2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining

Fast Recommendation on Bibliographic Networks

Onur Küçüktunç*†, Kamer Kaya*, Erik Saule*, Ümit V. Çatalyürek*‡
{kucuktunc,kamer,esaule,umit}@bmi.osu.edu
*Department of Biomedical Informatics, The Ohio State University
†Department of Computer Science and Engineering, The Ohio State University
‡Department of Electrical and Computer Engineering, The Ohio State University

***Abstract*—Graphs and matrices are widely used in algorithms for social network analyses. Since the number of interactions is much less than the possible number of interactions, the graphs and matrices used in the analyses are usually sparse. In this paper, we propose an efficient implementation of a sparse-matrix computation which arises in our publicly available citation recommendation service called theadvisor. The recommendation algorithm uses a sparse matrix generated from the citation graph. We observed that the nonzero pattern of this matrix is highly irregular and the computation suffers from high number of cache misses. We propose techniques for storing the matrix in memory efficiently and reducing the number of cache misses. Experimental results show that our techniques are highly efficient on reducing the query processing time which is highly crucial for a web service.**

***Index Terms*—Citation recommendation; social network analysis; sparse matrices; hypergraphs; cache locality.**

I. Introduction

Sparse-matrix computations working exclusively on nonzero entries are usually not suitable for today's cache architectures if an ordinary ordering of the rows/columns/nonzeros is used. The difficulty arises from the fact that the memory access pattern in such computations depends on the nonzero distribution in the matrix which usually does not have a well-defined regular structure. If the access pattern is random, the number of cache misses through the computation increases. Since there will be a penalty for each cache miss, reordering the nonzero accesses in the matrix is a good idea to reduce the number of cache misses and hence, the execution time.

One of the most widely used operation in network analysis is sparse matrix-dense vector multiplication (SpMxV). This operation is assumed to be the computational bottleneck for the network analyses based on random walk with restart (RWR) which is used in PageRank [1], impact factor computations [2], recommendation systems [3], [4] and finding/predicting genetic interactions [5].

In this paper, we target a citation, venue, and expert recommendation problem in our publicly available web-service called the**advisor**[1]. The service takes a bibliography file in various formats (bib,ris,xml) that contains a set of *seed* papers to initiate the recommendation process. Then, it returns a set of papers ordered with respect to a ranking function. The user can guide the search or prune the list of suggested papers with positive or negative feedbacks by declaring some papers relevant or irrelevant. In this case, the service completely refines the set and shows the new results back to the user. In addition to papers, the**advisor** also suggests researchers or experts, and conferences or journals of interest. The service is designed to help to the researchers while performing several tasks, such as:

- literature search,
- improving the reference list of a manuscript being written,
- finding conferences and journals for attendance, subscription, or paper submission,
- finding a set of researchers in a field of interest to follow their work,
- finding a list of potential reviewers, which is required by certain journals in the submission process.

The algorithm we use in the**advisor** is based on RWR and can be naively implemented as an SpMxV operation. There exist several methods in the literature proposed to improve the cache locality for the SpMxV operations by reordering the rows and/or columns of the matrix by using graph/hypergraph partitioning [6], [7], [8], [9], [10] and other techniques [11], [12], [13], [14]. The recommendation algorithm used in the**advisor** is *direction aware*. That is, the user can specify that she is interested in classical papers or recent papers. This property brings a unique characteristic to the SpMxV operation used in the service which makes existing hypergraph partitioning based techniques [6], [8], [9] not directly applicable.

In this paper, our contribution is two-fold: First, we propose techniques to efficiently store the matrix used by our direction-aware algorithm. We then propose an efficient implementation of the algorithm and investigate several matrix ordering techniques based on a hypergraph partitioning model and ordering heuristics, such as the Approximate Minimum Degree (AMD) [15] and Reverse Cuthill-McKee (RCM) [16].

We give a thorough evaluation of the proposed approach and measure the efficiency of implementation and matrix storing/ordering techniques used in the**advisor**. The combination of all the techniques improved the response time of our service by 67% (3x). We believe that the techniques proposed here can also be useful for SpMxV-related sparse-matrix problems in social network analysis.

State-of-the-art hypergraph partitioners are typically too slow to be used to optimize just a couple of SpMxV operations. However, considering the**advisor**'s purpose, the algorithm will be executed many times whereas the ordering is required only

[1] http://theadvisor.osu.edu/

978-0-7695-4799-2 2012

DOI 10.1109/ASONAM.2012.82

once. The current version of our service is already using the implementation and ordering described in this paper.

The paper is organized as follows: In Section II, we describe the hypergraph partitioning problem and existing matrix ordering techniques for SpMxV. In Section III, we formally describe our direction-aware RWR algorithm and its efficient implementation used in the**advisor**. The ordering techniques we use are given in Section IV. Section V gives the experimental results and Section VI concludes the paper.

II. Background

A. Modeling sparse matrices with hypergraphs

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. A net $\eta \in \mathcal{N}$ is a subset of vertices, and the vertices in η are called its *pins*. The *size* of a net is the number of its pins, and the *degree* of a vertex is equal to the number of nets that contain it. A graph is a special instance of hypergraph such that each net has size two. Vertices can be associated with weights, denoted with $\mathsf{w}[\cdot]$, and nets can be associated with costs, denoted with $\mathsf{c}[\cdot]$.

A *K-way partition* of a hypergraph $\mathcal{H}$ is denoted as $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ where parts are pairwise disjoint, each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, and union of K parts is equal to $\mathcal{V}$.

In a partition Π, a net that has at least one pin (vertex) in a part is said to *connect* that part. The number of parts connected by a net η, i.e., *connectivity*, is denoted as λ_η. A net η is said to be *uncut* (*internal*) if it connects exactly one part, and *cut* (*external*), otherwise (i.e., $\lambda_\eta > 1$).

Let W_k denote the total vertex weight in $\mathcal{V}_k$ and W_{avg} denote the weight of each part when the total vertex weight is equally distributed. If each part $\mathcal{V}_k \in \Pi$ satisfies the *balance criterion*

$$W_k \leq W_{avg}(1+\varepsilon), \quad \text{for } k = 1, 2, \ldots, K \tag{1}$$

we say that Π is *balanced* where ε represents the maximum allowed imbalance ratio.

The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. Let $\chi(\Pi)$ denote the cost, i.e., *cutsize*, of a partition Π. There are various cutsize definitions [17]. In this work, we use

$$\chi_{conn}(\Pi) = \sum_{\eta \in \mathcal{N}} \mathsf{c}[\eta](\lambda_\eta - 1) \ . \tag{2}$$

The cutsize metric given in (2) will be referred to as *connectivity*-1 metric. Given ε and an integer $K > 1$, the hypergraph partitioning problem can be defined as the task of finding a balanced partition Π with K parts such that $\chi(\Pi)$ is minimized. The hypergraph partitioning problem is NP-hard [17] with any of the above objective functions. We used a state-of-the-art partitioning tool PaToH [18].

There are three well-known hypergraph models for sparse matrices. These are the column-net [19], row-net [19], and fine-grain models [20]. Here, we describe the column-net model we used for a sparse matrix $\mathbf{A}$ of size $n \times n$ with m nonzeros. In the *column-net model*, $\mathbf{A}$ is represented as a unit-cost hypergraph $\mathcal{H}_\mathcal{R} = (\mathcal{V}_\mathcal{R}, \mathcal{N}_\mathcal{C})$ with $|\mathcal{V}_R| = n$ vertices, $|\mathcal{N}_C| = n$ nets, and m pins. In $\mathcal{H}_\mathcal{R}$, there exists one vertex $v_i \in \mathcal{V}_\mathcal{R}$ for each row i. Weight $\mathsf{w}[v_i]$ of a vertex v_i is equal to the number of nonzeros in row i. There exists one unit-cost net $\eta_j \in \mathcal{N}_\mathcal{C}$ for each column j. Net η_j connects the vertices corresponding to the rows that have a nonzero in column j. That is, $v_i \in \eta_j$ if and only if $a_{ij} \neq 0$. The *row-net model* is the column-net model of the transpose of $\mathbf{A}$.

B. Matrix ordering techniques for improving cache locality in SpMxV

The SpMxV operation is defined as $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ where $\mathbf{A}$ is an $n \times n$ sparse matrix with m nonzeros, $\mathbf{x}$ is the $n \times 1$ input vector, and $\mathbf{y}$ is the $n \times 1$ output vector. Let $\mathbf{P}$ and $\mathbf{Q}$ be two $n \times n$ permutation matrices. That is, $\mathbf{P}$ and $\mathbf{Q}$ have only a 1 in each of their rows and columns and the rest is 0. When the matrix $\mathbf{A}$ is ordered as $\mathbf{A}' = \mathbf{PAQ}$, the SpMxV operation can be written as $\mathbf{y}' \leftarrow \mathbf{A}'\mathbf{x}'$ where $\mathbf{y}' = \mathbf{P}\mathbf{y}$ and $\mathbf{x}' = \mathbf{Q}^T\mathbf{x}$. Some existing cache-locality optimization techniques use this fact and permute the rows and columns of $\mathbf{A}$ to improve cache locality. To find good $\mathbf{P}$ and $\mathbf{Q}$, several approaches are proposed in the literature: Bandwidth reduction is proven to be promising for decreasing cache misses [14]. For this reason, the reverse Cuthill-McKee heuristic [16] has been frequently used as a tool and a benchmark by several researchers [7], [13], [21]. RCM has also been frequently used as a fill-in minimization heuristic for sparse LU factorization. Another succesful fill-in minimization heuristic, the approximate minimum degree (AMD) [15], is also used for improving cache locality [21]. Graph and hypergraph partitioning models and techniques have been extensively studied for reducing cache misses [6], [7], [8], [9], [10]. Among those, the most similar ones to our work are [8], [9] and [6], which use hypergraph partitioning as the main tool to reduce the number of cache misses.

As should be evident, the sparse matrix storage format and the cache locality are related. In this work, we use two of the most common formats. The coordinate format (COO) keeps an array of m triplets of the form $\langle a_{ij}, i, j \rangle$ for a sparse matrix $\mathbf{A}$ with m entries. Each triplet contains a nonzero entry a_{ij} and its row and column indices (i, j). The COO format is suitable for generating arbitrary orderings of the non-zero entries. The compressed row storage format (CRS) uses three arrays to store a $n \times n$ sparse matrix $\mathbf{A}$ with m nonzeros. One array of size m keeps the values of nonzeros where the nonzeros in a row are stored consecutively. Another array parallel to the first one keeps the column index of each nonzero. The third array keeps the starting index of the nonzeros at a given row where the ending index of the nonzeros at a row is one less than the starting index of the next row. A matrix represented in CRS is typically 30% smaller than the COO since the m entries representing i in COO are compressed in an array of size n in CRS.

III. Direction-aware RWR for citation recommendation

Citation analysis-based paper recommendation has been a popular problem since the 60's. There are methods that only take local neighbors (i.e., citations and references) into account, e.g., bibliographic coupling [22], cocitation [23], and CCIDF [24]. Recent studies, however, employ graph-based algorithms, such as Katz [25], random walk with restarts [26], or well-known PageRank algorithm to investigate the whole citation network. PaperRank [27], ArticleRank [28], and Katz distance-based methods [25] are typical examples.

For our recommendation service, the**advisor**, we designed an algorithm based on RWR on a directed citation graph $G = (V, E)$ where $V = \{1, \ldots, n\}$ is the vertex set and E, the edge set, contains an edge (i, j) if paper i cites paper j. We defined a *direction awareness* parameter $\kappa \in [0, 1]$ to obtain more recent or traditional results in the top-k documents.

A. Random Walk with Restart

Let $deg^+(i)$ and $deg^-(i)$ be the number of references of and citations to paper i, respectively. Given a query with inputs k, a *seed* paper set $\mathcal{M}$, damping factor d, and direction awareness parameter κ, our algorithm computes the steady-state probability vector $\mathbf{p}$. For an iterative RWR implementation, at iteration t, the two types of contributions of paper i to a neighbor paper are defined as:

$$c_t^+(i) = \mathbf{p}_{t-1}(i)\frac{d(1-\kappa)}{deg^+(i)}, \tag{3}$$

$$c_t^-(i) = \mathbf{p}_{t-1}(i)\frac{d\kappa}{deg^-(i)}, \tag{4}$$

where $\mathbf{p}_{t-1}$ is the probability (rank) vector after the $(t-1)$th iteration, $c_t^+(i)$ is the contribution of paper i to a paper in its reference list and $c_t^-(i)$ is the contribution of paper i to a paper which cites i. The rank of paper i after iteration t is computed with,

$$\mathbf{p}_t(i) = \mathbf{r}(i) + \sum_{(i,j)\in E} c_t^-(j) + \sum_{(j,i)\in E} c_t^+(j), \tag{5}$$

where $\mathbf{r}$ is the restart probability vector due to jump backs to the papers in $\mathcal{M}$, computed with,

$$\mathbf{r}(i) = \begin{cases} \frac{1-d}{|\mathcal{M}|}, & \text{if } i \in \mathcal{M} \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Hence, each iteration of the algorithm can be defined with the following linear equation:

$$\mathbf{p}_t = \mathbf{r} + \mathbf{A}\mathbf{p}_{t-1}, \tag{7}$$

where $\mathbf{r}$ is an $n \times 1$ restart probability vector calculated with (6), and $\mathbf{A}$ is a structurally-symmetric $n \times n$ matrix of edge weights, such that

$$a_{ij} = \begin{cases} \frac{d(1-\kappa)}{deg^+(i)}, & \text{if } (i,j) \in E \\ \frac{d\kappa}{deg^-(i)}, & \text{if } (j,i) \in E \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

The algorithm converges when the probability of the papers are stable, i.e., when the process is in a *steady state*. Let

$$\Delta_t = (\mathbf{p}_t(1) - \mathbf{p}_{t-1}(1), \ldots, \mathbf{p}_t(n) - \mathbf{p}_{t-1}(n)) \tag{9}$$

be the difference vector. We say that the process is in the steady state when the L2 norm of Δ_t is smaller than a given value ξ. That is,

$$\|\Delta_t\|_2 = \sqrt{\sum_{i\in V} \left(\mathbf{p}_t(i) - \mathbf{p}_{t-1}(i)\right)^2} < \xi. \tag{10}$$

Assume that $\mathbf{A}$ is stored in CRS format. Algorithm 1 shows the pseudocode of our RWR-based approach where at each iteration, (7) is computed.

Algorithm 1: RWR with CRS-Full

```
Input: A: n × n input matrix in CRS format, seed paper set M
Output: Steady state probability vector p
p_t ← 0
for each paper i ∈ M do
    p_t(i) ← 1/|M|
e ← ||p_t||_2
while e > ξ do
    p_{t-1} ← p_t
    p_t ← 0
    for each paper i = 1 to n do
1       if p_{t-1}(i) > 0 then
            for each nonzero a_ij in row i of A do
                p_t(j) ← p_t(j) + a_ij p_{t-1}(i)
    for each paper i ∈ M do
        p_t(i) ← p_t(i) + (1-d)/|M|
    e ← ||p_t − p_{t-1}||_2
return p ← p_t
```

B. A more efficient RWR implementation

To compute (7), one needs to read all of $\mathbf{A}$ at each iteration. Note that for each nonzero in $\mathbf{A}$, there is a possible update on $\mathbf{p}_t$. The number of vertices and edges in the citation graph G are given in Table I. As described above, $\mathbf{A}$ contains $2|E|$ nonzeros which is approximately equal to 12×10^6. This size allows us to index rows and columns using 32-bit values. However, the probabilities and matrix entries are stored in 64-bit. Assuming it is stored in CRS format, the size of $\mathbf{A}$ in memory is roughly 147MB.

TABLE I
Statistics for the citation graph G.

$\|V\|$	$\|E\|$	avg deg	max deg^+	max deg^-
982,067	5,964,494	6.07	617	5418

Here, we propose two modifications to reduce $\mathbf{A}$'s size and the number of multiplications required to update $\mathbf{p}_t$. The first modification is compressing the nonzeros in $\mathbf{A}$: we know that during an iteration, the contributions of paper i to the papers in its reference list are all $c_t^-(i)$. Similarly, the contributions

of i to the papers which cite i are $c_t^+(i)$. Let $\mathbf{s}_R$ and $\mathbf{s}_C$ be the row and column scaling vectors defined as

$$\mathbf{s}_R(i) = \begin{cases} \frac{d(1-\kappa)}{deg^+(i)}, & \text{if } deg^+(i) > 0 \\ 0, & \text{otherwise} \end{cases} \quad (11)$$

$$\mathbf{s}_C(i) = \begin{cases} \frac{d\kappa}{deg^-(i)}, & \text{if } deg^-(i) > 0 \\ 0, & \text{otherwise.} \end{cases} \quad (12)$$

For the papers without any references ($deg^+(i) = 0$) or citations ($deg^-(i) = 0$), the contributions are adjusted so that they not lost and only towards the other direction, i.e., $\mathbf{s}_C(i) = d/deg^-(i)$ or $\mathbf{s}_R(i) = d/deg^+(i)$, respectively. Let $\mathbf{B}$ be the 0-1 adjacency matrix of G defined as

$$b_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise.} \end{cases} \quad (13)$$

Then (7) can be rewritten as

$$\mathbf{p}_t = \mathbf{r} + \mathbf{B}(\mathbf{s}_R * \mathbf{p}_{t-1}) + \mathbf{B}^T(\mathbf{s}_C * \mathbf{p}_{t-1}), \quad (14)$$

where $*$ denote the pointwise vector multiplication. In this form, the total size of $\mathbf{B}$, $\mathbf{B}^T$, $\mathbf{s}_R$, and $\mathbf{s}_C$ is roughly 71MB assuming we only store the indices of nonzeros in $\mathbf{B}$ and $\mathbf{B}^T$. This modification not only reduces the size of $\mathbf{A}$, but also decreases the number of multiplications required in each iteration. Here, we only need to do pointwise multiplications $\mathbf{s}_R * \mathbf{p}_{t-1}$ and $\mathbf{s}_C * \mathbf{p}_{t-1}$ before traversing the nonzero indices. Hence, we only need to do $2|V|$ multiplications per iteration. Assuming $\mathbf{p}_t(i) > 0$ for all $i \in V$, Algorithm 1 (which follows Equation (7)) performs $2|E|$ multiplications. Hence this modification can lead up to 6 fold reduction on the number of multiplications on our dataset.

We can further reduce the memory usage by using the fact that $b_{ij} = 1$ if and only if $b_{ji}^T = 1$. We can only store $\mathbf{B}$, and when we read a nonzero b_{ij}, we can do the updates on $\mathbf{p}_t$ both for b_{ij} and b_{ji}^T. By not storing $\mathbf{B}^T$, we reduce the size roughly to 43MB. Furthermore, we actually read two nonzeros when we bring b_{ij} from the memory. However, we still need to do two different updates. A similar optimization has been proposed for some particular SpMxV operations [29]. Algorithm 2 shows the pseudocode of the RWR computation with the modifications described above.

Although the proposed modifications reduce the size of $\mathbf{A}$ and the number of multiplications, there is a drawback. In Algorithm 1, line 1 first checks if $\mathbf{p}_{t-1}(i) > 0$. If this is not the case there is no need to traverse any of the a_{ij}s. This shortcut is especially useful when $\mathbf{p}_{t-1}$ contains only a few positive values which is the case for the first few iterations. However, such a shortcut, as in line 1 of Algorithm 2, only works for nonzeros corresponding to the outgoing edges when the matrix is reduced. That is, if b_{ij} is nonzero Algorithm 2 does the update $\mathbf{p}_t(j) \leftarrow \mathbf{p}_t(j) + \mathbf{sp}_C(i)$ even though $\mathbf{sp}_C(i)$ is zero. Hence, some updates with no effects are done in Algorithm 2, although they are skipped in Algorithm 1.

If $\mathbf{B}$ is stored in COO format one needs roughly 63MB in memory. In this format, the nonzeros are read one by one. Hence, a shortcut for the updates with no effect is not practical. On the other hand, with COO, we have more flexibility for nonzero ordering, and techniques like blocking can be implemented without any overhead. We give the COO based pseudocode in Algorithm 3.

Algorithm 2: RWR with CRS-Half

Input: $\mathbf{B}$: $n \times n$ adjacency matrix in CRS format, seed paper set $\mathcal{M}$, row and column scaling vectors $\mathbf{s}_C$ and $\mathbf{s}_R$
Output: Steady state probability vector $\mathbf{p}$
$\mathbf{p}_t \leftarrow \mathbf{0}$
for each *paper* $i \in \mathcal{M}$ **do**
  $\mathbf{p}_t(i) \leftarrow \frac{1}{|\mathcal{M}|}$
$e \leftarrow \|\mathbf{p}_t\|_2$
while $e > \xi$ **do**
  $\mathbf{sp}_R \leftarrow \mathbf{p}_t * \mathbf{s}_R$
  $\mathbf{sp}_C \leftarrow \mathbf{p}_t * \mathbf{s}_C$
  $\mathbf{p}_{t-1} \leftarrow \mathbf{p}_t$
  $\mathbf{p}_t \leftarrow \mathbf{0}$
  for each *paper* $i = 1$ *to* n **do**
1   **if** $\mathbf{sp}_C(i) > 0$ **then**
        for each *nonzero* b_{ij} *of* $\mathbf{B}$ **do**
          $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \mathbf{sp}_R(j)$
          $\mathbf{p}_t(j) \leftarrow \mathbf{p}_t(j) + \mathbf{sp}_C(i)$
      else
        for each *nonzero* b_{ij} *of* $\mathbf{B}$ **do**
          $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \mathbf{sp}_R(j)$
  for each *paper* $i \in \mathcal{M}$ **do**
    $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \frac{1-d}{|\mathcal{M}|}$
  $e \leftarrow \|\mathbf{p}_t - \mathbf{p}_{t-1}\|_2$
return $\mathbf{p} \leftarrow \mathbf{p}_t$

Algorithm 3: RWR with COO-Half

Input: $\mathbf{B}$: $n \times n$ adjacency matrix in COO format, seed paper set $\mathcal{M}$, row and column scaling vectors $\mathbf{s}_C$ and $\mathbf{s}_R$
Output: Steady state probability vector $\mathbf{p}$
$\mathbf{p}_t \leftarrow \mathbf{0}$
for each *paper* $i \in \mathcal{M}$ **do**
  $\mathbf{p}_t(i) \leftarrow \frac{1}{|\mathcal{M}|}$
$e \leftarrow \|\mathbf{p}_t\|_2$
while $e > \xi$ **do**
  $\mathbf{sp}_R \leftarrow \mathbf{p}_t * \mathbf{s}_R$
  $\mathbf{sp}_C \leftarrow \mathbf{p}_t * \mathbf{s}_C$
  $\mathbf{p}_{t-1} \leftarrow \mathbf{p}_t$
  $\mathbf{p}_t \leftarrow \mathbf{0}$
  for each *nonzero* b_{ij} *of* $\mathbf{B}$ **do**
    $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \mathbf{sp}_R(j)$
    $\mathbf{p}_t(j) \leftarrow \mathbf{p}_t(j) + \mathbf{sp}_C(i)$
  for each *paper* $i \in \mathcal{M}$ **do**
    $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \frac{1-d}{|\mathcal{M}|}$
  $e \leftarrow \|\mathbf{p}_t - \mathbf{p}_{t-1}\|_2$
return $\mathbf{p} \leftarrow \mathbf{p}_t$

IV. Exploiting cache locality in reduced matrix operations

As explained in the previous section, one of the techniques we use for compressing the matrix is to store $\mathbf{B}$, but not $\mathbf{B}^T$. After this modification, when a nonzero b_{ij} is read, $\mathbf{p}_t(i)$ and $\mathbf{p}_t(j)$ are updated accordingly. Hence, when we order $\mathbf{B}$'s rows with a permutation matrix $\mathbf{P}$, we need to use the same $\mathbf{P}$ to order the columns if we want to find the nonzero indices in $\mathbf{B}^T$. Due to this reason, permuting the adjacency matrix as $\mathbf{B}' = \mathbf{P}\mathbf{B}\mathbf{P}^T$ is good practice for our problem. Note that the original SpMxV problem does not have such a restriction. Hence, existing hypergraph-partitioning-based approaches cannot be directly applied for our problem [6], [8], [9]. Note that we can still use symmetric permutations such as the ones obtained by RCM and AMD.

Similar to existing partitioning-based approaches, we use a two-phase permutation strategy which first partitions the rows of $\mathbf{B}$ into K and sorts them in the increasing order of their part numbers. The intra-part row ordering is decided later by using RCM or AMD and the final permutation matrix $\mathbf{P}$ is obtained. Our column-net hypergraph $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ is created with n vertices and n nets corresponding to the rows and columns of $\mathbf{B}$, respectively, as described in Section II-A. In $\mathcal{H}_{\mathcal{R}}$, two vertices v_i and $v_{i'}$ are connected via a net η_j if both b_{ij} and $b_{i'j}$ is equal to 1. To handle the above mentioned restriction of using the same permutation for rows and columns, we set $v_i \in \eta_i$ for all $i \in \{1, \ldots, n\}$. That is, we set all diagonal entries of $\mathbf{B}$, which originally has a zero diagonal, to 1 and partition it. With this modification, a net j can be internal if and only if the pins of j are in the same part with vertex j. Hence, when we permute the rows and columns with respect to the part numbers of the rows, the columns corresponding to the internal nets of a part will be accessed by the rows only in that part.

Since we store the matrix in CRS format, we know that $\mathbf{sp}_C$ is accessed sequentially (even for COO-Half, our nonzero ordering respects to row indices to some degree). Hence, accessing to $\mathbf{p}_t$ and $\mathbf{sp}_R$ with column indices will possibly be the main bottleneck. We use PaToH [18] to minimize $connectivity - 1$ metric (2) and improve cache locality. Throughout the computation, the entry $\mathbf{sp}_R(j)$ will be put to cache at least once assuming the jth column has at least one nonzero in it. If column j is internal to part ℓ then $\mathbf{sp}_R(j)$ will be only accessed by the rows within part ℓ. Since the internal columns of each part are packed close in the permutation, when $\mathbf{sp}_R(j)$ is put to the cache, the other entries of $\mathbf{sp}_R$ which are part of the same cache line are likely to be internal columns of the same part. On the other hand, when an external column j is accessed by a part ℓ' which is not the part of j, the cache line containing $\mathbf{sp}_R(j)$ is unlikely to contain entries used by the rows in part ℓ'. Minimizing the $connectivity - 1$ metric equals to minimizing the number of such accesses. Note that the same is true for the access of $\mathbf{p}_t$ with column indices.

We find intra-part row/column orderings by using RCM and AMD. These heuristics have previously been used for fill-in minimization in sparse LU factorization. RCM is used to find a permutation σ which reduces the bandwith of a symmetric matrix $\mathbf{A}$ where the bandwith is defined as $b = \max(\{|\sigma(i) - \sigma(j)| : a_{ij} \neq 0\})$. When the bandwith is small, the entries are close to the diagonal, and the cache locality will be high. The AMD heuristic also has the same motivation of minimizing the number of fill-ins, which usually densifies nonzeros in different parts of the matrix. Since having nonzeros closed to each other is good for cache locality, we used these heuristics to order rows and columns inside each part.

For all of the algorithms described in Section III, we used the proposed ordering scheme. For CRS-Full, we permutated $\mathbf{A}$, and for CRS-Half and COO-Half, we permuted $\mathbf{B}$ as described above. For COO-Half, we also apply blocking after permuting $\mathbf{B}$: we divide $\mathbf{B}$ into square blocks of size 2048×2048 and traverse the nonzeros with respect to their block ids (and rowwise within a block). The block size is tuned on the architecture the**advisor** is running on.

V. Experimental Results

We used three different architectures to test the algorithms. The main (target) one has a 2.4GHz *AMD Opteron* CPU and 4GB of main memory. The CPU has 64KB L1 and 1MB L2 caches. Our service, the**advisor**, is currently running on a cluster with 50 nodes each having the above mentioned architecture. For each query, the service opens a socket to a running process, submits the query, and returns the results to the user.

For completeness, we also test the algorithms on two other more recent architectures. The second architecture has a 2.27GHz quad-core *Intel Xeon* (Bloomfield) CPU and 48GB of main memory. Each core has 32KB L1 and 256KB L2 caches and each socket has an 8MB L3 cache. The third architecture has a 2.4GHz quad-core *AMD Opteron* (Shanghai) CPU and 32GB of main memory. Each core has 64KB L1 and 512KB L2 cache and each socket has a 6MB L3 cache. All of the algorithms are implemented in C++. The compiler is icc version 11.1 and the -O2 optimization flag is used. For the experiments, we use only one core from each processor.

We generated 70 queries where each query is a set $\mathcal{M}$ of paper ids obtained from the bibliography files submitted by the users of the service who agreed to donating their queries for research purposes. The number of seed nodes $|\mathcal{M}|$ of the queries vary between 1 and 72, with an average of 17.9. For RWR, we use $d = 0.8$ and $\kappa = 0.75$ which are the default values in the**advisor**. We did not use a threshold ξ for convergence. We observed that RWR in our citation graph takes approximately 20 iterations to converge. Because computing the error between iterations takes some time and to be consistent in the experiments, we let the algorithms iterate 20 times. While generating the partitions, we set the imbalance ratio of PaToH to 0.4.

As mentioned in Section III, the algorithms CRS-Full and CRS-Half avoid some updates but COO-Half cannot, even they have no effect on $\mathbf{p}_t$. In our query set, the average number of seed papers is 17.9. In the first iteration, $\mathbf{p}_{t-1}$ has only

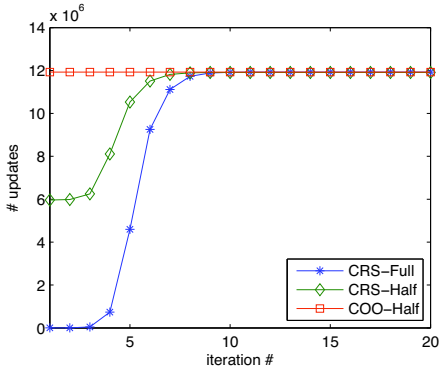


Fig. 1. Number of updates per iteration of the algorithms.

a few positive values on average, and CRS-Full updates $\mathbf{p}_t$ only for the corresponding papers in $\mathcal{M}$. Since $n \gg 17.9$, CRS-Full avoids roughly 12 million nonzeros/updates in the first iteration. This number is roughly 6 million for CRS-Half. COO-Half traverses all 12 million nonzeros and does the corresponding updates even if nearly all of them have no effect. However, the number of positive values in $\mathbf{p}_{t-1}$ increases exponentially. As Figure 1 shows, the shortcuts in CRS-based algorithms are not useful after the 8th iteration. The figure also implies that the citation graph is highly connected since the algorithms seem to traverse almost all the nonzeros in $\mathbf{A}$. That is, random walks can reach to almost all vertices in the graph. We observed that 97% of the vertices of the citation graph G are in a single connected component.

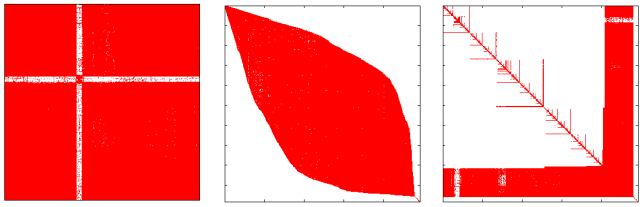
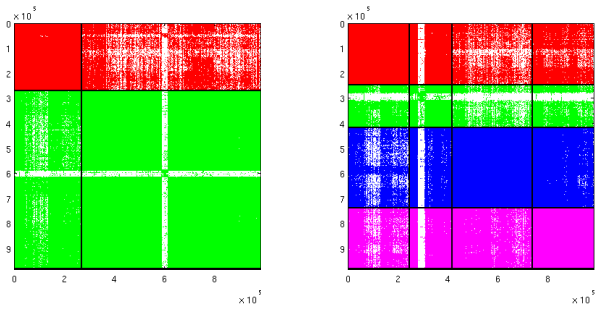
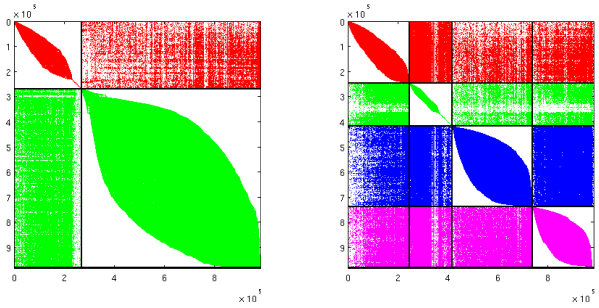


Fig. 2. The nonzero pattern of $\mathbf{B}$ (left) when ordered with RCM (middle) and AMD (right). Nonzeros are colored with red and white areas show empty regions.

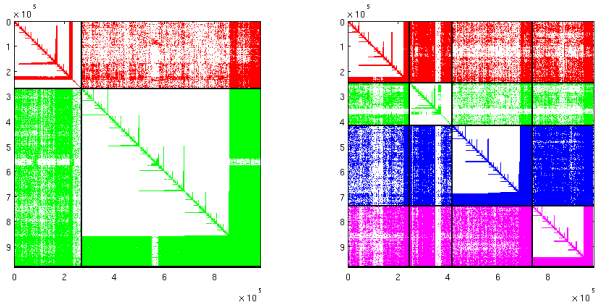
The nonzero pattern of the adjaceny matrix $\mathbf{B}$ is given in Figure 2 (left). As the figure shows, the nonzeros are distributed in all matrix. In our experiments, the papers are originally numbered with respect to the order we parse their metadata. When $\mathbf{B}$ is ordered by using the RCM heuristic, the nonzero pattern (middle) is densified near the diagonal as expected. The bandwidths of the original and RCM ordered $\mathbf{B}$ matrices are 981287 and 460288, respectively. Although the bandwidth is reduced more than half, it is still large. Figure 2 (right) shows the nonzero pattern of $\mathbf{B}$ when ordered with the AMD heuristic. The nonzeros are densified inside one horizontal and one vertical block. We observed that 80% of the nonzeros are inside this region. As the figure shows, the remaining nonzeros are located in smaller horizontal and vertical regions which also may be helpful to reduce the



(a) $\mathbf{B}'$ for $K = 2$ (left) and 4 (right)



(b) $\mathbf{B}'$ with RCM for $K = 2$ (left) and 4 (right)



(c) $\mathbf{B}'$ with AMD for $K = 2$ (left) and 4 (right)

Fig. 3. The nonzero pattern of the permuted adjacency matrix $\mathbf{B}'$ with different partitions and reordering heuristics. The row set of each part is shown with a different color. The diagonal blocks contain b_{ij}s where row i and row j of $\mathbf{B}$ are in the same part.

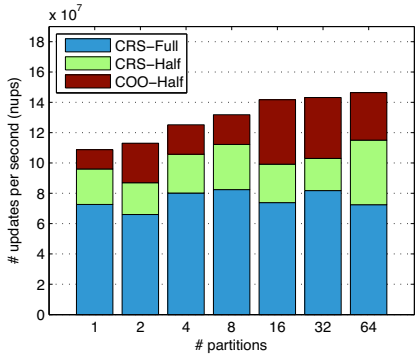
number of cache misses.

As described in Section IV, we first partition $\mathbf{B}$ in the column-net model to reorder it. To do that, we use $K = \{2, 4, 8, 16, 32, 64\}$ and create 6 different partitions. For each partition, we create a permutation matrix $\mathbf{P}$ and reorder $\mathbf{B}$ as $\mathbf{B}' = \mathbf{PBP}^T$. Figure 3(a) shows the structure of the nonzero pattern of $\mathbf{B}'$ for $K = 2$ and 4. In the figure, the horizontal (vertical) lines separate the rows (columns) of the matrix w.r.t. their part numbers. The diagonal blocks in the figure contain the nonzeros b_{ij}s where the ith and jth row of $\mathbf{B}$ are assigned to the same part. Although the nonzeros seem evenly distributed in $\mathbf{B}'$, as Table II shows, for $K = 2$ and 4, the percentage of the nonzeros inside diagonal blocks of $\mathbf{B}'$ are 94% and 83%, respectively. We can argue that the nonzeros are densified inside these blocks as expected.

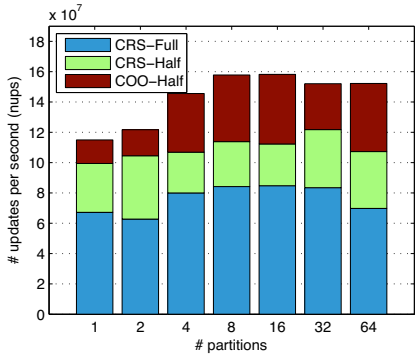
We permute the rows and columns of the diagonal blocks

TABLE II
NUMBER OF NONZEROS INSIDE AND OUTSIDE OF THE DIAGONAL BLOCKS OF $\mathbf{B}'$ AFTER REORDERING.

K	2	4	8	16	32	64
nnz in	5.62M	4.92M	4.39M	3.80M	3.46M	2.95M
nnz out	0.34M	1.04M	1.57M	2.16M	2.50M	3.01M



(a) RCM



(b) AMD

Fig. 4. Number of updates per second for each algorithm and ordering.

by using the ordering heuristics RCM and AMD. Figure 3(b) and (c) show the nonzero patterns of this further permuted matrix for $K = 2$ and 4.

Figure 4(a) and (b) show the number of updates per seconds (*nups*) for each algorithm when RCM and AMD heuristics are used, respectively. This experiment counts the number of updates that occur in memory, even if they are nilpotent. When the matrix is ordered with RCM, the partitioning has a positive effect on *nups* of CRS-Full which increases from around 7×10^7 with no partitioning to over 8×10^7 with $K = 8$. Although the number of nonzeros outside the diagonal blocks is more when K increases, the diagonal blocks tend to get smaller and denser which may improve the cache locality. We tested the algorithms with different K values to find the best configuration. For example, when K increases for RCM, the maximum *nups* is obtained with $K = 64$. Such an improvement is also observed when AMD is used. Compressing the matrix increases the *nups* for CRS-Half up to 12×10^7 with $K = 32$. And with blocking used in COO-Half, *nups* increases to 16×10^7 with $K = 8$ which is

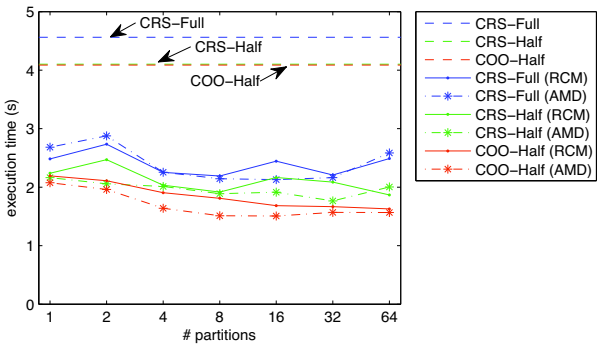


Fig. 5. Execution times in seconds for each algorithm with different Ks and ordering heuristics on *AMD Opteron* architecture with 1MB cache. The values are the averages of the times for 70 queries.

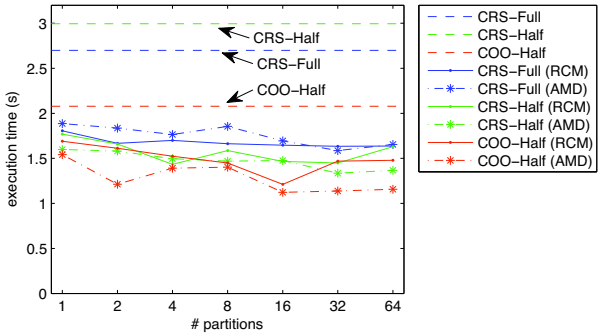


Fig. 6. Execution times in seconds for each algorithm with different Ks and ordering heuristics on *AMD Opteron* (Shanghai) architecture with 6MB cache. The values are the averages of the times for 70 queries.

the maximum for this experiment.

Figure 5 shows the execution times of the algorithms for different Ks and ordering heuristics on the target architecture with 1MB L2 cache. As concordant with Figure 4, the fastest algorithm is COO-Half where $K = 8$ and the diagonal blocks are ordered with AMD. The average query response time for this configuration, which is being used in the**advisor**, is 1.51 seconds. Compared with the execution time of CRS-Full with the original ordering, which is 4.55 seconds, we obtain 3 times improvement. When $K = 1$, i.e., if there is no partitioning, the execution time of COO-Half is 2.08. Hence, we obtain 25% improvement due to partitioning.

We tested our modifications also on two other architectures described above. On *AMD Shanghai*, the results are similar as shown in Figure 6. COO-Half is the fastest algorithm, this time with $K = 16$ and the AMD heuristic. However, matrix compression seems to have a negative effect on CRS-Half. Its execution time is more than CRS-Full with the original ordering. This is unexpected since as shown in Figures 5 and 7, both on the target architecture and *Intel Xeon*, CRS-Half is much faster than CRS-Full. On the *Intel* architecture, the fastest algorithm is again COO-Half where the average query response time is 0.68 seconds with $K = 4$ and the AMD heuristic. Compared to the time of CRS-Full with no ordering, which is 1.30 seconds, the improvement is 42%. If

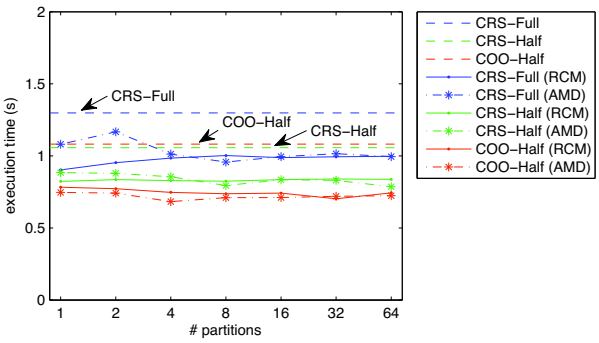


Fig. 7. Execution times in seconds for each algorithm with different Ks and ordering heuristics on *Intel Xeon* architecture with 8MB cache. The values are the averages of the times for 70 queries.

we apply only matrix compression and blocking with COO-Half, the query response time is 1.07 seconds. Hence, we can argue that we obtained 36% improvement by permuting the reduced matrix. If we only use AMD, i.e., when $K = 1$, the query response time of COO-Half is 0.75. This shows roughly 10% improvement due to partitioning alone. Since *Intel Xeon*'s cache is larger than the others, we believe that when the matrix gets large, i.e., when the number of papers in our database increases, the improvements will be much higher on all architectures but especially on *Intel Xeon*.

VI. Conclusion and Future Work

In this paper, we proposed an efficient implementation of an SpMxV-type problem which arises in our publicly available citation, venue, and expert recommendation service, the**advisor**. We proposed compression and bandwidth reduction techniques to reduce the memory usage and hence, the bandwidth required to bring the matrix from the memory at each iteration. We also used matrix ordering techniques to reduce the number cache misses. Experimental results show these modifications greatly help to reduce the query execution time.

As a future work, we are planning to develop new ideas to further reduce the query response time. As far as the service is running, this will be one of the tasks we are interested in. Note that in SpMxV operations, it is very hard to obtain linear speedup with shared memory parallelization. Hence, to maximize the throughput we chose to use one processor per query. However, we believe that such parallelism can still be effective for the**advisor** especially when the number concurrent requests is less than the number of processors allocated in the cluster.

Acknowledgment

This work was supported in parts by the DOE grant DE-FC02-06ER2775 and by the NSF grants CNS-0643969, OCI-0904809, and OCI-0904802.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, TR 1999-66, 1999.

[2] J. Bollen, M. A. Rodriguez, and H. V. de Sompel, "Journal status," *Scientometrics*, vol. 69, no. 3, pp. 669–687, 2006.

[3] H.-N. Kim and A. El Saddik, "Personalized PageRank vectors for tag recommendations: inside FolkRank," in *Proc. ACM Recommender Systems*, 2011, pp. 45–52.

[4] Z. Yin, M. Gupta, T. Weninger, and J. Han, "A unified framework for link recommendation using random walks," in *Proc. ASONAM*, 2010, pp. 152–159.

[5] K. C. Chipman and A. K. Singh, "Predicting genetic interactions with random walks on biological networks," *BMC Bioinformatics*, vol. 17, no. 10, 2009.

[6] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph-partitioning-based models and methods for exploiting cache locality in sparse-matrix vector multiplication," *CoRR*, vol. abs/1202.3856, 2012.

[7] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication," *IBM J.Res.Dev.*, vol. 41, no. 6, pp. 711–726, 1997.

[8] A. N. Yzelman and R. H. Bisseling, "Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods," *SIAM J. Sci. Comput.*, vol. 31, pp. 3128–3154, 2009.

[9] ——, "Two-dimensional cache-oblivious sparse matrix-vector multiplication," *Parallel Comput.*, vol. 37, pp. 806–819, 2011.

[10] J. B. White and P. Sadayappan, "On improving the performance of sparse matrix-vector multiplication," in *Proc. HIPC*, 1997.

[11] R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A high performance algorithm using pre-processing for the sparse matrix-vector multiplication," in *Proc. ACM/IEEE Supercomputing*, 1992, pp. 32–41.

[12] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera, "Performance optimization of irregular codes based on the combination of reordering and blocking techniques," *Parallel Comput.*, vol. 31, no. 8+9, pp. 858–876, 2005.

[13] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proc. ACM/IEEE Supercomputing*, 1999.

[14] O. Temam and W. Jalby, "Characterizing the behavior of sparse algorithms on caches," in *Proc. ACM/IEEE Supercomputing*, 1992.

[15] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM J. Matrix Anal. Appl.*, vol. 17, no. 4, pp. 886–905, 1996.

[16] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proc. ACM national conference*, 1969, pp. 157–172.

[17] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Willey–Teubner, 1990.

[18] Ü. V. Çatalyürek and C. Aykanat, *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*, Bilkent University, Computer Eng., Ankara, Turkey. Available at http://bmi.osu.edu/ umit/software.htm, 1999.

[19] ——, "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication," *IEEE T. Parall. Distr.*, vol. 10, pp. 673–693, 1999.

[20] ——, "A fine-grain hypergraph model for 2D decomposition of sparse matrices," in *IPDPS 15*, 2001.

[21] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera, "Increasing data reuse of sparse algebra codes on simultaneous multithreading architectures," *Concurr. Comput.: Pract. Exper.*, vol. 21, no. 15, pp. 1838–1856, 2009.

[22] M. M. Kessler, "Bibliographic coupling between scientific papers," *American Documentation*, vol. 14, pp. 10–25, 1963.

[23] H. Small, "Co-citation in the scientific literature: A new measure of the relationship between two documents," *J. Am. Soc. Inf. Sci.*, vol. 24, no. 4, pp. 265–269, 1973.

[24] S. Lawrence, C. L. Giles, and K. Bollacker, "Digital libraries and autonomous citation indexing," *Computer*, vol. 32, pp. 67–71, 1999.

[25] D. Liben-Nowell and J. M. Kleinberg, "The link-prediction problem for social networks," *JASIST*, vol. 58, no. 7, pp. 1019–1031, 2007.

[26] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu, "Automatic multimedia cross-modal correlation discovery," in *Proc. ACM SIGKDD*, 2004.

[27] M. Gori and A. Pucci, "Research paper recommender systems: A random-walk based approach," in *Proc. IEEE/WIC/ACM Web Intelligence*, 2006.

[28] J. Li and P. Willett, "ArticleRank: a PageRank-based alternative to numbers of citations for analyzing citation networks," *Proc. ASLIB*, vol. 61, no. 6, 2009.

[29] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proc. IPDPS*, 2011, pp. 721–733.