Scalable Graph Clustering with Pregel

Bryan Perozzi[1], Christopher McCubbin[2], Spencer Beecher[3], and J.T. Halbert[3]

Abstract We outline a method for constructing in parallel a collection of local clusters for a massive distributed graph. For a given input set of (vertex, cluster size) tuples, we compute approximations of personal PageRank vectors in parallel using Pregel, and sweep the results using MapReduce. We show our method converges to the serial approximate PageRank, and perform an experiment that illustrates the speed up over the serial method. We also outline a random selection and deconfliction procedure to cluster a distributed graph, and perform experiments to determine the quality of clusterings returned.

1 Introduction

Recent developments in clustering algorithms have allowed the extraction of local clusters using a localized version of the PageRank algorithm known as “Personalized PageRank” [2]. Personal PageRank vectors can be efficiently computed by approximation. The “Approximate personal PageRank” (APR) approach concentrates the starting location of a normal PageRank in one vertex of the graph, and limits the distance that the PageRank walk and teleportation can progress.

One may then sort the surrounding vertices in decreasing order by their degree weighted probability from the APR vector, and then *sweep* them to search for the

Bryan Perozzi
Department of Computer Science, Stony Brook University, Stony Brook, NY, USA
e-mail: `bperozzi@cs.stonybrook.edu`

Christopher McCubbin
Sqrrl Data, Inc., Boston, MA, USA e-mail: `chris@sqrrl.com`

Spencer Beecher
TexelTek, Inc., Columbia, MD, USA e-mail: `sbeecher@texeltek.com`

J.T. Halbert
TexelTek, Inc., Columbia, MD, USA e-mail: `jhalbert@texeltek.com`

presence of a local cluster. This method generates a local good cluster if it exists, and runs in time proportionate to the size of the cluster.

In this work, we outline a method for constructing a collection of local clusters of a distributed graph in parallel. There are many possible applications for this technique, and we illustrate this with an example which uses local clusters to generate a clustering.

Our process to generate a clustering of a graph is composed of four steps that represent an extension of the work of Spielman and Teng [18] and Andersen et. al. [2].

The four steps are as follows.

1. We pick random pairs of source vertices and cluster sizes. The vertices are drawn by degree from the stationary distribution and the cluster sizes are drawn from a modification of Spielman's *RandomNibble* procedure. [18]
2. We compute the approximate personal PageRank vectors in parallel using Pregel [13] for each of these seed pairs.
3. We perform a sweep using MapReduce to produce the local clusters.
4. We reconcile cluster overlaps by assigning vertices to the cluster with lowest conductance. This is an implementation of an idea put forward by Andersen et. al. in an unpublished technical report.

Our contributions are the algorithms *Parallel Approximate PageRank* (PAPR) and *MapReduce Sweep*(MRSweep), which together can find local clusters in parallel on large graphs. We refer to their combination as *ParallelNibble*. We also provide proofs of convergence and asymptotic running time and experimental investigation of both the quality of clusterings produced and the algorithm's scalability on a variety of real world graphs.

2 Background

We will be considering an undirected graph $G = \{V, E\}$ where V is the set of vertices and $E \subseteq \{V \times V\}$ is the set of edges. Let $n = |V|$ and $m = |E|$.

As is commonly known many graphs exhibit community structure: that is, it is possible to group vertices into densely interconnected sets $C = \{C_i | C_i \subseteq V\}$. Extracting these communities from large graphs is something of an art since finding methods to evaluate the quality of a community is still an active area of research. We will discuss two of the more popular measures here.

One metric to describe the quality of a community C_i in a graph G is its *conductance*, $\phi(C_i)$. Intuitively $\phi(C_i)$ is the ratio between the perimeter of the cluster and its size [10]. It is defined as:

$$\phi(C_i) = \frac{|\text{outgoing edges of } C_i|}{\min(\text{Vol}(C_i), \text{Vol}(V \setminus C_i))}$$

Where Vol$(C_i) = \sum_{j \in C_i}$ degree(j). A lower conductance score therefore indicates a better cluster; the vertices are more tightly connected to each other than to vertices outside their community.

The conductance $\phi(C)$ of a clustering C in graph G is defined to be the minimum conductance of its clusters. This measure is favored by many authors (in particular Spielman et. al. in [18]) because of its connection to global spectral clustering via the Cheeger inequality. Additionally Andersen et. al. build a local version of the Cheeger inequality in [2].

Another metric to evaluate the quality of a clustering C is its modularity [16]. Modularity is designed to calculate the ratio of the internal edges of clusters in a given clustering to the number of edges that would be expected given a random assignment of edges. We calculate modularity with the following formula:

$$q(C) = \sum_{C_i \in C} \left\{ \frac{|E(C_i)|}{m} - \left(\frac{\sum_{v \in C_i} deg(v)}{2m} \right)^2 \right\}$$

Modularity satisfies the inequality $\frac{-1}{2} \leq q \leq 1$, and higher modularities are considered better clusterings.

3 Related Work

In this section we describe current publications in local graph algorithms, clustering, and distributed systems as it pertains to our work.

Graph Clustering Algorithms An excellent overview of graph clustering is given in [17]. With the advent of so-called “big data”, graphs with node and edge cardinalities in the billions or more have become common. This has created the need for algorithms that are scalable. Beginning with the famous PageRank algorithm, spectral methods for analyzing graphs have gained popularity in the past decade. Local spectral clustering methods, introduced by Speilman and Teng [18], and advanced by Andersen et. al. in [2] seek to apply these techniques scalably. The best time complexity to date comes from EvoCut [3], but does not beat the approximation guarantee of [2], which our work is an extension of.

An alternative to computing personal PageRank vectors in parallel is presented by Bahmani et al [5], who developed a fast Monte Carlo method for approximating a personal PageRank vector for every vertex in a graph using MapReduce. Our work uses a different approximation technique for personal PageRank vectors and is built on Pregel, but could perhaps be enhanced by their technique.

Local spectral methods and other local methods often require that a seed set of nodes is chosen. The problem of selecting the best starting vertices for local graph clustering has attracted some attention in the literature. Methods typically try to quickly compute metrics associated with good communities, and then use these results to seed community detection algorithms based on personalized PageRank. Recent work from Gleich and Seshadhri proposes a heuristic based on triangle counting in the vertex neighborhood [8].

A simple local graph clustering technique called *semi-clustering* was discussed in [13]. Our approach is computed in a different way, is optimizing a different cluster quality metric, and has different theoretical guarantees.

Nibble: An Algorithm for Local Community Detection The Nibble algorithm was first sketched in [18] and more fully described in [19]. The algorithm finds a good cluster, of a specified size, near a given vertex. It runs nearly linearly in the size of the desired cluster, but is not guaranteed to succeed (i.e. such a set may not exist).

Nibble finds local clusters by computing an approximate distribution of a truncated random walk starting at the "seed" vertex. They extend the work of Lovász and Simonovits [12] to describe a set of conditions that find a low conductance cut based on these approximations quickly.

The PageRank-Nibble algorithm introduced by Andersen, Chung, Lang [2] improves upon this approach by using personal PageRank vectors to define nearness. They similarly extend the mixing result of Lovász and Simonovits to PageRank vectors.

Hadoop, MapReduce, and Giraph Hadoop is an open-source implementation of the Distributed File System and MapReduce programming paradigm introduced in [7]. It has been used to implement a variety of algorithms for large graphs. [15, 21]

Bulk Synchronous Parallel (BSP) processing is more flexible parallel computing framework than MapReduce. In BSP, a set of processors do computations. During these computations, the processors may send messages to other processors, either by name or as a broadcast. The computation will proceed until a barrier is reached in the algorithm. When the processor reaches a barrier, the system ensures that the processing will not continue until all the processors have reached the barrier. The system can then be seen as proceeding through a set of supersteps, marked by the barriers. Usually termination is done when all processors vote to halt at a barrier. If the virtual processors coincide with the nodes of a graph, we can perform many useful graph algorithms with BSP. This is the model used by the Pregel system [13] and later implemented in open source by the Apache Giraph project [4].

4 Algorithm

We use a three step process to compute a clustering of a graph. First, in PAPR, we compute many approximate personal PageRank vectors in parallel using the Pregel computing model. Next, in MRSweep, we perform a sweep of the vectors in parallel using Hadoop. These two algorithms together are a parallel version of the PageRank Nibble algorithm put forward in [2]. A critical difference of our *ParallelNibble* algorithm is that it produces local clusters which are overlapping. This prevents us from clustering graphs with a straightforward application of the Partition algorithm from [18]. The final step of our process transforms these overlapping local clusters into a non-overlapping clustering of the graph.

Parallel Approximate PageRank The computation of full PageRank has been intimately associated with the Pregel framework [14]. Here we present the approach for computing an approximate personal PageRank vector in Pregel, an outline of the proof of correctness showing that approximate personal PageRank vectors computed in this way still converge to personal PageRank vectors, and an analysis of the amount of work required to perform this computation.

Computing an approximate personal PageRank vector. We compute the approximate personal PageRank vector with a direct parallelization of the approach of Andersen, et al [2]. We start with a PageRank vector p of all zeros and a residual vector r initially set to $r = \chi_v$, i.e. the vector of all zeros except a 1 corresponding to the source vertex v.

The two inputs to this algorithm are $\{v\}$, the set of source nodes, and b, the log volume of the desired cluster. In the following proofs we follow the notation of [2] which uses the energy constant $\varepsilon \in [0,1]$, and the teleportation constant $\alpha \in [0,1]$. The energy constant ε controls the approximation level of the personal PageRank vector. As the size of the desired cluster grows, a finer approximation is necessary. In practice, we require $\varepsilon = O(\frac{1}{2^b})$ and typically initialize $\alpha = 0.10$.

Algorithm 1 PAPR(v,ε,α)

```
At each vertex $u$, for each superstep:

1. If this vertex has any messages $i$ from a neighbor pushing weight $w_i$ from the last step, set
   $r_u = r_u + w_i$
2. If $\frac{r_u}{d_u} > \varepsilon$ perform the push operation at this vertex.
3. If $\frac{r_u}{d_u} < \varepsilon$, vote to halt.

We define the push operation at a vertex to be:

1. set $p_u = p_u + \alpha r_u$
2. set $r_u = (1-\alpha) r_u / 2$
3. for each of my neighbors, send a message with weight $w = (1-\alpha) r_u / 2d_u$ attached.
```

In Pregel, each vertex has a corresponding processor and state. We realize the vectors p and r in our implementation by storing the scalar associated with vertex i in its processor. Along with p_i and r_i, we also store the values of the global parameters m, α, and ε. The algorithm for each vertex in a superstep is given in Algorithm 1.

Run this algorithm until all nodes vote to halt. We will show that PAPR halts and converges to an approximate PageRank vector; the number of push operations performed by PAPR is $O(\frac{1}{\varepsilon\alpha})$; and the complexity of PAPR is $O(\frac{1}{\varepsilon\alpha\omega})$ where ω is the number of workers. The basic ideas of these proofs follow in spirit along with proofs in [2], except where one step at at time is considered in those proofs, multiple steps may be performed in parallel by our algorithm. We can show that equivalent steps will be performed at each vertex as in the original algorithm up to a reordering, and therefore the same results hold due to the linearity of the functions involved.

Proof that PAPR Terminates and Converges to Approximate PageRank

Lemma 1 Let U be the set of nodes that experience the push operation in a superstep. After the push operation, our algorithm will produce the vectors:

$$p' = p + \sum_{u \in U} \alpha r(u) \chi_u \tag{1}$$

$$r' = r - \sum_{u \in U} \{ r(u) \chi_u + (1-\alpha) r(u) \chi_u W \} \tag{2}$$

Proof:
It is evident that p' is of the form described by the definition of the algorithm.
We can simplify the equation for r' to:

$$r - \sum_{u \in U} \left\{ r(u)\chi_u + \frac{1}{2}(1-\alpha) r(u) \left(\chi_u + \frac{\chi_u A}{d(u)} \right) \right\} \tag{3}$$

Using this simplification, we can compare components with what the algorithm will produce. If an element v of r corresponds to a vertex that is not in U or U's neighbors, then all the components in equation 3 besides the first are 0, so $r'(v) = r(v)$ like we expect. Otherwise, if v is not in U but is a neighbor of U, equation 3 has as the components of $r(v)$

$$r(v) - \sum_{u \in U} \left\{ \cancelto{0}{r(u)\chi_u(v)} + \frac{1}{2}(1-\alpha) r(u) \left(\cancelto{0}{\chi_u(v)} + \frac{\chi_u A(v)}{d(u)} \right) \right\}$$

$$r(v) - \sum_{(v,u) \in E} \left\{ \frac{1}{2}(1-\alpha) r(u) \frac{1}{d(u)} \right\}$$

Which is what you would expect. When v is in U, the χ_u factors cancel to 1 when $v = u$ so we get

$$\cancelto{0}{r(v) - r(v)} + \frac{r(v)(1-\alpha)}{2} + \sum_{(v,u) \in E} \frac{(1-\alpha) r(u)}{2d(u)}$$

which is also what we expect, proving the lemma.

Lemma 2 To show that PAPR converges to APR, we need to show that in PAPR as in APR, $p + pr(\alpha, r) = p' + pr(\alpha, r')$.

Using equation 5 in [2] and the linearity of the pr function,

$$\begin{aligned} p+pr(\alpha,r) =& p+pr(\alpha, r-\sum_{u\in U} r(u)\chi_u)+\sum_{u\in U} pr(\alpha, r(u)\chi_u) \\ =& p+pr(\alpha, r-\sum_{u\in U} r(u)\chi_u)+\sum_{u\in U} [\alpha r(u)\chi_u+(1-\alpha)pr(\alpha, r(u)\chi_u W)] \\ =& p'+pr(\alpha, r-\sum_{u\in U} [r(u)\chi_u)+(1-\alpha)r(u)\chi_u W]) \\ =& p'+pr(\alpha, r') \end{aligned}$$

Lemma 3 Let T be the total number of push operations performed by Parallel Approximate PageRank, S be the number of supersteps, ω be the number of workers and d_i be the degree of the vertex u used in the ith push. We would like to show that $\sum_{i=1}^{T} d_i \leq \frac{1}{\varepsilon\alpha}$

Proof: The proof follows as the proof in [2]. However, in PAPR many push operations are performed in each superstep. We can number the push operations using an index i, using the constraint that a push operation in an earlier superstep than another always has a lower index (numbering within a superstep is arbitrary). Since we use the same condition to choose vertices to perform the push operation on as in [2], each individual push operation on a vertex taken by itself still decreases $|r|_1$ by an amount greater than $\varepsilon\alpha d_i$. The result follows.

Complexity of PAPR. Consider S, the vector of super step lengths.

We partition the algorithm into $|S|$ super steps, such that $\sum_{S_i\in S} S_i = T$, i.e. S_i represents the number of pushes in step i. So then

$$\sum_{i=0}^{T} d_i = \sum_{S_j\in S}\sum_{i=0}^{S_j} d_i \implies \varepsilon\alpha\sum_{i=0}^{T} d_i = \sum_{S_j\in S}\varepsilon\alpha\sum_{i=0}^{S_j} d_i$$

Consider ω workers, each of which have been assigned $\frac{|Supp(p)|}{\omega}$ i.i.d. vertices for computation in parallel (i.e. the vertices with non-zero entries in p are divided uniformly among ω). We can then write the total amount of work in terms of the expected amount of work performed by each worker per superstep:

$$\varepsilon\alpha\sum_{i=0}^{T} d_i = \sum_{S_j\in S}\varepsilon\alpha\sum_{i=0}^{S_j} d_i = \sum_{S_j\in S}\varepsilon\alpha\omega\sum_{i=0}^{S_j/\omega} d_i$$

This implies $\varepsilon\alpha\omega\sum_{S_j\in S}\sum_{i=0}^{S_j/\omega} d_i = \varepsilon\alpha\sum_{i=0}^{T} d_i \leq 1$, as in the proof by [2], because $||r||_1 = 1$. This would them imply that the total running time satisfies the relationship $\sum_{S_j\in S}\sum_{i=0}^{S_j/\omega} d_i \leq \frac{1}{\varepsilon\alpha\omega}$. Therefore PAPR's complexity is $O(\frac{1}{\alpha\varepsilon\omega})$.

Computing multiple APRs simultaneously In the previous section, we showed that we can compute one APR from a starting vertex v using a parallel algorithm. To compute more APRs from a set of starting points S, we simply store a scalar pagerank entry p_j and residual entry r_j for each starting vertex $s_j \in S$, and initialize appropriately. We then modify the algorithm to compute each scalar quantity in turn for each starting vertex.

MapReduce Sweep In [2], each APR vector is converted into a good clustering using a *sweeping* technique. One orders the nodes in the graph using the corresponding probability value in the personal PageRank vector divided by the degree: $\frac{p_n}{d_n}$. If the PageRank vector has a support size equal to a number N_p, this creates an ordering on the nodes $n_1, n_2, \ldots, n_{N_p}$ and induces *sweep sets* $S_j^p = \{n_i | i \leq j\}$. A set with good conductance is found by finding the set with minimum conductance out of these sweep sets, but will output nothing if the set's conductance is greater than ϕ_{min}.

In the graphs that we are considering, we wish to compute many such good sets in parallel and also leverage the power of the MapReduce framework to aid in the algorithm computation. Between the Map and Reduce phases of MapReduce, the keys emitted by the Mapper are both partitioned into separate sets, and within each partition the keys are sorted according to some comparator. Keys present in a partition are guaranteed to be processed by the same Reducer, in sorted order.

In our MapReduce implementation of the Sweep algorithm, the Mapper will iterate over the vertices output by the Pregel APR algorithm. This output contains the probability value for each APR vector computed that affected that vertex, as well as the vertex's degree. We create keys emitted by the Mapper that are partitioned by the APR start vertex, and are sorted by the sweep metric $\frac{p_n}{d_n}$. Therefore after the Map phase the Reducers will receive all the probability and degree values for a single APR vector, sorted the correct way to produce the sweep sets. The Reducer then can compute the conductance for each APR's sweep sets and find the minimum conductance value. As an additional optimization, the data structure used to compute prior conductance values can be re-used to quickly compute the conductance value for the same set with an additional vertex. One simply stores the structures needed to compute conductance, such as the set of vertices adjacent to the cluster but not in the cluster, and updates them as new vertices arrive with their neighbor data.

4.1 Clustering Whole Graphs

The ParallelNibble procedure presented above provides a way of computing local clusters in parallel on distributed graphs. The ability to detect local communities is useful in a variety of real-world graph analysis tasks when one wants to know more about a source node (e.g. in a social network such as Twitter, one could model a node's local community affiliation and use it to determine interest in trending topics.)

To further explore the power of these local methods, we now consider the problem of generating a clustering for an entire graph. In order to do this we require an approach to generate good candidate tuples of source nodes and cluster sizes to build local clusters from, and a method for dealing with overlapping clusters.

Selection of source vertices and cluster sizes As with all local clustering methods, the selection of the starting vertices will make a significant difference in the final clustering. To generate tuples (v_i, b_i) as input for ParallelNibble we take inspiration from Spielman and Teng's RandomNibble [18]. Specifically, for each desired candidate nibble i, we randomly select a vertex v_i from the stationary distribution and a cluster size b_i in the range $\frac{\lceil \log m \rceil}{2}, \ldots, \lceil \log m \rceil$ according to $\Pr[b = i] = 2^{-i}/(1 - 2^{-\lceil \log m \rceil})$.

This distribution for b is a truncated version of RandomNibble's; it focuses on finding larger clusters instead of smaller ones. Choosing b this way makes sense for performing a coarse clustering of G, but it does have a disadvantage - this approach will be unable to detect small clusters. A remedy for this is to recursively apply the same procedure to the generated clusters.

Postprocessing overlapping clusters Once we have computed the local clusters for all the source vertices, we wish to convert them into a good global clustering. There are a variety of ways that these overlapping local clusters could be combined. We choose a simple method put forward by Andersen, et. al. which has the desirable property of preserving the minimum conductance of the final clustering. The method amounts to resolving conflicts in local cluster membership by always assigning a vertex to its cluster with the least conductance. It is accomplished by the following procedure.

First, we sort the generated clusters by their conductance. Then we iterate through the clusters, adding them to our final clustering. As we add each cluster the final clustering, we mark all of the vertices in it as 'used'. Clusters with higher (worse) conductances can not use these vertices again.

This is clearly not optimal for maximizing the modularity of the clustering, but provides a straightforward approach for dealing with a complicated problem.

5 Experimental Results

Here we present results obtained from running our algorithm on real graphs. We focus on two types of metrics: the quality of the clustering in terms of conductance or modularity, and the algorithm scalability measured by the running time vs. number of worker processes.

Test Environment We used two different environments, a 12 machine cluster for the quality of clustering tests, and a 32 machine cluster for the scalability test. Each machine has 96 GB of RAM, and 2 Intel Xeon processors. The cluster was using Apache Hadoop 0.20.203 on CentOS 6. All experiments were written in Scala, using Apache Giraph 0.2.

Evaluation of Clustering In order to evaluate the quality of the clustering found by our algorithm, we have benchmarked it against a variety of real world graphs found from the SNAP Network Datasets[1]. We compare our results against *Louvain Fast Unfolding* [6] a popular modularity optimization algorithm that performs a local search followed by a cluster contraction phase which repeats until it finds a

[1] Available: `http://snap.stanford.edu/data/index.html`

maximum modularity. Fast Unfolding is not optimal, but it is quite fast and has been shown to achieve excellent results on a variety of real world graphs.

We emphasize that Fast Unfolding is an algorithm optimizing a *global* criteria (modularity) using local changes, while Nibble and its derivatives are completely local algorithms optimizing a local criteria (the conductance of *local cuts*). In some cases, Fast Unfolding is barely able to run and must be supplied with multiple gigabytes of memory. We compare the results of our clustering process against the first phase of Fast Unfolding before cluster contraction is applied. We refer to this as the *Baseline modularity*. The conductance of this clustering we use as our *baseline conductance*

For all these tests, we used the number of Giraph workers $\omega = 10$, the teleportation constant $\alpha = 0.10$, and the minimum acceptable conductance of MRSweep $\phi_{min} = 0.15$.

Table 1 shows that PAPR is able to find low conductance clusters, but that the complete clustering performs worse than the baseline modularity. Potential ways to improve performance include taking more samples, changing the way sources are selected, or calculating a more precise PageRank vector.

Graph Name	$\|V\|$	$\|E\|$	ϕ	Baseline ϕ	q	Baseline q
soc-livejournal	4,847,571	68,993,773	0.0376	0.1764	0.488	0.665527
web-google	875,713	5,105,039	0.027	0.015	0.689	0.76056
web-stanford	281,903	2,312,497	0.017	0.001	0.584	0.815849
amazon0302	262,111	1,234,877	0.109	0.03846	0.617	0.637707
ca-AstroPh	18,772	396,160	0.1507	0.0666	0.244	0.54332
ca-GrQc	5,242	28,980	0.106	0.004	0.538	0.708325
ca-HepPh	12,008	237,010	0.120	0.002	0.473	0.587588
email-Enron	36,692	367,662	0.066	0.15	0.384	0.557363
loc-gowalla	196,591	950,327	0.107	0.1515	0.459	0.639371
oregon1-010526	10,670	22,002	0.145	0.1594	0.438	0.458858
soc-Epinions1	75,879	508,837	0.164	0.2	0.271	0.405964
web-Stanford	281,903	2,312,497	0.022	0.001	0.54	0.81631
wiki-Vote	7,115	103,689	0.178	0.111	0.295	0.42207

Table 1 The quality of clusterings produced by our method on some publicly available graph datasets. We have ignored clusters with $\phi = 0$ (this indicates a disconnected cluster was discovered). Baseline modularity (q) and baseline ϕ are derived from a clustering made with the Fast Unfolding method [6]

PAPR Scalability To verify the scalability of our local clustering approach we computed a fixed number of clusters on the biggest graph we considered, *soc-livejournal*, and varied the number of workers, ω, available to the Apache Giraph job. Source vertices were selected randomly, but the cluster size was fixed. Other parameters are as used earlier. The total time to run the algorithm includes the time for Giraph to load the graph, and the time to run PAPR. We present the total time and the PAPR running time in Figure 1.

As expected, increasing the number of workers decreases the running time. After a certain number of workers (ω=10 in this case) the synchronization and communication costs begin to dominate the computation, and there is no benefit from additional parallelization.

One of the prime contributors to this communication overhead comes from the difficulty of partitioning graphs which follow a power-law distribution. When faced with such a graph Pregel randomly assigns the vertices to workers. This results in most of the edges running between different workers and requires network overhead for messages passed over these edges. Recent work by Gonzalez et. al. [9] presents the problem in detail and provides a computational approach using vertex-cuts instead of edge-cuts which allows for much greater parallelization.

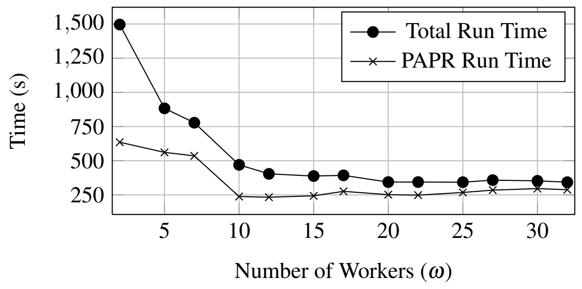

Fig. 1 Effects of increasing the number of workers on running time. Initially, adding workers has a big effect, however the synchronization and communication overhead limits the contributions of workers past a certain point (here, at $\omega = 10$).

6 Conclusions and Future Work

We have shown that a parallel technique can be used to create Approximate PageRank vectors and turn those vectors into local clusters using the parallel processing techniques of Pregel and MapReduce. We have shown that the calculation of these vectors is highly parallelizable and results in significant time savings as workers are added. This time and memory parallelization allows the use of these local spectral clustering techniques on larger graphs than would traditionally be possible by simply adding more commodity hardware to the analysis system.

Recent work on community detection [1, 20] shows that allowing communities to overlap better captures the behavior observed in in real world networks. Methods based on local clustering have already been used to analyze the profile of network communities at different size scales [11], and there is reason to believe that these techniques can aide in other aspects of the analysis of large graphs. We plan to perform more scalability analysis of the technique using more hardware and "Internet scale" graphs reaching into the billions of nodes, where traditional methods have serious difficulty providing meaningful results.

References

1. Y.Y. Ahn, J.P. Bagrow, and S. Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, 2010.
2. R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 475–486. IEEE, 2006.
3. Reid Andersen and Yuval Peres. Finding sparse cuts locally using evolving sets. *CoRR*, abs/0811.3779, 2008.
4. Apache giraph, February 2012. http://incubator.apache.org/giraph/.
5. 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*, SIGMOD '11, pages 973–984, New York, NY, USA, 2011. ACM.
6. V.D. Blondel, J.L. Guillaume, R. Lambiotte, and E.L.J.S. Mech. Fast unfolding of communities in large networks. *J. Stat. Mech*, page P10008, 2008.
7. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
8. David F. Gleich and C. Seshadhri. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *KDD*, pages 597–605, 2012.
9. Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.
10. Ravi Kannan, Santosh Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, May 2004.
11. Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 695–704, New York, NY, USA, 2008. ACM.
12. L. Lovász and M. Simonovits. Random walks in a convex body and an improved volume algorithm. *Random Structures & Algorithms*, 4(4):359–412, 1993.
13. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.
14. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.
15. Christopher McCubbin, Bryan Perozzi, Andrew Levine, and Abdul Rahman. Finding the 'needle': Locating interesting nodes using the k-shortest paths algorithm in mapreduce. *2011 IEEE International Conference on Data Mining Workshops*, 0:180–187, 2011.
16. M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.
17. Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.
18. D.A. Spielman and S.H. 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. ACM, 2004.
19. Daniel A. Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *CoRR*, abs/0809.3232, 2008.
20. Jaewon Yang and Jure Leskovec. Structure and overlaps of communities in networks. *CoRR*, abs/1205.6228, 2012.
21. Z. Zhao, G. Wang, A.R. Butt, M. Khan, VS Kumar, and M.V. Marathe. Sahad: Subgraph analysis in massive networks using hadoop. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 390–401. IEEE, 2012.