Fast Personalized PageRank on MapReduce

Bahman Bahmani*
Stanford University
bahman@stanford.edu

Kaushik Chakrabarti
Microsoft Research
kaushik@microsoft.com

Dong Xin†
Google Inc.
dongxin@google.com

ABSTRACT

In this paper, we design a fast MapReduce algorithm for Monte Carlo approximation of personalized PageRank vectors of all the nodes in a graph. The basic idea is very efficiently doing single random walks of a given length starting at each node in the graph. More precisely, we design a MapReduce algorithm, which given a graph G and a length λ, outputs a single random walk of length λ starting at each node in G. We will show that the number of MapReduce iterations used by our algorithm is optimal among a broad family of algorithms for the problem, and its I/O efficiency is much better than the existing candidates. We will then show how we can use this algorithm to very efficiently approximate all the personalized PageRank vectors. Our empirical evaluation on real-life graph data and in production MapReduce environment shows that our algorithm is significantly more efficient than all the existing algorithms in the MapReduce setting.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; F.1.2 [**Computation By Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Personalized PageRank, MapReduce

1. INTRODUCTION

Very large scale datasets and graphs are ubiquitous in today's world: world wide web, online social networks, and huge search and query-click logs regularly collected and processed by search engines. Because of the massive scale of these datasets, doing analyses and computations on them is infeasible for individual machines. Therefore, there is a growing need for distributed ways of storing and processing these datasets. MapReduce, a simple model of computation, first introduced by Dean and Ghemawat [9], has recently emerged as a very attractive way of doing such analyses. Its effectiveness and simplicity has resulted in its implementation by different internet companies [9, 13, 5, 22], and widespread adoption for a wide range of applications [19], including large scale graph computations [15, 16].

One of the most well known graph computation problems is computing personalized PageRanks (PPR) [12]. Personalized PageRanks (and other personalized random walk based measures) have proved to be very effective in a variety of applications, such as link prediction [17] and friend recommendation [3] in social networks, and there are many algorithms designed to approximate them in different computational models [14, 3, 10, 25].

In this paper, we study the problem of Fully Personalized PageRank ($FPPR$) approximation on MapReduce. Specifically, we study the problem of approximating the personalized PageRank vectors of all nodes in a graph in the MapReduce setting, and present a fast MapReduce algorithm for Monte Carlo approximation of these vectors. Even though some of the previously designed personalized PageRank approximation algorithms can be implemented in MapReduce, we will show that our algorithm takes much better advantage of the parallel computation model of MapReduce and is hence significantly more efficient than the existing candidates in this setting. We also note that our algorithm can be used for computing other personalized random walk based measures (such as personalized SALSA [3]) in MapReduce as well.

In this introduction, we first provide some background on personalized PageRank and MapReduce, and then give the problem statements, and also outline our results.

1.1 Background

Here we review personalized PageRank, the Monte Carlo approach for PageRank computation, and MapReduce. Here, and throughout the paper, we assume to have a weighted directed graph $G = (V, E)$ with n nodes and m edges. We denote the weight on an edge $(u, v) \in E$ with $\alpha_{u,v}$ and, for the sake of simplifying the presentation of some of the formulae, assume for the rest of the paper that the weights on the outgoing edges of each node sum up to 1.

*Work done while visiting Microsoft Research.

†Work done while at Microsoft Research.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

1.1.1 Personalized PageRank

PageRank is the stationary distribution of a random walk that at each step, with a probability ϵ, usually called the teleport probability, jumps to a random node, and with probability $1-\epsilon$ follows a random outgoing edge from the current node. Personalized PageRank is the same as PageRank, except all the random jumps are done back to the same node, denoted as the "source" or "seed" node, for which we are personalizing the PageRank.

One can easily see that the personalized PageRank of node v, with respect to a source node u, denoted by $\pi_u(v)$, satisfies:

$$\pi_u(v) = \epsilon\delta_u(v) + (1-\epsilon) \sum_{\{w|(w,v)\in E\}} \pi_u(w)\alpha_{w,v} \qquad (1)$$

Where $\delta_u(v) = 1$ if and only if $u = v$ (and 0 otherwise).

The fully personalized PageRank computation problem is to compute all the vectors $\overrightarrow{\pi}_u$ for all $u \in V$. Of course, most applications, such as friend recommendation or query suggestion, only require the top-k values (and corresponding nodes) in each PPR vector (for some suitable value of k).

1.1.2 Monte Carlo Approach

There are two broad approaches to computing Personalized PageRank. The first approach is to use linear algebraic techniques, such as Power Iteration [23]. The other approach is Monte Carlo, where the basic idea is to approximate Personalized PageRanks by directly simulating the corresponding random walks and then estimating the stationary distributions with the empirical distributions of the performed walks. Based on this idea, Fogaras et al [10] and later Avrachenkov et al [2] proposed the following method for PPR approximation: Starting at each node $u \in V$, do a number, R, of random walks starting at u, called "fingerprints", each having a length geometrically distributed as Geom(ϵ). Then, the frequencies of visits to different nodes in these fingerprints will approximate the personalized PageRanks. Our algorithm also belongs to the Monte Carlo family.

1.1.3 MapReduce

MapReduce [9] is a simple computation model for processing huge amounts of data in massively parallel fashion, using a large number of commodity machines. By automatically handling the lower level issues, such as job distribution, data storage and flow, and fault tolerance, it provides a simple computational abstraction.

In MapReduce, computations are done in three phases. The *Map* phase reads a collection of values or key/value pairs from an input source, and by invoking a user defined *Mapper* function on each input element independently and in parallel, emits zero or more key/value pairs associated with that input element. The *Shuffle* phase groups together all the Mapper-emitted key/value pairs sharing the same key, and outputs each distinct group to the next phase. The *Reduce* phase invokes a user-defined *Reducer* function on each distinct group, independently and in parallel, and emits zero or more values to associate with the group's key. The emitted key/value pairs can then be written on the disk or be the input of a Map phase in a following iteration.

1.2 Problem Statement

In this paper, we study the problem of FPPR approximation on MapReduce (**FPPR-MapReduce**): *Design an efficient MapReduce algorithm that given a weighted directed graph $G = (V, E)$, approximately computes the personalized PageRank vectors $\overrightarrow{\pi_u}$ of all nodes $u \in V$.*

As stated earlier, we adopt the Monte Carlo approach, which requires simulating a number, R, of random walks (fingerprints) from each node. Therefore, we will need to solve the following sub-problem, that we call the Single Random Walk problem (**SRW-MapReduce**): *Design a MapReduce algorithm that given a graph G and a length λ, outputs one random walk of length λ starting from each node in the graph.*

1.3 Our Contribution

Intuitively speaking, to fully leverage the power of parallel computation supported by MapReduce, a good algorithm should have the following properties: (1) *high parallelization* and (2) *small number of MapReduce iterations*. The Monte Carlo approach for FPPR approximation naturally has the first property, as any fingerprint starting at any source node can be computed in parallel with and independently from all other fingerprints (for the same or different source nodes). However, as pointed out in [10], some of the fingerprints may be very long, and hence require a large number of MapReduce iterations using the straightforward implementation (e.g., one MapReduce iteration for each step in the walk). For instance, with $\epsilon = 0.2$, a fingerprint can be longer than 10 steps with probability 0.11, and can be longer than 20 steps with probability 0.012. These long walks will become the bottleneck of the algorithm, blocking the entire computation, and causing it to take too long to run.

In this paper, we develop an algorithm to compute single random walks of a given length for all nodes in a graph, and show that it is optimal in terms of the number of MapReduce iterations among a broad class of algorithms. Based on that, we then develop an efficient algorithm to approximate fully personalized PageRanks on MapReduce, and also analyze its I/O cost. Our empirical evaluation on real-life graph data and in production MapReduce environment demonstrates that our algorithm outperforms the state of the art FPPR approximation algorithms, in terms of efficiency and approximation error.

The rest of the paper is organized as follows. Section 2 gives the background for computing FPPR on MapReduce. The single random walk algorithm is presented in section 3, and the FPPR approximation algorithm is presented in section 4. We show experimental results in section 5, review the related work in section 6, and finally conclude this paper in section 7.

2. PRELIMINARIES

In this section, we first review the related programming model on MapReduce, which consists of two high level primitives, and then use these two operators to describe how existing PPR approximation algorithms can be implemented on MapReduce.

2.1 Programming Model on MapReduce

The basic MapReduce abstraction can sometimes prove highly restrictive, as it requires any computation to be translated into the rigid framework of one Map and one Reduce operation. As the computations get logically more complicated, this becomes increasingly more challenging. Therefore, many models and system implementations have been proposed to extend the basic MapReduce framework [1, 22,

5, 26]. These extensions provide higher level languages for the programmers (such as PigLatin or SCOPE), which include higher level primitives such as joins.

In this paper, we will use two high level primitives: Reducer and Combiner, where the Reducer is defined as:

$$\langle output\rangle = Reduce\ \langle input\rangle\ On\ \langle key\rangle$$
$$Using\ ReducerUDF$$

and the Combiner is defined as:

$$\langle output\rangle = Combine\ \langle input1\rangle\ With\ \langle input2\rangle$$
$$ON\ \langle input1.key1\rangle = \langle input2.key2\rangle$$
$$Using\ CombinerUDF$$

Clearly, a Reducer groups data from an input by its key, processes the data by a user defined function ReducerUDF, and sends results to output. Note that here the Reducer is a high level programming primitive, which is different from the concept of the reducer in the reduce phase of the traditional MapReduce framework (as discussed in the previous section). In the remaining of this paper, we will use the term Reducer to refer to such a high level primitive.

Given a graph data which is represented by triples $G = \langle u, v, weight\rangle$ (i.e., edges), the task of computing a random neighbor for each source node u can be implemented by a Reducer using u as the key, grouping all $\langle u, v, weight\rangle$ triples (for all v), and generating one random neighbor v for u and outputting $\langle u, v\rangle$ in ReducerUDF.

A Combiner is essentially a join primitive, which joins $input1$ and $input2$ by $input1.key1$ and $input2.key2$, and processes the data by a user defined function CombinerUDF. Suppose the random neighbor output in the above example is $N = \langle u, v\rangle$. The task of further extending $\langle u, v\rangle$ to $\langle u, v, t\rangle$ by finding a random neighbor t of v can be implemented by a Combiner by joining N and G on condition $N.v = G.u$, and finding a random neighbor t of $N.v$ from the graph G and outputting $\langle u, v, t\rangle$ in CombinerUDF.

2.2 FPPR Algorithms

FPPR approximation is a very well-studied problem. So, we start with briefly reviewing the existing algorithms, specially from a MapReduce perspective.

2.2.1 Power Iteration, Dynamic Programming, and Rounding

The simplest method for computing Personalized PageRanks is Power Iteration [23]. It starts with initializing $\pi_u^{(0)}(v) = \delta_u(v)$ (where u, v are arbitrary nodes in the graph), and then it repeatedly performs the following update:

$$\pi_u^{(i)}(v) = \epsilon\delta_u(v) + (1-\epsilon)\sum_{\{w|(w,v)\in E\}} \pi_u^{(i-1)}(w)\alpha_{w,v} \quad (2)$$

Similar to but subtly different from this method is the dynamic programming algorithm, first introduced by Jeh and Widom [14]. This algorithm starts with the same initialization as Power Iteration, but then performs the following update iteratively:

$$\overrightarrow{\pi_u}^{(i)} = \epsilon\overrightarrow{\delta_u} + (1-\epsilon)\sum_{\{v|(u,v)\in E\}} \alpha_{u,v}\overrightarrow{\pi_v}^{(i-1)} \quad (3)$$

Both of these algorithms entail exponential reduction in the approximation error, which is desirable. However, using them for large scale FPPR approximation is not feasible, because the sizes (i.e., number of non-zero elements) of their approximate PPR vectors grow very quickly as more iterations are done, and the algorithms will need as much as $\Theta(n^2)$ space, which is far from feasible for the typical sizes of today's large scale graphs. Of course, it should be mentioned that if all one needs to compute is the PPR vectors of a small subset of the nodes in the graph, then Power Iteration can be used effectively.

Sarlos et al. [25] observed that even though the sizes of the approximate PPR vectors of these algorithms grow very quickly, many of the (non-zero) values in these vectors are very small, and hence can be ignored (i.e., assumed to be 0), without causing a significant approximation error. Based on a refinement of this observation, they proposed an algorithm, that we denote as the Rounding algorithm, which performs similar iterations to the simple dynamic programming algorithm, but also rounds its approximate PPR vectors at the end of each iteration. More precisely, defining $r_i = \sqrt{(1-\epsilon)^i}$, and initializing $\overrightarrow{\pi_u}^{(0)} = \psi_0(\epsilon\overrightarrow{\delta_u})$, the algorithm repeatedly performs the following iteration:

$$\overrightarrow{\pi_u}^{(i)} = \psi_i\left(\epsilon\overrightarrow{\delta_u} + (1-\epsilon)\sum_{\{v|(u,v)\in E\}} \alpha_{u,v}\overrightarrow{\pi_v}^{(i-1)}\right) \quad (4)$$

where the function ψ_i, defined as $\psi_i(x) = r_i\lfloor x/r_i\rfloor$, rounds down its input to an integer multiple of r_i. Any value smaller than r_i gets rounded down to 0 by ψ_i, and this allows the algorithm to control the growth of the sizes of its approximate PPR vectors, and scale to full personalization.

Heuristics similar to the idea in the Rounding algorithm have been used in some other algorithms as well [20, 4], but they do not provide performance guarantees as Rounding does, and moreover Rounding has been previously observed to perform better in practice as well [25]. So, in this paper, we will compare our FPPR approximation algorithm with the Rounding algorithm, by analyzing their I/O efficiency and also empirically.

The power iteration algorithm and its variants can be easily implemented on MapReduce [18]. Here, we briefly discuss how to implement them using the Reducer and Combiner primitives described in the last subsection. Given a graph $G = \langle u, v, \alpha_{u,v}\rangle$, the initialization of $\overrightarrow{\pi_u}^{(0)}$ is simply a Reducer of graph G on key u. Each following iteration, where $\overrightarrow{\pi_u}^{(i)}$ is updated (i.e., Equation 2, 3, or 4) can be implemented as a Combiner joining $\overrightarrow{\pi_u}^{(i-1)}$ and G.

2.2.2 Monte Carlo Baseline

As we described in section 1.1.2, the Monte Carlo approach simulates R random walks from each source node u. Here we outline a straightforward implementation of this method on MapReduce. The algorithm consists of three phases: an initial Reducer to initialize R random walks from u, a sequence of Combiner iterations to extend each random walk until it teleports to u, and a final Reducer to aggregate the frequencies of visits to every node v in all R walks (for each source node), and approximate the PPR values.

The initial Reducer and the Combiner are very similar to the examples that we gave in subsection 2.1. The Combiner will be invoked multiple times. At each iteration, every walk which has not yet finished is extended by one randomly chosen neighbor from its endpoint, and then it is decided if the walk should teleport at this step. If a walk is finished, the

Combiner stores it on disk. After all walks are finished, we call the final Reducer for each u, and compute the visiting frequencies of all nodes v (w.r.t. the source node u).

We observe that the baseline implementation, which we will refer to in the rest of the paper as MCBL, needs to execute the Combiner many times for long walks. To deal with this issue, Fogaras et al [10] propose heuristic methods, such as ignoring the tails of the longer walks, or approximating them using the Global PageRanks. These methods decrease the quality of approximations, and are after all only heuristics, and don't actually solve the problem. In this paper, we present an algorithm to implement the Monte Carlo approach using a very small number of iterations, which eliminates the need for such heuristics. Even if one wants to cut the longer walks, it will still be more efficient to implement the resulting walks using our algorithm.

2.2.3 The SQRT Algorithm

Das Sarma et al. [8] present an algorithm to efficiently compute a single random walk from a source node in the graph in the streaming computation model. To do a walk of length λ, it first samples one short segment of length θ out of each node in a randomly sampled subset of the nodes, and then tries to merge these segments to form the longer walk. However, to keep the random walk truly random, it cannot use the segment at each node more than once. So, if it hits a node for the second time, it gets stuck, in which case it samples (through one more pass over the data on disk) a number of edges out of a properly defined set of nodes, and brings them to the main memory to continue the walk. But, this can not be done in MapReduce, as there is no globally shared memory available. So, the algorithm can not be easily implemented on MapReduce.

However, there are two easy ways out of this problem. First, whenever hitting a stuck node, one can simply take one random outgoing edge, and extend the walk only by that edge. This removes the need for shared memory. But, then the problem is that, it can reduce the algorithm essentially to the MCBL algorithm (in the worst case). Second, one can make sure there are no stuck nodes during the run of the algorithm. This can be done by sampling more segments per node. More precisely, if we sample λ/θ segments (of length θ) per node, then no node is ever going to be stuck (as we don't use more than λ/θ segments in the whole walk of length λ). We call this algorithm the SQRT algorithm.

Assuming, for simplicity of discussion, that λ/θ is an integer, SQRT(G, λ, θ), presented in Algorithm 2, first samples λ/θ independent segments of length θ using edge-by-edge extension by GenSeg(G, λ, θ), presented in Algorithm 1. Each segment is labeled by id $i = 1, \ldots, \lambda/\theta$. Then, it merges these segments together, based on the order of their ids, to make longer and longer segments and eventually end up with one random walk of length λ starting at each node. We reemphasize that SQRT is a simple modification of the algorithm proposed by Das Sarma et al. [8] that we are introducing to make it possible to be implemented on MapReduce, and hence, in the later parts of this paper, we will treat it as prior work. An example using this algorithm is given below.

EXAMPLE 1. *Assume $\lambda = 17$, $\theta = 3$. The algorithm first generates $\eta = \lceil 17/3 \rceil = 6$ segments for each node u. Let the 6 segments from u be $S[u, i]$ $(1 \leq i \leq 6)$. Denote the walk starting from u as $W[u]$. Initially, we have $W[u] = u$. Let $W[u].LastNode$ be the last node of the current (partial) walk $W[u]$. In the first iteration, $W[u].LastNode$ is u, and we append $S[u, 1]$ to $W[u]$. Suppose the last node of $S[u, 1]$ is v (hence $W[u].LastNode = v$). In the second iteration, we append $S[v, 2]$ to $W[u]$. Suppose the last node of $S[v, 2]$ is x (hence $W[u].LastNode = x$). In the third iteration, we append $S[x, 3]$ to $W[u]$. Continue the process for 6 iterations, and $W[u]$ reaches length $\lambda = 17$.*

Algorithm 1 GenSeg(G, λ, θ)

Input: A (weighted) graph $G = (V, E)$, parameters λ and θ
Output: $\lceil \lambda/\theta \rceil$ independent random walk segments starting at each node, each of length θ, except the last one which, if λ/θ is not an integer, has length $\lambda - \theta\lfloor \lambda/\theta \rfloor$ $(\lambda \bmod \theta)$

```
//Initial Reducer
Initialization: For each u ∈ V, and 1 ≤ i ≤ ⌈λ/θ⌉, set S[u,i] = u
//θ iterations of Combiner
for j = 1 to θ do
  for i = 1 to ⌈λ/θ⌉ do
    if (i = ⌈λ/θ⌉ and 0 < λ − θ⌊λ/θ⌋ < j) then
      Break; //the last segment
    end if
    for each u ∈ V do
      w = RandomNeighbor(S[u,i].LastNode);
      S[u,i] = S[u,i].Append(w);
    end for
  end for
end for
Return S;
```

Algorithm 2 SQRT(G, λ, θ)

Input: A (weighted) graph $G = (V, E)$, the desired length of the walks λ, the length of initial segments θ
Output: A solution to the SRW problem, i.e., one random walk of length λ starting at each graph node

```
Let S = GenSeg(G, λ, θ);
Define ∀u ∈ V: W[u] = u. //Initial Reducer
//⌈λ/θ⌉ iterations of Combiner
for i = 1 to ⌈λ/θ⌉ do
  for each u ∈ V do
    W[u] = W[u].Append(S[W[u].LastNode, i]);
  end for
end for
Return W;
```

The SQRT algorithm consists of two parts: segment generation, that needs θ iterations to generate the initial segments, and segment merge, that needs λ/θ iterations to merge the segments. Hence, it does a total of $\theta + \lambda/\theta$ iterations, which is optimized when $\theta = \sqrt{\lambda}$, resulting in $2\sqrt{\lambda}$ iterations. SQRT improves the number of required iterations compared to the baseline algorithm (needing λ iterations). However, for long walks, it can still need a large number of iterations. In the next section, we show that one can compute long random walks in the MapReduce framework, using a very small number of iterations. We then show its application for FPPR approximation in section 4.

3. SINGLE RANDOM WALK

Our MapReduce algorithm for FPPR approximation is based on very efficiently doing single random walks of a given length λ, starting at each node in the graph. Notice that we do not require the random walks at different nodes to be independent, but the random walk starting at each

node is required to be a true random walk. As mentioned in section 1.2, we call this problem the SRW-MapReduce problem. In this section, we first propose our new Doubling algorithm, then prove it achieves the optimal number of iterations among a large family of SRW algorithms, and finally analyze and compare its I/O cost with other candidates.

3.1 The Doubling Algorithm

Similar to the SQRT algorithm, Doubling(G, λ, θ), shown in Algorithm 3, starts (assuming λ/θ to be an integer and a power of 2, for simplifying the discussion here) by generating λ/θ independent segments of length θ out of each node (using the GenSeg subroutine as shown in line 1 of Algorithm 3). However, the subsequent iterations are completely different: in contrast to SQRT which performs just one merge per node (per iteration) and grows a single walk per node, Doubling performs multiple merges and grows multiple walks per node. In the first iteration, for each node, Doubling merges pairs of segments generated by GenSeg (there are $\frac{\lambda}{2\theta}$ such pairs) to construct $\frac{\lambda}{2\theta}$ walks of length 2θ; in the second iteration, it merges pairs of 2θ-step walks (there are $\frac{\lambda}{4\theta}$ such pairs) to construct $\frac{\lambda}{4\theta}$ walks of length 4θ, and so on.

Doubling has two main advantages. First, it terminates (i.e., obtains a single random walk of length λ for each node) in much fewer iterations than SQRT ($\log \frac{\lambda}{\theta}$ iterations vs. $\frac{\lambda}{\theta}$ iterations, after initial segments are generated). Second, by performing more merges per node in parallel, Doubling takes much better advantage of the parallel computation model of MapReduce compared with SQRT. This results in a much faster execution in terms of end-to-end time as confirmed by our experiments.

The challenge is to ensure that the final walk starting at each node is a true random walk. Depending on the graph structure, the growing partial walks can have complicated correlations, and if two correlated partial walks get merged, the result is no longer a proper random walk. We address this challenge by proposing a simple ID logic that, independently of the graph structure, ensures that the same segment is never used twice for any random walk. We prove this in Theorem 3.

We now illustrate the algorithm using an example.

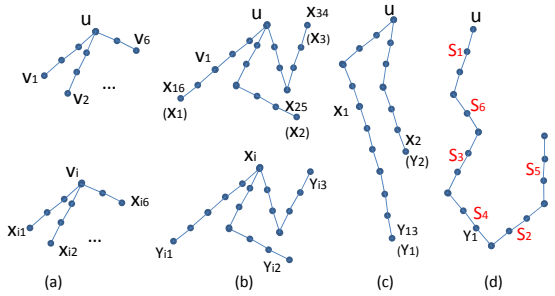

Figure 1: Example for Doubling Algorithm

EXAMPLE 2. *Assume $\lambda = 17$, $\theta = 3$. The algorithm first generates $\eta = \lceil 17/3 \rceil = 6$ segments, $S[u, i]$ $(1 \le i \le 6)$ for each node $u \in V$. Note each segment $S[u, i]$ is labeled by ID i. $S[u, 6]$ is of length 2 while the other segments $S[u, i]$ $(i < 6)$ are of length 3.*

Unlike the SQRT algorithm, where only one walk $W[u]$ from u is defined at each iteration, the Doubling algorithm maintains multiple walks $W[u, i, \eta]$ at the same time, where i is the ID of $W[u, i, \eta]$, and η is the maximum ID at the current iteration. In the beginning, $\eta = 6$, and $W[u, i, \eta] = S[u, i]$ for $i = 1, \ldots, 6$. Then, the algorithm starts merging segments. It merges two segments $W1$ and $W2$ (i.e., appends $W2$ to $W1$) if $W1.LastNode = W2.FirstNode$, $W1.ID < W2.ID$, and $W1.ID + W2.ID = \eta + 1$. This seemingly simple ID logic ensures that each segment is a proper random walk even when multiple segments are maintained for each node. After appending $W2$ to $W1$, we keep $W1.ID$ unchanged. Thus, one merging iteration will reduce the maximum ID from η to $\lfloor \frac{\eta+1}{2} \rfloor$. In this example, as shown in Figure 1(a), for a node u, $W[u, i, 6].LastNode = v_i$, and for each v_i, $W[v_i, j, 6].LastNode = x_{ij}$. Therefore, we merge $W[u, i, 6]$ with $W[v_i, 7 - i, 6]$ for $i = 1, 2, 3$, and get 3 new segments: $W[u, 1, 3]$ that ends at x_{16} (simplified to x_1), $W[u, 2, 3]$ that ends at x_2, and $W[u, 3, 3]$ that ends at x_3, as shown in Figure 1(b). Note that the multiple merges for each node as well as the merges across different nodes can be done independently of each other and hence in parallel. Therefore, at each x_i, there are also three merged segments ending at y_{ij}.

Now $\eta = 3$, and we will merge $W[u, 1, 3]$ with $W[x_1, 3, 3]$. Since η is an odd number, $W[u, 2, 3]$ does not have a matching segment to merge, and we will keep $W[u, 2, 3]$ as same. The merging results are shown in Figure 1(c). For the node u, we now have two segments $W[u, 1, 2]$ that ends at y_1 and $W[u, 2, 2]$ that ends at y_2 (originally x_2). In the final step, we merge $W[u, 1, 2]$ with $W[y_1, 2, 2]$, and get $W[u, 1, 1]$, which has length $\lambda = 17$.

Notice that if we denote a generic initial segment $S[u, i]$ as $S[i]$, by only referring to its ID, we have

$$W[u, 1, 1] =< S[1], S[6], S[3], S[4], S[2], S[5] >$$

(as shown in Figure 1(d)). The walk is clearly composed of initial segments with different IDs (which are thus independent, even if starting at the same node), and is hence a true random walk of length 17. Also, notice how the algorithm outputs a random walk of length 17 in just 6 iterations (i.e., 3 iterations for GenSeg, and then 3 more iterations for Doubling merges), while SQRT would need at least 8 iterations (e.g., with $\theta = 4, 5,$ or 6) to do the same thing.

3.1.1 Correctness of The Doubling Algorithm

Here we prove that the algorithm works correctly, that is, its output walk starting at each node is a true random walk of the desired length. We first give a definition:

DEFINITION 1. *For any integer $\eta \ge 1$, define the following (finite) recursive sequence:*

$$T_0^\eta = \eta;\ while(T_j^\eta > 1)\ T_{j+1}^\eta = \lfloor \frac{T_j^\eta + 1}{2} \rfloor$$

Denote the set of these numbers by $T^\eta = \{T_j^\eta\}_{j \ge 0}$.

With $\eta = \lceil \lambda/\theta \rceil$, T^η is simply the set of all maximal IDs in different merge iterations of Doubling(G, λ, θ). For instance, in Example 2, $T^\eta = \{6, 3, 2, 1\}$. Note that T_i^η is also the number of segments starting from each node after i^{th} merge iteration.

In order to prove the correctness of the algorithm, we need to show that $W[u, 1, 1]$ (the output walk) is a walk in which every node (or edge) is chosen randomly, and that the length of $W[u, 1, 1]$ is λ. Since all segments (across all

Algorithm 3 Doubling(G, λ, θ)

Input: A (weighted) graph $G = (V, E)$, the desired length of the walks λ, the length of initial segments θ
Output: One random walk of length λ starting at each graph node

Let $S = \text{GenSeg}(G, \lambda, \theta)$;
Define $\eta = \lceil \lambda/\theta \rceil$, and $\forall\, u \in V$, $1 \le i \le \eta : W[u, i, \eta] = S[u, i]$, $E[u, i, \eta] = S[u, i].LastNode$; //Initial Reducer
//Combiner iterations
while $\eta > 1$ **do**
  $\eta' = \lfloor \frac{\eta+1}{2} \rfloor$;
  for $i = 1$ to η' **do**
    for each $u \in V$ **do**
      if $i = \frac{\eta+1}{2}$ **then**
        $W[u, i, \eta'] = W[u, i, \eta]$;
        $E[u, i, \eta'] = E[u, i, \eta]$;
        Continue;
      end if
      $v = E[u, i, \eta]$;
      $W[u, i, \eta'] = W[u, i, \eta].Append(W[v, \eta - i + 1, \eta])$;
      $E[u, i, \eta'] = E[v, \eta - i + 1, \eta]$;
    end for
  end for
  $\eta = \eta'$;
end while
For each $u \in V$, output $W[u, 1, 1]$; //the computed random walk

different nodes in the graph) are initialized randomly and independently, and we generate a walk by merging segments, to show $W[u, 1, 1]$ is a random walk, we need to show that $W[u, 1, 1]$ does not use a same initial segment twice. As we have seen in Example 2, the Doubling algorithm ensures that only segments with different IDs will be merged. Therefore, all segments in each walk are different. Formally, we have the following lemma, proved in the Appendix:

LEMMA 1. *Given λ, θ, for any $\eta \in T^{\lceil \lambda/\theta \rceil}$, $u, v \in V$, and $1 \le i, j \le \eta$, $i \ne j$, we have:*

1. *$W[u, i, \eta]$ is a proper random walk.*
2. *$W[u, i, \eta]$ and $W[v, j, \eta]$ are independent.*

Now we show that the length of $W[u, 1, 1]$ is λ. Denoting the length of a segment $W[u, i, \eta]$ by $|W[u, i, \eta]|$, we have the following lemma, proved in the Appendix:

LEMMA 2. *Given λ, θ, for any $\eta \in T^{\lceil \lambda/\theta \rceil}$ and $u \in V$: $\sum_{i=1}^{\eta} |W[u, i, \eta]| = \lambda$*

Given the above two lemmas, we have the following theorem:

THEOREM 3. *The output of the Doubling algorithm is a valid solution to the SRW problem.*

PROOF. After the last iteration, for any node $u \in V$, we have only one segment, namely $W[u, 1, 1]$. From lemma 1, we know that this segment is a proper random walk. Also, from lemma 2, we have $|W[u, 1, 1]| = \lambda$. □

So, Doubling is a correct SRW algorithm. Next, we analyze the number of its MapReduce iterations.

3.2 The Number of Iterations

After the generation of initial segments, the Doubling algorithm merges them in a few iterations, until it has one segment per node. We have the following lemma, proved in the appendix, and theorem:

LEMMA 4. *For any integer η, T_j^{η} is a monotone decreasing sequence, and $|T^{\eta}| = 1 + \lceil \log_2 \eta \rceil$.*

THEOREM 5. *The Doubling algorithm with parameters λ, θ finishes in $\theta + \lceil \log_2 \lceil \frac{\lambda}{\theta} \rceil \rceil$ MapReduce iterations.*

PROOF. $\text{GenSeg}(G, \lambda, \theta)$ needs θ iterations to produce the initial segments, after which, we have $T_0^{\lceil \lambda/\theta \rceil}$ segments per node. Then, after j segment merge iterations, we have $T_j^{\lceil \lambda/\theta \rceil}$ segments per node, and we continue merging the segments until we end up with one segment per node, that is until $T_j^{\lceil \lambda/\theta \rceil} = 1$. But, from the above lemma, we know this requires $\lceil \log_2 \lceil \frac{\lambda}{\theta} \rceil \rceil$ iterations. Therefore, the total number of iterations is $\theta + \lceil \log_2 \lceil \frac{\lambda}{\theta} \rceil \rceil$. □

For any λ, we have: $\text{argmin}_{\theta \in [1,\lambda]} \{\theta + \lceil \log_2 \lceil \frac{\lambda}{\theta} \rceil \rceil\} = 1$. So, the optimal number of iterations for the Doubling algorithm is achieved with $\theta = 1$, in which case the algorithm finishes in $1 + \lceil \log_2 \lambda \rceil$ iterations. We now show that this is actually the minimum number of iterations possible for a large family of SRW algorithms.

3.2.1 Optimality of the Number of Iterations

We first define a family of algorithms for doing random walks, that we call the *Natural* family of random walk algorithms. A typical Natural algorithm takes as input a directed weighted graph $G = (V, E)$, and a family of length sets $\{\overline{L_v}\}_{v \in V}$, and outputs a random walk of length λ_v starting at v ($\forall\, v \in V, \lambda_v \in \overline{L_v}$). The only operations that the algorithm is allowed to use are:

1. $Extend(\mathcal{R})$: takes a random walk $\mathcal{R}$, possibly of length 0 (i.e., just a node), and extends it by taking a random edge out of its last node.
2. $Merge(\mathcal{R}_1, \mathcal{R}_2, \lambda)$: takes two random walks $\mathcal{R}_1$, of length $\lambda_1 \ge 1$, and $\mathcal{R}_2$, of length $\lambda_2 \ge 1$, such that the last node of $\mathcal{R}_1$ is the same as the first node of $\mathcal{R}_2$, as well as a length λ, such that $\lambda_1 < \lambda \le \lambda_1 + \lambda_2$, stitches $\mathcal{R}_1$ and $\mathcal{R}_2$ together and truncates the resulting walk (of length $\lambda_1 + \lambda_2$) at length λ.

We do not require the output walks starting at different nodes to be independent, but we do require the different walks starting at the same node to be true and independent random walks. One can easily see that GenSeg, SQRT, and Doubling are all Natural. Intuitively speaking, the operators allowed for Natural algorithms are all one can do when one wants to make random walks, unless one has some non-local structural information about the graph. For instance, if one knows that the graph is a cycle of length, say, 4, denoted as (u_1, u_2, u_3, u_4), then without doing anything further, one already knows exactly what a random walk of a length λ starting at a node $v = u_i$ is. Use of such structural information is not allowed for Natural algorithms, but this is a very reasonable restriction, as such information are usually (specially, for the massive scale graphs typical in the MapReduce setting) not available in practice anyway.

Next, we give another definition and prove a simple claim.

DEFINITION 2. *A Natural Binary Tree is a labeled binary tree such that for any leaf v, $label(v) = 1$, and for any non-leaf node z with children x, y, we have $label(x) + label(y) \ge label(z)$.*

CLAIM 1. *The number of levels of a Natural binary tree with root having label λ is at least $1 + \lceil \log_2 \lambda \rceil$.*

PROOF. Define $D(\lambda)$ to be the minimum number of levels of a Natural binary tree with λ as the label of the root. Then, we prove the claim by induction. For $\lambda = 1$, the tree is just a single node, so we have 1 level, and the claim is obviously true. Now assume the claim is true for all numbers smaller than $\lambda > 1$, and we will prove it for λ. Assume, the children of the root have labels λ_1, λ_2. Then, $\lambda_1 + \lambda_2 \geq \lambda$, and hence $\max\{\lambda_1, \lambda_2\} \geq \lfloor \frac{\lambda+1}{2} \rfloor$. Therefore:

$$D(\lambda) \geq 1 + D(\lfloor \frac{\lambda + 1}{2} \rfloor)$$

Then, a simple induction shows that $D(T_j^\lambda) \geq |T^\lambda| - j$, and hence $D(\lambda) = D(T_0^\lambda) \geq |T^\lambda| = 1 + \lceil \log_2 \lambda \rceil$. □

Now, we show, by induction, that considering a node u and a length $\lambda \geq 1$, the operations used by a Natural algorithm, ALG, to do a random walk of length λ starting at u can be modeled by a Natural binary tree, $\mathcal{T}(u, \lambda)$. If $\lambda = 1$, then algorithm can only use $Extend(u)$ to generate a walk of length 1 starting at u, and we model this using a single node (leaf, with label 1). If $\lambda > 1$, then the last operation done by ALG to generate the desired walk can be one of the two following options:

1. extending a walk $\mathcal{R}'$ of length $\lambda - 1$, starting at u and ending at a node v, using $Extend(\mathcal{R}_1)$
2. merging a walk $\mathcal{R}_1$ of length $1 \leq \lambda_1 < \lambda$ starting at u and ending at a node v, with a walk $\mathcal{R}_2$ of length $\lambda - \lambda_1 \leq \lambda_2 < \lambda$ and starting at v, using $Merge(\mathcal{R}_1, \mathcal{R}_2, \lambda)$

In both cases $\mathcal{T}(u, \lambda)$ will be a tree with root having label λ. In the first case, the root will have a child with label 1, which is a leaf, and a child with label $\lambda - 1$ which is the root of a copy of $\mathcal{T}(v, \lambda - 1)$ (which exists by the inductive hypothesis). In the second case, the root will have a child with label λ_1 which is the root of a copy of $\mathcal{T}(v, \lambda_1)$, and a child with label λ_2 which is the root of a copy of $\mathcal{T}(v, \lambda_2)$.

From claim 1, it follows that the number of levels of $\mathcal{T}(u, \lambda)$ is at least $1 + \lceil \log_2 \lambda \rceil$. But, if ALG is implemented in MapReduce, each level in $\mathcal{T}(u, \lambda)$ will need at least one MapReduce operation, and hence the depth of this tree will be a lower bound for the number of MapReduce iterations needed by ALG to make a random walk of length λ starting at u. This shows that any MapReduce Natural random walk algorithm needs at least $1 + \lceil \log_2 \lambda \rceil$ MapReduce iterations to produce random walks of length λ. Specifically, any Natural SRW algorithm needs $1 + \lceil \log_2 \lambda \rceil$ MapReduce iterations to produce random walks of length λ. But, this is exactly the number of iterations used by Doubling (with parameter value $\theta = 1$) to do random walks of length λ. So, in terms of the number of iterations to produce random walks of given lengths, Doubling is optimal among all Natural SRW algorithms.

3.3 I/O Cost Analysis

In this section, we analyze and compare the I/O efficiency of the Doubling, SQRT, and GenSeg algorithms. Realistic analysis of the efficiency of MapReduce algorithms is not straightforward, as the algorithms' efficiency in practice depends on many complicated factors, such as the distribution of data and computation by the job schedulers, proximity of the communicating machines, etc. So, I/O cost analysis is just a proxy to give an overall understanding of the expected performance of different algorithms. However, since for many MapReduce algorithms, including the ones we are considering in this section, the actual computational work performed by the computing nodes is very simple, data I/O is indeed a very important factor in the performance of the algorithms. The exact I/O costs also depend on many details such as the architecture of the network of Mappers and Reducers (which may be directly decided by the job scheduler and not be under the programmer's control), the exact implementation of the algorithms, the schemas and datatypes used to represent the data, and so on. So, here we only present asymptotic analyses of the I/O costs, by counting the number of basic data items (such as node ids) that need to be accessed for each operation. As stated earlier in the paper, we assume that we use the two primitives Reducer and Combiner, introduced in section 2.1, to implement the algorithms. Denoting the number of basic data items in a dataset $\mathcal{R}$ by $|\mathcal{R}|$, we assume for these two primitives that:

- when an input dataset $\mathcal{R}_I$ is Reduced to produce an output dataset $\mathcal{R}_O$, the input and output costs are, respectively, $O(|\mathcal{R}_I|)$ and $O(|\mathcal{R}_O|)$.
- when two input datasets $\mathcal{R}_1$ and $\mathcal{R}_2$ are Combined to produce and output dataset $\mathcal{R}_3$, the input and output costs are, respectively, $O(|\mathcal{R}_1| + |\mathcal{R}_2|)$ and $O(|\mathcal{R}_3|)$.

We now start by analyzing GenSeg(G, λ, θ). In the i^{th} iteration of the algorithm ($1 \leq i \leq \theta$), each node has λ/θ segments of length $i-1$. The algorithm Combines the graph table and the segments table, and extends each segment by one edge, and gets the new segments (of length i). Recalling, from section 1.1, that the number of nodes and edges in the graph are, respectively, n and m, we get that the input and output costs of iteration i are, respectively, $O(m + n(i-1))$ and $O(ni)$. Summing up all these costs for $1 \leq i \leq \theta$, we get that the total I/O cost of GenSeg(G, λ, θ) is:

$$O((m + n\lambda)\theta) \tag{5}$$

Next, we analyze the I/O cost of Doubling(G, λ, θ). After generating the initial segments, the algorithm Combines the current segment dataset with itself to make the new (longer) segments, which then get output by the Combiner. From lemma 2, we know that the total size of the segments starting at each node in any iteration is equal to λ. Thus, the input and output costs at each iteration are $O(n\lambda)$. From theorem 5, we know the number of segment merge iterations is $\log_2(\lambda/\theta)$ (ignoring the ceilings). So, including the I/O cost of GenSeg as computed above, the total I/O cost of Doubling(G, λ, θ) is:

$$O((m + n\lambda)\theta + n\lambda \log_2 \frac{\lambda}{\theta}) \tag{6}$$

A similar analysis shows the I/O cost of SQRT(G, λ, θ) is:

$$O((m + n\lambda)\theta + n\frac{\lambda^2}{\theta}) \tag{7}$$

Now, comparing expressions 6 and 7, we clearly see that for any λ, θ, Doubling(G, λ, θ) has better I/O efficiency than SQRT(G, λ, θ). Also, note that the simple algorithm which extends the random walks edge-by-edge can be seen as the special case of GenSeg with $\theta = \lambda$ (i.e., GenSeg(G, λ, λ)), which, from expression 5, has I/O cost $O((m + n\lambda)\lambda)$, which is again less efficient than Doubling. Finally, notice that no

SRW algorithm can have an I/O cost smaller than the size of its final output, which is $\Omega(n\lambda)$. So, choosing θ to be a small constant number, we see that for not very dense graphs (e.g., as long as $m/n = O(\lambda \log \lambda)$), Doubling$(G, \lambda, \theta)$ is within a logarithmic factor (in the length of the walk) from this lower bound for all SRW algorithms.

4. FULLY PERSONALIZED PAGERANKS

In this section, we present our algorithm for the FPPR-MapReduce problem. As discussed in section 1.1.2, Frogras et al. [10] propose that to approximate the PPR vector of a source node u, one can simulate a number, R, of PPR fingerprints (with geometrically distributed lengths) starting at u, and then use the empirical distribution of the performed walk as the PPR estimation. To approximate FPPR, one can do the same thing for every source node. We note two simple points: (1) The lengths of the fingerprints can be the same for different source nodes, and (2) the fingerprints at different source nodes do not need to be independent. Based on these two points, we can design the DoublingPPR algorithm presented in Algorithm 4 for FPPR approximation.

The algorithm is based on our Doubling algorithm for doing single random walks. It takes as input a (weighted) graph G, the expected total length of the PPR walk to be done, and the teleport probability ϵ. Then, it generates $R = \epsilon L$ random numbers λ_i from the distribution Geom(ϵ). Note that in general, we only have $E[\sum_{i=1}^{R} \lambda_i] = L$, but a simple application of Chernoff bound shows that if L is large compared with $1/\epsilon$ (e.g., $L = \widetilde{\Omega}(1/\epsilon)$), then $\sum_{i=1}^{R} \lambda_i$ is actually very close to L (i.e., $L \pm o(L)$) with high probability. It then generates the set of parameters θ_i $(1 \leq i \leq R)$ using a call to the function ChooseTheta. We will further discuss the details of this function later in this section. Then, the algorithm uses Doubling(G, λ_i, θ_i) $(\forall\, 1 \leq i \leq R)$ to generate a walk, $W_i[u]$, of length λ_i, starting at each node u in the graph. Finally, it finds the aggregate count of the number of visits, $C(u, v)$, to each node v in the R walks of source node u, and returns the visiting frequencies as PPR estimations.

Note that, in Algorithm 4, for the sake of simplicity of presentation, we are making R separate calls to the Doubling algorithm. But, each of these calls will first access the graph a few times to generate its initial segments, and one can share these graph accesses between all the different calls to Doubling. In other words, with a single access to the graph, we can extend all the unfinished initial segments (by one edge). This clearly saves some data I/O, and it is what we actually did in our implementation. One can analyze this implementation in a completely similar way to the analyses done in section 3.3 to obtain the following expression for its I/O cost:

$$O(m \max_{1 \leq i \leq R} \{\theta_i\} + n \sum_{i=1}^{R} (\lambda_i \theta_i + \lambda_i \log_2 \frac{\lambda_i}{\theta_i})) \qquad (8)$$

This also provides a way for choosing the values of the parameters θ_i $(1 \leq i \leq R)$. We note that θ_i's should be in general chosen to be small numbers, and that the performance of the algorithm (as also observed in our experiments, whose results are presented later in this paper), is robust to small changes to these parameters. However, a heuristic that one can use is to choose the values of θ_i's to minimize the I/O cost expression 8. Our experiments showed that this choice of θ_i's did actually provide some (though modest) improvement to the performance of the algorithm (e.g., compared to the simpler choices of all equal θ_i's), so we suggest one can use this heuristic to implement the function ChooseTheta.

Algorithm 4 DoublingPPR$(G,\ L,\ \epsilon)$

Input: A (weighted) graph $G = (V, E)$, (expected) total PPR walk length L, teleport probability ϵ
Output: Monte Carlo approximation $\widehat{\pi}_u(.)$ of the personalized PageRank vector of any $u \in V$, obtained by doing PPR walks of (expected) length L.

Let $R = \epsilon L$, and for $1 \leq i \leq R$, let $\lambda_i \sim$ Geom(ϵ) be a random number generated from the geometric distribution Geom(ϵ). Also, let $\overrightarrow{\theta}$ = ChooseTheta$(G, \overrightarrow{\lambda})$, where $\overrightarrow{\theta} = [\theta_1, \ldots, \theta_R]$ and $\overrightarrow{\lambda} = [\lambda_1, \ldots, \lambda_R]$.

For $1 \leq i \leq R$, let $W_i =$ Doubling(G, λ_i, θ_i).
For any $u, v \in V$ let $C(u, v) = 0$.
for $i = 1$ to R **do**
  for $u, v \in V$ **do**
    $C(u, v) +=$ number of visits to v in $W_i[u]$
  end for
end for
for $u, v \in V$ **do**
  $\widehat{\pi}_u(v) = \frac{C(u,v)}{\sum_{i=1}^{R} \lambda_i}$.
end for
Return $\widehat{\pi}_u(.)$ for all $u \in V$.

We can also compare the I/O efficiency of DoublingPPR to other FPPR approximation algorithms. Since we have already shown that Doubling is more I/O efficient than edge-by-edge extension of random walks, it is clear that DoublingPPR is also more efficient than the simple implementation of the Monte Carlo approach proposed in [10]. So, here we provide a comparison with the other existing FPPR approximation method, namely the Rounding algorithm. One can easily see that the I/O cost of the Rounding algorithm in the MapReduce framework is the same as its computational complexity analyzed in [25]. So, to guarantee a constant error, the I/O cost of Rounding is:

$$O(\frac{m+n}{\epsilon^2}) \qquad (9)$$

As shown in [10, 2], a simple application of Chernoff bound shows that for the Monte Carlo approach to provide a constant error with high probability, $L = O(\frac{\log n}{\epsilon})$ steps are enough. Now, as mentioned above, one can choose θ_i's to optimize the I/O cost 8, but even with the simplest choice $\theta_i = 1$ $(\forall\, 1 \leq i \leq R)$, the I/O cost of DoublingPPR is $O(m + n \sum_{i=1}^{R} \lambda_i \log_2 \lambda_i)$. But, one can easily show (using concavity of the log function, and the fact that λ_i's are geometrically distributed) that: $\sum_{i=1}^{R} \lambda_i \log_2 \lambda_i = O(L \log \frac{1}{\epsilon})$. So, the I/O cost of DoublingPPR to guarantee constant error with high probability is $O(m + \frac{n}{\epsilon} \log n \log \frac{1}{\epsilon})$. Therefore, comparing with expression 9, we get that if the graph is not very sparse, that is if $m/n = \omega(\epsilon \log \frac{1}{\epsilon} \log n)$ (which is a reasonable bound for many real-world networks, such as social networks, worldwide web, etc.), then DoublingPPR has better I/O efficiency than Rounding as well.

5. EXPERIMENTS

In this section, we present the results of the experiments that we did to test the performance of our methods. As efficiency measures, we considered the total clock time and total machine time of the algorithms, and as quality measure, we considered the approximation error for the top k

nodes (for suitable values of k). We will define and explain these measures and also present our experimental results in more detail in the following subsections. But, the main results of our experiments can be summarized as follows:

• In comparison to the MCBL method, Doubling not only improves the machine time but also improves the clock time by an order of magnitude.
• Doubling significantly outperforms Rounding not only in both clock and machine times but also in quality of the approximations. Also, even though the efficiency of Rounding decreases significantly (and non-linearly) as more iterations are performed, Doubling's performance linearly changes with the total length of the PPR walk it performs.
• For doing a single random walk per node, Doubling performs similarly to SQRT in terms of machine time but significantly outperforms it in clock time. This shows Doubling performs the same amount of work as SQRT but does it in a more parallel fashion.
• The performance of Doubling is robust with respect to the values of the parameters θ_i, and as long as they are chosen to be small numbers, it gets similar performances. However, choosing these values to minimize the I/O cost expression 8 improves the performance of the algorithm modestly.

5.1 Experimental Setup

Datasets: We used real world query-click data in our experiments. Using random walks on query-click graphs has been shown to be very successful in finding related queries [7]. We sampled frequent <query, url click> pairs from a web search engine. All distinct queries and clicked urls are considered as nodes in the graph, and each query and each of its clicked urls are connected by an edge, whose weight is $\log_2 f$, where f is the number of clicks that the url received for that query. The graph has more than 112M total nodes, and 256M (undirected) edges. We replaced each edge with two directed edges between its endpoints, so the final graph that we did our experiments on had more than 513M directed edges. We assigned each node in the graph an integer id, and used these ids for all of our experiments. The programming environment that we used was a standard, unmodified, production MapReduce environment. All algorithms were executed concurrently with other production jobs, at the normal cluster workload.
Algorithms: For single random walk, we compared our Doubling algorithm with the SQRT algorithm. For fully personalized PageRank, we compared our DoublingPPR algorithm with the Rounding algorithm [25], which, to the best of our knowledge, is the state-of-the-art approach in the literature. We do not compare our algorithm with other Power Iteration type algorithms, since, as mentioned in section 2, they either do not scale to full personalization or do not perform as well as Rounding. We also implemented the MCBL algorithm. The high level MapReduce implementation ideas of Rounding and MCBL are described in section 2.2. For the Rounding algorithm, we used integers (instead of floats) to store and manipulate the rounded probabilities. For the MCBL implementation, since we were eventually only interested (for PPR approximation) in the visiting frequencies to different nodes, we only kept aggregate counts of the number of times the walks had visited different nodes (as well as the current last nodes of the walks). We did the same thing for each of the partial walks in our own algorithm, during its segment merge iterations. Also, in all experiments, we used the (typical) value $\epsilon = 0.2$ as our PageRank teleport probability.
Efficiency Measures: To measure the efficiency of algorithms, we considered two measures:

1. Total Clock Time: How long, in terms of wall clock time, it took each job to run from the beginning to the end.
2. Total Machine Time: $\sum_x \tau_x$ where τ_x was the amount of time a machine x was used during the job.

Quality Measure: To measure the quality of the results, for a node u and a value k, we define Z_u^k to be the set of nodes with the k largest personalized PageRanks for u. In most applications of personalized PageRanks, for a source node u, only the nodes in Z_u^k (for some appropriate value of k) are of interest. For instance, in friend recommendation (or query suggestion), only the users (queries) with highest personalized PageRanks will get recommended (suggested). Note that real applications such as query suggestion, or site or friend recommendation often involve lots of factors. Personalized PageRank usually acts as one of the (important) features for those tasks. So it is also important to measure how well we are approximating the true numeric values of the personalized PageRanks of the nodes in Z_u^k. Therefore, to measure the quality of the approximation of both Z_u^k and the actual numeric PPR values, we use:

$$Err(u,k) = \frac{\sum_{v \in Z_u^k} |\pi_u(v) - \hat{\pi}_u(v)|}{\sum_{v \in Z_u^k} \pi_u(v)} \tag{10}$$

Where $\hat{\pi}_u(v)$ is the approximate personalized PageRank (e.g., by either the Rounding algorithm or our algorithm). We sampled 100 source nodes from the graph, used 70 iterations of Power Iteration to compute the "ground-truth" personalized PageRanks for each source node, and averaged the above error measure over the sampled source nodes (at different values of k) to measure the quality of the results of different algorithms. Further details about our sampling method and the results of these experiments are given later in this section.
Choice of θ_i's: The DoublingPPR algorithm has three main parts: generating segments, merging segments, and aggregating the walks (to compute the personalized PageRanks). The balance between the times spent on segment generation and segment merge depends on the choices of θ_i's. As explained in section 4, we chose the values of θ_i's so as to minimize the I/O cost expression 8. We observed that this is not a very crucial factor in the performance of the algorithm, since the values of θ_i's are all small numbers, and the algorithm's performance is robust with respect to these parameters, so as long as θ_i's are chosen to be small constants, we get similar performances. However, as shown in Figure 8, our way of choosing θ_i's did modestly improve the algorithm's performance (in comparison to the simpler choices of equal θ_i's). So, for the rest of the experiments, we just kept using these values of θ_i's.

5.2 Experimental Results

We first compare the quality-efficiency tradeoff of DoublingPPR with MCBL and Rounding, and then drill down to the various issues which may relate to the performance of the algorithms.

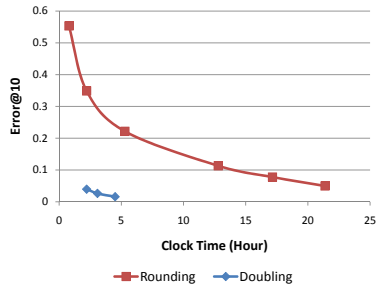

Figure 2: $\overline{Err}(.,10)$ vs. Clock Time

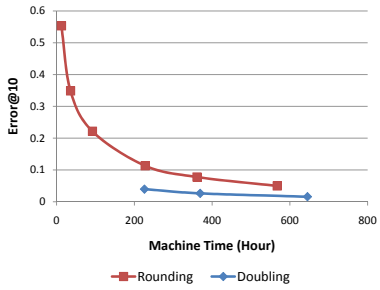

Figure 3: $\overline{Err}(.,10)$ vs. Machine Time

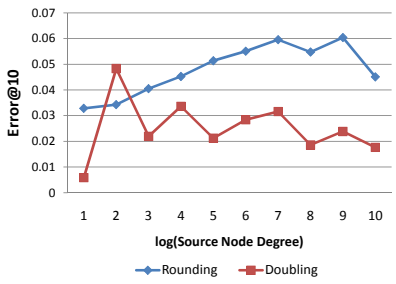

Figure 4: $\overline{Err}(.,10)$ vs. Bucket

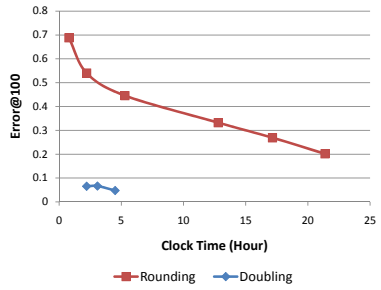

Figure 5: $\overline{Err}(.,100)$ vs. Clock Time

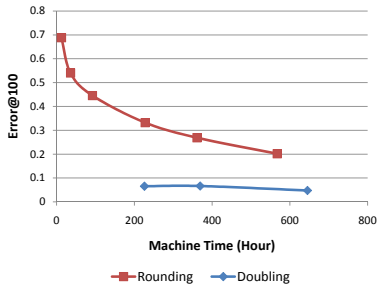

Figure 6: $\overline{Err}(.,100)$ vs. Machine Time

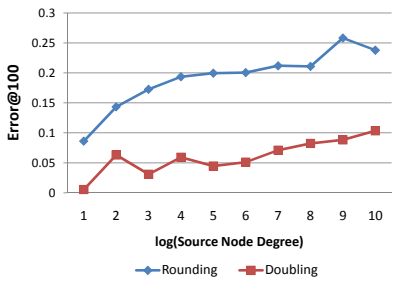

Figure 7: $\overline{Err}(.,100)$ vs. Bucket

5.2.1 *Quality-Efficiency Tradeoffs*

We first present the results of the efficiency comparison between DoublingPPR and MCBL. Note that since both algorithms are based on the Monte Carlo approach, their quality is the same (modulo the randomness due to the actually performed random walks). To compute FPPR on the experimental graph data using total walk length 500, MCBL uses 22.03 hours in clock time, and 333.60 hours in machine time; while for the same task, DoublingPPR uses 2.21 hours in clock time, and 225.82 hours in machine time. The Doubling algorithm improves the clock time by one order of magnitude, while using 2/3 of the machine time.

We also compared the quality-efficiency tradeoff of DoublingPPR and Rounding. The nodes in our graph had widely different degrees. So, in order to observe the behavior of the algorithms for different source degrees, we divided the nodes in the graph into a number of buckets, with bucket s ($s \geq 1$) including all the nodes whose degree was in the $[2^{s-1}, 2^s)$ interval. There are a very small number of nodes which belong to buckets $s > 10$. So, we only considered 10 different buckets ($1 \leq s \leq 10$). Then, we sampled 10 nodes uniformly at random from each bucket, computed the error measure defined in equation 10, at different values of k, for each sampled source node, and averaged the results. Figures 2, 5, 3, 6 show the averaged errors for $k = 10, 100$ versus the machine and clock times, for DoublingPPR and Rounding algorithms. For DoublingPPR, we did three experiments with walk lengths 500, 1000, and 2000. For Rounding, we computed the average errors and machine and clock times for 10, 20, 30, 40, 45, and 50 iterations. Our guiding heuristic in choosing these lengths and iteration numbers was (rough) measures of absolute error for the algorithms, set at 10^{-3}, suggesting the length of the Doubling walk to be in the order of 10^3 and the number of iterations of Rounding to be in the order of 10^1. As it is clear from the figures, DoublingPPR consistently performs significantly better than Rounding in average errors. We observe that even with walk length 500, DoublingPPR achieves a better quality than 50 iterations of Rounding.

We observe that the efficiency of the Rounding algorithm decreases significantly as more iterations are done. This is because in later iterations, the rounding threshold becomes smaller, and more PPR values need to be carried over to the next iteration. That introduces more I/O and computational costs. For instance, from iteration 45 to 50, the Rounding algorithm uses about 200 machine hours, while the first 45 iterations in total only took less than 400 machine hours. On the other hand, we clearly see that both clock time and machine time of the DoublingPPR algorithm grow linearly with the length of the walks.

Finally, we computed the average errors for source nodes in different buckets. The results in Figures 4, 7, using length 1000 for DoublingPPR and 50 iterations for Rounding, show that DoublingPPR consistently achieves lower error than Rounding[1]. This, in accordance with the I/O cost comparison done in section 4, suggests that although we used a relatively sparse query click graph (i.e., average degree is low), DoublingPPR is also expected to outperform Rounding in dense graphs.

5.2.2 *Performance Drill-Down*

Here we drill down to several aspects of the algorithms to better understand their computational performance.

Sensitivity to the choice of θ_i's: To compute personalized PageRank, the Monte Carlo approach needs to compute multiple random walks. For our algorithm, we can adaptively choose different θ values for different walks, or

[1]The numbers 1000 and 50 were chosen, because, as can be seen from Figures 2, 5, 3, 6, a 500-step walk already gives a better quality than 50 iterations of Rounding, and a 1000-step Doubling walk is still significantly faster than 50 Rounding iterations.

uniformly assign a single θ value for all walks. As we mentioned earlier in this section, the DoublingPPR algorithm is not very sensitive to the choice of θ_i's, but carefully chosen θ values, as explained in section 4, can modestly improve the overall computational performance. Figure 8 shows the machine time used to compute PPRs by walk length 1000, using adaptive θ (i.e., a different θ value for each individual walk), uniform $\theta = 1$, and uniform $\theta = 3$. We observe that adaptive θ values try to balance the cost between segment generation and segment merge, and achieve an overall better performance.

Comparison with Rounding: To measure how well the algorithms DoublingPPR and Rounding could benefit from the parallelization capability in MapReduce, we varied the resource allocation (i.e., the number of allocated machines) for both algorithms by factors $2, 4$, and 8 based on the default resource allocation provided by our execution engine, and computed the corresponding machine times and clock times. The result, given in Figure 10, and using 1000 steps for DoublingPPR and 50 iterations for Rounding, shows that with different resource allocations also DoublingPPR is significantly more efficient than Rounding. It also shows that machine time and clock time typically behave in opposite ways: For the same computational task, smaller clock times require higher machine times. This is often because to achieve a smaller clock time, tasks are divided into finer granularities, and then there are extra overheads in distributing the data and scheduling sub-tasks to more machines.

Comparison with SQRT: We measured the efficiency of doing a single random walk using each of the Doubling and SQRT algorithms. We performed three experiments with walk lengths $4, 8$, and 16. The machine times were very close for all three experiments, but the clock times showed that Doubling was much faster than SQRT. The result is presented in Figure 9.

6. RELATED WORK

MapReduce algorithms have been designed and proposed in the literature for a wide range of applications, such as machine learning, text processing, and bioinformatics (refer to [19, 18] and references therein). MapReduce provides an excellent tool for large scale graph processing as well, and graph algorithms have been designed for it in the literature [15, 16, 6]. In this paper, we study one of the most well known graph computation problems, i.e., computing personalized PageRanks [12, 11]. Our algorithms can also be applied to other personalized random walk based measures, such as personalized SALSA [21, 24].

We adopt the Monte Carlo approach, which has been previously used to design Personalized and Global PageRank approximation algorithms in different computational models. We have discussed several of these algorithms from the MapReduce perspective in section 2. We would like to also point out that Bahmani et al. [3] propose Monte Carlo algorithms for Personalized (and Global) PageRank approximation in the random access model. Their algorithm is designed to efficiently do a PPR random walk starting at a single node in the graph, assuming random access to the edges. This algorithm is however unsuitable for the MapReduce framework, for a multitude of reasons: heavy use of random access to the graph, complicated logic which is not at all clear how to coordinate on MapReduce for FPPR, and needing a lot of MapReduce iterations.

7. CONCLUSIONS

In this paper, we present a new algorithm for massive scale fully personalized PageRank approximation on MapReduce. The algorithm belongs to the Monte Carlo family of PageRank approximation methods. The main contribution of this paper is to propose a very efficient MapReduce algorithm for computing single random walks starting from all nodes in a graph. The number of MapReduce iterations needed by this algorithm is optimal among a broad family of algorithms for performing random walks. We use this algorithm to design our PPR approximation algorithm, which outperforms the state-of-the-art algorithm in the literature. To achieve a similar quality of approximation of personalized PageRank values, our algorithm uses significantly less clock time and machine time. We believe this algorithm will have a high impact on many real-world massive scale personalization problems, such as friend recommendation, query suggestion, site suggestion, etc. In addition, our algorithm for efficiently simulating random walks on MapReduce is applicable to other applications using random walks (e.g., computation of other personalized random walk based measures, such as personalized SALSA) on massive graphs.

8. REFERENCES

[1] Pig. http://hadoop.apache.org/pig.

[2] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM J. Numer. Anal.*, 45(2):890–904, 2007.

[3] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank. http://arxiv.org/abs/1006.2880, 2010.

[4] P. Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Math*, 3, 2006.

[5] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[6] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engg.*, 11(4):29–41, 2009.

[7] N. Craswell and M. Szummer. Random walks on the click graph. In *SIGIR*, pages 239–246, 2007.

[8] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Estimating PageRank on graph streams. In *PODS*, pages 69–78, 2008.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, pages 10–10, 2004.

[10] D. Fogaras and B. Rácz. Towards scaling fully personalized PageRank. In *WAW*, pages 105–117, 2004.

[11] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen. Combating web spam with trustrank. In *VLDB*, pages 576–587, 2004.

[12] T. H. Haveliwala. Topic-sensitive pagerank. In *WWW*, pages 517–526, 2002.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[14] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, number 1-58113-680-3, 2003.

[15] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Fast diameter estimation and mining in massive graphs with hadoop, 2008.

[16] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. *ICDM*, 0:229–238, 2009.

[17] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *CIKM*, pages 556–559, 2003.

[18] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[19] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in mapreduce. In *MLG*, pages 78–85, 2010.

[20] F. McSherry. A uniform approach to accelerated pagerank computation. In *WWW*, pages 575–582, 2005.

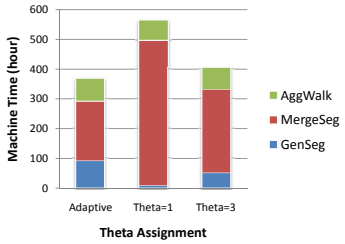

Figure 8: Machine Time vs. $\{\theta_i\}_i$

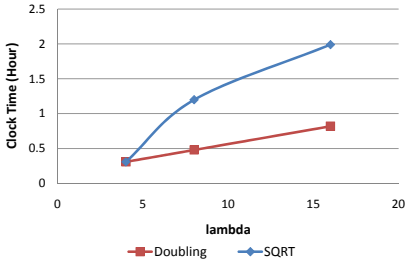

Figure 9: Clock Time vs. Walk Length

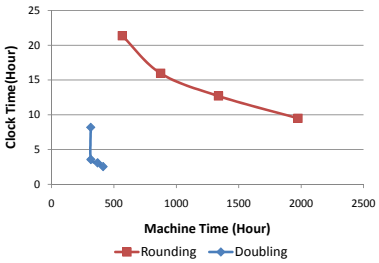

Figure 10: Clock Time vs. Machine Time

[21] A. Y. Ng, A. X. Zheng, and M. I. Jordan. Stable algorithms for link analysis. In *SIGIR*, pages 258–266, 2001.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[23] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[24] D. Rafiei and A. O. Mendelzon. What is this page known for? computing web page reputations. *Comput. Netw.*, 33(1-6):823–835, 2000.

[25] T. Sarlós, A. A. Benczúr, K. Csalogány, D. Fogaras, and B. Rácz. To randomize or not to randomize: space optimal summaries for hyperlink analysis. In *WWW*, pages 297–306, 2006.

[26] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, Beijing, China, 2007. ACM.

9. APPENDIX

9.1 Proof of Lemma 1

PROOF. $\eta \in T^{\lceil \lambda/\theta \rceil}$, so $\eta = T_x^{\lceil \lambda/\theta \rceil}$ for some $x \geq 0$. We prove the lemma by induction on x. For $x = 0$, the lemma follows from the definition of GenSeg(G, λ, θ). Now, we assume the result for $\eta = T_x^{\lceil \lambda/\theta \rceil}$ and prove it for $\eta' = T_{x+1}^{\lceil \lambda/\theta \rceil}$. First, for $u \in V$ and $1 \leq i \leq \eta'$, we prove that $W[u, i, \eta']$ is a proper random walk. If $i = \frac{\eta+1}{2}$, then $W[u, i, \eta'] = W[u, i, \eta]$ which is, by the inductive hypothesis, a proper random walk. So, assume $i < \frac{\eta+1}{2}$. Then, $W[u, i, \eta'] = W[u, i, \eta].Append(W[v, \eta - i + 1, \eta])$ where $v = E[u, i, \eta]$. By the inductive assumption, $W[u, i, \eta]$ and $W[v, \eta - i + 1, \eta]$ are proper random walks. Furthermore, since $i \neq \eta - i + 1$ (because, by assumption, $i < \frac{\eta+1}{2}$), these random walks are independent. Therefore, $W[u, i, \eta']$ is composed of two proper and independent random walks, and is hence a proper random walk itself.

Next, we prove, for $1 \leq i \neq j \leq \eta'$, $W[u, i, \eta']$ and $W[v, j, \eta']$ are independent. We consider two cases:

1. $\max\{i, j\} = \frac{\eta+1}{2}$. Assume $\max\{i, j\} = i$ (the other case is symmetric). Then $W[u, i, \eta'] = W[u, i, \eta]$, $j \neq \frac{\eta+1}{2}$ (because, $j \neq i$), and hence

$$W[v, j, \eta'] = W[v, j, \eta].Append(W[x, \eta - j + 1, \eta])$$

where $x = E[v, j, \eta]$. However, from the inductive hypothesis, since $i \neq j$ and $i \neq \eta - j + 1$, we know $W[u, i, \eta]$ is independent from both $W[v, j, \eta]$ and $W[x, \eta - j + 1, \eta]$. Thus, $W[u, i, \eta']$ is independent from $W[v, j, \eta']$, as desired.
2. $i, j < \frac{\eta+1}{2}$. Assume $i < j$. Then,

$$\begin{aligned} W[u, i, \eta'] &= W[u, i, \eta].Append(W[z, \eta - i + 1, \eta]) \\ W[v, i, \eta'] &= W[v, i, \eta].Append(W[x, \eta - j + 1, \eta]) \end{aligned}$$

Where $z = E[u, i, \eta]$, $x = E[v, j, \eta]$. But, since $i < j < \frac{\eta+1}{2} < \eta - j + 1 < \eta - i + 1$ both $W[u, i, \eta]$ and $W[z, \eta - i + 1, \eta]$ are independent from both $W[v, j, \eta]$ and $W[x, \eta - j + 1, \eta]$. Thus, $W[u, i, \eta']$ is independent from $W[v, j, \eta']$, as desired.

So, in either case, $W[u, i, \eta']$ is independent from $W[v, j, \eta']$. This finishes the inductive argument and hence the proof. □

9.2 Proof of Lemma 2

PROOF. We prove the result by induction on η. If $\eta = T_0^{\lceil \lambda/\theta \rceil}$, then the claim is obvious, by the definition of the GenSeg(G, λ, θ) algorithm. For the inductive step, assuming the claim to be true for $\eta = T_j^{\lceil \lambda/\theta \rceil}$, one can easily prove it for $\eta' = T_{j+1}^{\lceil \lambda/\theta \rceil}$, by separately considering the two cases of odd and even η, and using the recurive definition of $W[u, \cdot, \eta']$ in each case. We omit the details for the sake of space. □

9.3 Proof of Lemma 4

PROOF. For any $z \in \mathbb{N}$, $\lfloor \frac{z+1}{2} \rfloor \leq z$, so the sequence is monotone decreasing. Assume $2^\kappa \leq \eta < 2^{\kappa+1}$, and that the base 2 representation of η is $\eta = \sum_{r=0}^{\kappa} b_r 2^r$. Then, one can prove by induction that for $j \geq 0$: $T_j^\eta = \max_{i<j}\{b_i\} + \sum_{r=j}^{\kappa} b_r 2^{r-j}$. Then, we consider two cases:

1. $\eta = 2^\kappa$: then $\max_{0 \leq i \leq \kappa-1}\{b_i\} = 0$, and we get $T_\kappa^\eta = 1$, while $T_{\kappa-1}^\eta = 2 > 1$, which, together with the monotone decreasing property of the sequence, gives $|T^\eta| = 1 + \kappa = 1 + \lceil \log_2 \eta \rceil$.
2. $\eta > 2^\kappa$: then $\max_{0 \leq i \leq \kappa-1}\{b_i\} = 1$, and we get $T_{\kappa+1}^\eta = 1$, while $T_\kappa^\eta = 2 > 1$, which, together with the monotone decreasing property of the sequence, gives $|T^\eta| = \kappa + 2 = 1 + \lceil \log_2 \eta \rceil$ □