Soc. Netw. Anal. Min. (2013) 3:1097–1111
DOI 10.1007/s13278-013-0106-z

ORIGINAL ARTICLE

Fast recommendation on bibliographic networks with sparse-matrix ordering and partitioning

Onur Küçüktunç · Kamer Kaya · Erik Saule ·
Ümit V. Çatalyürek

Received: 1 December 2012 / Revised: 18 February 2013 / Accepted: 23 February 2013 / Published online: 20 March 2013

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 the**advisor** as well as in many other recommendation systems. 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 we reduced the number of cache misses with ordering and partitioning. Experimental results show that our techniques are highly efficient in reducing the query processing time which is highly crucial for a web service.

O. Küçüktunç (✉)
Department of Computer Science and Engineering,
The Ohio State University, Columbus, OH, USA
e-mail: kucuktunc.1@osu.edu

O. Küçüktunç · K. Kaya · E. Saule · Ü. V. Çatalyürek
Department of Biomedical Informatics,
The Ohio State University, Columbus, OH, USA

K. Kaya
e-mail: kamer@bmi.osu.edu

E. Saule
e-mail: esaule@bmi.osu.edu

Ü. V. Çatalyürek
e-mail: umit@bmi.osu.edu

Ü. V. Çatalyürek
Department of Electrical and Computer Engineering,
The Ohio State University, Columbus, OH, USA

Keywords Citation recommendation · Social network analysis · Sparse matrices · Hypergraphs · Cache locality · Ordering · Partitioning

1 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, or 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 network analyses based on a random walk with restart (RWR) which is used in PageRank (Page et al. 1999), impact factor computations (Bollen et al. 2006), recommendation systems (Kim and El Saddik 2011; Yin et al. 2010) and finding/predicting genetic interactions (Chipman and Singh 2009). The SpMxV kernel is also the core of other graph-based metrics such as Katz which is proposed by Liben-Nowell and Kleinberg (2007) for a study on the link prediction problem on social networks and used later for information retrieval purposes including citation recommendation by Strohman et al. (2007).

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 *query* 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 researchers while performing several tasks, such as:

- literature search,
- improving the reference list of a manuscript being written,
- finding conferences and journals to attend, get subscribed, or submit papers,
- 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 employed in the**advisor** is based on RWR and is implemented by using the SpMxV operation. There exist several methods in the literature proposed to improve the cache locality for the SpMxV operations by ordering the rows and/or columns of the matrix by using graph/hypergraph partitioning (Akbudak et al. 2012; Toledo 1997; White and Sadayappan 1997; Yzelman and Bisseling 2009, 2011) and other techniques (Agarwal et al. 1992; Pichel et al. 2005; Pinar and Heath 1999; Temam and Jalby 1992). 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 (Akbudak et al. 2012; Yzelman and Bisseling 2009, 2011) not directly applicable. We recently experimented on a direction-aware Katz-based algorithm and showed that it outperforms one without direction awareness when the objective is to find either traditional or recent papers (Küçüktunç et al. 2012a, b).

In this paper, our contribution is twofold: first, we propose techniques to efficiently store the matrix used by direction-aware algorithms. We then propose efficient implementations 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) (Amestoy et al. 1996), reverse Cuthill-McKee (RCM) (Cuthill and McKee 1969), and SlashBurn (Kang and Faloutsos 2011). 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 once. The current version of our service is already using the implementation and ordering described in this paper.

We give a thorough evaluation of the proposed approach and algorithms, and measure the efficiency of the 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 % (3×). We believe that the techniques proposed here can also be useful for SpMxV-related sparse-matrix problems in social network analysis.

A preliminary version of this work was published in Küçüktunç et al. (2012a, b). We extend in this paper the discussion to a wider class of algorithms and exemplify the discussion by using Katz-based metrics. We also investigate the SlashBurn ordering. We discuss the impact of the convergence of the method on the choice of representation and ordering of the matrix.

The paper is organized as follows: In Sect. 2, we describe the hypergraph partitioning problem and existing matrix ordering techniques for SpMxV. In Sect. 3, we formally describe our direction-aware RWR and Katz algorithms and their efficient implementation where the former RWR-based algorithm is used in the**advisor**. The ordering techniques we use are given in Sect. 4. Section 5 gives the experimental results and Sect. 6 concludes the paper.

2 Background

2.1 Citation analysis: random walk with restart and Katz

Citation analysis-based paper recommendation has been a popular problem since the 1960s. There are methods that only take local neighbors (i.e., citations and references) into account, e.g., bibliographic coupling (Kessler 1963), cocitation (Small 1973), and CCIDF (Lawrence et al. 1999). Recent studies, however, employ graph-based algorithms, such as Katz Liben-Nowell and Kleinberg (2007), random walk with restart (Pan et al. 2004), or well-known PageRank algorithm to investigate the whole citation network. PaperRank (Gori and Pucci 2006), ArticleRank (Li and Willett 2009), and Katz distance-based methods (Liben-Nowell and Kleinberg 2007) are typical examples.

We target the problem of paper recommendation assuming that the researcher has already collected a list of

[1] http://the**advisor**.osu.edu/.

papers of interest. Let $G = (V, E)$ be the directed citation graph with a vertex set $V = \{1, 2, \ldots, n\}$ and an edge set E, which contains (i, j) if paper i cites paper j. We define the problem as follows: given a set of query papers $\mathcal{Q} \subseteq V$, and a parameter k, return top-k papers which are relevant to the ones in $\mathcal{Q}$.

Let E' be the undirected version of E, i.e., $E' = \{\{i,j\} : (i,j) \in E\}$, and $G' = (V, E')$ be the undirected citation graph and $\delta(i)$ denote the degree of a vertex $i \in V$ in G'. Random walk with restart is a widely used method in many fields. In citation analysis, RWR directs the random walks towards both references and citations of the papers. In addition, the restarts are directed only to the query papers in $\mathcal{Q}$. Hence, random jumps to any paper in the literature are prevented. Starting from the query papers, we assume that a random walk ends in paper i continues with a neighbor with a damping factor $d \in (0, 1]$. And with probability $(1 - d)$, it restarts and goes back to the query papers. Let $\mathbf{p}_t(i)$ be the probability that a random walk ends at vertex $i \in V$ at iteration t. Hence, $\mathbf{p}_0(i) = \frac{1}{|\mathcal{Q}|}$ for a query paper i, and 0 for other papers. Let $c_t(i)$ be the contribution of i to one of its neighbors at iteration t. In each iteration, d of $\mathbf{p}_{t-1}(i)$ is distributed among i's references and citations equally, hence, $c_t(i) = d\frac{\mathbf{p}_{t-1}(i)}{\delta(i)}$.

The Katz distance (Liben-Nowell and Kleinberg 2007) is another measure which has been used for citation recommendation purposes (Strohman et al. 2007). The Katz distance between two papers $i, j \in V$ is computed as:

$$\text{Katz}(i,j) = \sum_{\ell=1}^{\infty} \beta^{\ell} |\text{paths}_{i,j}^{\ell}|, \tag{1}$$

where $\beta \in [0, 1]$ is the decay parameter, and $|paths_{i,j}^{\ell}|$ is the number of paths of length ℓ between i and j in the undirected citation graph G'. Such a path does not need to be elementary, i.e., the path i, j, i, j is a valid path of length 3. Therefore, the Katz measure might not converge for all values of β; it needs to be chosen smaller than the reciprocal of the largest eigenvalue of the adjacency matrix of G'.[2] In citation recommendation with multiple query papers, the relevance of a paper j is computed as $\pi(j) = \sum_{i \in \mathcal{Q}} \text{Katz}(i,j)$.

2.2 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) $\mathcal{N}$. A net $\eta \in \mathcal{N}$ is a subset of vertex set $\mathcal{V}$, 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. Figure 1a shows a simple hypergraph with five vertices and five nets. A graph is a special instance of hypergraph such that each net has size two. Vertices can be associated with weights, denoted by $\mathsf{w}[\cdot]$, and nets can be associated with costs, denoted by $\mathsf{c}[\cdot]$.

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 the 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., $\lambda_\eta > 1$). In Fig. 1a the toy hypergraph with four internal nets and an external net is partitioned into two.

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*

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

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

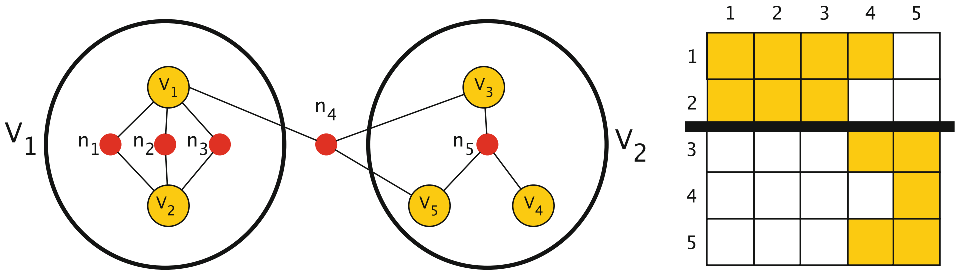
The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. Let $\chi(\Pi)$ denote the cost, i.e., *cutsize*, of a partition Π. There are various cutsize definitions (Lengauer 1990). In this work, we use

$$\chi_{\text{conn}}(\Pi) = \sum_{\eta \in \mathcal{N}} \mathsf{c}[\eta](\lambda_\eta - 1)\ , \tag{3}$$

as also used by Akbudak et al. (2012) for similar purposes. The cutsize metric given in (3) will be referred to as the *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 (Lengauer 1990) with the above objective function. We used a state-of-the-art partitioning tool PaToH (Çatalyürek and Aykanat 1999).

There are three well-known hypergraph models for sparse matrices. These are the column-net (Çatalyürek and Aykanat 1999), row-net (Çatalyürek and Aykanat 1999), and fine-grain models (Çatalyürek and Aykanat 2001). 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

[2] The Katz centrality of a node i can be computed as $\text{Katz}(i) = \sum_{j=1}^{n} \text{Katz}(i,j) = \sum_{j=1}^{n} \sum_{\ell=1}^{\infty} \beta^{\ell} (A^{\ell})_{ji}$ where A is the 0–1 adjacency matrix of the citation graph. When β is smaller than the reciprocal of the largest eigenvalue, the Katz centralities can be computed as $((I - \beta A^T)^{-1} - I)\vec{I}$ where I is the identity matrix and $\vec{I}$ is the identity vector.



(a) A toy hypergraph **(b)** Corresponding matrix

Fig. 1 A toy hypergraph with five vertices and five nets partitioned into two parts (*left*). Net n_4 is a cut net since it is connected to two parts, hence, $\lambda_4 = 2$. The rest of the nets are internal. The corresponding matrix (w.r.t. column-net model) whose nonzero entries are colored and zeros are shown in white (*right*)

column-net model, **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 $a_{ij} \neq 0$ (see Fig. 1). The *row-net model* is the column-net model of the transpose of **A**.

2.3 Matrix ordering techniques for improving cache locality in SpMxV

The SpMxV operation is defined as $\mathbf{y} \leftarrow \mathbf{Ax}$ where **A** is an $n \times n$ sparse matrix with m nonzeros, **x** is the $n \times 1$ input vector, and **y** is the $n \times 1$ output vector. Let **P** and **Q** be two $n \times n$ permutation matrices. That is, **P** and **Q** have only a 1 in each of their rows and columns and the rest is 0. When the matrix **A** is ordered as $\mathbf{A}' = \mathbf{PAQ}$, the SpMxV operation can be written as $\mathbf{y}' \leftarrow \mathbf{A}'\mathbf{x}'$ where $\mathbf{y}' = \mathbf{Py}$ and $\mathbf{x}' = \mathbf{Q}^T\mathbf{x}$. Some existing cache-locality optimization techniques use this fact and permute the rows and columns of **A** to improve cache locality.

To find good **P** and **Q**, several approaches are proposed in the literature: bandwidth reduction is proven to be promising for decreasing cache misses (Temam and Jalby 1992). For this reason, the reverse Cuthill-McKee heuristic (Cuthill and McKee 1969) has been frequently used as a tool and a benchmark by several researchers (Pichel et al. 2009; Pinar and Heath 1999; Toledo 1997). RCM has also been frequently used as a fill-in minimization heuristic for sparse LU factorization. Another successful fill-in minimization heuristic, the approximate minimum degree (AMD) (Amestoy et al. 1996), is also used for improving cache locality (Pichel et al. 2009). Another ordering heuristic SlashBurn has recently been proposed for graph compression and mining (Kang and Faloutsos 2011).

Graph and hypergraph partitioning models and techniques have been extensively studied for reducing cache misses (Akbudak et al. 2012; Toledo 1997; White and Sadayappan 1997; Yzelman and Bisseling 2009, 2011). Among those, the most similar ones to our work are Yzelman and Bisseling (2009, 2011) and Akbudak et al. (2012), 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 since the storage determines the order in which the nonzeros are processed. 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 **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 **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.

3 Direction-aware methods for citation recommendation

As described previously, our recommendation service, the**advisor**, is designed to solve the following problem: given the directed citation graph $G = (V, E)$, a set of query papers $\mathcal{Q} \subseteq V$, and a parameter k, return top-k papers which are relevant to the ones in $\mathcal{Q}$. We defined a *direction awareness* parameter $\kappa \in [0, 1]$ to obtain more recent or traditional results in the top-k documents (Küçüktunç et al. 2012a, b). Let $\delta^+(i)$ and $\delta^-(j)$ be the number of references of and citations to paper u, respectively. The citation graph we use in the**advisor** has been obtained by cross-referencing the data of four online databases: DBLP, CiteSeer, HAL-Inria and arXiv. The properties of the citation graph are given in Table 1.

Table 1 Statistics for the citation graph G

$\|V\|$	$\|E\|$	avg δ	max δ^+	max δ^-
982,067	5,964,494	6.07	617	5,418

In this work, we discuss efficient ways to compute the result set using two *direction-aware* algorithms. The first one is based on the direction-aware random walk with restart (DaRWR) and the second one is based on the direction-aware Katz similarity (DaKatz). The following sections present both methods by addressing their similarities and differences when they are implemented with SpMxV operations.

3.1 Direction-aware Random walk with restart (DaRWR)

Given $G = (V, E)$, k, $\mathcal{Q}$, d, and the direction awareness parameter κ, our algorithm computes the steady-state probability vector $\mathbf{p}$. For an iterative DaRWR implementation, at iteration t, the two types of contributions of paper i to a neighbor paper are defined as:

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

$$c_t^-(i) = \mathbf{p}_{t-1}(i)\frac{d\kappa}{\delta^-(i)}, \tag{5}$$

where $c_t^+(i)$ 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,

$$\mathbf{p}_t(i) = \mathbf{r}(i) + \sum_{(i,j)\in E} c_t^-(j) + \sum_{(j,i)\in E} c_t^+(j), \tag{6}$$

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

$$\mathbf{r}(i) = \begin{cases} \frac{1-d}{|\mathcal{Q}|}, & \text{if } i \in \mathcal{Q} \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

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

$$\mathbf{p}_t = \mathbf{r} + \mathbf{A}\mathbf{p}_{t-1}, \tag{8}$$

where $\mathbf{r}$ is an $n \times 1$ restart probability vector calculated with (7), and $\mathbf{A}$ is a structurally symmetric $n \times n$ matrix of edge weights, such that

$$a_{ij} = \begin{cases} \frac{d(1-\kappa)}{\delta^+(i)}, & \text{if } (i,j) \in E \\ \frac{d\kappa}{\delta^-(i)}, & \text{if } (j,i) \in E \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

The algorithm converges when the probability of the papers are stable, i.e., when the process is in a *steady state*. Let $\Delta_t = (\mathbf{p}_t(1) - \mathbf{p}_{t-1}(1), \ldots, \mathbf{p}_t(n) - \mathbf{p}_{t-1}(n))$ 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,

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

3.2 Direction-aware Katz (DaKatz)

The direction awareness can be also adapted to other similarity measures such as the graph-based Katz distance measure (Liben-Nowell and Kleinberg 2007). We extend the measure to weight the contributions to references and citations differently with the κ parameter as in DaRWR. Given $G = (V, E)$, $\mathcal{Q}$, κ, and an integer parameter L, the relevance score of paper j is computed as:

$$\mathbf{p}(j) = \sum_{i\in\mathcal{Q}} d_L(i,j), \tag{11}$$

where $d_L(i, j)$ is the direction aware Katz distance between a query paper i and paper j with the paths of length up to L, computed recursively as:

$$d_L(i,j) = \beta\kappa \sum_{(k,j)\in E} d_{L-1}(i,k) + \beta(1-\kappa) \sum_{(j,k)\in E} d_{L-1}(i,k), \tag{12}$$

with the stopping case $d_0(i,i) = 1$ if $i \in \mathcal{Q}$, and 0 otherwise.

The structure of the DaKatz computation is very similar to the one of DaRWR, therefore, it can be efficiently implemented with SpMxV operations as follows: the scores are decayed and passed to the references and citations, rather than distributed among them, hence:

$$a_{ij} = \begin{cases} \beta(1-\kappa), & \text{if } (i,j) \in E \\ \beta\kappa, & \text{if } (j,i) \in E \\ 0, & \text{otherwise,} \end{cases} \tag{13}$$

is used to build the structurally symmetric $n \times n$ transition matrix $\mathbf{A}$. There are no jumps to the query vertices; therefore, the linear equation in (8) is simplified to:

$$\mathbf{p}_t = \mathbf{A}\mathbf{p}_{t-1}, \tag{14}$$

where the Katz distance is initialized with $\mathbf{p}_0(i) = 1$ if $i \in \mathcal{Q}$, and 0 otherwise. The final relevance score $\mathbf{p}(i)$ of a vertex i aggregates all Katz distances with each path length up to L, i.e.,

$$\mathbf{p}(i) = \sum_{t=1}^{L} \mathbf{p}_t(i). \tag{15}$$

Even though the relevance scores $\mathbf{p}(i)$ monotonically increase with each iteration, the algorithm still converges because of the *decay* parameter β.

3.3 Implementations with standard CRS (CRS-Full)

Assume that **A** is stored in CRS format and let A_{i*} be the ith row of **A**. Algorithms 1 and 2 show the pseudocodes of DaRWR and DaKatz where (8) or (14) is computed at each iteration, respectively. Colored lines are used in order to distinguish the differences between the two computations.

Algorithm 1: DaRWR with CRS-Full

```
Input: $n \times n$ transition matrix $\mathbf{A}$ in
       CRS format, query paper set $\mathcal{Q}$
Output: relevance vector $\mathbf{p}$
$\mathbf{p}_t \leftarrow \mathbf{0}$
$\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow 1/|\mathcal{Q}|$
$e \leftarrow \|\mathbf{p}_t\|_2$
while $e > \xi$ do
    $\mathbf{p}_{t-1} \leftarrow \mathbf{p}_t$
    $\mathbf{p}_t \leftarrow \mathbf{0}$
    foreach paper $i = 1$ to $n$ do
        if $\mathbf{p}_{t-1}(i) > 0$ then
            foreach $a_{ij} \neq 0$ in $\mathbf{A}_{i*}$ do
                $\mathbf{p}_t(j) \leftarrow \mathbf{p}_t(j) + a_{ij}\mathbf{p}_{t-1}(i)$
    $\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + (1-d)/|\mathcal{Q}|$
    $e \leftarrow \|\mathbf{p}_t - \mathbf{p}_{t-1}\|_2$
return $\mathbf{p} \leftarrow \mathbf{p}_t$
```

Algorithm 2: DaKatz with CRS-Full

```
Input: $n \times n$ transition matrix $\mathbf{A}$ in
       CRS format, query paper set $\mathcal{Q}$
Output: relevance vector $\mathbf{p}$
$\mathbf{p}_t \leftarrow \mathbf{0}$   $\mathbf{p}_{total} \leftarrow \mathbf{0}$
$\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow 1$
$e \leftarrow \|\mathbf{p}_t\|_2$
while $e > \xi$ do
    $\mathbf{p}_{t-1} \leftarrow \mathbf{p}_t$
    $\mathbf{p}_t \leftarrow \mathbf{0}$
    foreach paper $i = 1$ to $n$ do
        if $\mathbf{p}_{t-1}(i) > 0$ then
            foreach $a_{ij} \neq 0$ in $\mathbf{A}_{i*}$ do
                $\mathbf{p}_t(j) \leftarrow \mathbf{p}_t(j) + a_{ij}\mathbf{p}_{t-1}(i)$
    $\mathbf{p}_{total} \leftarrow \mathbf{p}_{total} + \mathbf{p}_t$
    $e \leftarrow \|\mathbf{p}_t - \mathbf{p}_{t-1}\|_2$
return $\mathbf{p} \leftarrow \mathbf{p}_{total}$
```

To compute (8) or (14), one needs to read all of **A** at each iteration. Note that for each nonzero in **A**, there is a possible update on $\mathbf{p}_t$. As described above, **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 **A** in memory is roughly 147MB.

3.4 Implementations with halved CRS (CRS-Half)

Here, we propose two modifications to reduce **A**'s size and the number of multiplications required to update $\mathbf{p}_t$. The first modification is compressing the nonzeros in **A**: we know that during an iteration, the contributions of paper i to the papers in its reference list are all equal to $c_t^-(i)$. Similarly, the contributions of i to the papers which cite i are equal to $c_t^+(i)$. Let $\mathbf{s}_R$ and $\mathbf{s}_C$ be the row and column scaling vectors defined for DaRWR and DaKatz as:

$$\text{(DaRWR)} \quad \mathbf{s}_R(i) = \begin{cases} \frac{d(1-\kappa)}{\delta^+(i)} & \text{if } \delta^+(i) > 0 \\ 0 & \text{otherwise.} \end{cases} \qquad \text{(DaKatz)} \quad \mathbf{s}_R(i) = \begin{cases} \beta(1-\kappa) & \text{if } \delta^+(i) > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

$$\mathbf{s}_C(i) = \begin{cases} \frac{d\kappa}{\delta^-(i)} & \text{if } \delta^-(i) > 0 \\ 0 & \text{otherwise.} \end{cases} \qquad \mathbf{s}_C(i) = \begin{cases} \beta\kappa & \text{if } \delta^-(i) > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

Let **B** be the 0–1 adjacency matrix of G defined as

$$b_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise.} \end{cases} \tag{18}$$

Then (8) and (14) can be rewritten as

$$\mathbf{p}_t = \mathbf{r} + \mathbf{B}(\mathbf{s}_R * \mathbf{p}_{t-1}) + \mathbf{B}^T(\mathbf{s}_C * \mathbf{p}_{t-1}), \tag{19}$$

where * denote the pointwise vector multiplication. In this form, the total size of **B**, $\mathbf{B}^T$, $\mathbf{s}_R$, and $\mathbf{s}_C$ is roughly 71MB assuming we only store the indices of nonzeros in **B** and $\mathbf{B}^T$. This modification not only reduces the size of **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 $i \in V$, Algorithms 1 and 2 perform $2|E|$ multiplications. Hence this modification can lead up to sixfold 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 **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 (Buluç et al. 2011). Algorithms 3 and 4 show the pseudocodes of the DaRWR and DaKatz computation with the modifications described above.

```
Algorithm 3: DaRWR with CRS-Half
Input: $n \times n$ adjacency matrix $\mathbf{B}$ in CRS format, query paper set $\mathcal{Q}$, row and column scaling vectors $\mathbf{s}_R$, $\mathbf{s}_C$
Output: relevance vector $\mathbf{p}$
$\mathbf{p}_t \leftarrow \mathbf{0}$
$\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow 1/|\mathcal{Q}|$
$e \leftarrow \|\mathbf{p}_t\|_2$
while $e > \xi$ do
    $\mathbf{sp}_R \leftarrow \mathbf{p}_t * \mathbf{s}_R$
    $\mathbf{sp}_C \leftarrow \mathbf{p}_t * \mathbf{s}_C$
    $\mathbf{p}_{t-1} \leftarrow \mathbf{p}_t$
    $\mathbf{p}_t \leftarrow \mathbf{0}$
    foreach paper $i = 1$ to $n$ do
        if $\mathbf{sp}_C(i) > 0$ then
            foreach $b_{ij} \neq 0$ in $\mathbf{B}_{i*}$ 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
            foreach $b_{ij} \neq 0$ in $\mathbf{B}_{i*}$ do
                $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \mathbf{sp}_R(j)$
    $\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + (1-d)/|\mathcal{Q}|$
    $e \leftarrow \|\mathbf{p}_t - \mathbf{p}_{t-1}\|_2$
return $\mathbf{p} \leftarrow \mathbf{p}_t$
```

```
Algorithm 4: DaKatz with CRS-Half
Input: $n \times n$ adjacency matrix $\mathbf{B}$ in CRS format, query paper set $\mathcal{Q}$, row and column scaling vectors $\mathbf{s}_R$, $\mathbf{s}_C$
Output: relevance vector $\mathbf{p}$
$\mathbf{p}_t \leftarrow \mathbf{0}$   $\mathbf{p}_{total} \leftarrow \mathbf{0}$
$\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow 1$
$e \leftarrow \|\mathbf{p}_t\|_2$
while $e > \xi$ do
    $\mathbf{sp}_R \leftarrow \mathbf{p}_t * \mathbf{s}_R$
    $\mathbf{sp}_C \leftarrow \mathbf{p}_t * \mathbf{s}_C$
    $\mathbf{p}_{t-1} \leftarrow \mathbf{p}_t$
    $\mathbf{p}_t \leftarrow \mathbf{0}$
    foreach paper $i = 1$ to $n$ do
        if $\mathbf{sp}_C(i) > 0$ then
            foreach $b_{ij} \neq 0$ in $\mathbf{B}_{i*}$ 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
            foreach $b_{ij} \neq 0$ in $\mathbf{B}_{i*}$ do
                $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \mathbf{sp}_R(j)$
    $\mathbf{p}_{total} \leftarrow \mathbf{p}_{total} + \mathbf{p}_t$
    $e \leftarrow \|\mathbf{p}_t - \mathbf{p}_{t-1}\|_2$
return $\mathbf{p} \leftarrow \mathbf{p}_{total}$
```

Although the proposed modifications reduce the size of **A** and the number of multiplications, there is a drawback. In Algorithms 1 and 2, line 8 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 only works for nonzeros corresponding to the outgoing edges when the matrix is reduced. That is, if b_{ij} is nonzero, Algorithm 3 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 Algorithms 3 and 4, although they are skipped in Algorithms 1 and 2.

3.5 Implementations with halved COO Storage (COO-Half)

If we apply the optimizations described for the halved CRS, one needs roughly 63MB in memory to store **B** in COO format. 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, since we do not need to store the nonzeros in a row consecutively. Furthermore, techniques like blocking can be implemented with much less overhead. We give the COO-based pseudocodes of DaRWR and DaKatz in Algorithms 5 and 6.

```
Algorithm 5: DaRWR with COO-Half
Input: $n \times n$ adjacency matrix $\mathbf{B}$ in COO format, query paper set $\mathcal{Q}$, row and column scaling vectors $\mathbf{s}_R$, $\mathbf{s}_C$
Output: relevance vector $\mathbf{p}$
$\mathbf{p}_t \leftarrow \mathbf{0}$
$\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow 1/|\mathcal{Q}|$
$e \leftarrow \|\mathbf{p}_t\|_2$
while $e > \xi$ do
    $\mathbf{sp}_R \leftarrow \mathbf{p}_t * \mathbf{s}_R$
    $\mathbf{sp}_C \leftarrow \mathbf{p}_t * \mathbf{s}_C$
    $\mathbf{p}_{t-1} \leftarrow \mathbf{p}_t$
    $\mathbf{p}_t \leftarrow \mathbf{0}$
    foreach nonzero $b_{ij}$ of $\mathbf{B}$ do
        $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \mathbf{sp}_R(j)$
        $\mathbf{p}_t(j) \leftarrow \mathbf{p}_t(j) + \mathbf{sp}_C(i)$
    $\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + (1-d)/|\mathcal{Q}|$
    $e \leftarrow \|\mathbf{p}_t - \mathbf{p}_{t-1}\|_2$
return $\mathbf{p} \leftarrow \mathbf{p}_t$
```

```
Algorithm 6: DaKatz with COO-Half
Input: $n \times n$ adjacency matrix $\mathbf{B}$ in COO format, query paper set $\mathcal{Q}$, row and column scaling vectors $\mathbf{s}_R$, $\mathbf{s}_C$
Output: relevance vector $\mathbf{p}$
$\mathbf{p}_t \leftarrow \mathbf{0}$   $\mathbf{p}_{total} \leftarrow \mathbf{0}$
$\forall i \in \mathcal{Q},\ \mathbf{p}_t(i) \leftarrow 1$
$e \leftarrow \|\mathbf{p}_t\|_2$
while $e > \xi$ do
    $\mathbf{sp}_R \leftarrow \mathbf{p}_t * \mathbf{s}_R$
    $\mathbf{sp}_C \leftarrow \mathbf{p}_t * \mathbf{s}_C$
    $\mathbf{p}_{t-1} \leftarrow \mathbf{p}_t$
    $\mathbf{p}_t \leftarrow \mathbf{0}$
    foreach nonzero $b_{ij}$ of $\mathbf{B}$ do
        $\mathbf{p}_t(i) \leftarrow \mathbf{p}_t(i) + \mathbf{sp}_R(j)$
        $\mathbf{p}_t(j) \leftarrow \mathbf{p}_t(j) + \mathbf{sp}_C(i)$
    $\mathbf{p}_{total} \leftarrow \mathbf{p}_{total} + \mathbf{p}_t$
    $e \leftarrow \|\mathbf{p}_t - \mathbf{p}_{t-1}\|_2$
return $\mathbf{p} \leftarrow \mathbf{p}_{total}$
```

4 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 $\mathbf{B}^T$. Also, using the same permutation allows a simpler implementation of the iterative SpMxV operations. Note that although $\mathbf{s}_R$ and $\mathbf{s}_C$ can be permuted with different row and column permutations, we only have a single $\mathbf{p}_t$ array to process both b_{ij} and its transposed counterpart b_{ji} as shown in Fig. 2. Due to these reasons, permuting the adjacency matrix as $\mathbf{B}' = \mathbf{PBP}^T$ 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 (Akbudak et al. 2012; Yzelman and Bisseling 2009; Yzelman and Bisseling 2011). Note that we can still use symmetric permutations such as the ones obtained by RCM, AMD, and SlashBurn.

Similar to existing partitioning-based approaches, we use a two-phase permutation strategy which first partitions the rows of $\mathbf{B}$ into K parts and sorts them in the increasing order of their part numbers. The intra-part row ordering is decided later by using RCM, AMD, or SlashBurn, 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 Sect. 2.2. 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 (Çatalyürek and Aykanat 1999) to minimize *connectivity*-1 metric (3) 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 (e.g., nets n_1, n_2, n_3, and n_4, and the corresponding columns in Fig. 1). 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' (n_4 and the fourth column in Fig. 1). 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, AMD, and SlashBurn where RCM and AMD have previously been used for fill-in minimization in sparse LU factorization. RCM is used to find a permutation σ which reduces the bandwidth of a symmetric matrix $\mathbf{A}$ where the bandwidth is defined as $b = \max(\{|\sigma(i) - \sigma(j)|: a_{ij} \neq 0\})$. When the bandwidth 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 the nonzeros in different parts of the matrix. Since having nonzeros close to each other is good for cache locality, we used these heuristics to order rows and columns inside each part.

The last ordering heuristic, SlashBurn, has been proposed for matrix compression, i.e., to reduce the number of

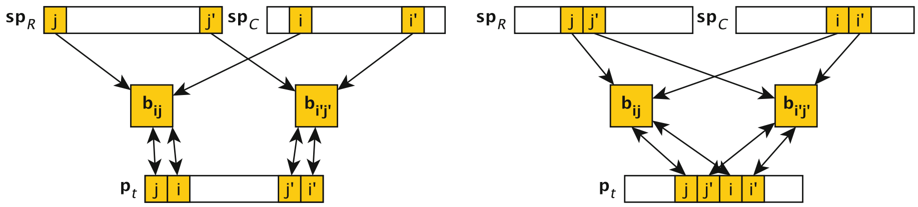


Fig. 2 Two memory-access scenarios with different row/column permutations and nonzero orderings while processing b_{ij} and $b_{i'j'}$ consecutively. The left scenario has a poor cache locality, since the locations accessed in $\mathbf{s}_R$, $\mathbf{s}_C$, and $\mathbf{p}_t$ are far from each other. On the other hand, in the right scenario, the locations are close to each other. Thus, with high probability, the required values to process $b_{i'j'}$ will already be in the cache after processing b_{ij}

fixed-size tiles required to cover all the nonzeros in the matrix, which also implies a reduced number cache-misses. For several social and web graphs, SlashBurn is proven to be very effective (Kang and Faloutsos 2011). However, its complexity is larger than that of RCM and AMD, and as a result, it is much slower in practice. Since the ordering will be executed only once as a preprocessing phase of the the**advisor**, for our application, we can ignore its complexity in the evaluation and concentrate on its benefits on the query response time.

For all the algorithms described in Sect. 3, we used the proposed ordering approach. For CRS-Full, we permuted **A**, and for CRS-Half and COO-Half, we permuted **B** as described above. For COO-Half, we also apply blocking after permuting **B**: we divide **B** into square blocks of size 1,024 × 1,024 and traverse the nonzeros with respect to their block ids (and row-wise within a block). The block size is tuned on the architecture the**advisor** is running on.

5 Experimental results

The setup for the experiments can be summarized as follows:

Architectures: We used three different architectures to test the algorithms. The target architecture (**Arch1**) 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 more recent architectures. The second architecture (**Arch2**) 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. The third architecture (**Arch3**) 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.

Implementation: All of the algorithms are implemented in C++. The compiler `gcc` and the `-O2` optimization flag is used. For the experiments, we use only one core from each processor.

Queries: We generated 286 queries where each query is a set $\mathcal{Q}$ 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 query papers, $|\mathcal{Q}|$, vary between 1 and 449, with an average of 24.7.

Parameters: For DaRWR, we use $d = 0.8$ and $\kappa = 0.75$ which are the default values in the**advisor**. For DaKatz, we use $\beta = 0.005$. While generating the partitions, we set the imbalance ratio of PaToH to 0.4.

Convergence: We did not use a threshold ξ for convergence. We observed that DaRWR in our citation graph takes about 20 iterations, and DaKatz takes about 10 iteration to converge (see Fig. 3). Computing the error between iterations takes some time, and since we want to be consistent in the experiments, we let the algorithms iterate a fixed number of times.

5.1 Effects of the storage schemes on the number of updates

As mentioned in Sect. 3, the algorithms CRS-Full and CRS-Half can avoid some updates but COO-Half cannot, even they have no effect on $\mathbf{p}_t$. In our query set, the average number of papers is 24.7. In the first iteration, $\mathbf{p}_{t-1}$ has only a few positive values on average, and CRS-Full updates $\mathbf{p}_t$ only for the corresponding papers in $\mathcal{Q}$. Since $n \gg 24.7$, 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 most of them have no effect for the first couple of iterations. However, the number of positive values in $\mathbf{p}_{t-1}$ increases exponentially. As Fig. 4 shows, the shortcuts in CRS-based

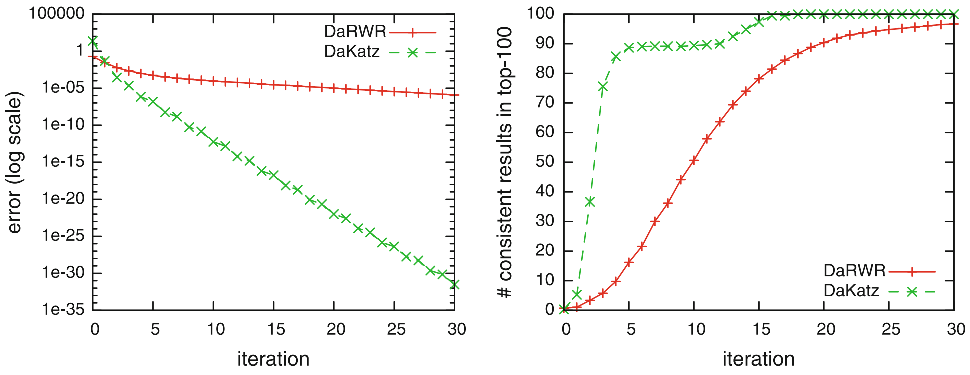


Fig. 3 Errors and number of consistent results within top-100 when DaRWR and DaKatz is run with the given number of iterations

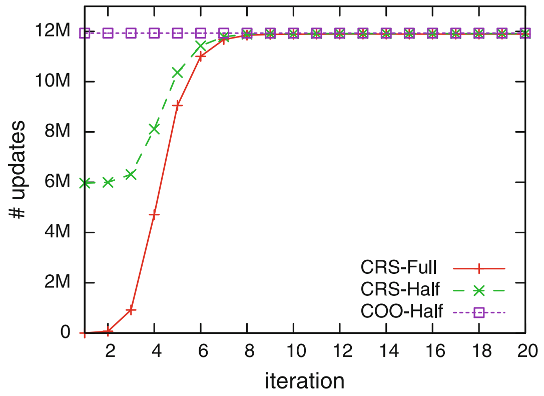


Fig. 4 Number of updates per iteration of the algorithms

algorithms are not useful after the eighth iteration. The figure also implies that the citation graph is highly connected since DaRWR and DaKatz seem to traverse almost all the nonzeros in **A**. That is, random walks and paths 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.

5.2 Effects of partitioning and ordering on nonzero patterns

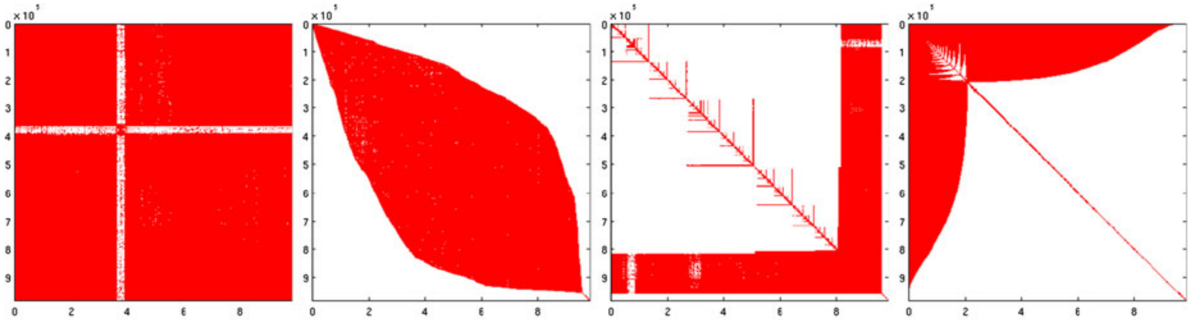
The nonzero pattern of the adjacency matrix **B** is given in Fig. 5a. As the figure shows, the nonzeros are distributed in all the matrix. In our experiments, the papers are originally numbered with respect to the order we parse their metadata. When **B** is ordered by using the RCM heuristic, the nonzero pattern (Fig. 5b) is densified near the diagonal as expected. The bandwidths of the original and RCM ordered **B** matrices are 981, 287 and 460, 288, respectively. Although the bandwidth is reduced more than half, it is still large. Figure 5c shows the nonzero pattern of **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 number of cache misses. The pattern of SlashBurn also possesses similar characteristics: Figure 5d shows the arrow-shaped pattern obtained after ordering **B** with SlashBurn. All the nonzeros are densified inside the pattern, and the number of cache misses is expected to be much less.

As described in Sect. 4, we first partition **B** in the column-net model to reorder it. To do that, we use $K = \{2, 4, 8, 16, 32, 64\}$ and create six different partitions. For each partition, we create a permutation matrix **P** and reorder **B** as $\mathbf{B}' = \mathbf{PBP}^T$. The left-most images in Figs. 6a–c show the structure of the nonzero pattern of $\mathbf{B}'$ for $K = 2,\ 4$, and 8, respectively. In the figures, 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 **B** are assigned to the same part. We permute the rows and columns of these blocks by using the ordering heuristics RCM, AMD, and SlashBurn. Figure 6 also shows the nonzero patterns of these further permuted matrices for each partition with $K = 2$, 4, and 8.

5.3 Performance analysis

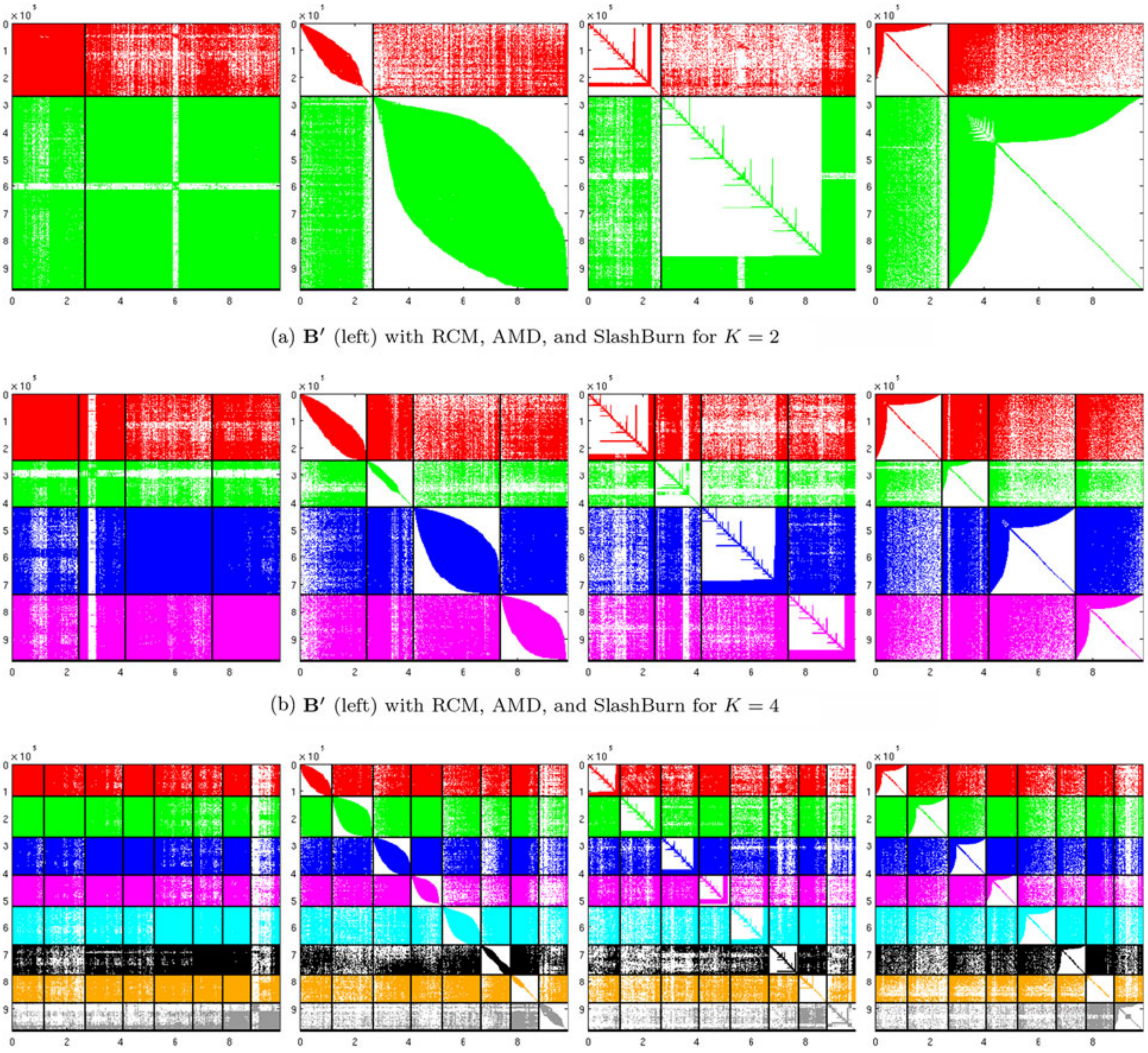
Figure 7a–c shows the number of updates per second (*nups*) for each algorithm when RCM, AMD, and SlashBurn are used, respectively. This experiment counts the number of updates that occur in memory, even if they are nilpotent, i.e., they do not change a value. The configuration which takes advantage of partitioning most is the COO-Half equipped with AMD ordering for which *nups* increases from 270 to 340 million. As the figure shows, SlashBurn does not bring a performance improvement relative to AMD and RCM. Hence, considering its complexity, we can suggest that using AMD and RCM is more practical, especially when the time spent for the ordering is important.

Although the nonzeros of $\mathbf{B}'$ seem evenly distributed in Fig. 6, as Table 2 shows, for $K = 2$ and 4, the percentage of the nonzeros inside diagonal blocks of $\mathbf{B}'$ are 94 and 83 %, respectively. Hence, the nonzeros are densified inside these blocks as expected. The number of nonzeros outside the diagonal blocks is more when K increases. However, the diagonal blocks tend to get smaller and denser which may improve the cache locality. As Fig. 7 shows, a consistent improvement can be observed for all the algorithms CRS-Full, CRS-Half, and COO-Half, and ordering heuristics RCM, AMD, and SlashBurn up to $K = 8$ and 16. But when K gets more than 16, no significant improvement can be observed and the performance can even get worse. There are two main reasons for this phenomena: first, the fraction of block-diagonal nonzeros continues to decrease, and second, the diagonal blocks become smaller than required. That is the cache locality may be optimized to the most, hence, a further reduction on the block size is unnecessary. Here, the best K in terms of performance depends on the configuration. We tested the algorithms with different K values to find the best configuration. As the figure shows, with CRS-Full, the maximum *nups* is around 80 million for $K = 8$ (RCM), 16 (AMD), and $K = 16$ (SlashBurn). Compressing the matrix increases the *nups* for CRS-Half up to 190 million with $K = 16$ (AMD). And with blocking used in COO-Half, *nups* increases to 340 million with $K = 16$ (AMD) which is the maximum for this experiment.



(a) Original (b) RCM (c) AMD (d) SlashBurn

Fig. 5 The nonzero pattern of **B** (a) when ordered with RCM (b), AMD (c), and SlashBurn (d). Nonzeros are colored with red and white areas show empty regions

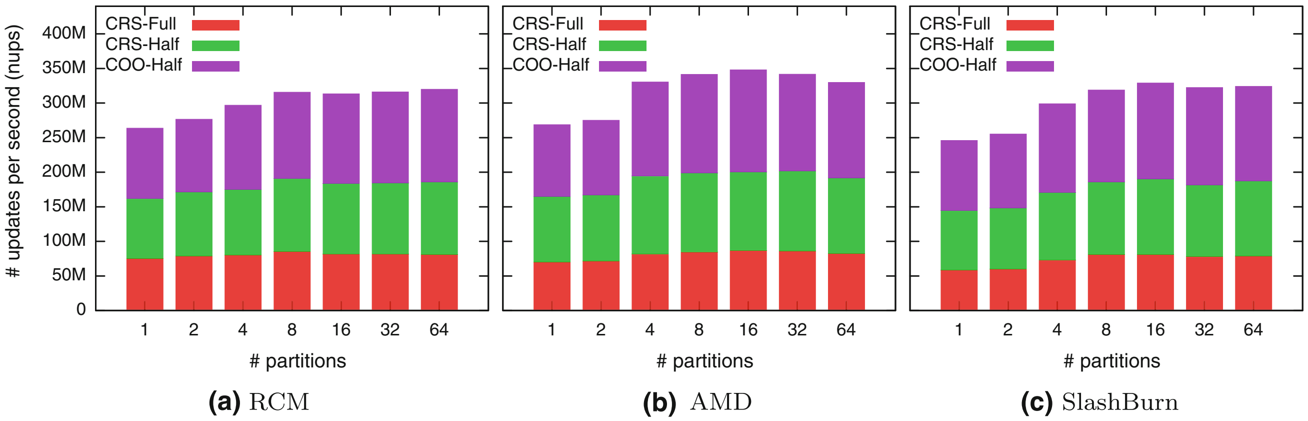


(a) $\mathbf{B}'$ (left) with RCM, AMD, and SlashBurn for $K = 2$

(b) $\mathbf{B}'$ (left) with RCM, AMD, and SlashBurn for $K = 4$

(c) $\mathbf{B}'$ (left) with RCM, AMD, and SlashBurn for $K = 8$

Fig. 6 The nonzero pattern of the permuted adjacency matrix **B** with different partitions and ordering 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 **B** are in the same part



(a) RCM **(b)** AMD **(c)** SlashBurn

Fig. 7 Number of updates per second (*nups*) for each algorithm and ordering on **Arch1** when DaRWR is executed for 20 iterations

Table 2 Number of nonzeros inside and outside of the diagonal blocks of $\mathbf{B}'$ after reordering

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

The *nups* of COO-Half seems to be much superior in Fig. 7. However, the algorithm itself needs to do more computation since it cannot skip any of the updates even if they are nilpotent. To compare the actual response times of the configurations, we used 286 queries and measured the average response time on each architecture. Figure 8 shows the execution times of DaRWR and DaKatz for different algorithms, Ks, and ordering heuristics on the target architecture **Arch1**. For the rest of the figures in the paper, SB denotes the SlashBurn ordering heuristic.

As concordant with *nups* values, for DaRWR, the fastest algorithm is COO-Half where $K = 16$ and the diagonal blocks are ordered with AMD. The average query response time for this configuration, which is being used in the**advisor**, is 1.61 s. Compared with the execution time of CRS-Full with the original ordering, which is 4.80 s, we obtain 3 times improvement. When $K = 1$, i.e., if there is no partitioning, the execution time of COO-Half is 2.29. Hence, we obtain a speedup of more than 2 due to ordering and 42 % additional improvement due to partitioning.

For DaKatz experiment on **Arch1**, the best configuration is not very visible. In our experiments, the minimum average query response time, 0.89 s, is obtained again by COO-Half when it is equipped with AMD and when K is 16. On the other hand, the best CRS-Full configuration answers a query in 0.90 s on average (with AMD and $K = 16$). Although the ordering and partitioning still help a lot, the improvements due to algorithmic modifications are only minor for DaKatz. As described above and shown by Fig. 4, the number of updates required by CRS-Full only matches that of COO-Half after the eighth iteration and DaKatz converges only in ten iterations for our citation graph. That is the total work required by CRS-Full is much less than both CRS-Half and COO-Half for DaKatz. Hence, the update overhead cannot be easily compensated by reducing the bandwidth and improving cache locality. Still, the average query response time is reduced from 1.89 to 0.89 thanks to ordering and partitioning.

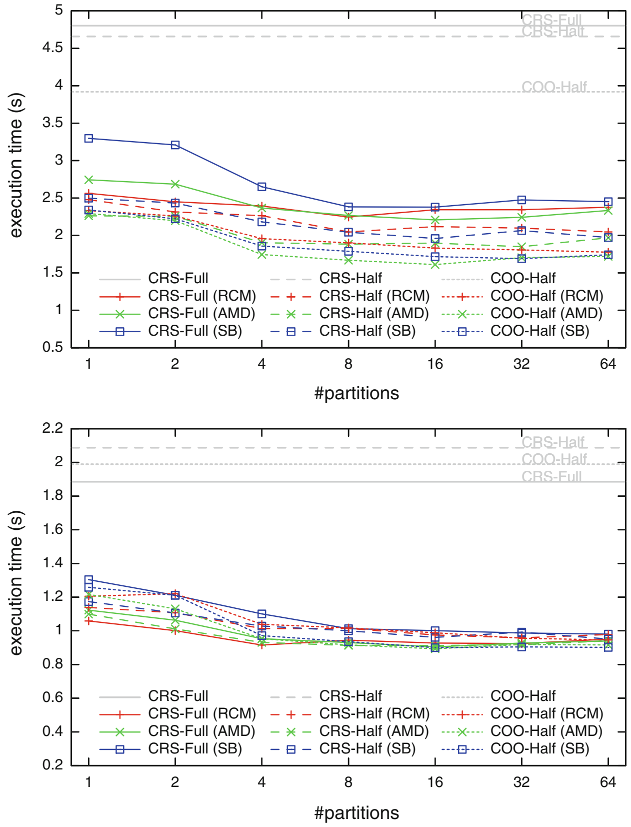


Fig. 8 Execution times in seconds for DaRWR (*top*) and DaKatz (*bottom*) using each algorithm with different Ks and ordering heuristics on **Arch1**. The values are the averages of the running times for 286 queries. For each query, the algorithms perform 20 DaRWR iterations and 10 DaKatz iterations

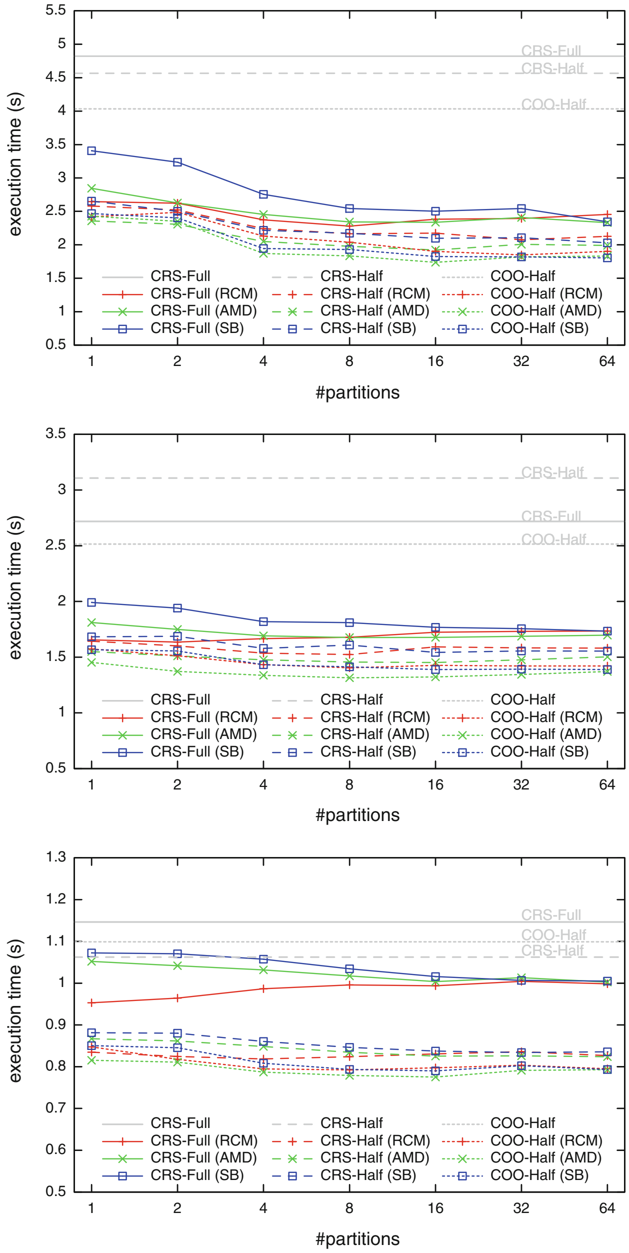


Fig. 9 Execution times of DaKatz algorithms in seconds on architectures on **Arch1** (*top*), **Arch2** (*middle*), and **Arch3** (*bottom*) with different *K*s and ordering heuristics. The values are the averages of the running times for 286 queries. For each query, 20 DaKatz iterations are performed

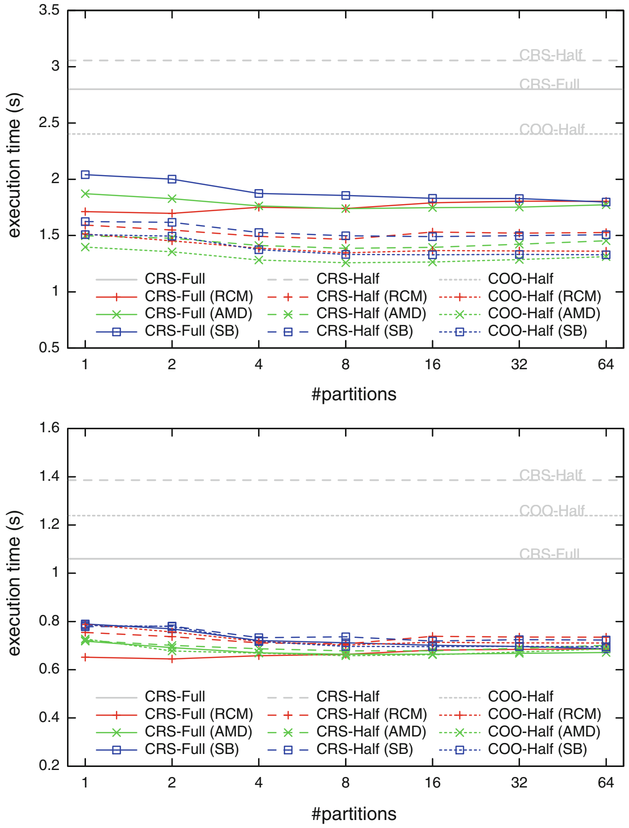


Fig. 10 Execution times in seconds for DaRWR (*top*) and DaKatz (*bottom*) using each algorithm with different *K*s and ordering heuristics on **Arch2**. The values are the averages of the running times for 286 queries. For each query, the algorithms perform 20 DaRWR iterations and 10 DaKatz iterations

For completeness, in Fig. 9, we give the results when 20 DaKatz iterations are performed. On all architectures, CRS-Full configurations are much slower than CRS-Half and COO-Half configurations as expected. And the differences are more visible. Furthermore, the relative performance of DaKatz algorithms is more similar to that of DaRWR algorithms with 20 iterations.

We tested our modifications on two other architectures described above. As shown in Fig. 10, the results are similar for DaRWR on **Arch2**. COO-Half with AMD is the fastest configuration, but this time with $K = 8$. 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 both on **Arch1** (Fig. 8) and **Arch3** (Fig. 11), CRS-Half is faster than CRS-Full. On **Arch3**, the fastest DaRWR algorithm is again COO-Half where the average query response time is 0.72 s with $K = 8$ and the AMD heuristic. Compared to the time of CRS-Full based DaRWR with no ordering, which is 1.18 s, the improvement is 39 %. If we apply only matrix compression with COO-Half, the query response time is 1.04 s. Hence, we can argue that we obtained 31 % improvement by permuting the reduced matrix. If we only use AMD, i.e., when $K = 1$, the query response time of COO-Half is 0.76. This implies roughly 5 % improvement due to partitioning alone. Since **Arch3**'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 the last architecture.

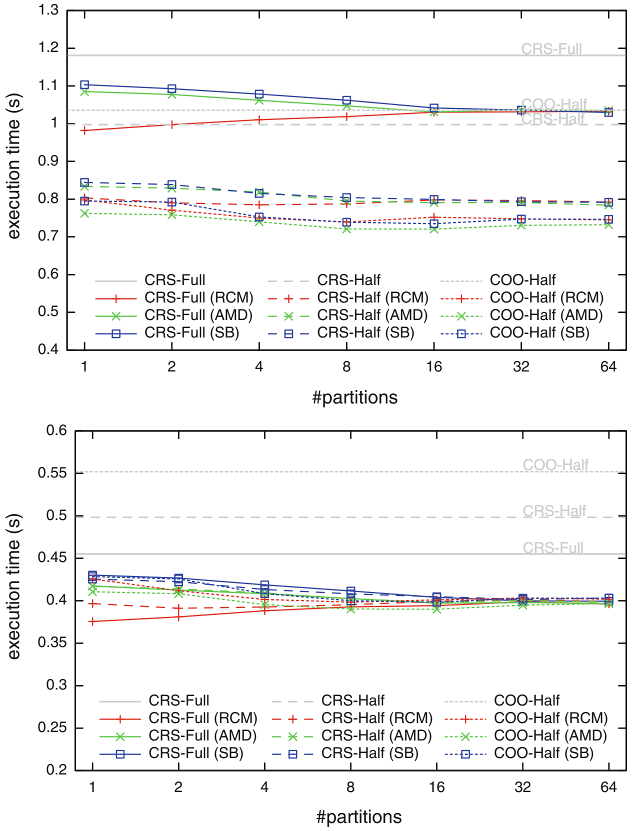


Fig. 11 Execution times in seconds for DaRWR (*top*) and DaKatz (*bottom*) using each algorithm with different *K*s and ordering heuristics on **Arch2**. The values are the averages of the running times for 286 queries. For each query, the algorithms perform 20 DaRWR iterations and 10 DaKatz iterations

For DaKatz on **Arch2** (Fig. 10) and **Arch3** (Fig. 11), the best algorithm is clearly CRS-Full. The RCM ordering yields 40 % (0.45–0.37 s) and 18 % (1.06–0.65 s) improvements on the average query response time, respectively. However, the partitioning is not useful in practice for DaKatz on these architectures since it improves the query response times of other configurations but has a negative effect on CRS-Full with RCM. A similar pattern is also visible for the same DaRWR and DaKatz configurations especially on architectures **Arch2** and **Arch3**. The partitioning and the corresponding permutation on **B** are designed while taking the halved matrix into account: an access to a nonzero b_{ij} yields also the processing of b_{ji}. That is, the nonzeros to be processed are coming from a two-dimensional region. Hence, having the nonzeros inside diagonal blocks, COO-Half should be the algorithm which utilizes the optimizations the most, especially considering its blocked access pattern. On the contrary, for the same reasons, CRS-Full should be the worst algorithm for exploiting the optimizations, since the upcoming accesses in CRS-Full are only in column dimension. Furthermore, partitioning and permutation increase the bandwidth of the matrix which is arguably the most important criterion for CRS-Full. Since RCM aims to minimize the bandwidth of the matrix, the performance can be reduced when an additional partitioning is used. On the other hand, when the cache is not enough or the bandwidth is still large after RCM, partitioning may take effect and improve the performance. This can be observed in Fig. 8 for the target architecture **Arch1** which has 6 and 8 times less cache than **Arch2** and **Arch3**, respectively.

Considering the number of updates of CRS-Full is much less than COO-Half for the first few iterations, it is expected to be faster than COO-Half. On the other hand, when 20 DaKatz iterations are performed instead of 10, COO-Half with AMD is again the best configuration as shown in Fig. 9.

6 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 of cache misses. Experimental results show that 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 of concurrent requests is less than the number of processors allocated in the cluster.

Another work we are interested in is to make the service much faster via a hybrid implementation of DaRWR (or DaKatz) which uses a combination of CRS-Full, CRS-Half, and COO-Half. In its simple form, the hybrid approach can use CRS-Full in the first few iterations then switch to COO-Half to utilize the efficiency of the algorithms to the most. The overhead of such a scheme is storing the citation matrix multiple times and a transition cost incurred while switching from one algorithm to another. We believe that a hybrid implementation is promising and we aim to do a thorough investigation in the near future.

Acknowledgments 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

Agarwal RC, Gustavson FG, Zubair M (1992) A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In: Proceedings of ACM/IEEE Supercomputing, pp 32–41

Akbudak K, Kayaaslan E, Aykanat C (2012) Hypergraph-partitioning-based models and methods for exploiting cache locality in sparse-matrix vector multiplication. CoRR abs/1202.3856

Amestoy PR, Davis TA, Duff IS (1996) An approximate minimum degree ordering algorithm. SIAM J Matrix Anal Appl 17(4): 886–905

Bollen J, Rodriguez MA, de Sompel HV (2006) Journal status. Scientometrics 69(3):669–687

Buluç A, Williams S, Oliker L, Demmel J (2011) Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium, pp 721–733

Çatalyürek ÜV, Aykanat C (1999) Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans Parallel Distrib Syst 10:673–693

Çatalyürek ÜV, Aykanat C (1999) PaToH: a multilevel hypergraph partitioning tool, Version 3.0. Bilkent University, Computer Engineering, Ankara, Turkey. http://bmi.osu.edu/∼umit/software.htm

Çatalyürek ÜV, Aykanat C (2001) A fine-grain hypergraph model for 2D decomposition of sparse matrices. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium

Chipman KC, Singh AK (2009) Predicting genetic interactions with random walks on biological networks. BMC Bioinformatics 10:17

Cuthill E, McKee J (1969) Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of ACM national conference, pp 157–172

Gori, M. Pucci A (2006) Research paper recommender systems: a random-walk based approach. In: Proceedings of IEEE/WIC/ACM Web Intelligence, pp 778–781

Kang U, Faloutsos C (2011) Beyond 'caveman communities': hubs and spokes for graph compression and mining. In: Proceedings of IEEE International Conference Data Mining, pp 300–309

Kessler MM (1963) Bibliographic coupling between scientific papers. American Documentation 14:10–25

Kim HN, El-Saddik A (2011) Personalized PageRank vectors for tag recommendations: inside FolkRank. In: Proceedings of ACM Recommender Systems, pp 45–52

Küçüktunç O, Kaya K, Saule E, Çatalyürek ÜV (2012a) Fast recommendation on bibliographic networks. In: Proceedings of Advances in Social Networks Analysis and Mining, pp 480–487

Küçüktunç O, Saule E, Kaya K, Çatalyürek ÜV (2012b) Direction awareness in citation recommendation. In: Proceedings of International Workshop on Ranking in Databases (DBRank'12) in Conjunction with VLDB'12

Lawrence S, Giles CL, Bollacker K (1999) Digital libraries and autonomous citation indexing. Computer 32:67–71

Lengauer T (1990) Combinatorial algorithms for integrated circuit layout. Wiley–Teubner, Berlin

Li J, Willett P (2009) ArticleRank: a PageRank-based alternative to numbers of citations for analyzing citation networks. Proc Assoc Inform Manag 61(6):605–618

Liben-Nowell D, Kleinberg JM (2007) The link-prediction problem for social networks. J Am Soc Inform Sci 58(7):1019–1031

Page L, Brin S, Motwani R, Winograd T (1999) The PageRank citation ranking: bringing order to the web. TR 1999-66, Stanford InfoLab

Pan JY, Yang HJ, Faloutsos C, Duygulu P (2004) Automatic multimedia cross-modal correlation discovery. In: Proceedings of ACM SIGKDD International Conference Knowledge Discovery and Data Mining, pp 653–658

Pichel JC, Heras DB, Cabaleiro JC, Rivera FF (2005) Performance optimization of irregular codes based on the combination of reordering and blocking techniques. Parallel Comput 31(8–9):858–876

Pichel JC, Heras DB, Cabaleiro JC, Rivera FF (2009) Increasing data reuse of sparse algebra codes on simultaneous multithreading architectures. Concurr Comput Pract Experience 21(15): 1838–1856

Pinar A, Heath MT (1999) Improving performance of sparse matrix-vector multiplication. In: Proceedings of ACM/IEEE Supercomputing

Small H (1973) Co-citation in the scientific literature: a new measure of the relationship between two documents. J Am Soc Inf Sci 24(4):265–269

Strohman T, Croft WB, Jensen D (2007) Recommending cictations for academic papers. In: Proceedings of International ACM SIGIR Conference Research and Development in Information Retrieval, pp 705–706

Temam O, Jalby W (1992) Characterizing the behavior of sparse algorithms on caches. In: Proceedings of ACM/IEEE Supercomputing, pp 578–587

Toledo S (1997) Improving the memory-system performance of sparse-matrix vector multiplication. IBM J Res Dev 41(6):711–726

White JB, Sadayappan P (1997) On improving the performance of sparse matrix-vector multiplication. In: Proceedings of International Conference High Performance Computing, pp 66–71

Yin Z, Gupta M, Weninger T, Han J (2010) A unified framework for link recommendation using random walks. In: Proceedings of Advances in Social Networks Analysis and Mining, pp 152–159

Yzelman AN, Bisseling RH (2009) Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. SIAM J Sci Comput 31:3128–3154

Yzelman AN, Bisseling RH (2011) Two-dimensional cache-oblivious sparse matrix-vector multiplication. Parallel Comput 37:806–819