A Simple Study of Pleasing Parallelism on Multicore Computers

Yanfei Ren and David F. Gleich

1 Introduction

Current single-machine computing environments are a mixture of high-power CPUs and GPUs mixed to large quantities of memory of various speeds. Often these are subsequently networked together into large distributed computational platforms. Cloud computing further complicates the scenario as advanced resources can be purchased for the time needed. These environments present a wide-range of opportunities to schedule what are often called *pleasingly parallel* computations, namely, those that have a large amount of independent computation that can be scheduled simultaneously.

We wish to investigate how to leverage and utilize such resources in the context of a large graph computation. While the focus on large graph computation is often in terms of solving problems on massive graphs with distributed computation, the downside to such computations is that they often involve *nearly linear-time* algorithms [16]. These have runtimes such as $O(n \log n)$ and typically involve a small number of passes over all the edges of the graph, for instance, running a connected components analysis or computing a PageRank vector. Consequently, the algorithm performance is largely dominated by how well the computation maps to the IO and memory system strategies of the platform.

Instead, the computation we investigate is the all-to-all personalized PageRank computation. Given an n-node graph, this involves computing the personalized PageRank vector associated with each node. We state the problem formally in Sect. 2. Consequently, there are n such computations that are all independent and decoupled. In terms of the scale, we are targeting graphs with up to a 100 million

Y. Ren · D. F. Gleich (✉)
Department of Computer Science, Purdue University, West Lafayette, IN, USA
e-mail: ren105@purdue.edu; dgleich@purdue.edu

A. Grama, A. H. Sameh (eds.), *Parallel Algorithms in Computational Science and Engineering*, Modeling and Simulation in Science, Engineering and Technology,
https://doi.org/10.1007/978-3-030-43736-7_11

edges and with up to 10 million nodes. Real-world instances of such graphs are the LiveJournal social network crawl with around 4 million vertices and 67 million edges and the Orkut social network crawl with 3 million vertices and around 220 million edges.

Because the output from the all-to-all problem would be $O(n^2)$ data, we seek to output only summary statistics of the personalized PageRank vectors including inverse participation ratios for the solutions that serve as a soft-measure of the number of non-zeros, as well as the largest 1000 entries of the vectors. The large values are commonly used as latent measures of node similarity [14, 41, 46]. Hence, a simple strategy for this computation is to load the graph into memory on all computers available, take the *fastest* single-core algorithm for personalized PageRank, and run it as many places as possible.

This picture becomes more interesting in light of the heterogeneous nature of computers. For instance, we can use vector or SIMD instructions to potentially compute multiple PageRank vectors at once, if the algorithm used is amenable to it. Second, large shared memory machines may have a large number of computing cores (over 200 is possible with commercially available systems that cost less than $250k). However, many of these cores share memory bandwidth resources that can impede some algorithms. This suggests that sharing access to a single graph may not scale. Furthermore, GPUs are constantly changing their underlying compute resources. Fourth, the algorithm performance itself is likely to be sensitive to choices of data distribution within the graph due to memory locality. Hence, even for this simple setting there is a rich set of complications to simplistically expecting a pleasingly parallel algorithm to scale.

Our goal is to investigate these performance differences in the context of the simple personalized PageRank computation. We chose that computation as it is representative of a wide swath of related computations on graphs including scalable methods for all-pairs shortest paths. Moreover, the algorithms to compute it are simple. They are specializations of well-known matrix computation algorithms including the power method and Gauss–Seidel method [19]. We can easily investigate a diverse collection of possible implementations that have different memory characteristics. Our focus was to keep the investigation simple and reflective of what might be expected from an informed, but non-expert, user of the algorithms. This is someone who understand how the algorithms works, where the relevant bottlenecks might be, but does not want to attempt to re-engineer the algorithm for the absolute maximum level of parallelism or performance. This individual is optimistic that the pleasingly parallel nature of the computation will be sufficient to drive performance. Towards that end, we discounted using GPUs at the moment as the toolkit for graph computations on GPUs is still evolving.

We have done all of these experiments in the Julia computing environment to make it easy for others to further investigate our ideas. It is also a high-level programming language that makes it simple to implement a variety of algorithms in a consistent fashion. Regarding the idea that high-level languages may be slow, we initially benchmarked the Julia implementation against a C++ implementation of a similar algorithm [25] and found the runtimes of the methods to be within 10% of each other.

2 The PageRank and AllPageRank Problem

The PageRank problem begins with a graph G, which could be both weighted and directed. However, in the interest of simplicity, we take G to be unweighted, directed, and strongly connected. This greatly simplifies the setting and puts the focus on the relevant pieces of the computation. Let us note that we lose no generality by doing so: a PageRank computation on a graph with multiple strongly connected components can be reduced to a sequence of PageRank computations on the individual strong components, and usually, these additional computations are much smaller because most real-world networks only have a single large strongly connected component (see, among others who make this observation, [35]).

Fix an ordering for G's vertices from 1 to n and identify each vertex with its index in this order. Let $\boldsymbol{A}$ be the resulting adjacency matrix of G

$$A_{ij} = \begin{cases} 1 & (i, j) \text{ is a directed } i \rightarrow j \text{ edge} \\ 0 & (i, j) \text{ is not an edge.} \end{cases}$$

We use the following additional notation:

$\mathbf{d}$ = vector of degrees $\quad d_i = \sum_j A_{ij}$

$\mathbf{p}$ = vector of inverse degrees $\quad p_i = 1/d_i$

$\boldsymbol{P}$ = the stochastic transition matrix $\quad \boldsymbol{P} = \boldsymbol{A}^T \text{Diag}(\mathbf{p})$

$\mathbf{e}_i$ = the ith column of the identity, $\mathbf{e}_i$ has a 1 in the ith row and 0 elsewhere,

where we use the Diag$(\cdot)$ operator to put the argument along the matrix diagonal. The PageRank problem [14, 28, 39, 40] is to compute the stationary distribution of a random walk that with probability α follows a standard random walk model on G and with probability $1 - \alpha$ jumps according a teleportation distribution vector $\mathbf{v}$, where $\mathbf{v}$ encodes the probability of jumping to each node. Typically α is between 0.5 and 0.99. Throughout this paper, we use what became the *standard value* of 0.85 [28]. The stationary distribution corresponds to a solution of the following nonsingular linear system:

$$(\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x} = (1 - \alpha)\mathbf{v}.$$

Personalized, or seeded, PageRank problems set $\mathbf{v}$ to be a single node, or in this case, a column of the identity matrix $\mathbf{e}_i$ and the linear system

$$(\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x}_i = (1 - \alpha)\mathbf{e}_i. \tag{1}$$

As an aside, we note that a standard feature of most PageRank constructions [14] is the dangling correction vector $\mathbf{c}$. In this case, we do not have this correction vector because we assume that we are given a strongly connected graph.

Our goal is to compute $\mathbf{x}_i$ for all i from 1 to n, or more simply, the matrix

$$\boldsymbol{X} = (1-\alpha)(\boldsymbol{I} - \alpha \boldsymbol{P})^{-1}.$$

We wish the entries of $\boldsymbol{X}$ to be of high accuracy, and intend to compute each column of $\boldsymbol{X}$ such that the 1-norm error is provably less than $(1-\alpha)/n$. Because the graph is strongly connected, the matrix $\boldsymbol{X}$ is dense when computed exactly. For a graph with one million vertices this graph is too large to store even on a large shared memory machine. We thus define the AllPageRank problem.

Problem 1 (AllPageRank) Fix a graph G, let $\boldsymbol{A}$ be a binary adjacency matrix indicating the presence of an edge, let $\boldsymbol{P}$ be a column stochastic matrix giving transition probabilities on the same graph. The AllPageRank problem is to compute the following entries of $(1-\alpha)(\boldsymbol{I} - \alpha \boldsymbol{P})^{-1}$:

- the participation ratio for each column $\mathbf{x}_i$, which is a soft measure of the number of non-zeros in the column
- the non-zero values of $\boldsymbol{X} \odot \boldsymbol{A}$ and $\boldsymbol{X}^T \odot \boldsymbol{A}$, and
- the k largest entries in each column, for $k = 100$ or $k = 1000$.

Here $\odot$ is the elementwise, or Hadamard, product. Note that the transition matrix $\boldsymbol{P}$ need not come from the transition described above and could come from anywhere, such as the common stochastic transformations in PageRank [14]. Nonetheless, we will always use $\boldsymbol{P} = \boldsymbol{A}^T \mathrm{Diag}(\mathbf{p})$ in this manuscript. The results of AllPageRank could be used to form a nearest neighbor approximation, to form PageRank affinities [46], or simply as a diffusion approximation of the underlying graph. Additional applications of such an output involve similar methods that solve protein function inference problems [23, 31]. Finally, this can be related to some idea of a "PageRank effective resistance" on an edge.

We stress that there *are* applications of the output for PageRank, but that our general goal is to use PageRank as a model computation that is representative of the challenges faced by more general numerical computing problems on graphs. This is akin to PageRank's widespread use to evaluate the performance of distributed graph computation engines [1, 9, 26, 33, 43, 44]. See additional examples of related computations in Sect. 5.

3 PageRank Algorithms

There are a few classic algorithms for PageRank computations: the power method, the Gauss–Seidel method, and the *push method* in two variations. We briefly explain these algorithms, give a small pseudocode for the computation, as well as an easy-to-compute error bound.

For the following set of algorithms, we will describe how to use them to compute a single vector, although we note that all of them are amenable to computing

multiple vectors simultaneously as discussed in Sect. 3.6. We will use the notation $\mathbf{x}$ to refer to the solution vector to $(\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x} = (1-\alpha)\mathbf{v}$ where $\mathbf{v} = \mathbf{e}_i$ for some fixed i. Each iterate in a high-level description of the method will be written $\mathbf{x}^{(k)}$; what exactly constitutes an iteration may vary among the discussions. For instance, for Gauss–Seidel and the Push Methods, it is often helpful to analyze a single update step within an iteration. We have endeavored to keep the discussion consistent and try to point to the pseudocode to clarify any ambiguities. Note that, in the pseudocode and discussions about it, however, we will be more clear about memory and use $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{r}$ to denote *vectors* of memory associated with an iteration rather than their interpretations about the solution.

3.1 The Power Method

What is usually called the power method for PageRank is probably better called the Richardson method for the linear system formulation of PageRank [14] because the two iterations are exactly identical in the scenario that each iterate is a probability distribution. The idea underlying both is to unwrap the linear system (1) into the fixed point iteration

$$\mathbf{x}_i^{(0)} = \mathbf{e}_i \qquad \mathbf{x}_i^{(k+1)} = \alpha \boldsymbol{P}\mathbf{x}_i^{(k)} + (1-\alpha)\mathbf{e}_i\,.$$

The main work at each iteration is the matrix vector product $\boldsymbol{P}\mathbf{x}^{(k)}$. This can be done either by computing a sparse matrix $\boldsymbol{P}$ where the non-zero value is the probability $\boldsymbol{A}^T \mathrm{Diag}(\mathbf{p})$ or instead, by storing just the graph structure of $\boldsymbol{A}^T$ alone without any values for the non-zero entries along with the vector $\mathbf{p}$. To find a point where $||\mathbf{x}^{(k)} - \mathbf{x}||_1 \leq \tau$, this method requires at most $2\log(\tau)/\log(\alpha)$ iterations [7]. As noted in [14], we can terminate this earlier when $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_1 \leq \tau(1-\alpha)$ because that guarantees the same error condition. This helpful circumstance arises due to the relationship between $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_1$ and the residual of the linear system.

In our implementation, this iteration is implemented using two vectors of memory for a compressed sparse column representation of the adjacency matrix $\boldsymbol{A}$. The pseudocode is in Fig. 1. In this algorithm, we store an iteration in $\mathbf{x}$ and use the memory in $\mathbf{y}$ to compute the next iterate $\mathbf{x}^{(k+1)}$. After the entire update is done, we compare the vectors and swap.

3.2 Gauss–Seidel

Gauss–Seidel is a simple variant of the power method where we update the solution vector immediately after computing the value `update` in Fig. 1. This requires only

```
1  Inputs:
2   - adjacency matrix A (in compressed column storage),
3   - p as the inverse degree vector of A,
4   - 0 < α < 1,
5   - v is an integer giving the column of X to compute
6   - error tolerance τ = (1 − α)/n
7   - two vectors of memory x and y
8  initialize x as 0
9  set x[v] = 1
10 set maxiter = 2 log(τ)/ log(α)
11 for iter=1:maxiter
12   for i=1:n
13     update = 0
14     for j in nonzeros(A[:,i])
15       update += x[j] * p[j]
16     end
17     y[i] ← α ∗ update
18   end
19   y[v] ← y[v] + 1 − α
20   set δ = ‖x − y‖_1
21   x, y ← y, x # swap x, y
22   if δ/(1 − α) ≤ τ
23     return x and converged
24   end
25 end
```

Fig. 1 Pseudocode for the power method to compute the vth column of $\boldsymbol{X} = (1-\alpha)(\boldsymbol{I} - \alpha \boldsymbol{P})^{-1}$. This algorithm takes two vectors of memory and performs random reads from the memory in **x** and **p**, but then linearly ordered writes to the memory **y**

one vector of memory. Writing this update formally is often annoyingly intricate—it involves an idea called a regular splitting [45]—but is an extremely simple change in terms of the code. Thus, we start with the pseudocode in Fig. 2.

There are a few subtle differences from the pseudocode of the power method. First, we initialize the vector **x** from zero. This choice will turn out to make tracking the error in the Gauss–Seidel iteration much easier [8]. Second, the algorithm actually stores $\mathbf{x}^{(k)} \odot \mathbf{p}$ in the memory **x** where $\odot$ is an elementwise product. This choice is made so that we can compute the quantities in `update` on lines 14–17 without looking up the values in **p**. Note that we could have done the same transformation for the power method, but we found it slightly decreased performance. Here, the value of δ tracks the total sum of $\mathbf{x}^{(k)} = \mathbf{x} \odot \mathbf{d}$ after the loop 13–25. This corresponds to the sum of an iterate $\mathbf{x}^{(k)}$ of the Gauss–Seidel method. As we will see next, for Gauss–Seidel starting from 0, we have that $\sum_{i=1}^{n} [\mathbf{x}^{(k)}]_i$ gives the 1-norm of the error.

The error analysis of the method is fairly straightforward. The iterations we analyze are the *unscaled iterations* that would correspond to multiplying **x** in the code by **d** (elementwise) at each step. We call these $\mathbf{x}^{(k)}$ as discussed in the previous

```
1  Inputs:
2   - adjacency matrix A (in compressed column storage)
3   - p as the inverse degree vector of A,
4   - d as the degree vector of A,
5   - 0 < α < 1,
6   - v is an integer giving the column of X to compute
7   - error tolerance τ = (1 − α)/n
8   - one vector of memory x
9  initialize x as 0
10 set maxiter = 2 log(τ)/ log(α)
11 for iter=1:maxiter
12   set δ = 0
13   for i=1:n
14     update = 0
15     for j in nonzeros(A[:,i])
16       update += x[j]
17     end
18     update *= α
19     if i == v # handle the right hand side
20       update += 1 − α
21     end
22     δ ← δ + update
23     x[i] ← update ∗ p[i]
24   end
25   if δ ≥ 1 − τ
26     x ← x ⊙ d # scale x by d element−wise
27     return x and converged
28   end
29 end
```

Fig. 2 Pseudocode for the Gauss–Seidel method to compute the vth column of $\boldsymbol{X} = (1-\alpha)(\boldsymbol{I} - \alpha \boldsymbol{P})^{-1}$. This algorithm takes one vector of memory. It maintains $\mathbf{x}$ as the Gauss–Seidel iterate elementwise scaled by $\mathbf{p}$. This performs random reads from the memory in $\mathbf{x}$, and then linearly ordered writes to the same memory $\mathbf{x}$. This works like the power-method from Fig. 1 where the updates are immediately applied in the vector $\mathbf{x}$

paragraph. In what follows $\mathbf{x}$ is the solution vector. However, let us note that what constitutes an iteration is not the loop on line 11, but the loop on line 13. This is because this method is easiest to analyze if we only consider what happens when a single element of $\mathbf{x}^{(k)}$ is changed on Line 23. In our analysis, we will show that each iterate $\mathbf{x}^{(k)}$ is bounded above by the true solution $\mathbf{x}$. Formally, this can be stated as $\mathbf{x}^{(k)} \leq \mathbf{x}$. We will establish this by showing that iterates only increase the value of $\mathbf{x}^{(k)}$ and they never get too large. If $\mathbf{x}^{(k)} \leq \mathbf{x}$ is the case, then the error

$$\|\mathbf{x} - \mathbf{x}^{(k)}\|_1 = \sum_i [\mathbf{x} - \mathbf{x}^{(k)}]_i = \sum_i [\mathbf{x}]_i - \sum_i [\mathbf{x}^{(k)}]_i = 1 - \sum_i [\mathbf{x}^{(k)}]_i.$$

Here, we only used that the sum of the entire PageRank vector $\sum_i [\mathbf{x}]_i = 1$ for the true solution on a strongly connected graph. Note that in line 22, we update δ which is tracking the sum of the unscaled vector $\mathbf{x}^{(k)}$ and after the full loop on 13–24, we have computed $\delta = \sum_i [\mathbf{x}^{(k)}]_i$. Consequently, this termination criteria maps to what we use in the algorithm.

Now, it remains to show that we indeed have the solution upper bounding each unscaled iterate. Note that, because $\mathbf{x}^{(0)} = 0$ we immediately have $\mathbf{x}^{(0)} \leq \mathbf{x}$. We will also strengthen our setup and note that the residual of the linear system (1)

$$\mathbf{r}^{(0)} = (1-\alpha)\mathbf{v} - (\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x}^{(0)}$$

is also non-negative. The importance of the relationship with the residual is that the residual and error satisfy the following system of equations:

$$(\boldsymbol{I} - \alpha \boldsymbol{P})(\mathbf{x} - \mathbf{x}^{(k)}) = \mathbf{r}^{(k)}.$$

The matrix $(\boldsymbol{I} - \alpha \boldsymbol{P})$ is an M-matrix [27] with a non-negative inverse, so the error vector $\mathbf{x} - \mathbf{x}^{(k)} \geq 0$ when $\mathbf{r}^{(k)} \geq 0$. Thus, it suffices to show that $\mathbf{r}^{(k+1)} \geq 0$ given $\mathbf{r}^{(k)} \geq 0$. In this case, we know that $\mathbf{x}^{(k)}$ and $\mathbf{x}^{(k+1)}$ are the same in all but one coordinate. Let u correspond to the index i that is changed in iteration k. We compute

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mu_k \mathbf{e}_u,$$

where μ_k is the value of `update`$-\mathbf{e}_u^T\mathbf{x}^{(k)}$. Expanding out the code to get μ_k gives

$$\mu_k = \begin{cases} \alpha \sum_{j \to u} x_j^{(k)}/d_j - x_u^{(k)} & u \neq v \\ \alpha \sum_{j \to u} x_j^{(k)}/d_j + (1-\alpha) - x_u^{(k)} & u = v \end{cases}.$$

Note that μ_k is exactly the uth element of the residual of the linear system (1)

$$\mathbf{r}^{(k)} = (1-\alpha)\mathbf{v} - (\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x}^{(k)} \Rightarrow \mu_k = \mathbf{e}_u^T \mathbf{r}^{(k)}. \tag{2}$$

We have that $\mu_k \geq 0$ because $\mathbf{r}^{(k)} \geq 0$ by assumption. At this point, we still need to show that $\mathbf{r}^{(k+1)} \geq 0$, and we have

$$\begin{aligned} \mathbf{r}^{(k+1)} &= (1-\alpha)\mathbf{v} - (\boldsymbol{I} - \alpha \boldsymbol{P})(\mathbf{x}^{(k)} + \mu_k \mathbf{e}_u) = \mathbf{r}^{(k)} - \mu_k(\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{e}_u \\ &= (\mathbf{r}^{(k)} - \mu_k \mathbf{e}_u) + \underbrace{\mu_k \alpha \boldsymbol{P} \mathbf{e}_u}_{\text{non-negative}} \end{aligned}$$

Now, we also have that

$$[\mathbf{r}^{(k)} - \mu_k \mathbf{e}_u]_i = \begin{cases} 0 & i = u \\ [\mathbf{r}^{(k)}]_i & i \neq u \end{cases} \geq 0$$

because μ_k is the uth component of $\mathbf{r}^{(k)}$ (see (2)). Thus we have $\mathbf{r}^{(k+1)} \geq 0$.

This justifies that the algorithm in Fig. 2, if it terminates, will have the correct error. To see that it will terminate, note that this same analysis shows that we reduce the sum of the residual at each step of the algorithm. We can also get convergence through classical results about the convergence of Gauss–Seidel on M-matrices [45].

Although there is no sub-asymptotic theory about Gauss–Seidel compared with the power method, ample empirical evidence suggests that, for most graphs, Gauss–Seidel runs in about half the iterations of PageRank. The asymptotic theory in Varga [45] shows that Gauss–Seidel is asymptotically faster than the power method. However, this is in terms of the spectral radius alone. This asymptotic theory, however, can be misleading for PageRank as an example with a random graph from [13] shows. To foreshadow our results, Gauss–Seidel will be the method to beat for computing PageRank with a single thread. This mirrors results found in other scenarios as well [15, 36].

3.3 The Cyclic Push Method

One challenge with Gauss–Seidel is that it requires in-neighbor access to the edges of the graph. These are still accessed consecutively, which makes streaming solutions a possibility. There are nevertheless many graph systems that provide the most efficient access to the out-neighbors of a directed graph. It turns out that there is a way to implement the Gauss–Seidel for these systems using something called the *push method* for PageRank, the big difference, however, is that we maintain two vectors of memory. The first variant of the *push* method we will describe will exactly map to the Gauss–Seidel computation above. The key difference is that it explicitly maintains a residual vector.

Suppose we kept a solution vector $\mathbf{x}^{(k)}$ along with a residual vector $\mathbf{r}^{(k)}$. Then the single-entry update in Gauss–Seidel corresponds to

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{e}_u \mathbf{e}_u^T \mathbf{r}^{(k)}.$$

(This expression arises from (2) combined with the μ_k variation on the Gauss–Seidel update.) This is easy to compute, but then we have to update $\mathbf{r}^{(k)}$ to get the new residual $\mathbf{r}^{(k+1)}$. In the push method, this second update dominates the work.

Recall the expression for the residual update that arose in our theory on Gauss–Seidel

$$\mathbf{r}^{(k+1)} = (\mathbf{r}^{(k)} - \mu_k \mathbf{e}_u) + \mu_k \alpha \boldsymbol{P} \mathbf{e}_u.$$

```
1  Inputs:
2   - adjacency matrix A (in compressed row storage),
3   - p as the inverse degree vector of A,
4   - 0 < α < 1,
5   - v is an integer giving the column of X to compute
6   - error tolerance τ = (1 − α)/n
7   - two vector of memory x, r
8  initialize x, r as 0
9  set r[v] ← (1 − α)
10 set maxiter = 2 log(τ)/ log(α)
11 for iter=1:maxiter
12   set δ = 0
13   for i=1:n
14     μ = r[i]
15     x[i] += μ
16     δ ← δ + x[i]
17     # now handle the residual update
18     r[i] = 0
19     ρ = α ∗ μ ∗ p[i]
20     for j in nonzeros(A[i,:])
21       r[j] += ρ
22     end
23   end
24   if δ ≥ 1 − τ
25     return x and converged
26   end
27 end
```

Fig. 3 Pseudocode for the cyclic push method to compute the vth column of $\boldsymbol{X} = (1-\alpha)(\boldsymbol{I} - \alpha\boldsymbol{P})^{-1}$. This algorithm takes two vectors of memory. It maintains $\mathbf{x}$ as the Gauss–Seidel iterate and $\mathbf{r}$ as the residual $(1-\alpha)\mathbf{e}_v - (\boldsymbol{I} - \alpha\boldsymbol{P})$. This performs random writes to the memory in $\mathbf{r}$. Note that this iteration is mathematically identical to Fig. 2, but it uses compressed row storage for $\boldsymbol{A}$ instead of compressed column storage

To perform this update, all we need to do is set the uth element of $\mathbf{r}^{(k)}$ to 0, and then lookup the values of the uth column of $\boldsymbol{P}$. Note that the matrix $\boldsymbol{P} = \boldsymbol{A}^T\text{Diag}(\mathbf{p})$ and so the uth column is just the uth row of $\boldsymbol{A}$, which encodes the out-neighbors, scaled by $p[u]$. The resulting algorithm is given by the pseudocode in Fig. 3.

This iteration is mathematically identical to Gauss–Seidel. The iteration in this form was described by McSherry [36] as an alternative way of computing PageRank that was more amenable to optimization because we can use properties of the residual to choose when to revisit or skip updating a node. The term "push" comes from the idea that when you update $x[i]$ you "push" an update out to the neighbors of i in the residual vector.

3.4 The Push Method With a Work Queue

The name "push method" actually comes from [2]. That paper utilized the push method to compute a personalized PageRank vector of an undirected graph in *constant* time (where the constant depends on α and the accuracy τ) for a weaker notion of error. This weak notion corresponds to finding an iterate with error that satisfies $0 \leq \mathbf{x} - \mathbf{x}^{(k)} \leq \tau \mathbf{d}$. So the error on a node with a large degree could be large. This enabled a number of clever ideas to show that this can be done in work that does not depend on the size of the graph. One of the key ideas is that this algorithm maintains a *queue* of vertices to process, and hence, avoids storing or working with vectors that are the size of the graph.

In this case, we adopt similar ideas and add a work queue of vertices that have not yet satisfied their tolerance. In comparison with the cyclic push method, this maintains the same amount of memory, in addition, when the residual associated with a vertex goes above a threshold, we add it to a queue to process in the figure. Namely, if the residual on a node is ω, then we can show that the maximum change to the solution vector due to that element is $\omega(1-\alpha)$. There might be as many as n items in the residual, so if we want a solution that is accurate to 1-norm error τ, then we can check if the residual is smaller than $(1-\alpha)\tau/n$. If it is smaller than this, we can show it will not impact the solution.

The pseudocode with the queue is in Fig. 4. The algorithm is identical to Fig. 3, except that we visit vertices in the order that they have been added to the queue. The only small subtlety is that we can check if a vertex is in the queue in order to avoid adding it multiple times based on the current value of the residual. In Line 25, we check if this is the first time that the element increased beyond the threshold ω. The other small detail is that we keep a running sum of the vector $\mathbf{x}$ in δ, which is incremented based on the value of μ at each step. In a low-precision implementation, this sum would need to be accumulated at a higher precision as it involves an extremely large running sum. As such, we can use the previous error analysis which justifies that when the total sum of the vector $\mathbf{x}$ exceeds τ, then we have converged.

3.5 Related Algorithmic Advances

It was [21] and [36] that realized that the *push* formulation offered a number of additional opportunities to accelerate PageRank computation by skipping and optimizing potential updates in a Gauss–Seidel-like fashion. These were later improved upon by [6] and [2] with the idea of the workqueue. The connection to Gauss–Seidel only arose later [8, 11]. The algorithms in our paper do not use the full flexibility of these methods as they are often specialized techniques that arise for web-graphs.

```
1  Inputs:
2   - adjacency matrix A (in compressed row storage),
3   - p as the inverse degree vector of A,
4   - 0 < α < 1,
5   - v is an integer giving the column of X to compute
6   - error tolerance τ = (1 − α)/n
7   - two vector of memory x, r
8  initialize x, r as 0
9  set r[v] ← (1 − α)
10 set maxiter = 2n log(τ)/ log(α)
11 set δ = 0
12 let Q be a queue initialized with vertex v
13 ω = (1 − α)τ/n
14 while Q is not empty
15   i ← next(Q)
16   μ = r[i]
17   x[i] += μ
18   δ ← δ + μ
19   # now handle the residual update
20   r[i] = 0
21   ρ = α ∗ μ ∗ p[i]
22   for j in nonzeros(A[i,:])
23     rj = r[j]
24     r[j] = rj + ρ
25     if rj < ω and rj + ρ ≥ ω
26       add j to the end of the Q
27     end
28   if δ ≥ 1 − τ
29     return x and converged
30   end
31 end
```

Fig. 4 Pseudocode for the push method with a work queue to compute the vth column of $\boldsymbol{X} = (1-\alpha)(\boldsymbol{I} - \alpha \boldsymbol{P})^{-1}$. This algorithm takes two vectors of memory. It maintains $\mathbf{x}$ as the Gauss–Seidel iterate and $\mathbf{r}$ as the residual $(1-\alpha)\mathbf{e}_v - (\boldsymbol{I} - \alpha \boldsymbol{P})$. This performs random writes to the memory in $\mathbf{r}$ and picks what amounts to a randomly scattered entry of i to process next

We have ignored here a wide class of methods for PageRank that work via Monte Carlo approaches [3–5, 32]. These methods all have trouble getting high accuracy entries, although they tend to get the top-k lists correct and should be considered for applications that only desire that type of information. Krylov methods are only competitive for PageRank when α is extremely large [18]. There are have been numerous attempts to parallelize the computation of a single PageRank vector [17]—especially on graph processing systems [1, 9, 26, 33, 43, 44]. In particular, these methods often utilize ideas closely related to the workqueue notion of the push method. Analysis of these results show that they often fail to be useful

parallelizations of the underlying problem and have significant overhead compared to simple implementations [37].

3.6 Multivector Transformations

The algorithms described so far here—and most of the discussions of PageRank that we are aware of—deal with computing a single PageRank or personalized PageRank vector. (The biggest exception are a number of techniques to attempt to approximate all PageRank vectors [4, 21].) With the idea in mind that we are considering an educated, but non-expert, user of these algorithms we note the following idea. Modern processors feature vector execution units often called SIMD (single instruction, multiple data) or simply vector instructions. Because the data access pattern for the power method, Gauss–Seidel, and the cyclic push method are entirely independent of both the choice of the right hand side $\mathbf{e}_i$ and any elements of the vectors, then we can conceptually execute the same iteration on multiple vectors simultaneously. This involves few changes to the code assuming that the language supports some notion of treating a *vector of entries* like a scalar. Thus, for each of the methods above, we create a variation that processes multiple vectors simultaneously. Our technique to do this in the Julia programming language is to replace a one dimensional array of data with a one dimension array of statically sized vectors. This enables the compiler to unroll and auto vectorize code that involves multiple entries at once in a way that is consistent with our informed user persona. The code is essentially unchanged from the previous cases and we refer interested readers to our online codes to reproduce these ideas. (See Sect. 6.)

4 Results

We now conduct a set of experiments using these four PageRank algorithms in the setting of the *AllPageRank* problem. That is, we run them to compute *multiple columns* of the matrix $\boldsymbol{X}$. The primary performance measure we are considering is the *number of columns computed* per unit time. We run the algorithms for one of two time intervals: 14.4 min and 5 min. Note that 14.4 min is exactly 1/100th of a day, and so the number of vectors computable in 24 h is exactly 100 times greater. For 5 min, the factor is roughly 300 times larger. Note that the *AllPageRank* problem involves a great deal of computation, and so it is natural to, perhaps, think of running this for a few days. Months or weeks are less reasonable, though.

We consider two parallelization strategies: threads and processes. In the threaded implementation, we load the graph information into memory once and use the high-level language's threading library to launch a given number of computation threads. These threads continue to compute single columns, or multiple columns simultaneously, of the solution until the time limit is exhausted. They all access

the same shared memory copy. The process scenario is largely the same, except we launch independent processes that all have their own copy of the graph information. Note that we do not consider any parallel setup or IO time; but let us state that this was negligible for our experiments—it might take 1–5 min to set up an execution which we expect to run for hours. Our code for these experiments is all available online (see Sect. 6 for the reference).

Also in keeping with our informed user persona, we did not perform any heroic measures to eliminate all simultaneous usage of the machine. We asked other users not to use the machines during our tests, which, we believe was respected. There were a few processes from other users that would appear to be doing intermittent work. (As an example, we may see someone running the unix "top" command to see if the machine was being used).

4.1 Data and Machines

We report on two datasets, each of which is a strongly connected component of a larger graph. These data come from [29, 38].

- **Orkut** has 2, 997, 355 nodes and 220, 259, 736 directed edges.
- **LiveJournal** has 3, 828, 682 nodes and 65, 825, 429 directed edges.

The two machines we use are:

- A 64-core (4 × 16-core) shared memory server with Xeon E7-8867 v3 (2.50 GHz) CPUs and 2 TB of RAM; this is configured in a fully connected topology. Each processor has four memory channels, 45 MB of L3 cache, and 256 KB × 16 of L2 cache.
- A 192-core (8 × 24-core) shared memory server with Xeon Platinum 8168 (2.70 GHz) CPUs and 6 TB of RAM; this is configured in a hypercube topology with three connections per CPU. Each processor has six memory channels, 33 MB L3 Cache, and 24 × 1 MB of L2 cache.

4.2 Performance on a 64-Core System

We begin our discussion by looking at the results of all the algorithms on the 64-core server as these are the simplest to understand. These are summarized in Table 1, which shows how performance varies on 1, 32, and 64 threads and processes when we compute 1, 8, or 16 vectors simultaneously. In principle, using multiple vectors simultaneously will result in the Julia compiler generating AVX and SIMD instructions on the platform, which can greatly increase the computational power. We see that this increases performance by around a factor of 4 or 5. We see only a

Table 1 Vectors computed on the LiveJournal graph within 14.4 min

(a) *Threads*									
	Vectors								
	1			8			16		
	Threads								
Method	1	32	64	1	32	64	1	32	64
Power	25	399	731	88	1984	2752	96	1584	2784
Gauss–Seidel	46	675	1324	152	3128	5320	176	3248	5680
Cyclic push	41	626	849	104	2240	3192	96	1840	3040
Queue push	12	213	323	56	944	1504	64	1232	2048
(b) *Processes*									
	Vectors								
	1			8			16		
	Processes								
Method	1	32	64	1	32	64	1	32	64
Power	24	438	697	88	1800	2920	96	1696	2912
Gauss–Seidel	28	690	965	144	3064	5704	176	3616	5632
Cyclic push	27	544	723	96	1992	2848	96	1776	2480
Queue push	9	189	302	40	880	1536	48	1296	1872

These results are from a threads (a) and processes (b) implementation on the 64-core server on the 64-core server. For each method, we vary the number of vectors computed simultaneously among 1, 8, and 16 along with varying the number of threads from 1, 32, to 64

small change going from 8 to 16 vectors computed simultaneously, and sometimes this will decrease performance (see the threaded results on the power method and processes results for Gauss–Seidel). The results with processes are generally, but not always, faster than the results with the same number of threads.

Note that the power method uses more iterations than either the Gauss–Seidel and cyclic push methods, and so we expect it to be slower from an algorithm perspective (although the memory access patterns are more amenable to parallelization). Gauss–Seidel and the cyclic push methods are mathematically identical and so execute the same number of iterations. The difference in performance is entirely due to the memory access patterns. These results show that it is better to have random reads than random writes as the power method is faster than the cyclic push method. Although the Queue Push method should do the least work of all, it seems that the additional cost of maintaining the queue causes the method to run the slowest.

In summary, these results point to challenges in linearly scaling the work involved in this pleasingly parallel computation. They also highlight the need to compute multiple vectors simultaneously. Note that running Gauss–Seidel with one process or thread produces about half the output of the power method with 32 threads computing only one vector at a time.

4.3 Sparse Matrix Ordering

The next experiment we consider is using a sparse matrix ordering scheme to improve the locality of reference among the operations. This is a standard technique in sparse matrix computations that is commonly taught in graduate curricula. We use the METIS algorithm [24] and generate 50 and 100 partitions. We then re-order the matrix so that each partition is a consecutive block. Since the computations with multiple vectors all had uniformly higher performance, we only report the results for the methods that compute eight vectors simultaneously.

Again, these results show a considerable increase in performance for most methods. The performance of Gauss–Seidel increases by 30%, for instance. Notable exceptions include the power method and Queue Push methods on Orkut. The partitions took less than an hour to compute. Since we envision running these computations for over 10 h, the permuted method would overtake the non-permuted one after about 4 h. Consequently, it seems this technique is still worth doing even for these pleasingly parallel computations. In particular, note that the cyclic push method shows a very large change in performance and largely runs faster than the power method in all cases. Given the random write nature of this work, this is perhaps unsurprising, but it is useful to know that this type of algorithm is especially sensitive to ordering (Table 2).

4.4 Performance on a 192-Core System

Next, we investigate how performance changes on a 192-core system for the algorithms that run eight vectors simultaneously. Table 3 shows the results for the threaded and independent process scenarios. This table highlights the problem with scaling threaded computation on this particular system. As the number of threads

Table 2 The change in the number of vectors computed on LiveJournal and Orkut as we vary the sparse matrix order shows that a small bit of careful ordering dramatically improves performance

	(a) LiveJournal			(b) Orkut		
	Ordering			Ordering		
Method	Native	50	100	Native	50	100
Power	2752	3312	3544	904	896	944
Gauss–Seidel	5320	7128	6864	1744	1920	2120
Cyclic push	3192	4272	4928	696	1104	1176
Queue push	1504	1808	1936	512	424	512

These results are from the 64-core server in a threading environment with the 14.4 min interval. The ordering varies from the native order of the file as it emerged from the strongly connected component computation to one computed using 50 and 100 partitions with METIS. The algorithms all compute eight vectors simultaneously. Performance improves for all algorithms except the queue push and power methods on Orkut. Note the dramatic increase in the performance of cyclic push on Orkut

Table 3 Vectors computed by the 192-core server show problems with scaling the threaded computation

(a) *LiveJournal*						
	Threads or processes					
Method	1T	1P	96T	96P	192T	192P
Power	104	120	3232	5960	272	7768
Gauss–Seidel	264	272	10,248	13,920	3592	19,160
Cyclic push	160	168	6432	7728	4752	9920
Queue push	64	64	2752	3256	1808	4496
(b) *Orkut*						
	Threads or processes					
Method	1T	1P	96T	96P	192T	192P
Power	40	40	1048	1768	0	888
Gauss–Seidel	88	88	2384	4184	0	5800
Cyclic Push	48	48	1584	1864	1040	2016
Queue Push	24	24	680	872	0	920

These results show both the threaded (T) and process (P) environment with the 14.4 min interval. These results all used the ordering computed with 50 partitions and the algorithms all compute eight vectors simultaneously. We repeated the experiment with 192 threads and verified a similar result from another trial; we were unable to determine a cause for why these results showed *no* vectors computed

increases, the performance decreases. We investigate this finding in the next section (Sect. 4.5) as well. In fact, on the Orkut networks, there is no work done when using 192 threads within 14.4 min for most of the trials. We repeated this trial to verify that the result was consistent—it was.

Overall, these results show challenges when using threads on a machine with a more complex memory topology, even when using pleasingly parallel computations.

4.5 Performance Scaling

The final experiment we conduct is a performance scaling study for the best algorithm we found: the SIMD Gauss–Seidel algorithm. We use eight-vectors as there was only a minor performance difference (if any) for the 16-vector variant. Here we also use the data that has been reordered with METIS in order to get a sense of scaling when the computation is performing well. We vary the number of threads or processes in each system and report the scaling results in Fig. 5. These show that the threading performance quickly degrades on the machine with 192 cores and the per-process implementation is needed to get good scaling results. Note also that neither setup *scales* particularly well for a pleasingly parallel computation.

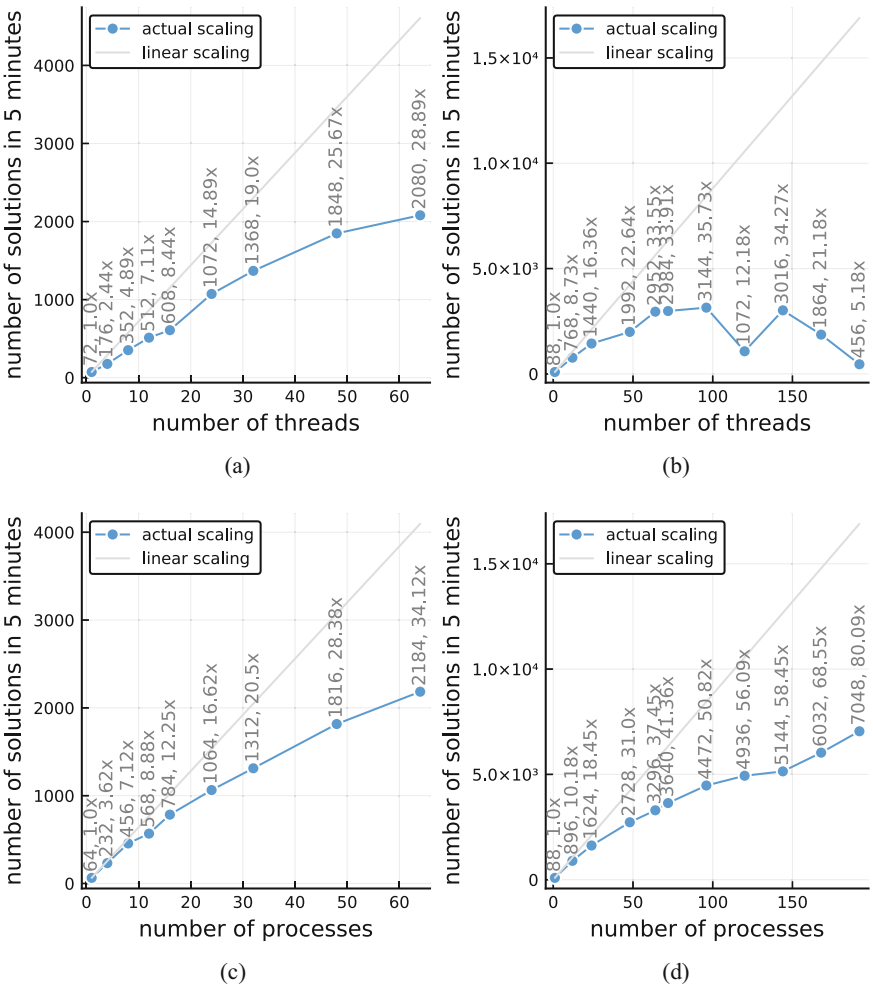

Fig. 5 Scaling results for threads and processes implementations. The text annotations give the raw number of vectors computed by that method within 5 min as well as the speedup ratio over the 1 thread or process result. (**a**) 64-core, threaded. (**b**) 192-core, threaded. (**c**) 64-core, processes. (**d**) 192-core, processes

5 Related Problems

AllPageRank is just a simple instance of a more general need for this type of computation. A closely related methodology underlies the *network community profile* calculation [12, 22, 30, 34]. This setting involves running a local clustering algorithm for hundreds or thousands of times—independently—on a shared graph. These computations often take hours to run on graphs of similar size.

A related computation is the GHOST technique used for network alignment [42]. This calculation extracts a subset of vertices from a large graph and then computes an eigenvalue histogram on the induced subgraph. These histograms are used as a invariant and characteristic feature for network alignment methods. (As an aside, we note that there are better ways to get a related concept called the *network density of states* [10].)

In summary, the style of computation used for the AllPageRank problem occurs repeatedly and is worth understanding given that the computations often consume considerable time and informed users.

6 Conclusion

The focus of this manuscript is on a *pleasingly parallel* computation: the AllPageRank problem we introduce. When we investigated computing the vectors involved in this problem on two shared memory parallel systems, it showed that expecting *linear* speedup on these problems is unrealistic. Even in this simple case, our results show that two ideas are crucial to get reasonable performance:

- computing multiple vectors simultaneously
- using matrix ordering techniques.

Both of these are easy to incorporate into parallel execution libraries that could be designed for this class of tasks, which is distinct from the current focus of distributed graph computation libraries. Our code is available online for others to reproduce our findings on new and emerging systems: https://github.com/YanfeiRen/pagerank

Back to the problem at hand, we are able to compute around 20,000 columns of $\boldsymbol{X}$ for the LiveJournal graph in 14.4 min. This shows that it would take around two days with 192 cores to generate all the information for the AllPageRank problem. For the Orkut graph, it would take around 5 days. We note that both are reasonable and acceptable runtimes to generate an interesting derived dataset. Waiting a week for an experiment is a fairly standard scenario in the physical sciences.

That said, this is still an expensive computation. Making these techniques commonplace on graphs of this scale would likely require another factor of 10 increase in performance so that the results come in 5 h on 192 cores, or say, 15 h on 64 cores. Monte Carlo techniques may be one possibly, along with reduced precision computation. Our experiments all used 64-bit floating point values. The computation may be possible in 32-bit floating point values although it will require some care as values such as $1/4,000,000$ are within a factor of 10 of the unit roundoff value for 32-bit floats. Finally, we note that there are methods that should further accelerate Gauss–Seidel, such as successive over-relaxation. While there is a negative finding about SOR on general PageRank systems [20], there are many PageRank systems and near relative PageRank systems that would use symmetric positive definite matrices [34] where SOR, with the optimal choice of ω, might be

productive. Preliminary tests show this yields another 2–3 fold improvement for undirected graphs.

We realize that there are additional strategies that an expert could take to improve performance such as developing custom routines to control memory placement and thread locality. We note, however, that these tools are difficult to access from high-level libraries where our hypothetical informed user resides.

Acknowledgments This work was supported in part by NSF IIS-1546488, CCF-1909528, the NSF Center for Science of Information STC, CCF-0939370, DOE award DE-SC0014543, NASA, and the Sloan Foundation.

References

1. Aberger, C.R., Tu, S., Olukotun, K., Ré, C.: Emptyheaded: A relational engine for graph processing. arXiv **cs.DB**, 1503.02368 (2015). URL http://arxiv.org/abs/1503.02368
2. Andersen, R., Chung, F., Lang, K.: Local graph partitioning using PageRank vectors. In: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (2006). URL http://www.math.ucsd.edu/~fan/wp/localpartition.pdf
3. Avrachenkov, K., Litvak, N., Nemirovsky, D., Osipova, N.: Monte Carlo methods in PageRank computation: When one iteration is sufficient. SIAM J. Numer. Anal. **45**(2), 890–904 (2007). DOI 10.1137/050643799. URL http://dx.doi.org/10.1137/050643799
4. Avrachenkov, K., Litvak, N., Nemirovsky, D., Smirnova, E., Sokol, M.: Quick detection of top-k personalized PageRank lists. In: A. Frieze, P. Horn, P. Prałat (eds.) Algorithms and Models for the Web Graph, pp. 50–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
5. Bahmani, B., Chakrabarti, K., Xin, D.: Fast personalized PageRank on MapReduce. In: Proceedings of the 2011 international conference on Management of data, SIGMOD '11, pp. 973–984. ACM, New York, NY, USA (2011). DOI 10.1145/1989323.1989425
6. Berkhin, P.: Bookmark-coloring algorithm for personalized PageRank computing. Internet Mathematics **3**(1), 41–62 (2007). URL http://www.internetmathematics.org/volumes/3/1/Berkhin.pdf
7. Bianchini, M., Gori, M., Scarselli, F.: Inside PageRank. ACM Transactions on Internet Technologies **5**(1), 92–128 (2005). DOI 10.1145/1052934.1052938
8. Boldi, P., Vigna, S.: The push algorithm for spectral ranking. arXiv **cs.SI**, 1109.4680 (2011). URL https://arxiv.org/abs/1109.4680
9. Ching, A., Kunz, C.: Giraph: Large-scale graph processing infrastructure on Hadoop. In: Proceedings of the Hadoop Summit (2011)
10. Dong, K., Benson, A.R., Bindel, D.: Network density of states. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD '19. ACM Press (2019). DOI 10.1145/3292500.3330891. URL https://doi.org/10.1145/3292500.3330891
11. Esfandiar, P., Bonchi, F., Gleich, D.F., Greif, C., Lakshmanan, L.V.S., On, B.W.: Fast Katz and commuters: Efficient approximation of social relatedness over large networks. In: Algorithms and Models for the Web Graph (2010). DOI 10.1007/978-3-642-18009-5_13
12. Fountoulakis, K., Gleich, D.F., Mahoney, M.W.: A short introduction to local graph clustering methods and software. In: Book of Abstracts for 7th International Conference on Complex Networks and Their Applications, pp. 56–59 (2018)
13. Gleich, D.F.: Models and algorithms for PageRank sensitivity. Ph.D. thesis, Stanford University (2009). URL http://www.stanford.edu/group/SOL/dissertations/pagerank-sensitivity-thesis-online.pdf

14. Gleich, D.F.: PageRank beyond the web. SIAM Review **57**(3), 321–363 (2015). DOI 10.1137/140976649
15. Gleich, D.F., Gray, A.P., Greif, C., Lau, T.: An inner-outer iteration for PageRank. SIAM Journal of Scientific Computing **32**(1), 349–371 (2010). DOI 10.1137/080727397
16. Gleich, D.F., Mahoney, M.W.: Mining large graphs. In: P. Bühlmann, P. Drineas, M. Kane, M. van de Laan (eds.) Handbook of Big Data, Handbooks of modern statistical methods, pp. 191–220. CRC Press (2016). DOI 10.1201/b19567-17
17. Gleich, D.F., Zhukov, L.: Scalable computing with power-law graphs: Experience with parallel PageRank. In: SuperComputing 2005 (2005). URL http://www.cs.purdue.edu/homes/dgleich/publications/gleich2005-parallelpagerank.pdf. Poster.
18. Golub, G., Greif, C.: An Arnoldi-type algorithm for computing PageRank. BIT Numerical Mathematics **46**(4), 759–771 (2006). DOI 10.1007/s10543-006-0091-y
19. Golub, G.H., van Loan, C.: Matrix Computations. Johns Hopkins University Press (2013)
20. Greif, C., Kurokawa, D.: A note on the convergence of SOR for the PageRank problem. SIAM Journal on Scientific Computing **33**(6), 3201–3209 (2011). DOI 10.1137/110823523. URL https://doi.org/10.1137/110823523
21. Jeh, G., Widom, J.: Scaling personalized web search. In: Proceedings of the 12th international conference on the World Wide Web, pp. 271–279. ACM, Budapest, Hungary (2003). DOI 10.1145/775152.775191
22. Jeub, L.G.S., Balachandran, P., Porter, M.A., Mucha, P.J., Mahoney, M.W.: Think locally, act locally: Detection of small, medium-sized, and large communities in large networks. Phys. Rev. E **91**, 012821 (2015). DOI 10.1103/PhysRevE.91.012821
23. Jiang, B., Kloster, K., Gleich, D.F., Gribskov, M.: AptRank: an adaptive PageRank model for protein function prediction on bi-relational graphs. Bioinformatics **33**(12), 1829–1836 (2017). DOI 10.1093/bioinformatics/btx029
24. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998). DOI 10.1137/S1064827595287997
25. Kurokawa, D., Gleich, D.F., Greif, C.: Prpack. Github repository, https://github.com/dgleich/prpack (2013). URL https://github.com/dgleich/prpack
26. Kyrola, A., Bllelloch, G., Guestrin, C.: GraphChi: Large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (2012)
27. Langville, A.N., Meyer, C.D.: Deeper inside PageRank. Internet Mathematics **1**(3), 335–380 (2004). URL http://www.ams.org/msnmain?fn=130&form=fullsearch&pg4=MR&s4=2111012
28. Langville, A.N., Meyer, C.D.: Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press (2006)
29. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (2014)
30. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Statistical properties of community structure in large social and information networks. In: WWW '08: Proceeding of the 17th international conference on World Wide Web, pp. 695–704. ACM, New York, NY, USA (2008). DOI 10.1145/1367497.1367591
31. Lin, C.H., Konecki, D.M., Liu, M., Wilson, S.J., Nassar, H., Wilkins, A.D., Gleich, D.F., Lichtarge, O.: Multimodal network diffusion predicts future disease–gene–chemical associations. Bioinformatics p. bty858 (2018). DOI 10.1093/bioinformatics/bty858
32. Lofgren, P.A., Banerjee, S., Goel, A., Seshadhri, C.: FAST-PPR: Scaling personalized PageRank estimation for large graphs. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, pp. 1436–1445. ACM, New York, NY, USA (2014). DOI 10.1145/2623330.2623745
33. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: A framework for machine learning and data mining in the cloud. In: Proceedings of the VLDB Endowment, vol. 5, pp. 716–727 (2012)

34. Mahoney, M.W., Orecchia, L., Vishnoi, N.K.: A local spectral method for graphs: With applications to improving graph partitions and exploring data graphs locally. Journal of Machine Learning Research **13**, 2339–2365 (2012). URL http://www.jmlr.org/papers/volume13/mahoney12a/mahoney12a.pdf
35. McGlohon, M., Akoglu, L., Faloutsos, C.: Weighted graphs and disconnected components: patterns and a generator. In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '08, pp. 524–532. ACM, New York, NY, USA (2008). DOI 10.1145/1401890.1401955
36. McSherry, F.: A uniform approach to accelerated PageRank computation. In: Proceedings of the 14th international conference on the World Wide Web, pp. 575–582. ACM Press, New York, NY, USA (2005). DOI 10.1145/1060745.1060829
37. McSherry, F., Isard, M., Murray, D.G.: Scalability! but at what cost? In: 15th Workshop on Hot Topics in Operating Systems (HotOS XV). USENIX Association, Kartause Ittingen, Switzerland (2015). URL http://blogs.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry
38. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC '07, pp. 29–42. ACM, New York, NY, USA (2007). DOI 10.1145/1298306.1298311
39. Page, L.: Method for node ranking in a linked database (2001). URL http://www.freepatentsonline.com/6285999.pdf
40. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the web. Tech. Rep. 1999-66, Stanford University (1999). URL http://dbpubs.stanford.edu:8090/pub/1999-66
41. Pan, J.Y., Yang, H.J., Faloutsos, C., Duygulu, P.: Automatic multimedia cross-modal correlation discovery. In: KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 653–658. ACM, New York, NY, USA (2004). DOI 10.1145/1014052.1014135
42. Patro, R., Kingsford, C.: Global network alignment using multiscale spectral signatures. Bioinformatics **28**(23), 3105–3114 (2012). DOI 10.1093/bioinformatics/bts592
43. Perez, Y., Sosic, R., Banerjee, A., Puttagunta, R., Raison, M., Shah, P., Leskovec, J.: Ringo: Iinteractive graph analytics on big-memory machines. In: Proceedings of the ACM SIGMOD Conference (2015)
44. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Mendez-Lojo, M., Prountzos, D., Sui, X.: The tao of parallelism in algorithms. In: Proceedings of the 32nd Conference on Programming Language Design and Implementation (2011)
45. Varga, R.S.: Matrix Iterative Analysis. Prentice Hall (1962)
46. Voevodski, K., Teng, S.H., Xia, Y.: Spectral affinity in protein networks. BMC Systems Biology **3**(1), 112 (2009). DOI 10.1186/1752-0509-3-112