2016 IEEE International Conference on Big Data (Big Data)

Massive Graph Processing on Nanocomputers

Bryan Rainey and David F. Gleich
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907–2107
Email: {raineyb,dgleich}@purdue.edu

Abstract—**Many recently proposed graph-processing frameworks utilize powerful computer clusters with dozens of cores to process massive graphs. Their usability and flexibility often come at a cost. We demonstrate that custom software written for "nanocomputers," including a credit-card-sized Raspberry Pi, a low-cost ARM server, and an Intel Atom computer, can process the same graphs. Our implementations of PageRank and connected components stream graphs from external storage while performing computation in the limited main memory on these nanocomputers. The results show that a $100 computer with an Intel Atom core can compute PageRank and connected components on a 1.5-billion-edge Twitter graph as quickly as graph-processing systems running on machines with up to 48 cores. As people continue to apply graph computations to large datasets, this research suggests that there may be cost and energy advantages to using nanocomputers.**

I. Introduction

Recent developments in graph-processing frameworks allow researchers to perform graph computations on large datasets such as social networks, web crawls, and science-derived networks. These systems offer usability and flexibility while executing iterative graph-processing tasks like PageRank, connected components, shortest paths, and community detection. Graph-processing systems tend to run on powerful hardware with dozens of processors, and they typically scale to graphs with millions of nodes and billions of edges [1]–[11].

An entirely different trend in computing is the production of so-called "nanocomputers" like the Raspberry Pi, a $35 computer the size of a credit card. Nanocomputers tend to be budget friendly, and they draw significantly less power than larger computers. Recent nanocomputers, such as the $100, 2-by-3-inch Kangaroo with an Intel Atom core, are effective as general-purpose personal computers. Our paper begins with the following question: can we perform large-scale graph computations on nanocomputers?

In our experiments, we successfully process billion-edge graphs on the Raspberry Pi, Scaleway – an ARM-based cloud server available for €3 per month – and the Kangaroo. We achieve running times comparable to several graph-processing systems running on as many as 48 cores, which suggests that the flexibility of such systems comes with a substantial reduction in computational efficiency. In order to benchmark the nanocomputers, we wrote custom implementations of two standard graph-processing tasks: computing PageRank using the power iteration and computing connected components using label propagation. These two graph algorithms are representative of several iterative graph procedures that stream through the edges in a graph while performing computation [12]. Due to main memory constraints on nanocomputers, we stream graphs from storage while performing computation. We take advantage of the fact that all three of our nanocomputers have 4-core processors by partitioning graphs into blocks and then dividing the blocks among multiple threads of execution. Our experiments culminate in a performance model that predicts the running times of graph computations on nanocomputers within a factor of two based on input/output bandwidth, memory bandwidth, and memory latency. Our software is available online in the interest of reproducibility:

https://github.com/brainey421/badjgraph/

Like graph-processing platforms with dozens of cores, our implementations of graph computations on nanocomputers scale to social networks and web crawls with millions of nodes and billions of edges. One of the reasons that nanocomputers can process massive graphs is that many graph algorithms, such as PageRank and connected components, scale linearly with the number of edges. With linear algorithms, the computational bottleneck is often simply reading the graph. The authors of another graph-processing system, GraphChi, made a similar observation in the context of microcomputers [7]. Nanocomputers offer input/output systems that are nearly as fast as those of much larger computers, which leads to reasonable edge-streaming performance on nanocomputers. These results suggest that innovations in hardware may be outpacing increases in the sizes of graphs that researchers are analyzing.

In practice, we see potential advantages of using nanocomputers to support low-throughput but highly parallel workloads. Many real-world graph analysis scenarios include a battery of highly-related analysis routines on a given graph. For example, a variety of clustering and classification tasks rely on computing PageRank with multiple values of teleportation coefficients and teleportation vectors [13], [14]. In this kind of scenario, we can export a large graph from a production system, preprocess it lightly, and then analyze the graph on nanocomputers in several ways in parallel.

II. Related Work

Many of the recently proposed graph-processing frameworks are designed for flexibility first and scalability second. This makes it easy to implement a variety of graph algorithms

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

and quickly explore novel insights about graphs [1]–[11]. However, the flexibility and scalability of these systems often come at a cost. McSherry, Isard, and Murray proposed a simple metric to evaluate the performance of graph-processing systems simply called COST, the Configuration that Outperforms a Single Thread [15]. The authors implemented standard iterative graph algorithms on a 2014 laptop using a single thread of execution. They demonstrated that the COST of many graph-processing frameworks is often hundreds of cores, if not unbounded – that is, in some cases, no configuration of a graph-processing system can outperform one thread on a standard laptop. Our work follows in this same line and shows that even readily accessible nanocomputers have performance that is comparable to, if not better than, many of these systems.

In response to the limitations of scalability on many graph-processing systems, there is a growing number of in-memory frameworks, including GraphChi [7], Ringo [8], EmptyHeaded [9], Galois [10], and Ligra [11]. These systems attempt to bring large-scale computation to single-node, shared-memory architectures. The GraphChi system most closely resembles our work. We share a similar goal: demonstrating that large-scale graph computation is feasible and efficient on small-scale hardware. The GraphChi system illustrated that large-scale graph computation is possible on a Mac Mini computer while retaining some of the flexibility of more highly scalable systems. In our research, we show that simple implementations of graph algorithms on nanocomputers perform comparably to the GraphChi system.

III. Background

We briefly review the two graph algorithms that we implement: the power iteration algorithm to compute PageRank [16] and a label propagation algorithm to compute connected components [17]. PageRank and connected components are useful in benchmarking graph-processing systems because they follow a common paradigm: maintain a piece of data for each node in the graph, and during an iteration, update each node's data based on a function of its neighbors' data. This strategy of reading through all of the edges in a graph and propagating information along those edges is common to many standard graph algorithms from finding shortest paths to detecting communities [12], [17], [18]. Not all graph algorithms fit this paradigm, namely algorithms that involve only a specific region of a graph or algorithms that require random access to the edges in a graph. However, there are often approximation algorithms for these tasks that do fit this paradigm, such as approximate triangle counting [19]. Observing how a graph-processing system performs while computing PageRank and connected components offers broad insight into the bottlenecks of the system and its performance relative to other systems.

A. PageRank

A common construction of the PageRank problem for a graph $G = (V, E)$ with n nodes is a linear system [20]. Let $\mathbf{A}$ be the adjacency matrix of G, where $A_{uv} = 1$ if and only if $(u, v) \in E$. Let $\mathbf{d}$ be the vector whose entries are the out-degrees of the nodes in V. Let $\alpha \in (0, 1)$ be PageRank's teleportation parameter, commonly set to 0.85. The PageRank linear system solves for the PageRank vector $\mathbf{x}$:

$$(\mathbf{I} - \alpha\mathbf{P})\mathbf{x} = (1 - \alpha)\mathbf{e}/n, \quad (1)$$

where $\mathbf{I}$ is the identity matrix, $P_{uv} = A_{vu}/d_v$, and $\mathbf{e}$ is the vector of all ones. The data in the PageRank vector $\mathbf{x}$ provides a measure of importance for each node in V based purely on the edges in E.

```
1: x^(1) ← e/n
2: for k ← 1 to maxiter do
3:   x^(k+1) = 0
4:   for all nodes u with out-degree δ do
5:     for all edges (u, v) do
6:       x_v^(k+1) ← x_v^(k+1) + α x_u^(k) / δ
7:     end for
8:   end for
9:   ρ ← 1 − ‖x^(k+1)‖_1
10:  x^(k+1) ← x^(k+1) + (ρ/n)e
11:  Check convergence based on ‖x^(k+1) − x^(k)‖_1
12: end for
```

Fig. 1. The power iteration algorithm to compute PageRank [16]. The three parameters are $\alpha \in (0, 1)$, commonly set to 0.85; maxiter, the maximum number of iterations; and ε, the convergence tolerance. During an iteration, the algorithm updates the vector $\mathbf{x}^{(k+1)}$ by propagating the data in $\mathbf{x}^{(k)}$ along every edge in E. The algorithm tests for convergence using the residual one-norm, which for PageRank is equivalent to $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_1$. We detail the parallelization in section IV-E.

The power iteration algorithm in figure 1 computes the PageRank vector. Line 1 initializes the PageRank scores in $\mathbf{x}^{(1)}$ to be uniform. During the kth iteration, the algorithm updates $\mathbf{x}^{(k+1)}$ by propagating the data in $\mathbf{x}^{(k)}$ along every edge in E. The algorithm in figure 1 is written to emphasize that every iteration cycles through the edges in E and propagates data along each edge, but lines 3–10 could be abbreviated:

$$\mathbf{x}^{(k+1)} \leftarrow \alpha\mathbf{P}\mathbf{x}^{(k)} + (1 - \alpha)\mathbf{e}/n. \quad (2)$$

After an iteration, the algorithm computes the residual one-norm $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_1 = \sum_{u=1}^{n} |x_u^{(k+1)} - x_u^{(k)}|$ to gauge convergence.

B. Connected Components

A connected component in G is a subgraph in which every pair of nodes is connected by some path of edges in E. The connected components that we consider are weakly connected, so the directions of the edges in E are not important.

The algorithm that we use to find the connected components in a graph G is similar to the power iteration algorithm that computes PageRank in that it also performs a linear pass over the edges in the graph. The label propagation algorithm in figure 2 begins by assigning every node a unique label in the vector $\mathbf{x}$. During the kth iteration, the algorithm updates $\mathbf{x}$ by visiting every edge in E whose endpoints have distinct labels and then propagating the minimum label to both

```
1:  for all nodes u do
2:      x_u ← u
3:  end for
4:  for k ← 1 to maxiter do
5:      counter ← 0
6:      for all edges (u, v) such that x_u ≠ x_v do
7:          x_u, x_v ← min(x_u, x_v)
8:          Increment counter
9:      end for
10:     if counter = 0 then
11:         break
12:     end if
13: end for
```

Fig. 2. A label propagation algorithm to compute connected components [17]. The only parameter is maxiter, the maximum number of iterations. During an iteration, the algorithm updates **x** by propagating the minimum label along every edge in E. The algorithm tests for convergence by counting the number of label propagations per iteration and terminating when there are none. We detail the parallelization in section IV-E.

nodes. Meanwhile, it keeps track of how many propagations occur during an iteration to gauge convergence. The algorithm converges when all adjacent nodes share the same label, and at that point, all nodes in a connected component will share the same label unique to that component.

IV. Methods

A. Hardware

Unlike typical graph-processing systems, we implement graph algorithms on small, low-power computers, commonly called "nanocomputers." Their specifications are summarized in table I. Both the Raspberry Pi and Scaleway use ARM processors, and the Kangaroo comes with a low-power Intel Atom processor. All of these processors have 4 cores. The primary constraints on these machines are input/output capacity, main memory, and cache memory. Because the datasets in our experiments are too large to fit in main memory, we rely on an external hard disk drive for the Raspberry Pi, a remote solid-state drive for Scaleway, and a local eMMC solid-state drive for the Kangaroo. Therefore, it is possible that the bottleneck of certain graph computations on nanocomputers is not the speed of the processor but the speed at which the computer can stream graphs from storage.

B. Graph Format

We store graphs in a binary format that mirrors the structure of an adjacency list. It contains a simple header with the number of nodes and edges. The remainder of the graph is a list of nodes, where each node is specified by its out-degree and a list of adjacent nodes, all 4-byte integers. This simple format suffices because all of the graphs in our experiments have fewer than 4 billion nodes. The format makes it straightforward to implement many iterative graph algorithms that require the degree of a node for each update.

TABLE I
HARDWARE SPECIFICATIONS FOR THE THREE SMALL, LOW-POWER COMPUTERS IN OUR EXPERIMENTS, COMMONLY REFERRED TO AS "NANOCOMPUTERS." THE PRIMARY LIMITATIONS ON THESE COMPUTERS ARE I/O CAPACITY, MAIN MEMORY, AND CACHE MEMORY.

Raspberry Pi 2	Scaleway Server	Kangaroo
$35	€2.99/month	$99.99
4-core ARMv7	4-core ARMv7	4-core Intel Atom
1GB RAM	2GB RAM	2GB RAM
2TB HDD ($89.99)	50GB SSD	32GB SSD

We chose this simple format after attempting to use data compressed using the WebGraph framework [21]. Our benchmarks showed that WebGraph's compression schemes were too slow to offset the decrease in I/O bandwidth. In order to test the cost of decompression, we ran a single-threaded implementation of the power iteration algorithm to compute PageRank on both WebGraph's compressed graphs and our decompressed graphs. We began by running the algorithm on a 200-million-edge graph of Hollywood actors on the two ARM nanocomputers. One iteration of PageRank on the Raspberry Pi took 58 seconds on the compressed graph, compared to 33 seconds on the decompressed graph. Scaleway saw a similar speedup from 39 seconds to 11 seconds. We also ran an iteration of the algorithm on a 1.5-billion-edge Twitter graph on the Kangaroo, which took 355 seconds on the compressed graph and 76 seconds on the decompressed graph.

To facilitate parallel computation, we split the graph file into several blocks of maximum size 16MB, large enough to accommodate the node with the greatest out-degree in our experiments, which is 3 million. We ensure that the block partitions do not split up a particular node's information about its out-degree and neighbors. For example, splitting a graph into, say, 100 blocks would allow 4 threads to perform simultaneous computation on 25 blocks each.

C. Datasets

Most of the datasets in our experiments come from the Laboratory for Web Algorithmics at the University of Milan. They provide compressed social networks and web crawls by utilizing compression techniques from the WebGraph framework [21] and by utilizing a label propagation clustering tool that permutes the nodes in a graph to improve locality [22]. For the web crawls, they group websites by host to improve locality and compression without using the clustering tool [23].

While we take advantage of WebGraph's reordered datasets, we convert them into the decompressed binary format described above, which eliminates the cost of decompression on low-power processors. Overall, the preprocessing in our experiments consists solely of converting the data into the binary format, which is fast even on nanocomputers. For instance, the Kangaroo can convert the 1.5-billion-edge Twitter graph into the binary format in roughly 5 minutes. As mentioned in the introduction, many of the graphs that researchers analyze arise from dumps from a production system, and analysts often run

algorithms like PageRank or connected components multiple times on graphs with slightly different parameters or subsets of edges. Therefore, a small amount of preprocessing on these graphs is a minimal cost.

D. Implementation

Our custom software, written in C and available at the link in section I, is designed to process graphs quickly on nanocomputers. Both algorithms that we implement maintain vectors of length n in main memory. The PageRank algorithm in figure 1 maintains a sequence of vectors $\{\mathbf{x}^{(k)}\}$, but our implementation maintains only two vectors during an iteration: the current vector $\mathbf{x}^{(k)}$ and the next vector $\mathbf{x}^{(k+1)}$. The connected components algorithm in figure 2 maintains only one vector $\mathbf{x}$. Each entry in these vectors consumes 4 bytes of memory: for the PageRank algorithm, the entries are 4-byte floating-point numbers, and for the connected components algorithm, the entries are 4-byte unsigned integers. In our experiments, we restrict ourselves to graphs whose vectors fit in main memory on the nanocomputers, which still allows us to analyze massive web crawls and social networks like Twitter and Friendster.

E. Parallelization

The Raspberry Pi, Scaleway, and the Kangaroo all have 4-core processors, which we utilize to parallelize computation among multiple threads of execution. The graph-processing tasks that we consider are algorithms that propagate information along every edge in a graph, and the order of the edges is not important. Therefore, we can parallelize the sections of these algorithms that loop through the edges in a graph. The parallelizable loops of the algorithms are lines 4–8 in figure 1, and lines 6–9 in figure 2. We utilize the OpenMP API to divide the iterations of these loops among any number of threads [24].

In order to ensure that no two threads simultaneously update an entry in a vector, we can make the update steps atomic using OpenMP's built-in primitive. However, recent research has considered eliminating thread safety to decrease the running times of certain algorithms. For example, HOGWILD! is a framework for parallelizing stochastic gradient descent without using locks [25]. Although HOGWILD! still assumes that its updates are atomic, Avron, Druinsky, and Gupta explored eliminating atomic updates in their linear system solver for symmetric, positive-definite matrices [26]. Avron et al. observed only a slightly worse convergence rate without atomicity and no significant difference in running time. Since we do observe better running times after removing atomicity, we argue below that eliminating atomic updates only marginally affects the convergence of our PageRank algorithm, and that atomic updates are not necessary for our connected components algorithm.

For our PageRank algorithm, the potential problem with eliminating atomicity is that simultaneous updates to $x_v^{(k+1)}$ on line 6 of figure 1 can collide, which leads to skipped updates to the vector $\mathbf{x}^{(k+1)}$. Skipping updates to the PageRank vector is not a new idea, and researchers have noted the benefits of skipping updates to areas of the vector that have already converged [27], [28]. Because our algorithm in figure 1 does not access entries in $\mathbf{x}^{(k+1)}$ sequentially – rather, it accesses entries in $\mathbf{x}^{(k)}$ sequentially – we expect relatively few update collisions during an iteration. Also, we expect that within a short number of iterations, all of the updates propagate at least once. We verified these assumptions in some initial microstudies of our code.

What is more problematic with eliminating atomic updates from PageRank is gauging convergence. Using non-atomic updates precludes us from utilizing a stopping criteria based on the change in the PageRank vector during each iteration, a convenient way to evaluate the residual. In order to guarantee a specific solution error, it would be necessary to perform a final check on the atomically computed residual, which is proportional to the error for PageRank [20]. As performing this check is uncommon, in this paper, we just run the algorithm for a fixed number of iterations to compare our system with other implementations of PageRank.

For our connected components algorithm, atomicity is not an issue at all. The following argument demonstrates that eliminating atomicity does not impede convergence. As with PageRank, the potential problem is skipping updates to the vector $\mathbf{x}$ on line 7 of figure 2. Consider the case when a thread of execution skips an update to some entry x_u. At that time, a second thread must be successfully updating x_u with another value, so x_u is guaranteed to have a new, smaller value after the collision. Thus, during a given iteration, every entry of $\mathbf{x}$ that requires an update will inherit a smaller value after that iteration, although it might not be the smallest possible value. Under the assumptions above, this property guarantees that after enough iterations, all of the entries in $\mathbf{x}$ will reach their minimum values, so the algorithm will still converge. In this case, each thread should maintain independent copies of the `counter` variable to merge upon conclusion of the parallel loop over edges. Because atomicity is unnecessary for convergence, we never use atomic updates in our connected components experiments.

V. Experiments

A. Datasets

Table II summarizes the details of the directed graphs that we use in our experiments based on datasets from the Labaratory for Web Algorithmics at the University of Milan [21]–[23]. The Hollywood graph is a network of actors and actresses from 2011. The two web crawls that we analyze are the UK and SK graphs from 2005. The 1.5-billion-edge Twitter graph is a social network from 2010, and the undirected 3.6-billion-edge Friendster graph is the largest network we analyze [29]. We include experiments on both the Twitter graph and its transpose to explore the difference between graphs with high out-degrees – such as the Twitter graph that models the flow of information on social media – and graphs with high in-degrees – such as the transposed Twitter graph that models followers on social media.

TABLE II
THE GRAPHS IN OUR EXPERIMENTS AND THEIR SIZES, INCLUDING THEIR FILE SIZES IN OUR BINARY FORMAT [21]–[23], [29].

Graph	Nodes	Edges	Size
hollywood-2011	2,180,759	228,985,632	0.8612GB
uk-2005	39,459,925	936,364,282	3.635GB
twitter-2010	41,652,230	1,468,365,182	5.625GB
sk-2005	50,636,154	1,949,412,601	7.451GB
com-friendster	65,608,366	3,612,134,270	13.70GB

TABLE III
THE SPEED AT WHICH EACH NANOCOMPUTER STREAMS GRAPHS FROM STORAGE. THE RASPBERRY PI STREAMS GRAPHS AT 23 MBPS, SCALEWAY AT 110 MBPS, AND THE KANGAROO AT 150 MBPS.

Graph	Raspberry Pi	Scaleway	Kangaroo
uk-2005	164 s	39 s	25 s
twitter-2010	244 s	53 s	38 s
sk-2005	324 s	74 s	50 s
com-friendster	600 s	127 s	93 s

B. Graph Streaming

In order to understand the baseline performance of nanocomputers on massive graph computations, we begin by measuring the speed at which these computers can stream all of the edges in a graph from storage. The Raspberry Pi streams graphs from an external hard disk drive connected via USB2.0; Scaleway remotely accesses a solid-state drive via 1GbE; and the Kangaroo has a local eMMC solid-state drive. The results in table III show that the Raspberry Pi streams graphs at an average speed of 23 MBps, despite the fact that USB2.0 theoretically supports 60 MBps – we suspect the half-duplex nature of the USB2.0 protocol was responsible for most of the performance decrease. Scaleway streams graphs from its SSD at 110 MBps, which nearly reaches the limit of its 1GbE connection, and the Kangaroo has the fastest streaming speed at 150 MBps. Both graph algorithms that we consider invoke this graph-streaming procedure, so these measurements form a baseline for the performance of the graph computations in the subsequent sections. For example, 20 iterations of PageRank on the Twitter graph must take at least 4880 seconds on the Raspberry Pi, compared to 760 seconds on the Kangaroo.

C. PageRank With Atomic Updates

The graph-processing tasks that we implement are the power iteration algorithm in figure 1 to compute PageRank and the label propagation algorithm in figure 2 to compute connected components. For all of these experiments, we use 8 threads to take advantage of the 4 cores on each of the nanocomputers and to ensure that every thread is either streaming the graph or performing computation. Our results show that nanocomputers can compute PageRank and connected components on billion-edge graphs in a matter of minutes.

The first implementation that we consider is the thread-safe power iteration algorithm, which uses atomic updates to protect the PageRank vector $\mathbf{x}$, so the algorithm is guaranteed to converge even with multiple threads. We choose the most common value for the parameter $\alpha = 0.85$. Table IV lists the running times for 20 iterations of PageRank, and figure 3 shows the scalability of the implementation. We report running times for both the Twitter graph and the transposed Twitter graph. Although the two graphs are the same size, PageRank runs more quickly on the transposed graph, which suggests that the running time of PageRank does not depend only on the size of a graph. The Raspberry Pi computes PageRank roughly at the rate at which the computer streams graphs from storage, or around 10% slower, which suggests that the Raspberry Pi is bottlenecked by input/output bandwidth. We explore the bottlenecks on all three computers in the performance model in section V-E.

D. Graph Algorithms Without Atomic Updates

In section IV-E, we justified anecdotally that eliminating atomic updates from PageRank computation only marginally affects convergence. Table V shows the running times of 20 iterations of PageRank with non-atomic updates, and figure 4 shows how the implementation scales with graph size. In most cases, eliminating atomicity decreases the running time of PageRank. However, using non-atomic updates rather than atomic updates does not affect the speed at which the computers read graphs from storage, so we would expect that eliminating atomicity is more effective at increasing computation speed only for experiments that are not already bottlenecked on reading graphs from storage. Indeed, the most drastic reductions in running time from table IV to table V are on Scaleway and the Kangaroo. As with the thread-safe PageRank computation, the Raspberry Pi is bottlenecked on input/output, which we discuss further in section V-E.

We demonstrate empirically that removing atomicity from the updates to the PageRank vector only marginally affects the convergence of the algorithm. Figure 5 shows the relative difference in the errors of the thread-safe and thread-unsafe implementations of PageRank for each graph. After 20 iterations, the transposed Twitter graph has the largest increase in error after eliminating atomicity at roughly 4 percent. To put these results into context, the errors of the two implementations are of the same order of magnitude and are identical in the first few significant digits. One possible explanation of why the difference abruptly increases during the last few iterations is that the nodes whose PageRank values have not yet converged are the same nodes that are likely to experience update collisions after eliminating atomicity. Regardless, the fact that the errors of the two implementations are nearly the same suggests that eliminating atomicity does not strongly impact the convergence of the power iteration.

In addition to PageRank, we implement the label propagation algorithm to compute connected components. As described in section IV-E, eliminating atomicity from the algorithm does not affect its convergence. We run as many iterations as necessary for the algorithm to converge. The running times for connected components are listed in table VI and visualized in figure 6. As with PageRank, the Twitter graph,

TABLE IV
RUNNING TIMES OF THE THREAD-SAFE IMPLEMENTATION OF 20 POWER ITERATIONS TO COMPUTE PAGERANK, INCLUDING THE PERCENT SLOWDOWN COMPARED TO THE GRAPH-STREAMING TIME.

Graph	Raspberry Pi		Scaleway		Kangaroo	
	Time	Slowdown	Time	Slowdown	Time	Slowdown
hollywood-2011	779 s	2.5%	90 s	310%	74 s	490%
uk-2005	3353 s	2.2%	775 s	-0.64%	512 s	2.4%
twitter-2010	5518 s	13%	2081 s	96%	1949 s	160%
twitter-2010-t*	5472 s	12%	1897 s	79%	1525 s	100%
sk-2005	6499 s	0.29%	1436 s	-3.0%	1017 s	1.7%
com-friendster	14318 s	19%	6837 s	170%	6632 s	260%

*We process both the Twitter graph and the transposed Twitter graph.

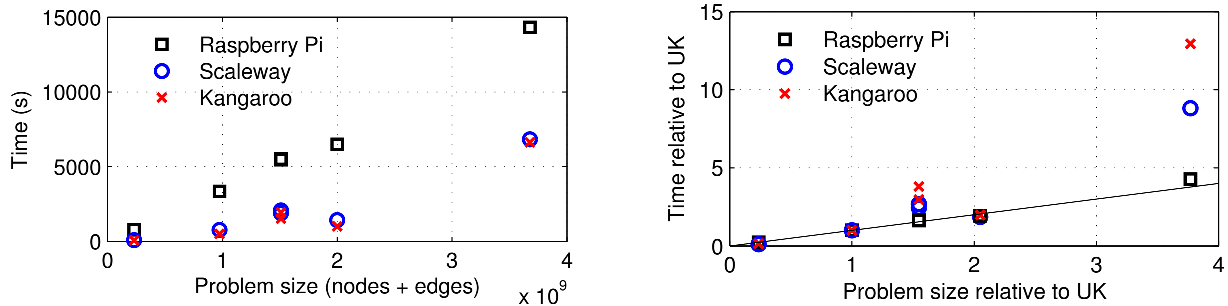

Fig. 3. The scalability of the thread-safe implementation of 20 power iterations to compute PageRank. The plot on the left shows the running times in seconds, and plot on the right shows the running times relative to the UK graph.

TABLE V
RUNNING TIMES OF THE THREAD-UNSAFE IMPLEMENTATION OF 20 POWER ITERATIONS TO COMPUTE PAGERANK, INCLUDING THE PERCENT SLOWDOWN COMPARED TO THE GRAPH-STREAMING TIME.

Graph	Raspberry Pi		Scaleway		Kangaroo	
	Time	Slowdown	Time	Slowdown	Time	Slowdown
hollywood-2011	778 s	2.4%	63 s	190%	39 s	210%
uk-2005	3350 s	2.1%	707 s	9.4%	499 s	2.4%
twitter-2010	5516 s	13%	2008 s	89%	1391 s	83%
twitter-2010-t	5323 s	9.1%	1772 s	67%	1213 s	60%
sk-2005	6491 s	0.17%	1434 s	-3.1%	1010 s	1.0%
com-friendster	14425 s	20%	6699 s	160%	4086 s	120%

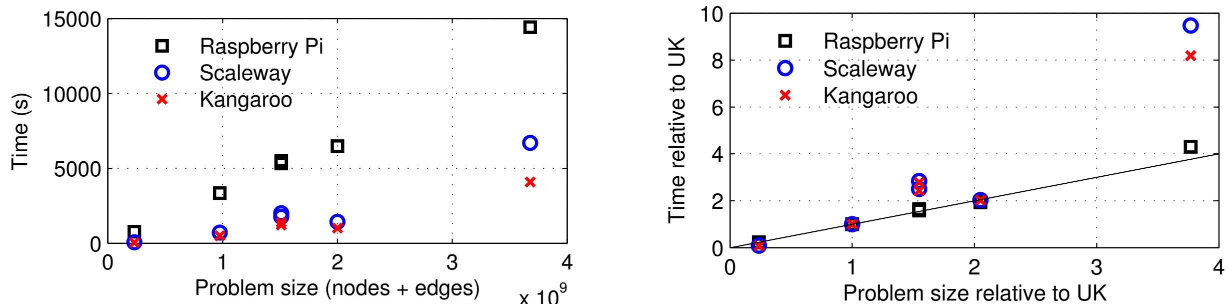

Fig. 4. The scalability of the thread-unsafe implementation of 20 power iterations to compute PageRank. The plot on the left shows the running times in seconds, and plot on the right shows the running times relative to the UK graph.

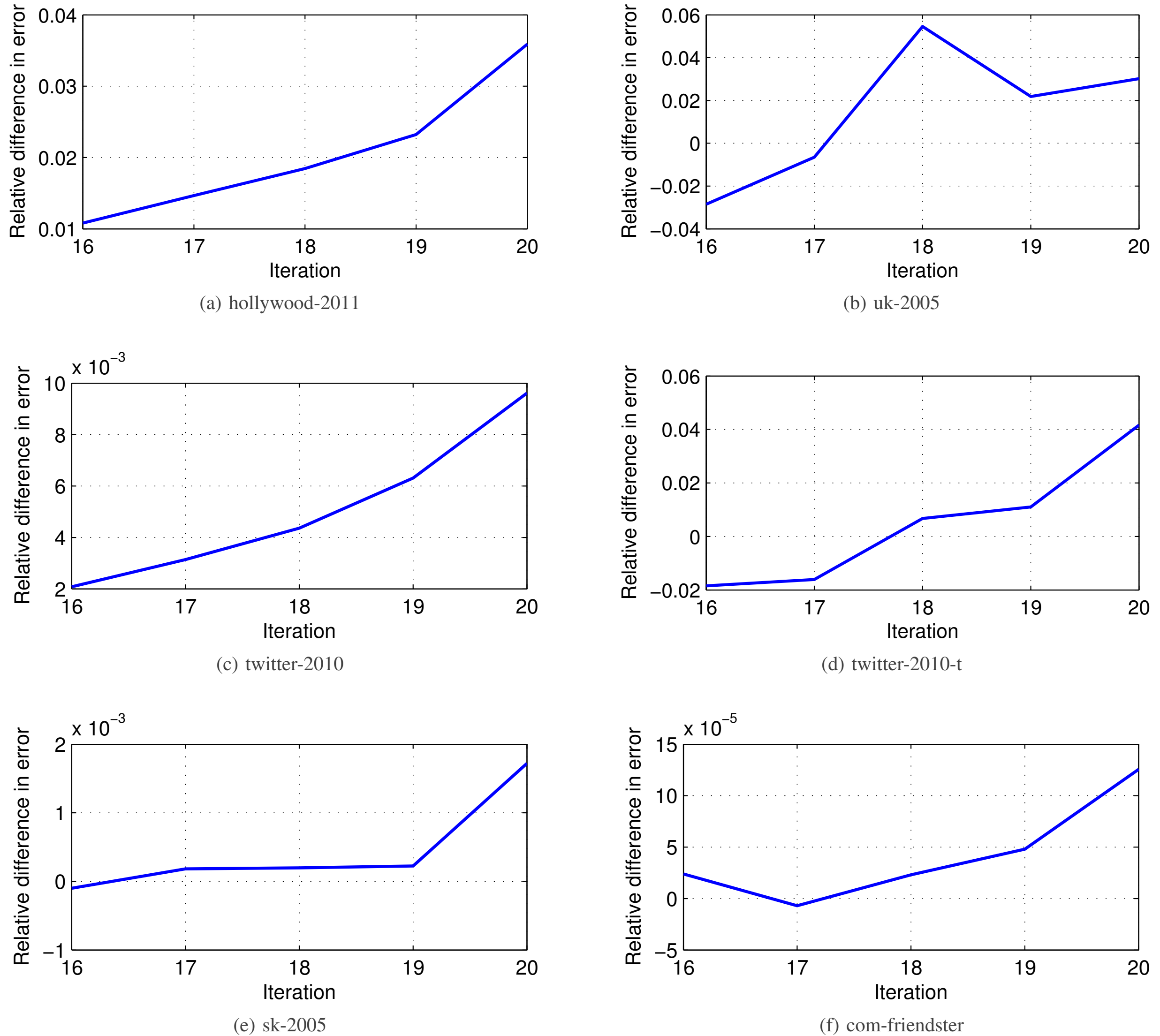

Fig. 5. For each graph, the increase in error incurred by eliminating thread safety from the power iteration algorithm to compute PageRank. These results show that the errors have the same order of magnitude and are identical in the first few significant digits. For iterations 16 through 20, we plot the relative difference in the errors $\|\mathbf{x}^{(k)} - \mathbf{x}\|_1$ of the thread-safe and thread-unsafe implementations. We estimate $\mathbf{x}$ using $\mathbf{x}^{(40)}$ in the thread-safe implementation. The transposed Twitter graph in plot 5d shows the largest increase in error after eliminating atomicity at only 4 percent.

which requires 7 iterations for convergence, generally takes longer to process than the transposed Twitter graph, which requires 6 iterations for convergence. Again, this discrepancy suggests that computation speed does not depend solely on the size of a graph, a major theme of our performance model.

E. Performance Model

In our performance model, we explore three potential bottlenecks of graph processing on nanocomputers: input/output bandwidth, memory bandwidth, and memory latency. As mentioned in table III, the I/O bandwidths are around 23 MBps on the Raspberry Pi, 110 MBps on Scaleway, and 150 MBps on the Kangaroo. (These are the parameters β_1 in the model.) Moreover, we estimate that the Raspberry Pi and Scaleway, both of which have a cache line size of 32B, have respective memory bandwidths of 810 MBps and 400 MBps; the Kangaroo, which has a cache line size of 64B, has a memory bandwidth of about 2100 MBps. (The parameter β_2 is the memory bandwidth, and C is the cache line size.) Finally, the memory latency for random accesses to the vectors in our experiments is roughly 250 ns on the Raspberry Pi, 130 ns on Scaleway, and 150 ns on the Kangaroo (the parameter L). The streaming speed was estimated by our software, and the memory parameters were determined via tinymembench [30].

In our experiments, we observe that the running time of a graph algorithm depends not only on a graph's size but also on its structure. The graph algorithms in figures 1 and 2 randomly access entries in a vector based on the endpoints of the edges in a graph. Some random memory accesses are more costly than others, especially remote memory accesses

TABLE VI

RUNNING TIMES OF THE LABEL PROPAGATION ALGORITHM TO COMPUTE CONNECTED COMPONENTS, INCLUDING THE PERCENT SLOWDOWN COMPARED TO THE GRAPH-STREAMING TIME. EACH GRAPH TAKES A DIFFERENT NUMBER OF ITERATIONS TO CONVERGE.

Graph (Iterations)	Raspberry Pi		Scaleway		Kangaroo	
	Time	Slowdown	Time	Slowdown	Time	Slowdown
hollywood-2011 (4)	193 s	38%	8.9 s	100%	5.2 s	110%
uk-2005 (11)	1816 s	0.67%	420 s	-2.1%	270 s	-1.8%
twitter-2010 (7)	1841 s	7.8%	490 s	32%	298 s	12%
twitter-2010-t (6)	1586 s	8.3%	401 s	26%	240 s	5.3%
sk-2005 (13)	4205 s	-0.17%	995 s	3.4%	649 s	-0.15%
com-friendster (7)	5350 s	27%	1289 s	45%	938 s	44%

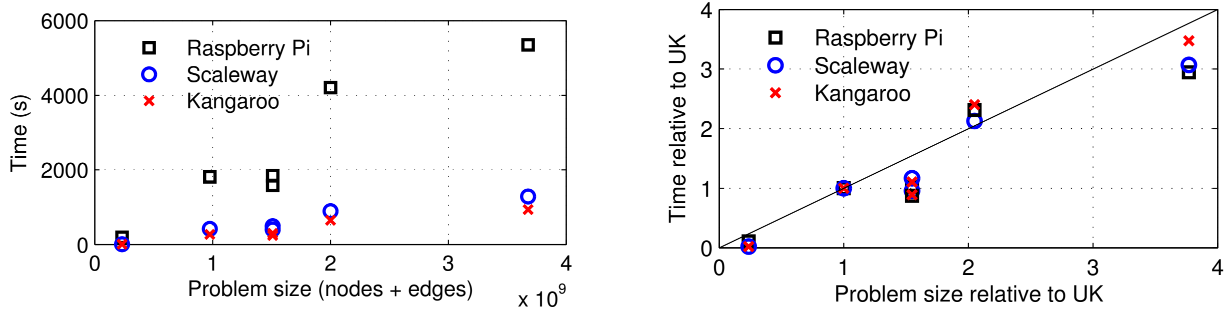

Fig. 6. The scalability of the label propagation algorithm to compute connected components. The plot on the left shows the running times in seconds, and plot on the right shows the running times relative to the UK graph.

TABLE VII

PREDICTED RUNNING TIMES TO COMPUTE 20 ITERATIONS OF PAGERANK BASED ON OUR PERFORMANCE MODEL, INCLUDING THE PERCENT OF THE ACTUAL RUNNING TIME OF 20 ITERATIONS OF PAGERANK WITHOUT ATOMIC UPDATES. "REMOTE ACCESSES" IS THE NUMBER OF MEMORY ACCESSES OUTSIDE OF AN APPROXIMATE L1 CACHE REGION.

Graph	Remote Accesses	Raspberry Pi		Scaleway		Kangaroo	
		Prediction	Percent of Actual	Prediction	Percent of Actual	Prediction	Percent of Actual
hollywood-2011	8.7%	766 s	98%	36 s*	57%	48 s*	120%
uk-2005	3.7%	3237 s	97%	677 s	96%	496 s	99%
twitter-2010	35%	5008 s	91%	1336 s	67%	1542 s	110%
twitter-2010-t	34%	5008 s	94%	1298 s	73%	1498 s	120%
sk-2005	7.9%	6635 s	100%	1387 s	97%	1017 s	100%
com-friendster	85%	15351 s	110%	7983 s	120%	9211 s	230%**

*Because the Hollywood graph is relatively small and fits in main memory on Scaleway and the Kangaroo, the I/O bandwidth bottleneck is not part of the prediction and the memory latency values are smaller (90 ns and 120 ns, respectively). **The performance model vastly overestimates the running time of PageRank on the Friendster graph on the Kangaroo; we suspect that the structure of the remote accesses on the Friendster graph is more complex than the structure of the other graphs.

that are far away from the previous memory accesses. In our performance model, we measure the percentage of remote memory accesses required for each graph – that is, we count how many memory accesses are a distance of at least 16KB from the previous memory access, based on the L1 cache size of a typical nanocomputer (the parameter R). While some graphs require few remote memory accesses, including the Hollywood graph (8.7%), the UK web crawl (3.7%), and the SK web crawl (7.9%), other graphs require more, including the Twitter graph (35%), the transposed Twitter graph (34%), and the Friendster graph (85%).

For one iteration of our graph algorithms, we need to stream edges over the I/O system, handle the memory traffic due to remote fetches, and account for the memory latency of those fetches. We expect one of these three factors to be the bottleneck. Thus, our performance model predicts the running time T of one iteration of PageRank or connected components based on the the nanocomputer and the graph:

$$T = \max\left\{\frac{S}{\beta_1}, \frac{mRC}{\beta_2}, mRL\right\}, \tag{3}$$

where S is the size of the graph in binary format, β_1 is the I/O bandwidth, m is the number of edges, R is the percentage of remote memory accesses, C is the cache line size, β_2 is the

TABLE VIII
COMPARISONS OF RUNNING TIMES TO COMPUTE 20 ITERATIONS OF PAGERANK ON THE TWITTER GRAPH AMONG SEVERAL GRAPH-PROCESSING SYSTEMS [1]–[10].

System	Cores	Memory	PageRank
Raspberry Pi	4	1GB	5516 s
Scaleway	4	2GB	2008 s
Kangaroo	4	2GB	1391 s
GraphChi	2	8GB	3160 s
Stratosphere	16	192GB	2250 s
X-Stream	16	64GB	1488 s
Vertica	48	192GB	1287 s
Giraph	128	1088GB	596 s
GraphX	128	1088GB	419 s
GraphLab	128	1088GB	249 s
Ringo	80	1024GB	121 s
Galois	48	1024GB	90 s
EmptyHeaded	48	1024GB	77 s

TABLE IX
COMPARISONS OF RUNNING TIMES TO COMPUTE CONNECTED COMPONENTS ON THE TWITTER GRAPH AMONG SEVERAL GRAPH-PROCESSING SYSTEMS [1]–[6].

System	Cores	Memory	Components
Raspberry Pi	4	1GB	1841 s
Scaleway	4	2GB	490 s
Kangaroo	4	2GB	298 s
X-Stream	16	64GB	1159 s
Stratosphere	16	192GB	950 s
Vertica	48	192GB	378 s
GraphX	128	1088GB	251 s
GraphLab	128	1088GB	242 s
Giraph	128	1088GB	200 s

memory bandwidth, and L is the memory latency. This model predicts that the bottleneck is always either I/O bandwidth or memory latency. It also predicts that on the Raspberry Pi, the only graph bottlenecked by latency is the Friendster graph, and on the other two nanocomputers, the only graphs bottlenecked by I/O are the UK and SK web crawls. Table VII lists our performance model's predicted running times to compute 20 iterations of PageRank and compares them to the actual running times of PageRank without atomic updates. The model generally predicts running times within a factor of two.

VI. SYSTEM COMPARISONS

A. Running Time

Many graph-processing systems report running times for computing PageRank and connected components on the Twitter graph, which allows us to compare the performance of these systems directly with the performance of nanocomputers [1]–[10]. The PageRank comparisons are in table VIII. In terms of absolute runtimes, the nanocomputers are not competitive. However, we note that only Galois and EmptyHeaded – both recent, highly optimized in-memory systems – outperform the Kangaroo on a per-core basis. Moreover, nanocomputers can compute PageRank on billion-edge graphs roughly an order of magnitude more slowly than most systems despite the vast differences in available computational power.

The most similar graph-processing system to our work is GraphChi, which reports results from a 2-core Mac Mini. For comparison, we ran GraphChi's C++ implementation of PageRank on the Twitter graph with the Kangaroo. Using an HDD rather than an SSD to store the Twitter graph in an edge-list format, GraphChi took 762 seconds per iteration, whereas our implementation, also using an HDD, took 266 seconds. The distinguishing feature of our work is the utilization of nanocomputers, and our software implementation seems to outperform GraphChi on these small computer architectures.

Table IX shows the connected components comparisons. Most in-memory graph-processing frameworks either omit running times for connected components or perform algorithms more optimized than the label propagation algorithm. This comparison shows that nanocomputers can compute connected components on billion-edge graphs at roughly the same speed as many systems running on dozens of cores.

B. Energy Consumption

We use a Kill A Watt meter to measure the power consumption of the Raspberry Pi and the Kangaroo. The maximum power consumption of the Raspberry Pi with its external hard drive was 7.0 W, and the maximum power consumption of the Kangaroo was 9.3 W. While the Raspberry Pi consumes the least power, the Kangaroo's faster computation detailed in table V makes it the most energy-efficient computer. Scaleway does not report the power consumption of its servers, but since its specifications in table I are comparable to the other two nanocomputers, we expect that it shares the same energy advantages as the other computers. Although graph-processing systems do not report how much power their computer clusters consume, energy is a legitimate cost of large-scale data processing. The fact that our nanocomputers consume less than 10 W while performing massive PageRank computation suggests that energy efficiency is a significant advantage of using nanocomputers to process graphs.

VII. DISCUSSION

These results show that simple, customized implementations of various graph algorithms can achieve system-limited performance on nanocomputers. To achieve better performance, we would need to optimize our implementations further – such as using fast integer encoding and decoding strategies [31] to optimize I/O on the Raspberry Pi, improving matrix orderings to improve locality [15], or improving algorithms to reduce the total work [28].

That said, flexible graph-processing systems have several distinct advantages over nanocomputers. First, their large computer clusters have enough memory to store PageRank and connected components vectors for exceptionally large networks like the World Wide Web, so we expect that many of these systems would vastly outperform nanocomputers while

processing such graphs. Additionally, our implementations of PageRank and connected components are customized, instead of being realized as special instances of a more general processing style. Moreover, while these two algorithms represent a host of graph-processing tasks that require streaming through all of the edges in a graph [12], they do not represent many other algorithms that analyze only a certain region of a graph or require random access to the edges in a graph.

Nonetheless, our results suggest that low-cost, energy-efficient nanocomputers can perform large graph computations, and the running times of our implementations are comparable to those of graph-processing systems running on dozens of cores. Much of the experience gained in optimizing computations on these low-power systems is likely to be useful for extracting maximum performance out of various subroutines in larger, heterogeneous parallel systems, which is likely an increasingly important strategy for high-performance computing systems. We suggest that the rate of innovation in small, energy-efficient computers relative to increases in the sizes of graphs that researchers analyze may be an important factor in the future development of graph-processing systems designed for processing a single graph in multiple ways.

ACKNOWLEDGMENTS

We would like to thank Nicole Eikmeier, Kyle Kloster, Huda Nassar, Varun Vasudevan, and Nate Veldt for their careful reading of an early draft. This work was supported by NSF CAREER award CCF-1149756.

REFERENCES

[1] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2014.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," in *Proceedings of the VLDB Endowment*, vol. 5, no. 8, Apr. 2012, pp. 716–727.

[3] A. Ching and C. Kunz, "Giraph: Large-scale graph processing infrastructure on Hadoop," in *Proceedings of the Hadoop Summit*, Jun. 2011.

[4] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Nov. 2013, pp. 472–488.

[5] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," in *Proceedings of the VLDB Endowment*, vol. 5, no. 11, Jul. 2012, pp. 1268–1279.

[6] A. Jindal, S. Madden, M. Castellanos, and M. Hsu, "Graph analytics using the Vertica relational database," *arXiv*, Dec. 2014, Available: http://arxiv.org/abs/1412.5263.

[7] A. Kyrola, G. Bllelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012.

[8] Y. Perez, R. Sosic, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec, "Ringo: Iinteractive graph analytics on big-memory machines," in *Proceedings of the ACM SIGMOD Conference*, Jun. 2015.

[9] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré, "EmptyHeaded: A relational engine for graph processing," *arXiv*, vol. cs.DB, Mar. 2015, Available: http://arxiv.org/abs/1503.02368. [Online]. Available: http://arxiv.org/abs/1503.02368

[10] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, Jun. 2011.

[11] J. Shun and G. E. Blelloch, "Ligra: A Lightweight graph processing framework for shared memory," in *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming*, Feb. 2013, pp. 135–146.

[12] D. F. Gleich and M. W. Mahoney, "Mining large graphs," in *Handbook of Big Data*, ser. Handbooks of Modern Statistical Methods, P. Bühlmann, P. Drineas, M. Kane, and M. van de Laan, Eds. CRC Press, 2016, pp. 191–220.

[13] A. Epasto, J. Feldman, S. Lattanzi, S. Lenoardi, and V. Mirrokni, "Reduce and aggregate: Similarity ranking in mutli-categorical bipartite graphs," in *Proceedings of the 23rd International Conference on World Wide Web*, Apr. 2014, pp. 349–360.

[14] Z. Gyongyi, H. Garcia-Molina, and J. Pedersen, "Combating web spam with TrustRank," in *Proceedings of the 30th International Conference on VLDB*, Sep. 2004, pp. 576–587.

[15] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST?" in *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*, May 2015.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford University, Tech. Rep., 1998.

[17] U. Kang, C. E. Tsourakakis, and C. Faloustos, "PEGASUS: A peta-scale graph mining system – implementation and observations," in *Proceedings of the 9th IEEE International Conference on Data Mining*, Dec. 2009, pp. 229–238.

[18] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[19] M. Jha, C. Seshadhri, and A. Pinar, "A space efficient streaming algorithm for triangle counting using the birthday paradox," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 2013, pp. 589–597.

[20] D. F. Gleich, "PageRank beyond the Web," *SIAM Review*, vol. 57, no. 3, pp. 321–363, 2015.

[21] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression techniques," in *Proceedings of the 13th International World Wide Web Conference*, May 2004, pp. 595–601.

[22] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International World Wide Web Conference*, Apr. 2011, pp. 587–596.

[23] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "UbiCrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[24] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[25] F. Niu, B. Recht, C. Re, and S. J. Wright, "HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent," in *Proceedings of the 25th Conference on Neural Information Processing Systems*, 2011.

[26] H. Avron, A. Druinsky, and A. Gupta, "Revisiting asynchronous linear solvers: Provable convergence rate through randomization," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, May 2014.

[27] S. Kamvar, T. Haveliwala, and G. Golub, "Adaptive methods for the computation of PageRank," Stanford University, Tech. Rep., 2003.

[28] F. McSherry, "A uniform approach to accelerated PageRank computation," in *Proceedings of the 14th International World Wide Web Conference*, May 2005, pp. 575–582.

[29] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the 12th International Conference on Data Mining*, Dec. 2012, pp. 745–754.

[30] S. Siamashka, "tinymembench," Available: https://github.com/ssvb/tinymembench.

[31] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, 2015.