Efficient and Local Parallel Random Walks

Anonymous Author(s)
Affiliation
Address
email

Abstract

Random walks are a fundamental primitive used in many machine learning algorithms with several applications in clustering and semi-supervised learning. Despite their relevance, the first efficient parallel algorithm to compute random walks has been introduced very recently (Łącki et al.) Unfortunately their method has a fundamental shortcoming: their algorithm is non-local in that it heavily relies on computing random walks out of all nodes in the input graph, even though in many practical applications one is interested in computing random walks only from a small subset of nodes in the graph. In this paper, we present a new algorithm that overcomes this limitation by building random walk efficiently and locally at the same time. We show that our technique is both memory and round efficient, and in particular yields an efficient parallel local clustering algorithm. Finally, we complement our theoretical analysis with experimental results showing that our algorithm is significantly more scalable than previous approaches.

1 Introduction

Random walks are key components of many machine learning algorithms with applications in computing graph partitioning [ST04, GKL+21], spectral embeddings [CPS15, CKK+18], or network inference [HMMT18], as well as learning image segmentation [MS00], ranking nodes in a graph [AC07] and many other applications. With the increasing availability and importance of large scale datasets it is important to design efficient algorithms to compute random walks in large networks.

Several algorithms for computing random walks in parallel and streaming models have been proposed in the literature. In the streaming setting, Sarma, Gollapudi and Panigrahy [DSGP11] introduced multi-pass streaming algorithms for simulating random walks, and recently Jin [Jin19] gave algorithms for generating a single random walk from a prespecified vertex in one pass. The first efficient parallel algorithms for this problem has been introduced in the PRAM model [KNP92, HZ96].

In a more recent line of work, Bahmani, Chakrabarti, and Xin [BCX11] designed a parallel algorithm that constructs a single random walk of length ℓ from every node in $O(\log \ell)$ rounds in the massively parallel computation model (MPC), with the important caveat that these walks are not independent (an important property in many applications). This was followed by the work of Assadi, Sun and Weinstein [ASW19], which gave an MPC algorithm for generating random walks in an undirected regular graph. Finally, Łącki et al. [ŁMOS20] presented a new algorithm to compute random walks of length ℓ from every node in an arbitrary undirected graph. The algorithm of [ŁMOS20] still uses only $O(\log \ell)$ parallel rounds, and walks computed are now independent.

From an high level perspective the main idea behind all the MPC algorithms presented in [BCX11, ASW19, ŁMOS20] is to compute random walks of length ℓ by stitching together random walks of length $\ell/2$ in a single parallel round and to compute walks of of length $\ell/2$ by stitching together random walks of length $\ell/4$ and so on. It is possible to prove that such strategy leads to algorithms

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

that run in $O(\log \ell)$ parallel rounds as shown in previous work (this is also optimal under the 1-vs-2 cycle conjecture, as shown in [ŁMOS20]). Note that this technique in order to succeed computes in round i several random walks of length 2^i for all the nodes in the network in parallel. This technique is very effective if we are interested in computing random walk from all the nodes in the graph, or, more precisely, when the number of walks computed out of a node is proportional to its stationary distribution. However, this approach leads to significant inefficiencies when we are interested in computing random walks only out of a subset of nodes or for a single node in the graph. This is even more important when we consider than in many applications as in clustering [GLMY11, GS12, WGD13] we are interested in running random walks only from a small subset of seed nodes. So this leads to a natural question: *Is it possible to compute efficiently and in parallel random walks only from a subset of nodes in a graph?*

In this paper we answer this question in the affirmative, and we show an application of such a result in local clustering. Before describing our results in details we discuss the precise model of parallelism that we use in this work.

The MPC model. We design algorithms in the massively parallel computation (MPC) model, which is a theoretical abstraction of real-world system, such as MapReduce [DG08], Hadoop [Whi12], Spark [ZCF+10] and Dryad [IBY+07]. The MPC model [KSV10, GSZ11, BKS13] is the de-facto standard for analyzing algorithms for large-scale parallel computing.

Computation in MPC is divided into synchronous *rounds* over multiple machines. Each machine has memory S and at the beginning data is partitioned arbitrarily across machines. During each round, machines process data locally and then exchange data with the restriction that no machine receives more than S bits of data. The efficiency of an algorithm in this model is measured by the number of rounds it takes for the algorithm to terminate, by the size of the memory of every machine and by the total memory used in the computation. In this paper we focus on designing algorithm in the most restrictive and realistic regime where $S \in O(n^\delta)$ for a small constant $\delta \in (0, 1)$ – this is called fully scalable.

Our contributions. Our first contribution is an efficient algorithm for computing multiple random walks from a single node in a graph efficiently.

Theorem 1 (Informal)**.** *There exists a fully scalable MPC algorithm that, given a graph $G = (V, E)$ with a root vertex r, and parameters B^*, ℓ and λ, can simulate B^* independent random walks on G from r of length ℓ with an arbitrarily low error, in $O(\log \ell \log_\lambda B^*)$ rounds and $\widetilde{O}(m\lambda\ell^4 + B^*\lambda\ell)$ total space.*

Our algorithm also applies to the more general problem of generating independent random walks from a subset of nodes in the graph:

Theorem 2 (Informal)**.** *There exists a fully scalable MPC algorithm that, given a graph $G = (V, E)$ and a collection of non-negative integer budgets $(b_u)_{u\in V}$ for vertices in G such that $\sum_{u\in V} b_u = B^*$, parameters ℓ and λ, can simulate, for every $u \in V$, b_u independent random walks on G of length ℓ from u with an arbitrarily low error, in $O(\log \ell \log_\lambda B^*)$ rounds and $\widetilde{O}(m\lambda\ell^4 + B^*\lambda\ell)$ total space. The generated walks are independent across starting vertices $u \in V$.*

The following remark clarifies the effect of parameter λ on the number of machines.

Remark 1. *The parameter λ has nothing to do with the input of the algorithm, but is a trade-off parameter between space and round complexity. It is useful to think of it as $\lambda = n^\epsilon$ for some ϵ (not necessarily a constant), in which case we get a round complexity of $O(\log \ell \log B^*/\epsilon \log n) \leq O(\log \ell/\epsilon)$ and a total memory of $\widetilde{O}(mn^\epsilon\ell^4 + B^*\lambda\ell)$. We can set ϵ to, for example, $1/\log\log n$, to get nearly optimal total space and $\widetilde{O}(\log \ell)$ round complexity.*

If we compare our results with previous works, our algorithm computes truly independent random walks as [ŁMOS20] does. This is in contrast with the algorithm of [BCX11], which introduces dependen constructs not independent walks. Our algorithm has significantly better total memory than [ŁMOS20], which would result in memory $\Omega(m \cdot B^*)$ for generating B^* walks out of a root node r. This comes at the cost of a slightly higher number of rounds ($\log_\lambda B^*$, a factor that in many applications can be considered constant).

The main idea behind our approach is to stitch the random walks adaptively. More precisely, in an initial phase we construct only few random walks per node, then we construct them iteratively in

$O(\log_\lambda B^*)$ cycles. In each cycle we increase the number of walks that we build for node r by a factor of λ and we construct the walks only by activating other nodes in the graphs that contribute actively in the construction of the random walks for r. In this way we obtain an algorithm that is significantly more work efficient in terms of total memory, compared with previous work.

Our second contribution is to present an application of our algorithm to estimating Personalized PageRank and to local graph clustering. To the best of our knowledge, our algorithm is the first local clustering algorithm that uses a number of parallel rounds that only have a logarithmic dependency on the length of the random walk used by the local clustering algorithm.

Theorem 3. *For any $\lambda > 1$, let $\alpha \in (0, 1]$ be a constant and let C be a set satisfying: $\Phi(C) \leq \alpha/10$, $\mathrm{Vol}(C) \leq \frac{2}{3}\mathrm{Vol}(G)$. Then there is an MPC algorithm for local clustering that uses $O(\log \ell \cdot \log_\lambda B^*) = O(\log\log n \cdot \log_\lambda(\mathrm{Vol}(C)))$ rounds of communication and total memory $\widetilde{O}(m\lambda\ell^4 + B^*\lambda\ell) = \widetilde{O}(m\lambda + \lambda\mathrm{Vol}(C)^2)$, where $B^* := \frac{10^6 \log^3 n}{\eta^2\alpha^2}$, $\ell := \frac{10 \log n}{\alpha}$ and $\eta = \frac{1}{10\mathrm{Vol}(C)}$, and outputs a cluster with conductance $O(\sqrt{\alpha \log(\mathrm{Vol}(C))})$.*

Finally we present an experimental analysis of our results where we show that our algorithm to compute random walk is significantly more efficient then previous work [ŁMOS20], and that our technique scale to very large graphs.

Additional related works. Efficient parallel random walks algorithm have also been presented in distributed computing [DSNPT13], although these algorithms require a number of rounds linear in the diameter of the graph. Several parallel algorithm have also been presented for estimating PageRank or PersonalizedPageRank [DSGP11, DSMPU15, BCX11], but those algorithms either have higher round complexity or introduce dependencies between the PersonalizedPageRank vectors computed for different nodes. Finally there has been also some work in parallelize local clustering algorithm [CS15], although all previously known methods have complexity linear in the number of steps executed by the random walk/process used in the algorithm(in fact, our method could potentially be used to speed-up this work as well).

Notation. We work on undirected unweighted graphs, which we usually denote by $G = (V, E)$, where V and E are the set of vertices and the set of edges, respectively. We also have $n := |V|$ and $m := |E|$, unless otherwise stated. We define matrix D as the diagonal matrix of degrees, i.e., $D_{i,i} = d(v_i)$. Also, we let A be the adjacency matrix, where $A_{i,j} = 1$ if and only if there is an edge joining v_i and v_j, and $A_{i,j} = 0$, otherwise.

2 MPC random walks

In this section, we present our main result to compute B^* random walks from a single vertex r up to a length of ℓ, then we generalize it to multiple sources. As mentioned before our main idea is to carefully stitch random walks adaptively to activate nodes locally. In the rest of the section, we start by presenting our main theorem and giving a brief overview of our algorithm. We then describe our stitching algorithm, and analyze the number of random walks we must start from each node so that our algorithms work. Finally, we present the extension of our result to the setting with multiple sources.

Theorem 4. *There exists a fully scalable MPC algorithm that, given an undirected graph $G = (V, E)$ with a root vertex r, and parameters B^*, ℓ and ϵ, can simulate B^* independent random walks on G from r to length ℓ with an arbitrarily low total variation distance error,*[1] *in $O(\log \ell \log_\lambda B^*)$ rounds and $\widetilde{O}(m\lambda\ell^4 \log n + B^*\lambda\ell)$ total space.*

2.1 Overview of our Algorithm

Here, we explain the frameworks of stitching and budgeting, which are the two key tools making up our algorithm. For simplicity and without loss of generality assume that each vertex v has its own machine that stores its neighborhood, its budgets, and its corresponding random walks.[2]

[1] n^{-C} for an arbitrary constant C.

[2] In reality multiple vertices may share a machine, if they have low degree; or if a vertex has a high degree it may be accommodated by multiple machines in a constant-depth tree structure.

Remark 2. *For ease of notation we assume that $\ell = 2^j$ for some integer j. One can see that this assumption is without loss of generality, because otherwise one can always round ℓ to the smallest power of 2 greater than ℓ, and solve the problem using the rounded ℓ. This affects the complexity bounds by at most a constant factor.*

Stitching. Here, we explain the framework of stitching, which is a key tool for our algorithm. For simplicity and without loss of generality assume that each vertex v has its own machine that stores its neighborhood, its budgets, and its corresponding random walks[3]. At each point in time the machine corresponding to v stores sets of random walks of certain lengths, each starting in v. Each edge of each walk is labeled by a number from 1 to ℓ, denoting the position it will hold in the completed walk. Thus, a walk of length s could be labeled from $k+1$ to $k+s$ for some k. At any given time let $W_k(v)$ denote the set of walks stored by v whose first label is k and $B(v,k)$ denotes their cardinality. After the initial round of edge generation $W_k(v)$ consists of $B(v,k)$ individual edges adjacent to v, for each v and k.

In the first round of stitching, for each edge (or length one walk) e in $W_k(v)$, for any *odd* k, v sends a request for the continuation of the walk to z, where z is the other endpoint of e. That is, v sends a request to z for an element of $W_{k+1}(z)$. Each vertex sends out all such requests simultaneously in a single MPC round. Following this each vertex replies to each request by sending a walk from the appropriate set. Crucially, each request must be answered with a *different, independent* walk. If the number of requests for $W_{k+1}(z)$ exceeds $|W_{k+1}(z)| = B(z,k+1)$, the vertex z declares failure and the algorithm terminates. Otherwise, all such requests are satisfied simultaneously in a single MPC round. Finally, each vertex v increases the length of each of its walks in $W_k(v)$ to two when k is odd, and deletes all remaining walks in $W_k(v)$ when k is even (see Figure 1).

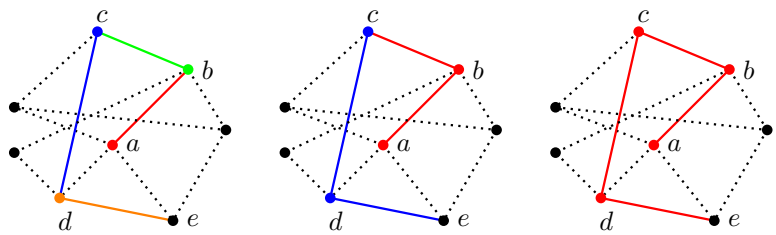
Budgeting. The main challenge in making stitching work with low space and few rounds is to set the vertex budgets ($B(v,k)$) accurately enough for stitching to succeed – this is the main technical contribution of this paper. If we wish to run walks from each vertex proportionally to the vertex degree (ie. from the stationary distribution) we can simply set $B(v,k) \sim \deg(v)$ for all v and k. This can be done because it is easy to calculate the stationary distribution in an undirected graph. However, if we wish to run random walks from a single root vertex r, setting the budgets becomes much more difficult.

Our technique is to run multiple cycles of stitching sequentially. In the first cycle, we simply start from the stationary distribution. Then, with each cycle, we shift closer and closer to the desired distribution, in which the budget of r is much greater than the budgets of other vertices. We do this by augmenting $B(r,1)$ by some parameter $\lambda \approx n^\epsilon$ each cycle. This forces us to augment other budgets as well: For example, for u in the neighborhood of r we expect to have a significantly increased budget $B(u,2)$. In order to estimate the demand on u (and all other vertices) we use data from the previous cycle: If κ rooted walks visited u as their second step in the previous cycle, we expect this number to increase to $\lambda \cdot \kappa$ this cycle. Hence, we can preemptively increase $B(u,2)$ to $\lambda \cdot \kappa$.

Informally, since the above method allows for an additive error of $\approx \lambda$ in our estimate, we are also forced to run $\approx \lambda$ additional walks from each vertex as a buffer. This contributes the dominant $m\lambda$ term in our space requirement. For more information on how the budgets evolve see Appendix C in the Appendix.

Budgeting. As described before the core idea behind our algorithm is to increase the number of pre-constructed random walks carefully. We do this by constructing the random walks in $\log_\lambda B^*$ phases. In the first phase we only generate B_0 random walks from v and to do we initialize the budget of every node to B_0. Then we proceed in phases and in each phase we increase the number of random walks from v by a factor λ(by increasing its budget). The key observation is that to do it we need to update the budget only for nodes that we observe frequently enough in the random walk constructed so far. In fact, by carefully setting a threshold θ we can guarantee two main properties. First, that for any node that we observe less than θ times the initial budget B_0 is sufficient. Second, for nodes that we observe more than θ times we can estimate accurately the budget needed to complete the random walks. Using this two facts, we can increase the budget gradually and locally and reduce the total number of random walks computed by the algorithm. For a precise description of this procedure please refer to thee pseudocode for Algorithm 2.

[3] In reality multiple vertices may share a machine, if they have low memory requirement; or if a vertex has a high memory requirement it may be accommodated by multiple machines in a constant-depth tree structure.



(a) Before round 1 (b) Before round 2 (c) Before round 3

Figure 1: Illustration of stitching algorithm for walk (a, b, c, d, e). The red, green, blue and orange walks in each figure correspond to walks in $W_1(a)$, $W_2(b)$, $W_3(c)$ and $W_4(d)$, respectively.

Algorithm 1 Stitching algorithm

1: **procedure** STITCH(G, B)
2: **for** $v \in V$ in parallel **do**
3: **for** $k \in [\ell]$ **do**
4: $W_k(v) \leftarrow$ a set of $B(v, k)$ independent uniformly random edges adjacent on v
5: **for** $j = 1 \ldots \log_2 \ell$ **do** ▷ See Remark 2
6: **for** $v \in V$ in parallel **do**
7: **for** $k \equiv 1 \ (mod\ 2^j)$ **do**
8: **for** walk $p \in W_k(v)$ **do**
9: **send** (v, k) **to** z, where $z \leftarrow$ end vertex of p
10: **for** $z \in V$ in parallel **do**
11: **for** each message (v, k) **do**
12: **if** $W_{k+2^{j-1}}(z) = \emptyset$ **then**
13: **return Fail**
14: **send** (q, k, z) **to** v, where the walk $q \leftarrow$ a randomly chosen walk in $W_{k+2^{j-1}}(v)$
15: $W_{k+2^{j-1}}(z) \leftarrow W_{k+2^{j-1}}(z) \backslash \{q\}$
16: **for** $v \in V$ in parallel **do**
17: **for** each message (q, k, z) **do**
18: $p \leftarrow$ any walk of length 2^{j-1} in $W_k(v)$ with end vertex z
19: $W_k(v) \leftarrow W_k(v) \backslash \{p\} \cup \{p + q\}$ ▷ $p + q$ is the concatenated walk
20: **for** $k \equiv 2^{j-1} \ (mod\ 2^j)$ **do**
21: $W_k(v) \leftarrow \emptyset$
22: **return** $W_1(v)$ for all $v \in V$

Analysis. We are now ready to present the main properties of our algorithm:

Lemma 1 (Correctness and complexity)**.** *Algorithm 2 takes $O(\log \ell \cdot \log_\lambda B^*)$ rounds of MPC communication,* STITCH *terminates without failures with high probability, and the total amount of memory used for walks is $\sum_{v \in V} \sum_{k=1}^{\ell} B(v, k) = O(m\lambda\ell^4 \log n + B^*\lambda\ell)$.*

Proof. Let P^k be distribution of random walks of length k starting from r. That is, the probability that such a walk ends up in $v \in V$ after k steps is $P^k(v)$.

To bound the round complexity we note that each call of STITCH takes only $O(\log \ell)$ rounds of communication, and it is called $\lfloor \log_\lambda B^* \rfloor + 1$ times; this dominates the round complexity. Further rounds of communication are needed to update the budgets. However this can be done in parallel for each vertex, and thus takes only one round per iteration of the outer for-loop (Line 5).

To prove that the algorithm fails with low probability, we must show the following crucial claim about the budgets $B(V, K)$. Recall that the ideal budget in the ith iteration would be $B(v, k) \approx B_0(v) + \lambda^i \cdot P^k(v)$. We show that in reality, the budgets do not deviate too much from this.

Algorithm 2 Main Algorithm (Budgeting)

1: **procedure** MAIN$(G, r, \ell, B^*, \lambda)$
2: $\theta \leftarrow 10C\ell^2 \log n$, $B_0 \leftarrow 30C\lambda\ell^3 \log n$, $\tau \leftarrow 1 + \sqrt{\frac{20C \log n}{\theta}}$ ▷ **Parameter settings**
3: $\forall v \in V$, $B_0(v) \leftarrow B_0 \cdot \deg(v)$
4: $\forall v \in V$, $\forall k \in [\ell]:\ B(v,k) \leftarrow B_0 \cdot \deg(v)$
5: **for** $i = 1 \ldots \lfloor \log_\lambda B^* \rfloor$ **do**
6: $W_1 \leftarrow$ STITCH(G,B)
7: $W \leftarrow W_1(r)$
8: **for** $v \in V,\ k \in [\ell]$ **do**
9: $\kappa \leftarrow |\{w \in W | w_k = v\}|$
10: **if** $\kappa \geq \theta$ **then**
11: $B(v,k) \leftarrow (B_0(v) + \lambda^i \cdot \frac{\kappa}{|W|}) \cdot \tau^{3k-3}$
12: **else**
13: $B(v,k) \leftarrow B_0(v) \cdot \tau^{3k-3}$
14: $W_1 \leftarrow$ STITCH(G, B)
15: $W \leftarrow W_1(r)$
16: **return** W

Claim 1. *After iteration i of the outer for-loop (Line 5) in Algorithm 2, with high probability B is set such that*

$$\forall v \in V,\ k \in [\ell]:\ B(v,k) \in \left[(B_0(v) + \lambda^i \cdot P^k(v)) \cdot \tau^{3k-4}, (B_0(v) + \lambda^i \cdot P^k(v)) \cdot \tau^{3k-2}\right].$$

Proof. We first note how $B(r,1)$ – the budget of walks starting at the root vertex – evolves. For $v = r$ and $k = 1$, κ is always equal to $|W|$ – since r is the root vertex – and greater than θ. Therefore, $B(r,1)$ is set in Line 11 of Algorithm 2 to $(B_0(r) + \lambda^i)$. This is important, because it means that when setting other budgets in iteration $i > 1$, $|W|$ is always $(B_0(r) + \lambda^{i-1})$, the number of walks rooted from r in the previous round. The exception is the first iteration, when $|W|$ is simply $|B_0(r)|$. In both cases we may say that $|W| \geq \lambda^{i-1}$.

There are two options we have to consider: If after the ith round of STITCH κ exceeded θ, in which case our empirical estimator $\kappa/|W|$ for $P^k(v)$ is deemed reliable. We then use this estimater to set the budget for the next round (see Line 11). Alternately, if κ did not exceed θ, the imperical estimator is deemed too unreliable; we then simply set $B(v,k)$ proportionally to $B_0(v)$ (see Line 13).

Case I ($\kappa < \theta$) then intuitively, κ is too small to provide an accurate estimator of $P^k(v)$. In this case we are forced to argue that the (predictable) term $B_0(v)$ dominates the (unknown) term $P^k(v)$. Since $\kappa < \theta$, $\mathbb{E}(\kappa) = P^k(v) \cdot |W| \leq 2\theta$. (The opposite happens with low probability[4] by Chernoff bounds, since $\theta \geq 10C \log n$.) Therefore,

$$B(v,k) = B_0(v) \cdot \tau^{3k-3} \leq (B_0(v) + \lambda^i \cdot P^k(v)) \cdot \tau^{3k-2},$$

and

$$B(v,k) = B_0(v) \cdot \tau^{3k-3} = (B_0(v) + \lambda^i \cdot P^k(v)) \cdot \left(1 - \frac{\lambda^i \cdot P^k(v)}{B_0(v) + \lambda^i \cdot P^k(v)}\right) \cdot \tau^{3k-3}.$$

So, we need to prove that $1 - \frac{\lambda^i \cdot P^k(v)}{B_0(v) + \lambda^i \cdot P^k(v)} \geq \tau^{-1}$. Now, by the above bound on $\mathbb{E}(\kappa)$ as well as the fact that $|W| \geq \lambda^{i-1}$, we have $2\theta \geq P^k(v) \cdot |W| \geq P^k(v) \cdot \lambda^{i-1}$, which results in $\lambda^i \cdot P^k(v) \leq 2\lambda\theta$. Consequently,

$$\left(1 - \frac{\lambda^i \cdot P^k(v)}{B_0(v) + \lambda^i \cdot P^k(v)}\right) \geq \left(1 - \frac{2\lambda\theta}{B_0(v) + 2\lambda\theta}\right) \geq \tau^{-1}.$$

Here we used $B_0(v) \geq B_0 \geq 3\lambda\theta \cdot (\sqrt{\theta/(20C \log n)})$, which holds by definition of $B_0(v)$ and our setting of parameters B_0 and θ.

[4] Throughout the proof, we say 'low probability' to mean probability of $n^{-\Omega(C)}$ where C can be set arbitrarily high.

Case II ($\kappa \geq \theta$), then intuitively κ is robust enough to provide a reliable estimator for $P^k(v)$. More precisely, $\kappa/|W| \in \left[\mathbb{E}(\kappa/|W|) \cdot \tau^{-1}, \mathbb{E}(\kappa/|W|) \cdot \tau\right]$ with high probability – indeed τ is defined in terms of θ deliberately in exactly such a way that this is guaranteed by Chernoff bounds. $\mathbb{E}(\kappa/|W|) = P^k(v)$, therefore

$$\lambda^i \cdot \frac{\kappa}{|W|} \in \left[\lambda^i \cdot P^k(v) \cdot \tau^{3k-4}, \lambda^i \cdot P^k(v) \cdot \tau^{3k-2}\right],$$

and

$$\begin{aligned} B(v,k) &= (B_0(v) + \lambda^i \cdot \frac{\kappa}{|W|}) \cdot \tau^{3k-3} \\ &\in \left[\left(B_0(v) + \lambda^i \cdot P^k(v)\right) \cdot \tau^{3k-4}, (B_0(v) + \lambda^i \cdot P^k(v)) \cdot \tau^{3k-2}\right]. \end{aligned}$$

□

STITCH only reports failure if for some $v \in V$ and $k \in [2, \ell]$, vertex v receives more requests for walks in $W_k(v)$ than $|W_k(v)| = B(v,k)$. Number of such request is upper bounded by the number of edges ending in v generated by neighbors of v, say w at level $k-1$. That is, the number of requests for $W_k(v)$ is in expectation at most

$$\begin{aligned} \sum_{w \in \Gamma(v)} \frac{1}{d(w)} B(w, k-1) &\leq \sum_{w \in \Gamma(v)} \frac{1}{d(w)} (B_0(w) + \lambda^i \cdot P^{k-1}(w)) \cdot \tau^{3k-5} \\ &= \left(\sum_{w \in \Gamma(v)} \frac{1}{d(w)} B_0(w) + \lambda^i \cdot P^k(v) \right) \cdot \tau^{3k-5} \\ &= \left(d(v) B_0 + \lambda^i \cdot P^k(v)\right) \cdot \tau^{3k-5} \\ &= \left(B_0(v) + \lambda^i \cdot P^k(v)\right) \cdot \tau^{3k-5}. \end{aligned}$$

Since this is greater than θ, the actual number of requests is at most $(B_0(v) + \lambda^i \cdot P^k(v)) \cdot \tau^{3k-4} \leq B(v,k)$, with high probability by Chernoff. Therefore, STITCH indeed does not fail.

Finally, we prove the memory bound. By setting of parameter θ, τ^{3k-2} is at most a constant. Also by the setting of parameters we have,

$$B(v,k) \leq (B_0(v) + \lambda^{\lfloor \log_\lambda B^* \rfloor + 1} \cdot P^k(v)) \cdot \tau^{3k-2} = O(B_0(v) + B^* \lambda \cdot P^k(v)),$$

and $\sum_{v \in V} \sum_{k=1}^{\ell} B(v,k) = O(1) \cdot \sum_{v \in V} \sum_{k=1}^{\ell} (B_0(v) + B^* \lambda \cdot P^k(v)) = O(m\lambda\ell^4 \log n + B^* \lambda \ell)$.

□

This gives us the proof of Theorem 4. Now, one can easily extend this result to the case when multiple sources for the starting vertex is considered. A formal version of Theorem 2 is proven in Appendix A.

3 From random walks to local clustering

We now present two applications for our algorithm to compute random walks. In particular we show how to use it to compute PageRank vectors and how to use it to compute local clustering. In interest of space, wee only state here our main results and we defer all the technical definition and proofs to the Appendix.

Approximating PageRank using MPC random walks Interestingly, we can show that we can use our algorithm as a primitive to compute PersonalizedPageRank for a node of for any input vector[5]

Theorem 5 (Approximating PersonalizedPageRank using MPC random walks)**.** *For any starting single vertex vector s (indicator vector), any $\alpha \in (0,1)$ and any η, there is a MPC algorithm that using $O(\log \ell \cdot \log_\lambda B^*)$ rounds of communication and the total amount of memory of $O(m\lambda\ell^4 \log n + B^* \lambda \ell)$, outputs a vector $\widetilde{q}$, such that $\widetilde{q}$ is a η-additive approximation to $\mathrm{pr}_\alpha(s)$, where $B^* := \frac{10^6 \log^3 n}{\eta^2 \alpha^2}$ and $\ell := \frac{10 \log n}{\alpha}$.*

[5] We note that [BCX11] also propose an algorithm to compute PersonalizedPageRank vector but with the limitation that this could be computed only for a single node and not for a vector.

The proof is deferred to Appendix C.

Using approximate PersonalizedPageRank vectors to find sparse cuts Now we can use the previous result on PersonalizedPageRank to find sets with relatively sparse cuts. Roughly speaking, we argue that for any set C of conductance $O(\alpha)$, for many vertices $v \in C$, if we calculate an approximate $\mathrm{pr}_\alpha(v)$ using our algorithms and perform a sweep cut over it, we can find a set of conductance $O(\sqrt{\alpha \log(\mathrm{Vol}(C))})$. This result is stated in Theorem 3. The proof of this result is very similar to the proofs of Section 5 of [ACL06], however since our approximation guarantees are slightly different, we need to modify some parts of the proof for completeness. The full proof is presented in Appendix Our main result of this subsection is stated below.

4 Empirical Evaluation

In this Section we present empirical evaluations of our algorithms for random walk generation, as well as clustering. As our datasets, we use several real-world graphs form the Stanford Network Analysis Project [LK14, LKF07, LLDM08, KY04, YL12]. The graphs are undirected and come mostly (though not in all cases) from 'Networks with ground truth communities', where the clustering application is most relevant. In order to demonstrate the scalability of our main Algorithm we use graphs of varying sizes, as demonstrated in the table below.

Table 1: Summary of the various graphs used in our empirical evaluations.

NAME	VERTICES	EDGES	DESCRIPTION
CA-GRQC	5424	14,496	COLLABORATION NETWORK
EMAIL-ENRON	36,692	183,831	EMAIL COMMUNICATION NETWORK
COM-DBLP	317,080	1,049,866	COLLABORATION NETWORK
COM-YOUTUBE	1,134,890	2,987,624	ONLINE SOCIAL NETWORK
COM-LIVEJOURNAL	3,997,962	34,681,189	ONLINE SOCIAL NETWORK
COM-ORKUT	3,072,441	117,185,083	ONLINE SOCIAL NETWORK

The experiments were performed on Amazon's Elastic Map-Reduce system using the Apache Hadoop library. The clusters consisted of 30 machines each of modest memory and computing power (Amazon's `m4.large` instance) so as to best adhere to the MPC setting. Each experiment described in this section was repeated 3 times to minimize the variance in performance inherent in distributed systems like this.

Practical considerations. In Section 2 we worked with the guarantee that no walks "fail" in the stitching phase. This assumption can be fulfilled at nearly no expense to the (asymptotic) guarantees in space and round complexity of Theorem 4, and make the proof much cleaner. In practice however, allowing some small fraction of the walks to fail allows for a more relaxed setting of the parameters, and thus better performance.

Each experiment is performed with 15 roots, selected uniformly at random. The main parameters defining the algorithm are as follows: ℓ — the length of a target random walk (16 and 32 in various experiments), C — The number of cycles (iterations of the for-loop in Line 5 of Algorithm 2) performed. , B_0 — the initial budget-per-degree of each vertex, λ — the approximate scaling of the budgets of the root vertices each cycle, τ — a parameter defining the amount of excess budget used in stitching. This is used somewhat differently here than in Algorithm 2. For more details see Appendix D.

4.1 Scalability

In this section we present the results of our experiments locally generating random walks simultaneously from multiple root vertices. We use the graphs COM-DBLP, COM-YOUTUBE, COM-LIVEJOURNAL, and COM-ORKUT, in order to observe how the runtime of Algorithm 2 scales with the size of the input graph. In each of these graphs, 15 root vertices have been randomly chosen. We ran three experiments with various settings of the parameters, of which one is presented below. For additional experiments see Appendix D. B_0 is set to be proportional to n/m – that is inverse

proportional to the average degree – since the initial budget of each of each vertex is set to B_0 times the degree of the vertex.

We report the execution time in the Amazon Elastic Map-Reduce cluster, as well as the number of rooted walks generated. Finally, under 'Walk failure rate', we report the percentage of *rooted* walks that failed in the *last cycle* of stitching. This is the crucial quantity; earlier cycles are used only to calibrate the vertex budgets for the final cycle.[6]

Table 2: Experiments with $\ell = 16$, $C = 3$, $B_0 = 6n/m$, $\lambda = 32$, $\tau = 1.4$.

GRAPH	TIME	B_0	ROOTED WALKS GENERATED	WALK FAILURE RATE
COM-DBLP	23 ± 7 MINUTES	1.812	$96,362 \pm 2597$	$14.6 \pm 0.6\%$
COM-YOUTUBE	34 ± 6 MINUTES	2.279	$53,076 \pm 1185$	$10.8 \pm 0.5\%$
COM-LIVEJOURNAL	76 ± 11 MINUTES	0.692	$184,246 \pm 756$	$7.9 \pm 0.1\%$
COM-ORKUT	64 ± 13 MINUTES	0.157	$200,924 \pm 1472$	$3.4 \pm 0.0\%$

We observe that Algorithm 2 successfully generates a large number of rooted walks – far more than the initial budgets of the root vertices. As predicted, execution time scales highly sublinearly with the size of the input (recall for example, that COM-ORKUT is more than a hundred times larger than COM-DBLP). The failure rate of walks decreases with the size of the graph in this dataset, with that of COM-ORKUT reaching as low as 3.4% on average; this may be due to the the higher average degree of our larger graphs, leading to the random walks spreading out more.

In Appendix D we report the results of two more experiments, including one with longer walks.

4.2 Comparison

In this section we compare to the previous work of [ŁMOS20] for generating random walks in the MPC model. This work heavily relies upon generating random walks from all vertices simultaneously, with the number of walks starting from a given vertex v being proportional $d(v)$. In many applications, however, we are interested in computing random walks from a small number of root vertices. The only way to implement this using the methods of [ŁMOS20] is to start with an initial budget large enough to guarantee the desired number of walks from each vertex.

We perform similar experiments to those in the previous section – albeit on much smaller graphs. Each graph has 15 root vertices chosen randomly, from which we wish to sample random walks. In Table 3 we set B_0 to 1 and perform $C = 3$ cycles of Algorithm 2 with $\lambda = 10$, effectively augmenting the budget of root vertices by a factor 100 by the last cycle. Correspondingly, we implement the algorithm of [ŁMOS20] – which we call UNIFORM STITCHING – by simply setting the initial budget 100 times higher, and performing only a single cycle of stitching, ie.: $B_0 = 100$, $C = 1$.

Table 3: Experiments with $\ell = 16$, $\lambda = 10$, $\tau = 1.3$. The row labeled Algorithm 2' corresponds to $B_0 = 1$, $C = 3$, while the row labeled 'Uniform Stitching' corresponds to $B_0 = 100$, $C = 1$.

ALGORITHM	CA-GRQC	EMAIL-ENRON	COM-DBLP
ALGORITHM 2	15 ± 1 MINUTES	19 ± 1 MINUTES	18 ± 1 MINUTES
UNIFORM STITCHING	7 ± 0 MINUTES	15 ± 0 MINUTES	66 ± 1 MINUTES

We observe that the running time of our Algorithm 2 barely differs across the three graphs, despite the nearly 100-factor difference between the sizes of CA-GRQC and COM-DBLP. At this small size, the execution time is dominated by setting up the 12 Map-Reduce rounds required to execute Algorithm 2 with these parameters. As expected, the baseline far outperforms our algorithm on the smallest graph; as the size of the input graph grows, however, its running time deteriorates quickly. For larger setting of the parameters UNIFORMSTITCHING can no longer complete on the cluster, due to the higher memory requirement, as we can see in Table 6 in Appendix D.

[6] For completeness, we report the empirical standard deviation everywhere. Note, however, that due to the high resource requirement of these experiments, each one was only repeated three times.

References

[AC07] Alekh Agarwal and Soumen Chakrabarti. Learning random walks to rank nodes in graphs. In *Proceedings of the 24th international conference on Machine learning*, pages 9–16, 2007.

[ACL06] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486, 2006.

[ASW19] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 461–470, 2019.

[BCX11] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. Fast personalized pagerank on mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 973–984, 2011.

[Ber07] Pavel Berkhin. Bookmark-coloring approach to personalized pagerank computing. Technical report, Internet Mathematics, 2007.

[BKS13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 273–284. ACM, 2013.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Networks*, 30(1-7):107–117, 1998.

[CKK+18] Ashish Chiplunkar, Michael Kapralov, Sanjeev Khanna, Aida Mousavifar, and Yuval Peres. Testing graph clusterability: Algorithms and lower bounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 497–508. IEEE, 2018.

[CPS15] Artur Czumaj, Pan Peng, and Christian Sohler. Testing cluster structure of graphs. In *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, pages 723–732, 2015.

[CS15] Fan Chung and Olivia Simpson. Distributed algorithms for finding local clusters using heat kernel pagerank. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 177–189. Springer, 2015.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DSGP11] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. *Journal of the ACM (JACM)*, 58(3):1–19, 2011.

[DSMPU15] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. Fast distributed pagerank computation. *Theoretical Computer Science*, 561:113–121, 2015.

[DSNPT13] Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. Distributed random walks. *Journal of the ACM (JACM)*, 60(1):1–31, 2013.

[FR04] Dániel Fogaras and Balázs Rácz. Towards scaling fully personalized pagerank. In Stefano Leonardi, editor, *Algorithms and Models for the Web-Graph: Third International Workshop, WAW 2004, Rome, Italy, October 16, 2004, Proceeedings*, volume 3243 of *Lecture Notes in Computer Science*, pages 105–117. Springer, 2004.

[GKL+21] Grzegorz Gluch, Michael Kapralov, Silvio Lattanzi, Aida Mousavifar, and Christian Sohler. Spectral clustering oracles in sublinear time. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1598–1617. SIAM, 2021.

[GLMY11] Ullas Gargi, Wenjun Lu, Vahab Mirrokni, and Sangho Yoon. Large-scale community detection on youtube for topic discovery and exploration. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 5, 2011.

[GS12] David F Gleich and C Seshadhri. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 597–605, 2012.

[GSZ11] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*, pages 374–383. Springer, 2011.

[Hav03] T.H. Haveliwala. Topic-sensitive pagerank: a context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):784–796, 2003.

[HMMT18] Jeremy G Hoskins, Cameron Musco, Christopher Musco, and Charalampos E Tsourakakis. Inferring networks from random walk-based node similarities. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 3708–3719, 2018.

[HZ96] Shay Halperin and Uri Zwick. An optimal randomised logarithmic time connectivity algorithm for the erew pram. *Journal of Computer and System Sciences*, 53(3):395–416, 1996.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[Jin19] Ce Jin. Simulating random walks on graphs in the streaming model. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, pages 46:1–46:15, 2019.

[JW03] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, page 271–279, New York, NY, USA, 2003. Association for Computing Machinery.

[KNP92] David R Karger, Noam Nisan, and Michal Parnas. Fast connected components algorithms for the erew pram. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 373–381, 1992.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

[KY04] Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *CEAS*, 2004.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[LKF07] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2, 2007.

[LLDM08] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[ŁMOS20] Jakub Łącki, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Walking randomly, massively, and efficiently. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 364–377, 2020.

[MS00] Marina Meila and Jianbo Shi. Learning segmentation by random walks. *Advances in neural information processing systems*, 13:873–879, 2000.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[ST04] Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90, 2004.

[WGD13] Joyce Jiyoung Whang, David F Gleich, and Inderjit S Dhillon. Overlapping community detection using seed set expansion. In *Proceedings of the 22nd ACM international conference on information & knowledge management*, pages 2099–2108, 2013.

[Whi12] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[YL12] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? **Yes**
   (b) Have you read the ethics review guidelines and ensured that your paper conforms to them? **Yes**
   (c) Did you discuss any potential negative societal impacts of your work? **N/A. This work is a theoretical result on the parallel simulation of random walks. It has no direct societal impact.**
   (d) Did you describe the limitations of your work? **Yes**
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? **Yes**
   (b) Did you include complete proofs of all theoretical results? **Yes**
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? **Yes. Code is submitted with the supplementary material.**
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? **N/A**
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? **Yes. (See tables.)**
   (d) Did you include the amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? **Yes. (See the second paragraph of Section 4.)**
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets... **The only existing assets used are the data-sets from the well-known Stanford Network Analysis Project.**
   (a) If your work uses existing assets, did you cite the creators? **Yes**
   (b) Did you mention the license of the assets? **N/A**
   (c) Did you include any new assets either in the supplemental material or as a URL? **No**
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? **N/A**
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? **N/A**
5. If you used crowdsourcing or conducted research with human subjects... **N/A**