs*; inner/outer asynchronous iterations [8] yield significant performance gains in parallel environments by reducing the number of outer iterations, which require heavy global synchronization.

The synchronous iterative *mat-vec* requires global synchronization, of all nodes, in every iteration. The above mentioned naive and partitioned *mat-vecs* are instances of synchronous *mat-vec*. The *reduce* phase performs the global synchronization, where every node computes its rank based on the ranks of all of its in-coming neighbors.

An asynchronous iterative *mat-vec* allows frequent partial synchronizations and infrequent global synchronizations, significantly decreasing the communication costs. Partitioned *mat-vec* can be augmented to implement inner/outer iterations by iterating over the *mat-vec* on the subgraph inside the *map*, using the (partial) ranks produced after every such (inner) iteration to compute the new set of ranks in the subsequent iteration. At the end of the inner iterations, the *reduce* can synchronize the partial ranks (updated with subgraph edges) to compute the global ranks. By performing the subgraph-specific updates locally, one can decrease the number of outer iterations; each outer iteration requiring a heavy *reduce* operation and the associated all-to-all communication. Even though the total computation is higher in the inner/outer approach, the high communication costs avoided by reduced number of outer iterations yields significant performance gains.

Figure 6 plots the execution times for an iterative *mat-vec* in a graph with  $2^{23}$  nodes for 20 iterations of the naive and partitioned *mat-vecs*, and the inner/outer approach running 8 outer iterations each with 5 inner iterations. Partitioned and inner/outer approaches use a partition-size of  $2^{18}$  nodes. As expected, the partitioned case performs

---

<sup>5</sup>Memcached. <http://memcached.org/>

<sup>6</sup>Apache HBase. <http://hbase.apache.org/>

<sup>7</sup>Apache Cassandra. <http://cassandra.apache.org/>

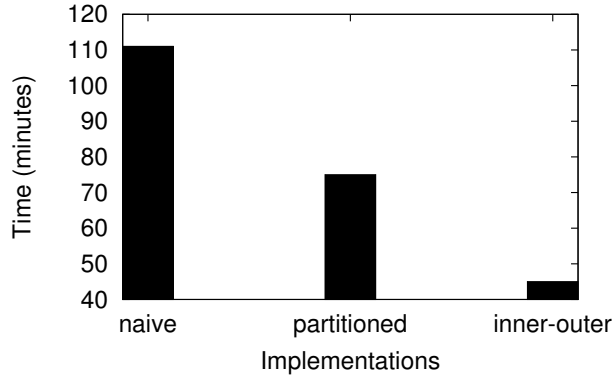


Figure 6: Iterative *Mat-Vec*: Naive vs Partitioned vs Inner/Outer

better than the naive case, and the inner/outer case performs significantly better than both the naive and partitioned cases.

#### 4. Evaluation: Graph Similarity

In this section, we evaluate the proposed approaches to compute iterative *mat-vec* in the context of computing topographical similarity within and across graphs. We compute self-similarity within graphs for convenience of operating on just one graph. Computing graph similarity through traditional methods in a scalable fashion is not possible. First, we describe a scalable decomposition technique to compute graph similarity using a series of *mat-vecs*. Then, we implement graph similarity using our proposed implementations of iterative *mat-vecs*.

##### 4.1. Network Similarity Decomposition

Here, we describe network similarity decomposition (NSD) [18] and its *MapReduce* adaptation to compute similarity between web-scale graphs. NSD models similarity computation as a series of *mat-vecs* for scalability both in terms of graph size and computing resources. We represent a graph,  $G_A(V_A, E_A)$ , by its adjacency matrix of outgoing edges,  $A$ , where  $a_{ij} = 1$  iff node  $i$  points to node  $j$  and zero otherwise.  $V_A$  and  $E_A$  denote the vertices and edges of  $G_A$  respectively, and  $n_A$  denotes the total number of nodes in  $G_A$ .  $\tilde{A}$  is the normalized version of  $A^T$ ; formally,  $(\tilde{A})_{ij} = a_{ji} / \sum_{i=1}^{n_A} a_{ji}$  for nonzero rows of  $A$  and zero otherwise.  $\mathbf{1}_{n_A}$  is the column vector of size  $n_A$  consisting of 1's.

Singh et al. [26] propose *IsoRank*, a two-phased approach to compute similarity. The first phase computes the similarity matrix, of two graphs  $G_B$  and  $G_A$  with  $m$  and  $n$  nodes respectively ( $m \leq n$ ), iteratively till the similarity scores converge.  $X_{m \times n}$  is a normalized matrix — its elements add to unity — where  $x_{ij}$  indicates the similarity score of the nodes  $i \in V_B$  and  $j \in V_A$ . The second phase uses a maximum-weight bipartite matching algorithm to find the best matching between nodes in  $G_A$  and  $G_B$  based on the final  $X$  scores computed earlier.

*IsoRank* iteration kernel is of the form

$$X \leftarrow \alpha \tilde{B} X \tilde{A}^T + (1 - \alpha) H \quad (1)$$

where  $H$  is the elemental similarity scores matrix as inferred from other information sources prior to computing *IsoRank*. For instance, graphs under *similarity comparison* represent Protein Protein Interaction (PPI) networks for pairs of species (nodes represent proteins and undirected edges connect physically interacting proteins) while  $H$  is the matrix of sequence similarities of proteins as produced by independent BLAST runs. The computation in Equation (1) —  $\alpha \tilde{B} X \tilde{A}^T$  (a triple-matrix product) — implements the recursive intuition behind this similarity computation approach that two nodes are similar if their neighboring nodes are (pairwise) similar.  $\alpha \in [0, 1]$  is the damping factor that denotes the relative contribution of this topology-only mechanism in building  $X$ ; the remaining  $1 - \alpha$  portion is injected at each iteration step by the independent similarity information embodied in matrix  $H$ .

The basic idea of this, basically diffusive, scheme is explained in Figure 7: For two networks  $G_A$  and  $G_B$ , correspondingly consisting of nodes  $\{a, b, c, d, e, f, g\}$  and  $\{1, 2, 3, 4, 5, 6\}$  it is given that between nodes of  $(b, 1)$ ,  $(f, 4)$  and  $(g, 6)$  pairs there is some similarity (i.e.  $H$  information). How to computationally upgrade a similarity affinity between e.g. nodes  $c$  and  $2$ ? The adopted approach is to sum the contributions of all 1-hop, 2-hop, 3-hop,... neighbors in the two networks (along red, green blue paths,...) that happen to have some similarity affinity already computed or given; powers of the  $\alpha$  parameter decrease higher-hop contributions and also non-terminal nodes in relevant paths divide by their degrees (multiplicatively entering the sum).

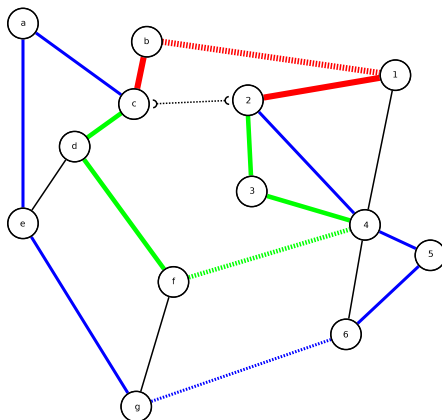


Figure 7: The basic idea of similarity computation as a diffusive process

Although semantically appealing, this iteration is hard to apply to graphs running into hundreds of thousands of nodes or larger as the storage requirements for  $X$  outgrow the physical memory capacity of a typical computing node. Parallelizing and thus distributing the storage across a computer cluster could be an interesting solution to this problem, but the triple-matrix-product parallelization introduces significant communication overheads and implementation subtleties.

Network Similarity Decomposition (NSD) adopts the approach of *IsoRank*, but models the similarity computation as a series of *mat-vecs* instead to avoid the communication costs of a triple-matrix-product. The series-of-*mat-vecs* formulation is described as follows —

Without loss of generality, we can use  $H$  as the initial condition  $X^{(0)}$  and so after  $t$  iterations,  $X^{(t)}$  takes the form

$$X^{(t)} = (1 - \alpha) \sum_{k=0}^{t-1} \alpha^k \tilde{B}^k H (\tilde{A}^T)^k + \alpha^t \tilde{B}^t H (\tilde{A}^T)^t \quad (2)$$

However, any  $H$  can always be decomposed into a set of  $s$  vector pairs (components), that can generally be expressed as:

$$H = \sum_{i=1}^s w_i z_i^T, \quad (3)$$

Substituting decomposition (3) in Equation (2) yields

$$X^{(t)} = \sum_{i=1}^s X_i^{(t)} \quad (4)$$

where

$$X_i^{(t)} = (1 - \alpha) \sum_{k=0}^{t-1} \alpha^k w_i^{(k)} z_i^{(k)T} + \alpha^t w_i^{(t)} z_i^{(t)T}, \quad (5)$$

and  $w_i^{(k)} = \tilde{B}^k w_i$ ,  $z_i^{(k)} = \tilde{A}^k z_i$ .

This provides us with the flexibility of computing  $w_i^{(k)}$  and  $z_i^{(k)}$  vectors independently through efficient sparse *mat-vec* iterations. The resulting vectors are partitioned and forwarded to a  $p \times q$  process grid (*i.e.*, partition  $w_i^{(k)}$  in  $p$  fragments,  $w_{i,1}^{(k)}, \dots, w_{i,p}^{(k)}$  and  $z_i^{(k)}$  in  $q$  fragments,  $z_{i,1}^{(k)}, \dots, z_{i,q}^{(k)}$ ) to generate the naturally distributed similarity matrix  $X^{(t)}$ .

However, in practice, parallelization leads to the following road-blocks:

- The number of components,  $s$ , can be very large leading to severe performance issues. Though approximation of  $H$  is possible using the  $r < s$  most dominant components, it is not as accurate.
- The subsequent stage of matching should also be performed in parallel (the auction algorithm as parallelized in [25] has been successfully tested as shown in [18], necessitating only a minimal adaptation of  $q = 1$  for the process grid).
- Even for moderate-sized clusters, it becomes infeasible to store  $X$  in main memory (distributedly) for graph pairs of just a few million nodes (a sparsification scheme like the one in [18] needs to be applied, further compromising accuracy).

However, for cases where elemental similarity scores are not available, we say all pairs of nodes are initially “equisimilar”. “Equisimilarity” can be achieved by representing  $X^{(0)}$  as  $1_m 1_n^T$  with suitable normalization (eg., by elementwise division respectively by  $m$  and  $n$ ), and using  $\alpha = 1$ .

From Equations 4 and 5, we have

$$X^{(t)} = w^{(t)} z^{(t)T} \quad (6)$$

where  $w^{(t)} = \tilde{B}^t 1_m$  and  $z^{(t)} = \tilde{A}^t 1_n$ .

The explicit assembly of the similarity matrix —  $X^{(t)}$  in Equation 6 — dominates the computation time and also introduces (physical memory) scalability problems for graphs beyond a few millions of nodes, even in the case of NSD. Avoiding this explicit construction leads to improved scalability and performance. The similarity matrix construction can be avoided by using the greedy matching algorithm [23] to find the best matching between nodes of  $G_A$  and  $G_B$ . Note that in the case of an explicitly constructed  $X$  the greedy matching algorithm works by iteratively pairing the nodes in the two graphs: Each iteration (i) finds the maximum score entry in  $X^{(t)}$  (eg., the  $i^j$ <sup>th</sup> entry), (ii) reports the corresponding matching pair, and (iii) zeroes the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column of the similarity matrix to enforce reporting a pair at most once. However in our case, where the explicit construction of  $X$  is avoided, the greedy matching operation can be captured by first sorting  $w^{(t)}$  and  $z^{(t)}$  vectors and then matching the nodes in this order. Further, while computing self-similarity (finding distinct nodes which are similar within a graph), greedy matching boils down to sorting the resulting (single) vector and matching its successive nodes (i.e., 1-2, 3-4, ...).

---

**Algorithm 1** Identify matching node pairs for graphs  $G_A$  and  $G_B$ .

Input: Adjacency matrices  $A, B$  and number of iterations  $t$

---

- 1: compute  $\tilde{A}, \tilde{B}$
  - 2:  $w_i^{(0)} \leftarrow 1_m/m, z_i^{(0)} \leftarrow 1_n/n$
  - 3: **for**  $k = 1$  to  $t$  **do**
  - 4:    $w^{(k)} \leftarrow \tilde{B} w^{(k-1)}, z^{(k)} \leftarrow \tilde{A} z^{(k-1)}$
  - 5: **end for**
  - 6: sort  $w^{(t)}, z^{(t)}$  in descending order and compute node indices  $I_w, J_z$  in this ordering
  
  - 7: **for**  $i = 1$  to  $m$  **do**
  - 8:    $I_w(i) \in V_B$  matches to  $J_w(i) \in V_A$
  - 9: **end for**
- 

Algorithm 1 describes the steps involved in computing similarity between nodes of two graphs using the scalable decomposition technique for node ranking followed by the greedy matching subphase (lines 7-9).

#### 4.2. Illustrations

Our approach to graph alignment is semantically intuitive; small-scale self-similarity tests also verify its predictive correctness. To this direction we experiment with two vi-

sually tractable graphs namely a synthetic random network of 10 nodes and a real, manually curated webgraph of 500 nodes (`harvard500`<sup>8</sup>), as depicted in Figure 8.

Ideally a self-similarity computation (computing the node matches of a network with itself) should map each node to itself. This sanity result is indeed the case for our instances: In Figures 9a, 10a all computed matching node ID pairs lie along the diagonal.

However it is interesting to note that by enforcing the constraint to skip this trivial solution, matches seem to happen between nodes of similar neighborhood structure. For example in (5, 6), (8, 9) and (2, 3) matching pairs, the nodes are of comparable input/output degrees (them or considering also their 1-hop neighbors' degree structure). In cases like the (8, 9) match, these two nodes share 3 of their neighbors (2, 3, 5) and 9 has two extra connections (to 1 and 10) compared to only one for 8 (to 6), i.e. different degree; however node 6 is more densely interconnected than either of 1, 10, thus "balancing" the initially annoying degree mismatch of the two matched nodes.

Essentially "important" nodes (in the PageRank ranking sense) are matched together in the non trivial solution case and this is evident from self-similarity results with `harvard500`: In Table 2 URLs of top matched nodes are listed and highly linked-to sites (corresponding e.g. to the home pages of key Harvard university schools/departments) are matched. These are the large deviations from the diagonal of the plot in Figure 10a; they relate the most important nodes of each of the subdomains - manifest as block submatrices in Figure 10 - across them.

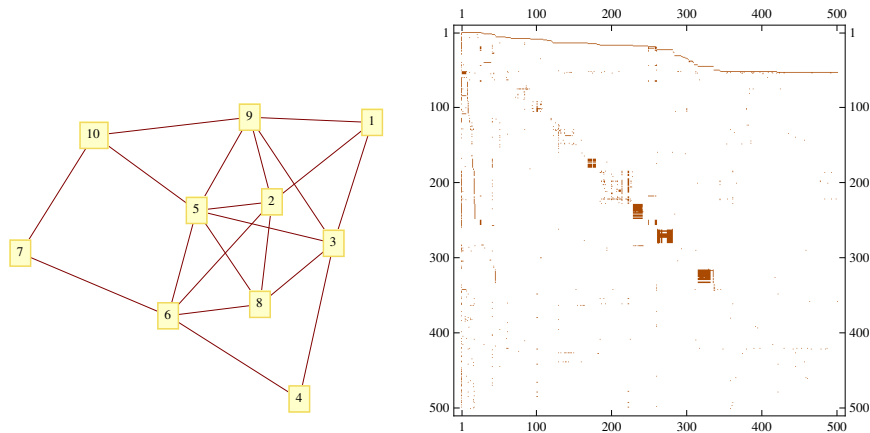


Figure 8: The synthetic directed random network of 10 nodes (right) and `harvard500` test graph (spy plot of its adjacency matrix, left) used in experiments.

<sup>8</sup>Also available from <http://www.cise.ufl.edu/research/sparse/matrices/MathWorks/Harvard500.html> as part of the University of Florida Sparse Matrix Collection.

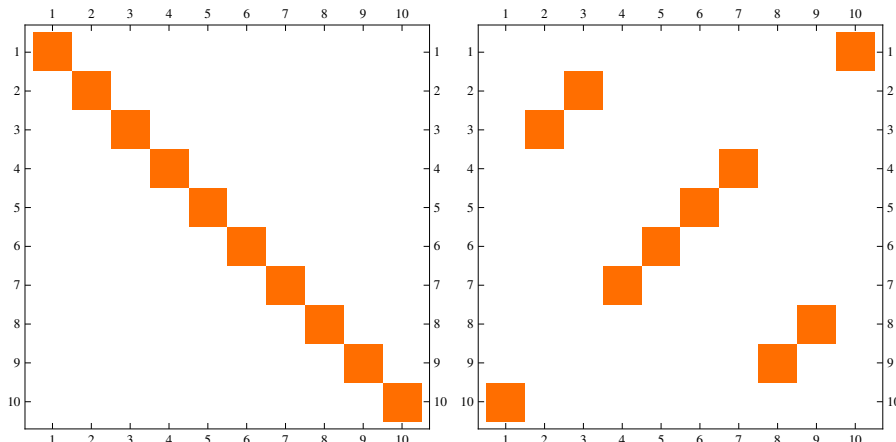


Figure 9: Self-similarity spy plots for the synthetic graph instance: On the right we have put the constraint to skip the natural matching of a node to itself that can be recovered by our algorithm (left).

<a href="http://www.harvard.edu">http://www.harvard.edu</a>	<a href="http://www.hbs.edu">http://www.hbs.edu</a>
<a href="http://www.med.harvard.edu">http://www.med.harvard.edu</a>	<a href="http://search.harvard.edu:8765/custom/query.html">http://search.harvard.edu:8765/custom/query.html</a>
<a href="http://www.hms.harvard.edu">http://www.hms.harvard.edu</a>	<a href="http://www.gocrimson.com">http://www.gocrimson.com</a>
<a href="http://www.gse.harvard.edu">http://www.gse.harvard.edu</a>	<a href="http://www.radcliffe.edu">http://www.radcliffe.edu</a>
<a href="http://www.hsdm.harvard.edu">http://www.hsdm.harvard.edu</a>	<a href="http://www.hsdm.med.harvard.edu">http://www.hsdm.med.harvard.edu</a>
<a href="http://www.hsph.harvard.edu">http://www.hsph.harvard.edu</a>	<a href="http://www.ksg.harvard.edu">http://www.ksg.harvard.edu</a>
<a href="http://whitepages.med.harvard.edu">http://whitepages.med.harvard.edu</a>	<a href="http://www.studentadvantage.com">http://www.studentadvantage.com</a>
<a href="http://gocrimson.ocsn.com/tickets/harv-tickets.html">http://gocrimson.ocsn.com/tickets/harv-tickets.html</a>	<a href="http://www.hds.harvard.edu">http://www.hds.harvard.edu</a>
<a href="http://www.dana-farber.org">http://www.dana-farber.org</a>	<a href="http://www.radcliffe.edu/events/index.html">http://www.radcliffe.edu/events/index.html</a>
<a href="http://www.radcliffe.edu/fellowships/index.html">http://www.radcliffe.edu/fellowships/index.html</a>	<a href="http://www.radcliffe.edu/research/index.html">http://www.radcliffe.edu/research/index.html</a>

Table 2: Top 10 matched URL pairs from the self-similarity computation over `harvard500` test graph (skip self-match constraint).

### 4.3. Results

Graph similarity, in our implementation, is implemented in *MapReduce*. The *mat-vec* iterations are implemented using the inner/outer approach; sorting and pairing steps are also implemented in *MapReduce*. All our approaches are inherently scalable as they use *MapReduce*. We compute similarity on a billion-node graph (double the number of Facebook users) on the largest of testbeds (32 machines) with a configuration as described in Table 1. The input graph is randomly generated using *MapReduce*, each *map* generating the adjacency lists for  $2^{19}$  nodes. The inner/outer approach with 8 outer and 5 inner iterations takes an acceptable **11 hours and 21 minutes** to compute self-similarity. Even larger clusters (as used in many research lab and industry settings) can compute the similarity even quicker.

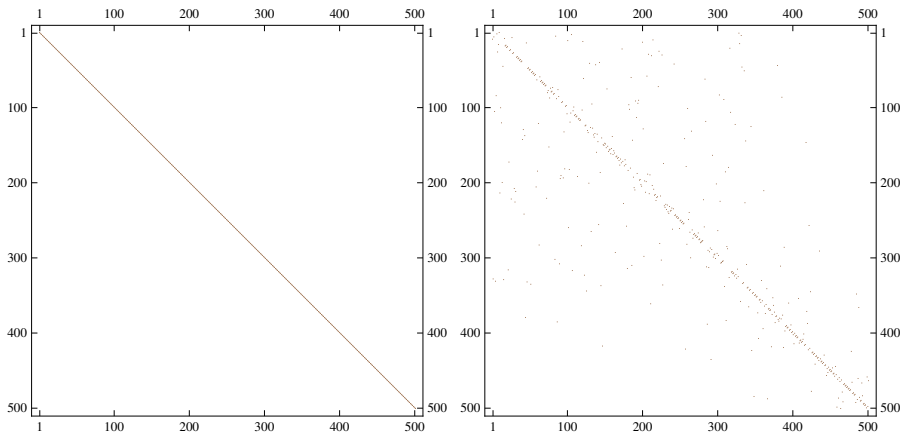


Figure 10: Self-similarity spy plots for harvard500: On the right we have put the constraint to skip the natural matching of a node to itself that can be recovered by our algorithm (left).

## 5. Conclusion

Large-scale graph analysis is more important than ever, thanks to the growing demands of modern applications using exceedingly large graph datasets. The scalability and fault-tolerance requirements of graph analysis can be addressed by existing distribution execution environments. In this paper, we study work-splitting in *mat-vec*, which is the key underlying operation in several graph algorithms. We identify that coarse-grained splitting (1-D and 1-D block partitioning) of the adjacency matrix yields best performance in *MapReduce*. We propose the use of inner/outer iterations for improved performance of iterative *mat-vec* when using 1-D block partitioning. We evaluate the proposed techniques in the context of graph similarity using scalable decomposition of the similarity matrix. The proposed coarse-grained parallelism allows us to compute graph similarity on a billion-node, 20 billion edges graph on just 32 Amazon EC2 machines.

## References

- [1] Amine Abou-rjeili and George Karypis. Multilevel Algorithms for Partitioning Power-Law Graphs. In *IPDPS*, 2006.
- [2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB*, 2009.
- [3] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. Van Dooren. A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching. *SIAM Review*, 46(4):647–666, 2004.

- [4] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.
- [5] Chen Chen, Xifeng Yan, Feida Zhu, and Jiawei Han. gApprox: Mining Frequent Approximate Patterns from a Massive Network. In *IEEE International Conference on Data Mining (ICDM)*, 2007.
- [6] Tyson Condie, Neil Conway, Peter Alvaro, Joseph Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *NSDI*, 2009.
- [7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
- [8] Gene H. Golub, Zhenyue Zhang, and Hongyuan Zha. Large Sparse Symmetric Eigenvalue Problems with Homogeneous Linear Constraints: The Lanczos Process with Inner-Outer Iterations. *Linear Algebra and its Applications*, 309(1-3):289–306, 2000.
- [9] Jun Huan, Wei Wang, and Jan Prins. Spin: Mining Maximal Frequent Subgraphs from Graph Databases. In *SIGKDD*, 2004.
- [10] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.
- [11] G. Jeh and J. Widom. SimRank: A Measure of Structural-Context Similarity. In *SIGKDD*, 2002.
- [12] Karthik Kambatla, Naresh Rapolu, Suresh Jagannathan, and Ananth Grama. Asynchronous Algorithms in MapReduce. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2010.
- [13] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System. In *IEEE International Conference on Data Mining (ICDM)*, 2009.
- [14] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing (JPDC)*, 48:96–129, 1998.
- [15] Hisashi Kashima and Akihiro Inokuchi. Kernels for Graph Classification. *ICDM Workshop on Active Mining*, 2002.
- [16] Nikhil S Ketkar. Subdue: Compression-Based Frequent Pattern Discovery in Graph Data. In *1st International Workshop on Open Source Data Mining (OSDM)*, 2005.
- [17] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

- [18] Giorgos Kollias, Madan Sathe, Olaf Schenk, and Ananth Grama. Parallel Algorithms for Graph Similarity. *submitted*, 2011.
- [19] Michihiro Kuramochi and George Karypis. Frequent Subgraph Discovery. In *IEEE International Conference on Data Mining (ICDM)*, 2001.
- [20] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *ICDE*, 2002.
- [21] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008. ACM ID: 1376726.
- [22] C. A. Philips. Parallel Graph Contraction. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1989.
- [23] R. Preis. Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In *International Symposium on Theoretical Aspects of Computer Science (STACS)*, 1999.
- [24] Nataa Prulj. Biological Network Comparison Using Graphlet Degree Distribution. *Bioinformatics*, 23(2):e177–e183, January 2007.
- [25] Madan Sathe, Olaf Schenk, Florian Miller, Ye Zhao, and Helmar Burkhart. Accelerating Maximum Weighted Matching Algorithms in Massive Graph Analysis. *submitted*, 2011.
- [26] R. Singh, J. Xu, and B. Berger. Global Alignment of Multiple Protein Interaction Networks with Application to Functional Orthology Detection. *Proceedings of the National Academy of Sciences*, 105(35):12763, 2008.
- [27] N. Vanetik, E. Gudes, and S. E. Shimony. Computing Frequent Graph Patterns from Semistructured Data. In *IEEE International Conference on Data Mining (ICDM)*, 2002.
- [28] Sebastian Wernicke. Efficient Detection of Network Motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 3:347359, October 2006. ACM ID: 1183306.
- [29] Xifeng Yan and Jiawei Han. gSpan: Graph-Based Substructure Pattern Mining. In *IEEE International Conference on Data Mining (ICDM)*, 2002.
- [30] Nishimura Yoshio, Washio Takashi, Yoshida Tetsuya, Motoda Hiroshi, Inokuchi Akihiro, Ibm J, and Okada Takashi. Fast Apriori-Based Graph Mining Algorithm and Application to 3-Dimensional Structure Analysis. *Transactions of the Japanese Society for Artificial Intelligence*, 18:257–268, 2003.
- [31] F Zhu, X Yan, J Han, and PS Yu. gPrune: A Constraint Pushing Framework for Graph Pattern Mining. In *Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)*, 2007.