SAND2015-3087C

Supercomputing for Web Graph Analytics

George M. Slota
Pennsylvania State University
University Park, PA
gslota@psu.edu

Sivasankaran Rajamanickam
Sandia National Laboratories
Albuquerque, NM
srajama@sandia.gov

Kamesh Madduri
Pennsylvania State University
University Park, PA
madduri@cse.psu.edu

ABSTRACT

The Web Data Commons 2012 hyperlink graph is one of the largest publicly-available hyperlink graphs, with over 3.5 billion vertices (web pages) and 128 billion edges (hyperlinks). Analyses of graphs of this scale are typically performed on large shared-memory systems, or using open-source Big Data computational frameworks such as Hadoop and Giraph. In this paper, we present a collection of algorithms and new parallel implementations for end-to-end analysis of this massive hyperlink graph. Our MPI- and OpenMP-based code can be run on supercomputers as well as smaller compute clusters. For instance, using 256 compute nodes of the Blue Waters supercomputer, the end-to-end processing and I/O time for six different computationally-challenging routines is about 70 minutes. Our new contributions include a distributed compressed graph representation, and efficient parallelization strategies for graph connectivity, harmonic centrality, PageRank centrality, and the label propagation community detection algorithm. Our algorithms lead to novel centrality and community structure results for this massive graph. Our work could further be used for benchmarking algorithms, software, and compute platforms for massive graph analytics.

1. INTRODUCTION

With the ever-increasing amounts of information being produced and consumed online, scalable web data management and analytics is growing in importance. Among the diverse structured and unstructured information sources available online, the *web hyperlink graph* is a unique resource that researchers have been exploring using graph-theoretic tools and methods. A hyperlink graph can be constructed by treating every web page that is accessible online as a vertex, and using directed edges to indicate links between pages. Understanding the structure of hyperlink graphs, of any scale, has many uses. The biggest and most direct application is perhaps web search. Web graph analysis may also shed insight into social and economic processes that may be responsible for the growth of the Internet. A third use case is to identify spam pages from legitimate and content-rich pages. Computer science researchers have been using hyperlink graphs as real-world instances for testing combinatorial algorithms and machine learning methods for ranking, community identification, inference, graph partitioning, and graph compression. In this paper, we study well-known computations on such web graphs (generally referred to as *web graph analytics*) from a High-Performance Computing (HPC) perspective. Our goal is to provide an informed commentary on the efficiency, scalability, and ease of implementation of graph analytics on current high-end computing systems. For this purpose, we consider the largest publicly-available hyperlink graph, the 2012 Web Data Commons graph, and design clean-slate parallel algorithms and implementations for the Blue Waters supercomputer, one of the world's most powerful computing platforms.

There is a lot of current work on graph algorithms, analytic frameworks, and graph data management systems. The following aspects motivate our present work and differentiate it from other related research efforts:

- Most parallel computing research efforts focus on a single computational routine and study its scalability for a collection of large graphs [9, 10, 13]. We want to simultaneously analyze performance of multiple analytics, and make decisions about graph data representations and decomposition strategies based on a single large-scale graph. We have picked the largest possible real-world graph that is publicly available.
- Synthetic graphs can never substitute real-world graph instances, as there are always some topological aspects that parsimonious models miss. For real-world graphs, we may be able to *exploit more structure* when designing analysis algorithms. There may also be some intricacies associated with real-world graphs that do not manifest in synthetic graphs. One of our objectives is to precisely quantify some challenges, for a collection of graph analytics, on an extremely large graph. We also want to identify known optimizations that work for this graph instance.
- Before choosing an algorithmic design paradigm (say, *vertex-centric*, *edge-centric*, *gather-apply-scatter*) or parallel platform (large shared memory, distributed memory, external memory, accelerators) for analyzing a family of graphs, we believe it is important to get a sense of how custom, specialized codes would perform on a certain class of graphs. We consider this massive graph a possibly challenging instance for any framework.
- Algorithms that are commonly used for graph analytics have per-iteration operation counts that that scale linearly (in the asymptotic case) with the number of vertices and edges. Further, memory requirements and communication costs are also linear. Thus, the constant factors in the implementations are what lead to large performance gaps on real systems. Distributed

graph analytic frameworks that provide linear-work algorithms should thus be evaluated on the largest-available graphs.
- I/O costs are often ignored when doing in-memory graph analytics. There is also quite a lot of research on external and semi-external memory graph algorithms and frameworks, where minimizing I/O costs, and not wallclock time, is the primary focus. An end-to-end evaluation of multiple analytics, considering I/O, memory, and network costs, would be more representative of real-world performance.
- We take a top-down approach and focus on high-level analytics in this work, rather than primitives that are commonly used within these analytics. Focusing on the higher-level computations provides more opportunities for testing algorithmic variants and optimizations.
- Our end-goal is also to provide a new benchmark for evaluating high-end computer hardware, software, as well as algorithms. Our current implementation and subsequent progress will be open-sourced. Given the size of the graph and some of its pecularities, we believe this hyperlink graph is a non-trivial and challenging graph instance for parallel graph analytics.

This paper presents the first in-memory, end-to-end analysis of the largest publicly available graph, including parallel I/O, graph construction and multiple useful analytics. In addition, this paper makes the following new algorithmic and parallel implementation contributions to analyze this massive graph: We present a fast and memory-efficient scheme to store the graph in a distributed setting. We then present algorithms for analyzing the connectivity of this large directed graph. Next, we discuss parallel implementations of the global path-based Harmonic centrality approach and the popular PageRank scheme. We finally develop a parallel community identification implementation based on the label propagation approach. Using these analytics, we are able to obtain new insights into the community structure and page-level centrality rankings of the web graph. Some of these insights are from the possibly the first in-memory global community structure experiment on this massive graph.

We also claim that graph analysis at scale need not be very daunting. Without resorting to tedious implementations and performance engineering, we show that end-to-end execution times can be reasonable, provided one makes informed algorithmic and data structure choices. All the analysis routines discussed in this paper fit into approximately 2500 C++ source lines, use only MPI and OpenMP for parallelization, and optionally, one additional library for compressing sorted integer lists. Using just 256 compute nodes of Blue Waters, we are currently able to perform all these analytics in under 70 minutes.

This paper is organized as follows. We discuss the data sources, the test platform, and prior work on analyzing this web graph in Section 2. In each sub-section of Section 3, we present an analytic, our parallel algorithm and data structure choices for this analytic, and performance results on the test platforms. In Section 4, we discuss cross-cutting issues related to performance, scalability, and portability. Given the fact that even a simple ingestion strategy of this graph requires nearly 2 TB of aggregate memory and extensive data massaging, we were unable to evaluate existing open-source frameworks. We however mention some relevant performance results from prior work in Section 5.

2. DATA AND SETUP

We analyze the 2012 Web Data Commons hyperlink graph[1] in this work. This graph was in-turn extracted from the open Common Crawl web corpus[2]. The Web Data Commons research team provides another hyperlink graph based on the Spring 2014 common crawl web corpus, but they recommend using the 2012 graph only, as the largest strongly connected component in the 2014 graph is much smaller. The 2012 graph is available for download with three levels of aggregation: at page-level, at the granularity of subdomains/hosts, and at the granularity of pay-level-domain (PLD). We primarily work with the page-level graph in this paper. This graph has 3563 million vertices and 128,763 million edges. In recent work, Meusel et al. [26] presented analysis of the page-level graph, and Lehmberg et al. [20] evaluated the PLD graph. They use the WebGraph library [5] to compress the graphs and have performed their analysis on large shared-memory systems. Meusel et al. make several interesting and novel observations in their work. They confirm that a giant strongly connected component (SCC) exists, which is along the lines of the observation by Broder et al. [7] in their seminal paper on web structure. Meusel et al. claim that the hypothesized *bow-tie* model for the web may actually be strongly dependent on the crawl strategy. Further, they show that degree and component size distributions do not follow a power law, but may be heavy-tailed. They mention using a 40-node Amazon EC2 instance to compress the graph, but the running times for various analytics discussed in their paper were not quantified. They also mention that computing the largest SCC 'was no easy task', and that they had to use a semi-external memory strategy to build it.

The PLD-level graph is considerably smaller with 43 million vertices and 623 million edges. The authors in this paper additionally look at centrality rankings of PLDs and present a new *two-layer model* to explain the structure of this aggregated graph. They also study clustering coefficient distributions and connectivity of high-degree PLDs among themselves. We verify some of the results from these two papers, but do not discuss any of the common observations in the next section. We also attempt to precisely quantify memory usage and communication volume of each analytic, and also the impact of load balance with the natural crawl-based vertex ordering. In order to understand effects of the degree distribution on the load balance, we also generate a synthetic $G(n, P)$ graph [16] of identical size. The inter-node communication behavior of graph analytics when tested with a $G(n, P)$ graph is very regular, and the edge cut can also be analytically determined. Hence, this provides a good instance for scalability and load balance comparisons.

We primarily use the NCSA Blue Waters supercomputer for analyzing this page graph. Blue Waters is a hybrid Cray XE6/XK7 system with around 22,500 XE6 compute nodes, and 4200 XK7 compute nodes. Each XE6 node contains two

[1] http://webdatacommons.org/hyperlinkgraph/
[2] http://commoncrawl.org

AMD Interlagos 6276 processors. There are four NUMA domains per node, each with four cores and sharing a 8 MB L3 cache. The memory capacity of each node is 64 GB and the peak memory bandwidth is 102.4 GB/s. We do not use the GPU-accelerated XK7 partition of Blue Waters. One of the main reasons we chose Blue Waters was the high-performance file system. The Lustre-based scratch file system uses 180 Scalable Storage Units (SSUs) and the rated I/O bandwidth is a remarkable 960 GB/s. We compile our C++ programs using Intel's compilers (version 15.0.0) and use the Intel Programming Environment (version 5.2.40) and Cray-MPICH version (7.0.3). Some of our experiments were also run on the NERSC Edison supercomputer. This is a Cray XC30 system with over 5500 nodes. Each node is a dual-processor Intel Ivy Bridge with 24 cores and 60 GB memory. Edison also has a high-performance I/O subsystem and the Lustre-based scratch directory we used has 96 OSTs and aggregate peak performance of 48 GB/s.

We emphasize that our goal is to use as few compute nodes as possible to run all the analytics on the page-level graph. Some of the analytics we discuss in this paper, such as community detection and largest strongly connected component extraction, serve as ways to prune large graphs and construct smaller coarser graphs or regions-of-interest, such as PLD graphs. Strong-scaling these pruning computations is perhaps not as important as achieving absolute speedups for analytics that operate on smaller and coarser graphs. Good examples of such analytics are various centrality measures, local community detection, link-based recommendation engines, and network alignment. Further, note that most of the computations we discuss in this paper perform per-iteration operations that scale linearly with the number of vertices and edges, and have memory requirements that are also linear with number of vertices and edges. In order to strong-scale these analytics, one can replicate vertex and edge state to reduce communication.

3. GRAPH ANALYSES AND ALGORITHMS

In this section, we describe the end-to-end analysis of the massive web graph, beginning with data ingestion, graph construction, and algorithms for each of the analytics in detail. We also present the insights gained from each one of the implemented analytics.

3.1 Data Ingestion

We converted the nearly 700 gzipped, text-formatted files with directed edge information into a single binary file (approximate size 1 TB). Each directed edge is represented by two vertices of 32-bit unsigned integer type $\langle v_0, v_1 \rangle$. We ingested data by calculating offsets for each MPI task, with each task given a nearly-identical portion of the file to read.

To achieve high read bandwidth from Blue Waters' shared Lustre-based scratch filesystem, we striped the 1 TB file across 160 storage units. Column 2 of Table 1 lists the total read time when varying the number of compute nodes, with a single MPI task per node. These times correspond to read bandwidths between 20-30 GB/s, or under a minute total to just read the edges into memory. This is well below the 1 TB/s theoretical peak. However, tests were performed during periods of high concurrent system use. We note that using a larger number of tasks generally corresponds to faster I/O. This is possibly due to the lower read volume requirement for each task, and so slowdowns from a single task due to network traffic and concurrent file system accesses may have a lesser effect on total time.

Table 1: Parallel performance for various stages of graph construction including the total read time (*Read*), time for edge exchange (*Excg*) and time for CSR representation (*LConv*).

# Nodes	Time (s) *Read*	*Excg*	*LConv*	*Total*	Perf Rate (GE/s)	Speedup
256	47	109	41	197	1.30	1.00×
512	45	90	33	168	1.52	1.17×
1024	42	61	27	130	1.97	1.51×
2048	34	55	24	113	2.27	1.74×
4096	39	68	23	130	1.97	1.51×
8192	35	56	19	110	2.33	1.79×

3.2 Graph Data Structure Construction

We choose a memory-efficient *one-dimensional* graph representation in this work, where each MPI task owns $\frac{n}{p}$ vertices and all the incoming and outgoing edges out of these vertices. After each task reads its share of outgoing edges, the edges are exchanged using an MPI `Alltoallv` step. As most of our analytics also require incoming edges, we then proceed to reverse the edges and do another `Alltoallv` exchange. Note that this second step usually completes much faster than the first, as the vertex identifiers as labeled have a moderate degree of locality. This is a byproduct of the labeling done by the Web Data Commons research team when they extracted the hyperlink graph from the Common Crawl corpus. Once each task has all of the outgoing and incoming edges for the vertices owned by this task, they can then convert the edge arrays into a compressed sparse row (CSR)-like representation.

Table 1 also shows the running times in seconds for exchanging both outgoing and incoming edges, as well as the time to create the final distributed CSR representation when varying numbers of tasks. We note a degree of strong scaling with increasing task count. We also include in Table 1 a performance rate in billions of edges processed per second (GE/s), corresponding to the total number of edges processed (128 billion in- and 128 billion out-edges). We note that the maximal time from start of execution to a complete CSR representation is under 200 seconds.

The graph ingestion and creation stage is the most memory-intensive part of our implementations. To hold the outgoing edge list in memory, we require $8m$ bytes of global memory, where m is the number of edges in the graph. This implies approximately 1 TB of aggregate memory. In order to use MPI collectives in the *Excg* step, we also need to create send and receive buffers, which require an additional $16m$ bytes of memory. Assuming each task has 64 GB memory and we have an even edge partitioning among tasks, and given the signed integer restriction in MPI collectives, the minimum number of tasks needed for this stage is about 94. However, as we shall see next, there could be up to 2× edge imbalance with task counts between about 100 and

512. This effectively bounds the minimum number of tasks required to be about 188, which we is what we observed in practice. We have also developed a graph ingestion scheme (explained next) that avoids the All-to-all shuffle.

3.2.1 *Compression*

Consider an alternate ingestion scenario where we can assume that edges are sorted first by head vertex, and then by the tail vertex. Given an edge list in binary format stored on disk, one could read $\frac{2m}{p}$ chunks of edge lists, figure out start offsets of vertices whose edge lists are cut, and then arrive at an edge-balanced partitioning of the graph. This ingestion strategy requires close to minimal inter-process communication and has very good potential to exploit parallel I/O. We could further use multiple threads per process to read the graph. Such an edge-balanced approach however may however imply an imbalance in per-process vertex counts, especially if the degree distributions are skewed and the vertex ordering is not random. We implement this strategy and were able to read only the out-edges in about 2 minutes, using just 12 nodes (about 732 GB of user-accessible aggregate main memory; 24 OpenMP threads per node) of NERSC Edison. This corresponds to an ingestion rate of 1.07 GE/s.

Further, based on the observation that the adjacency arrays are sorted, and that the crawl ordering of the vertices has considerable locality, we use a fast integer compression library [21] to compress the adjacency array of each vertex. The compression and decompression subroutines in this library are optimized for modern Intel architectures, and so we used Edison. We picked the compression and encoding algorithm to use based on vertex degree. Adjacencies of vertices with degree less than 4 were just stored uncompressed. Vertices with degree between 4 and 127 were compressed using the *varint* scheme. Adjacencies of vertices with degree greater than 127 were compressed using the *S4-BP128-D4* scheme. Both compression and decompression were extremely fast and effective. We could compress out-edges using **12.74 bits** per link. We also used a similar strategy to compress in-edges of each vertex. In-edges compression worked even better than out-edges at **8.39 bits** per link. Reading and compressing both in-edges and out-edges took about 220 seconds, and required just 10 nodes of Edison. This corresponds to a rate of 1.16 GE/s. We believe this is a significant result, as we can fit the largest web crawl entirely in memory using a very simple compression method. The catch here is the assumption on the sorted ordering of edges. The edges were initially sorted from head to tail (out-edge order), and so we did not need one sort. For generating sorted in-edges, we implemented a parallel, out-of-place histogram sort. This is not quite optimized. Using 50 nodes of Edison and 4 MPI tasks per node, we were able to sort all edges in about 4 minutes. Using implementations of faster parallel sorting algorithms, we could do even better, and sort using fewer nodes. We note that this compressed graph representation may also benefit semi-external memory frameworks such as FlashGraph [42]. Evaluating the performance of algorithms using this compressed graph representation is left for future work. One issue we encountered was that the compression library could not be ported to the AMD processor-based Blue Waters system because of the use of Intel specific intrinsics in the compression library. Note, however, that the algorithms for the analytics to be described next, require minimal changes, as we apply compression on a per-vertex basis, and there is no other indirection. We will consider the uncompressed CSR graph stored in-memory for the rest of the paper.

3.3 Computing Global Statistics

To verify the integrity of the ingested binary web crawl data, and to get a sense of the global structure, we computed the out and in-degree distributions. In Figure 1, we plot these distributions as cumulative fractions of total edges versus total vertices, with vertices sorted in decreasing order of degree. We also plot an in-degree distribution of the synthetic $G(n,p)$ random graph for comparison.

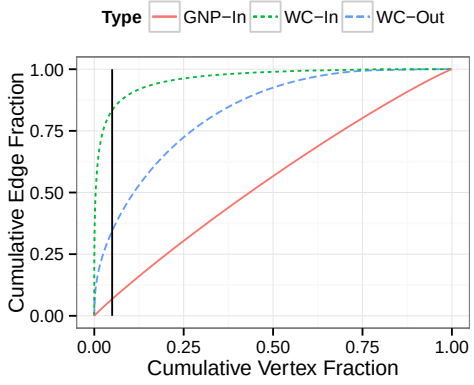

Figure 1: Cumulative vertex versus in and out edge fraction for the web crawl and random graph.

As was observed by Muesel et al. [26], we see a skewed degree distribution. A small fraction of vertices account for the vast majority of the total incoming edges. The solid vertical line indicates a 5% cumulative fraction of vertices, which own over 83% of all incoming edges. The highest in-degree vertex in the crawl, `www.youtube.com`, has about 93 million incoming links. The bias for outgoing edges is not as skewed, with 5% of vertices owning 34% of the edges, and a maximal degree of about 56,000. There were also some crawl settings preventing exploring pages with a large collection of hyperlinks, and so the out-degree is bounded.

We note two additional observations about the web crawl that were not mentioned in the Muesel et al. paper. Firstly, there are about 120 million vertices in the crawl having no incoming or outgoing connections. Secondly, there is a large number of repeated edges. For some of the high degree vertices, only about 75% of their incident edges are unique. This knowledge is useful for trimming and preprocessing the graph in order to reduce storage requirements and improve execution times of algorithms that ignore repeated edges during execution (e.g. SCC, harmonic centrality).

It is well-studied that skewed degree distributions present a severe challenge for scalable and portable graph algorithm implementations, both in shared-memory and distributed memory. As skew increases, a vertex-balanced partitioning can have severe inter-task work and communication load imbalances, as well as severe intra-task work imbalance when using a full-partitioned (MPI-only) data layout. Using MPI and multithreading alleviates load imbalance to an extent,

as the graph is not as partitioned, and there is some sharing of vertex information within a node. The number of partitions, and consequently, the number of MPI tasks, is therefore crucial.

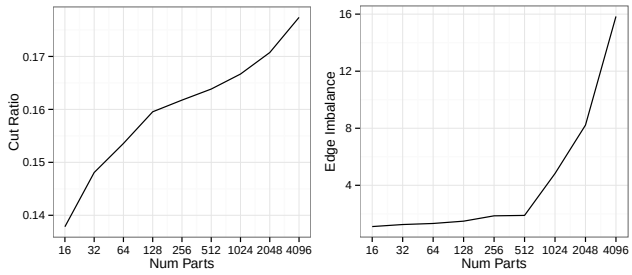

Figure 2: Edge cut and edge imbalance versus number of parts for $\frac{n}{p}$ vertex partitioning (or vertex balancing partitioning) of the web crawl using natural ordering.

We plot in Figure 2 the edge cut ratio (the number of cut edges to the total number of edges) and edge imbalance ratio (the maximum number of edges assigned to any partition, divide by the average number of edges per partition) for various task counts, using natural ordering and assuming an $\frac{n}{p}$ vertex partitioning on the crawl. We note that the edge cut ratio (between 0.14 and 0.18) is actually quite good, and seems to increase sub-linearly with the number of parts. It is not close to 1, as one would expect with a random partitioning. As mentioned, this is likely a byproduct of the crawl ordering. The edge imbalance remains moderate, staying under 2× until 1024 parts. The large jump is explained by the fact that 1024 parts is approximately the limit where the largest degree vertices can exist on a single given partition without imposing significant imbalance by themselves. This plot gives some insight into the number of tasks beyond which a simple $\frac{n}{p}$ partitioning becomes ineffective.

3.4 Connectivity

As part of our analytic collection, we implement several connectivity-based algorithms. We run analytics to extract the largest weakly-connected component (WCC), largest strongly connected component (SCC), as well as an algorithm for computing the approximate k-core values for vertices in the largest weakly connected component of the crawl. All of these algorithms require a variation of a distributed breadth-first search (BFS).

Our implementation assumes an MPI+OpenMP model, where each MPI task has a local queue of vertices to explore during discovery of a given level and an array of visitation statuses. The work in the local queue is divided among threads. Each thread also maintains two smaller next-level queues during adjacency list expansion, with the queues being filled either with vertices for the task to explore on the next level or with vertices to communicate to other tasks for their exploration. Each task also maintains two queues for next-level exploration and communication, and when a thread-owned queue reaches capacity, it atomically retrieves and updates an offset to one of the task-level queues. When the current work queue expires, all tasks communicate next-level work with an `Alltoallv` and the queues are reset.

Algorithm 1 FW-BW SCC Algorithm for finding the largest SCC (S) given a graph G, with vertices V, outedges $E(V)$ and inedges $E'(V)$

1: **procedure** FW-BW($G(V, E)$)
2: Select $v \in V$ for which $d_{in}(v) * d_{out}(v)$ is maximal
3: $D \leftarrow \text{BFS}(G(V, E(V)), v)$
4: $S \leftarrow D \cap \text{BFS}(G(D, E'(D)), v)$

We use this BFS implementation for finding weakly-connected components (WCC) and the largest strongly connected component (SCC). For SCC, we utilize the Forward-Backward approach [17], as given by Algorithm 1. We first discover all vertices reachable from our root by expanding out edge adjacencies (descendant set, D) and then subsequently discover all vertices that can reach our root through in edge adjacency expansion. The overlap between these two sets of vertices, S, is the strongly connected component that contains our original root. We maintain a group of potential root vertices based on the magnitude of in- and out-degrees. Prior work [38] has shown that on large real-world graphs such as web crawls, the largest degree vertices are almost always part of the largest weakly and strongly connected components. To extract the largest WCC, we simply need to run a single BFS starting from one of our roots, exploring both incoming and outgoing edges for each vertex discovered.

Note that an extension to find all the strongly or weakly connected components requires multiple iterations of the same FW-BW algorithm, or a color propagation algorithm similar in communication pattern to label propagation described below [38,39]. We choose to study that communication pattern for community structure algorithm and instead compute just the largest SCCs and WCCs here.

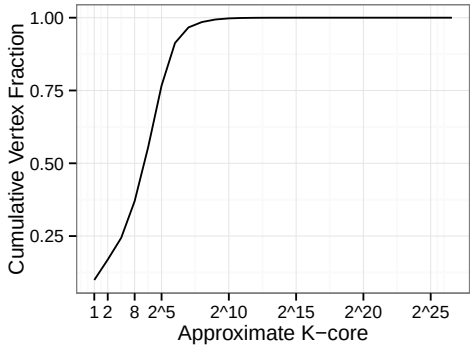

Figure 3: Cumulative fraction of vertices versus approximate k-core values.

We employ a similar approach to compute an approximate k-core decomposition. We use our set of seed vertices and iteratively explore the maximal components containing vertices of some minimal degree. This will effectively give us an upper bound on true k-core values. We capture the maximal approximate k-core sizes versus k-core values we retrieved during our analysis of the web crawl. These are plotted as cumulative fractions in Figure 3. We observe the sizes of the approximate k-cores are quite large, dropping below 500 million vertices only after reaching a k-core value of 128. However, by about a k-core value of 1024, only a small frac-

tion of vertices remain (20 million, or about 0.5%). The approximate k-core results can be refined, if necessary, to compute exact k-core values.

3.5 Centrality Measures

We additionally compute several centrality metrics as part of our distributed large-scale analytics test suite. Centrality measures generally indicate the relative importance of a single vertex in graph. We look at the centrality measures of harmonic centrality, PageRank, as well as in-degree and out-degree.

Our implementation for determining the highest-scoring vertices based on harmonic centrality follows the same general approach as what was described previously for our BFS-based analytics. We select up to 100 seed vertices sorted by decreasing in-degree to calculate harmonic centrality values. For each vertex, we then determine the distances of all vertices that are able to reach the seed vertex through a distributed BFS following incoming edges. The final harmonic centrality value is calculated as $\sum_{d=1}^{\infty} \frac{1}{d}|V_d|$, which corresponds to the sum over all distances d of the number of vertices $|V_d|$ that are d hops from the seed, scaled by d.

We also calculate all per-vertex PageRank values in order to retrieve the highest-scoring vertices. Our approximation to the PageRank scheme is given in Algorithm 2. Our implementation stores the scores for each task-owned vertex in a local array, $PageRanks$, and uses a hash map, $PageRankMap$, for storing scores of vertices within one incoming hop. These values are stored as vertex-value pairs. During each iteration, we compute all scores for each task set of vertices, V_t, using thread-level parallelism. We use `Alltoallv` collectives to communicate updated PageRank values of boundary vertices among tasks. Note that we do not explicitly check for convergence of the PageRank values, nor do we have any communication-reducing optimizations (such as propagating score of a vertex only if it has changed) in the current implementation.

Previously, the Web Data Commons team and Meusel et al. calculated centrality measures on the condensed host graph. In order to gain a different perspective on the highest vertices by centrality, we chose instead to calculate centrality measures on the full page graph. The top-scoring results for centrality values of out-degree, in-degree, PageRank, and harmonic centrality are given in Table 2. Note that we consider the PageRank values approximate, as we report the top scores after 20 iterations. The harmonic centrality values are also considered approximate, because we only calculated the values for the top 100 vertices by in-degree.

There are several conclusions that can be reached based on the results of Table 2. The foremost that out-degree does not have any significance as a centrality measure. Looking at the other three columns, we note consistent similarities in the sites that appear between our results, and those calculated on just the host level graph (`youtube`, `wordpress`, `google`, `gmpg`, `twitter`). However, we also note how pages belonging to the same host consistently appear together in the rankings. As we see in our next analytic, a number of these vertices are found to additionally appear in similar

Algorithm 2 Distributed PageRank

```
1: procedure PageRank-dist(G(V, E), δ)
2:     PageRankMap = HashMap()
3:     for all v ∈ V do
4:         PageRanks(v) ← 1/(|V|×|E(v)|)
5:     for i = 1 to iter do
6:         Q_update += Alltoallv(V_t, PageRanks)
7:         for all (v, p) ∈ Q_update do
8:             PageRankMap.set(v, p)
9:         for all v ∈ V_t do in parallel
10:            p_v ← 0
11:            for u ∈ E'(v) do
12:                if rank(u) = t then
13:                    p_v ← p_v + PageRanks(u)
14:                else
15:                    p_v ← p_v + PageRankMap.get(u)
16:            p_v ← p_v × δ
17:            p_v ← p_v × (1−δ)/|V|
18:            p_v ← p_v/|E(v)|
19:            PageRanks(v) ← p_v
20:    for all v ∈ V do
21:        PageRanks(v) ← PageRanks(v) × |E(v)|
```

dense communities. We observe this trend continuing well beyond just the top 10 scoring pages, though these additional results are omitted for brevity. Overall, it is apparent that using the full page-level graph can be quite noisy, indicating the necessity for preprocessing and aggregating the page-level graph. While considering only the host-level graph for analysis and extraction of useful information is one way to do this (such as how Meusel et al. did it), other potential approaches require further study.

3.6 Global Community Structure

Exploring dense clusters or communities in networks has been the focus of a lot of recent research [24, 31, 32, 37]. Recently, the label propagation algorithm [32] has received considerable attention due to the fact that it gives high-quality and stable communities, is very scalable, and is also easy to implement and parallelize. The general label propagation algorithm is as follows: each vertex is initially assigned a unique label. For some number of iterations, each vertex updates its label by picking the label that appears most frequently among all of its immediate neighbors. Ties are broken randomly.

For our label propagation implementation, we use an approach similar to the constrained label propagation implementation by Meyerhenke et al. [27]. We ignore directivity of edges and consider them all undirected; labels can propagate in either direction. For storing labels within one hop of the vertices owned by a given task, we use a hash map with vertex-label key-value pairs, similar to our PageRank implementation. We store owned vertices in an array. Due to the presence of several vertices with tens of millions of neighbors, the computationally expensive part of the algorithm during the implementation is determining the maximal label count over all neighbors. In the worst case, each neighbor has a unique label, and so the storage requirements are bounded by twice the number of neighbors (note that we

Table 2: The top 10 web pages according to different centrality indices (* Harmonic, PageRank centrality rankings are approximate).

Out-degree	In-degree	PageRank*	Harmonic*
photoshare.ru/..	www.youtube.com	www.youtube.com	wordpress.org
dvderotik.com/..	wordpress.org	www.youtube.com/t/..	twitter.com
www.zoover.be/..	www.youtube.com/t/..	www.youtube.com/testtube	twitter.com/privacy
cran.r-project.org/..	www.youtube.com/..	www.youtube.com/t/..	twitter.com/about
cran.rakanu.com/..	www.youtube.com/t/..	www.youtube.com/t/..	twitter.com/tos
www.linkagogo.com/..	www.youtube.com/..	www.tumblr.com	twitter.com/account/..
www.cran.r-project.org/..	www.youtube.com/t/..	www.google.com/intl/en/..	twitter.com/account/..
www.fussballdaten.de/..	gmpg.org/xfn/11	wordpress.org	twitter.com/about/resources
www.fussballdaten.de/..	www.google.com	www.google.com/intl/..	twitter.com/login
www.fussballdaten.de/..	www.google.com/intl/..	www.google.com	twitter.com/about/contact

need to store both labels and counts). This can be an issue in a multi-threaded environment where dozens of threads are concurrently tracking counts and each could potentially require up to this amount of storage. As a result, we need to limit the number of concurrent threads running on a given node during this portion in practice. The execution times we will present in the next section will be a reflection of this.

Table 3 gives resultant communities obtained after running our label propagation algorithm for 10 (top) and 30 (bottom) iterations on the web crawl. We give the number of vertices in the community, the number of intra-community edges, and well as the number of cut edges. These results are produced from separate runs, but demonstrate the previously-observed stability of communities produced from label propagation [25], as we note high similarity between the two lists (this is especially apparent with the Tumblr, WordPress, and TripAdvisor communities).

The biggest difference we observe when increasing the number of iterations of label propagation is that the communities become denser, and the intra-community versus inter-community edge ratio increases. Additionally, it is possible that large communities end up merging. Notably, it appears the two largest communities from the 10 iteration run would have eventually combined in the subsequent iterations, likely a result of the high number of outgoing edges from the `www.google.com/intl/en/..` community, quite possibly into the `www.youtube.com` community.

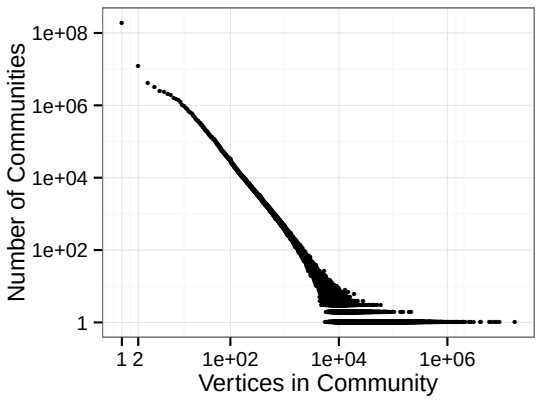

Figure 4: Frequency plot of community structure

Table 3: The top 10 communities ordered by vertex count, as given by our clustering output. The top half shows the list after 10 iterations, and the bottom list is after 30 iterations. m_{in} denotes intra-cluster edges, m_{cut} denotes edge cut. All values are rounded to millions.

n_{in}	m_{in}	m_{cut}	Representative vertex
57	1600	30	www.youtube.com
55	46	440	www.google.com/intl/en/..
17	370	400	www.tumblr.com
13	383	226	www.amazon.com
9	515	84	creativecommons.org/..
7	176	426	wordpress.org/..
5	38	194	www.flickr.com/..
4	120	147	www.google.com
4	281	18	tripadvisor.com
1	19	30	gmpg.org/xfn/
112	2126	32	www.youtube.com
18	548	277	www.tumblr.com
9	516	84	creativecommons.org/..
8	186	85	wordpress.org/..
7	57	83	www.amazon.com
6	41	21	www.flickr.com/..
6	39	58	askville.amazon.com
4	133	142	www.google.com
4	280	18	tripadvisor.com
3	78	13	www.househunt.com

We also plot the frequency of community sizes produced after 30 iterations of label propagation on the web crawl in Figure 4. This plot has a striking similarity to the frequency plots of in-degree, out degree, WCC, and SCC in Meusel et al. [26]. This lends further notion to the consistent and possibly heavy-tailed structural characteristic that have been observed previously in web crawls.

4. PERFORMANCE ANALYSIS

Table 5: Summary of memory, communication, and computational requirements for each task during execution of each of the six analytics.

Analytic	Memory	CommVol	CompVol
SCC	$(3m\delta_e + n)/p$	$2m\delta_c/p$	$2m\delta_e/p$
WCC	$(6m\delta_e + n)/p$	$2m\delta_c/p$	$2m\delta_e/p$
KC	$(3m\delta_e + n)/p$	$km\delta_c/p$	$km\delta_e/p$
PR	$(2m\delta_e + n)/p + m\delta_c/p$	$im\delta_c/p$	$im\delta_e/p$
HC	$(3m\delta_e + n)/p$	$m\delta_c/p$	$m\delta_e/p$
LP	$(4m\delta_e + n)/p + m\delta_c/p + td_{max}$	$i2m\delta_c/p$	$i2m\delta_e/p$

In this section, we report the execution times for our six implemented analytics, which include WCC, SCC, approximate k-core decomposition, PageRank, harmonic centrality, and label propagation. We reiterate that fine tuning the performance of any single algorithm was not our primary goal with this effort. However, all our hand-coded implementations are reasonably efficient for the algorithmic choices we made. This is demonstrated from the performance of these large kernels as shown below. We could implement any of the optimizations listed in Section 5, including some of our work such as 2D layouts for BFS or PageRank calculation, and improve each of these analytics. However, as a first step, we wish to observe global trends from running different analytics on this large-scale irregular and highly skewed graph, and, in turn, open up possible discussions about the direction for future research efforts. We believe that attempts at getting end-to-end analyses running in parallel using reasonably efficient implementations for graphs of such scale has not been done before. We hope this first step will motivate research in reducing the end-to-end times further. In the rest of this section, we report times of the six different analytics on Blue Waters.

Table 5 gives an overview of the theoretical per-task performance and memory requirement (we don't consider actual graph storage costs here) bounds for all of our implemented algorithms. In the table, m is the number of directed edges in the graph, n is the number of vertices, p is the number of tasks, δ_e is the edge imbalance among tasks, δ_c is the task edge cut imbalance, d_{max} is the maximal vertex degree, k is the maximal approximate k-core value, and i is the number of fixed iterations for PageRank and label propagation. We differentiate between δ_c and δ_e to more accurately capture the difference in upper theoretical bounds for aggregate per-task computational volume (number of edges, PageRanks, and labels read) and communication volume (number of adjacent vertices, PageRanks, and labels sent to other tasks).

We observe memory requirements are quite consistent, dominated by the two or three queues each of the algorithms used for send and receive communications and controlling current iteration workloads. For algorithms that consider both in and out edges (WCC and LP), we have an additional scaling of $2\times$. The n/p length arrays are used to store per-vertex information for each analytic, such as visitation status, distance from root, label, or PageRank. PageRank and Label Propagation require additional hash maps bounded by the one hop neighborhoods and size of the maximal degree vertex. Communication volumes are bounded by the total number of cut edges, and are dependent on the number of iterations being performed for each algorithm scaled by the edge cut imbalance for each task. Computational volumes are similar, but we consider the edge imbalance instead of edge cut imbalance. Using the bounds given in Table 5, we can expect better scalability while running the analyses on the $G(n, p)$ graph due to the δ_e and δ_c terms becoming trivial relative to the web crawl for even a moderate number of tasks. A difference of over $2\times$ for communication and computation requirements between the graphs can be expected. We can also expect PageRank, label propagation, and approximate k-core decompositions to have much higher execution times relative to the other analytics from the additional iterations terms on the computation and communication volumes.

We report the total execution time as well as the percent of the total time spent in the communication phase of each algorithm in Table 4. We run PageRank for 20 iterations and label propagation for 10 iterations. The time for harmonic centrality is the execution time for the vertex with the largest in degree and subsequently largest harmonic centrality magnitude. We report times on both the full web crawl as well as on a $G(n, p)$ random graph with approximately the same number of vertices, edges, and average degree.

When we observe the performance for our SCC and WCC implementations, we see good strong-scaling on the random graph, but the scalability is limited on the web crawl due to work and communication imbalance among tasks. As our implementations use synchronous communications, almost all tasks remain idle when waiting for the longest-running task to complete. However, for end-to-end execution and I/O time, we note that our current code is still able to retrieve the largest SCC or WCC from each graph on 256 nodes in under six minutes on Blue Waters. As harmonic centrality and approximate k-core decomposition algorithms have similar expected performance bounds, we observe characteristics as with WCC and SCC. We again note limited scalability due to computation and communication load imbalance for the web crawl, while the implementations strong scale quite well on the random graph. Future research directions of large scale real-world graph analytics should investigate edge-based partitions and more effective intra-task work distribution methods to overcome this. Hybrid partitions where high degree vertices use an edge-based partition and others use a vertex based partition also need to be investigated.

We next make some interesting observations about our PageRank implementation. This implementation especially suffers due to communication load imbalances. The unbalanced maximal per-part edge cuts limit scalability, as communications of almost all updated PageRank values are necessary for each iteration, and, since we perform multiple iterations and synchronizations, the general scalability of our implementation on the web crawl is limited relative to the ran-

Table 4: Summary of all analytics with total execution time and % communication time on 256 and 512 for the web crawl and a synthetic random graph.

	Web Crawl				GNP			
	256 nodes		512 nodes		256 nodes		512 nodes	
Analytic	Exec Time	% Comm	Exec Time	% Comm	Exec Time	% Comm	Exec Time	% Comm
WCC	131	34%	133	44%	84	12%	44	23%
SCC	148	25%	222	22%	88	15%	48	26%
Approx. K-core	1374	30%	1376	38%	731	12%	408	23%
PageRank	930	91%	900	95%	1210	30%	510	38%
Harmonic Cent.	92	34%	110	48%	41	12%	24	24%
Label Prop.	1438	70%	1425	76%	2406	23%	1382	25%

dom graph. We note that over 90% of total execution time is spent on the `Alltoallv` collective communications that distribute the updated PageRank values, and future implementations should attempt to balance the number of cut edges, or maximal per-part edge cut, of the graph part owned by any single task. Surprising, however, is how much more computation time is required on the random graph. This is explained by the fact that, although the random graph has better load balance, a majority of its PageRank lookups are done through the hash map rather than the array storing local vertex values. This is due to the inherently poor locality of random graphs. As a result, we believe methods such as edge partitions based up on vertex partitions [6] would be much more competitive than 2D random distributions.

We finally look at our label propagation performance results. Due to per-thread memory requirements that scale with the largest maximal in+out degree in the graph, we are unable to run a fully threaded implementation that utilizes all the available hardware threads on the web crawl due to memory constraints. We limit the maximum worker threads to 8 for these results. Similar to PageRank, we incur considerable communication costs for pushing label updates during each iteration when running on the web crawl. Running label propagation on the random graph incurs the previously observed high computation overheads due to the poor locality of the random graph and relatively high proportion of hash map accesses.

Overall, we note that the performance numbers observed in practice from the web crawl and random graph align quite well with the expected bounds given in Table 5. In general, these performance results demonstrate the importance of future efforts to minimize the δ_e and δ_c terms to ensure proper work balance for large and highly-skewed networks.

5. RELATED WORK

There are plenty of frameworks or libraries for different graph analytics on various platforms (supercomputers, multithreaded architectures, distributed clusters, accelerators, flash memory) implemented on top of different technologies (MPI, Sockets, OpenMP, CUDA, Hadoop) suitable for the target users. This section covers some of the work closest to this paper in terms of the technologies used and in terms of the scale of the graphs we have analyzed. We point out the largest analytics on supercomputers, distributed and shared-memory graph frameworks first. We also cover some related work on each of the analytic mentioned in Section 3.

The introduction of the Graph500 benchmark [28] has increased the relevance of supercomputers for graph data analytics. The top ten of the most recent Graph500 list are all supercomputers designed for general problems [1] similar to the way we use it. However, while the focus of Graph500 is on one benchmark application (BFS) with a synthetic graph (R-MAT). This has lead to renewed interest in the race for the achievable GTEPS and algorithms for BFS/R-MAT related problems [3, 9, 11–13, 29]. All these work focused on scaling the Graph500 benchmark or similar algorithm on various synthetic graphs. They vary on the data layout (1D vs 2D), handling of high degree vertices, communication layer and the BFS algorithm used (direction-optimizing vs traditional). We note that before this paper, these works corresponded to the analysis of some of the largest created synthetic graphs. In contrast, we analyze a real world graph of similar size and with richer analytics here.

Orthogonal to the above efforts that are focused on using supercomputers for Graph 500, there are a number of graph frameworks for graph analytics in distributed memory systems such as Giraph, PowerGraph, GraphLab, GraphX, SociaLite and FlashGraph [14, 18, 19, 22, 23, 35, 42]. Satish et al. [34] compare a subset of these frameworks along with other codes such as CombBLAS [8] and Galois [30]. They also compare these frameworks to their native code which is hand-optimized and demonstrate that the performance gap between native code and using a framework is huge: 2-30× in most cases but as high as 300× in some cases. The largest real world graph in their study is the Twitter graph with 61M vertices and 1.4B edges. For this particular graph, the hand optimized version takes ∼3 seconds for an iteration of the PageRank and ∼5 seconds for the BFS using four nodes. To provide some context, our implementation takes 45 and 70 seconds for an iteration of PageRank and BFS, respectively, using 256 nodes for the webscale graph, which is roughly 100× larger in term of the number of edges. Note that the frameworks mentioned above differ in different ways. CombBLAS [8] uses MPI as the programming paradigm and treats graphs as sparse matrices with 2D layouts (or edge-based distribution). Giraph [14] is based on Hadoop, Socialite [35] is built using Datalog and Galois [30] is built for shared-memory systems. We point out just the native implementation's performance as it has been shown that the native implementation performs better [34]. Flashgraph [42] is one framework that is not included in that study but has interesting similarities to our work. It uses (relatively) fast external memory to store large graphs and

"out-of-core"-like analytic algorithms. They have been able to run the large web-crawl used in this paper in a semi-external memory mode in order to calculate some of the analytics in Section 3. In later work, they also filter this graph and use the filtered graph for community detection on fewer active vertices [40]. In contrast, we use the entire graph in our analytic experiments, including the community detection algorithms, and we store the graph in-memory. Furthermore, we compute analytics like the approximate K-core decomposition that were not computed using Flashgraph.

In addition to the above frameworks, there are a number of different frameworks for shared-memory systems. Ligra, Galois, STINGER are all multithreaded frameworks [15, 30, 36] for graph analytics with different algorithmic capabilities. In the past, we have implemented some of the analytics described here in shared-memory architectures, such as the connected-components algorithms, and showed better performance than frameworks such as ligra [38]. More recently, we have also showed that the portable implementation of these algorithms are also possible in GPUs and Xeon Phis [39]. We could use a lot of these ideas on the shared memory portion of this work. However, pure shared memory analytics cannot handle massive graphs such as the one we are primarily targeting in this work.

Next we describe related work to some of the analytics we used in Section 3. The problem of PageRank is a well-studied one. There are implementations in different frameworks and comparisons of different frameworks are also available [34]. Other than the frameworks, it has been shown algorithms such as PageRank depend heavily on the data layout [6, 41]. However, computing these layouts are much more expensive than the combined cost of the analytics here and are left to future work.

The strongly connected components algorithm is one of the toughest algorithm to effectively parallelize. The two traditional parallel algorithms are what is called forward-backward [17] and color-propagation [2]. In the past work, we have demonstrated a combination of these two approaches is the best approach for real world graphs [38, 39]. The algorithm used in this work generalizes this approach to distributed memory (Section 3).

Scalable algorithms for community detection have generated a lot of interest in the recent years. We have implemented community detection algorithms such as label propagation in the shared memory architectures and used it for partitioning very large graphs [37]. The label propagation algorithm described in Section 3 is a generalization of this approach to distributed memory. Que et al. [31] achieve nearly one GTEPS with 4K nodes (with RMAT graphs) in their implementation of the Louvain algorithm while using a custom communication runtime. The largest real world graph they used has 39M vertices and 936M edges. There are also recent studies in community detection algorithms using shared memory systems [4, 24, 33]. The largest graphs in Lu et al. [24] are limited to 674M edges which takes ∼200 seconds using 32 threads. Reidy et al. [33] demonstrated performance of 261M edge graph on Cray XMT2 (285 secs) and Intel Xeon server (33s). In comparison to previous work our target graph is nearly three orders of magnitude larger in number of edges and our label propagation implementation takes ∼1400 seconds (or 1.78 GTEPS).

There is also work on all these architectures with other analytics of interest. Chakaravarthy et al. [10] focuses on single source shortest path algorithms, especially focusing on the scalability of the approach while handling high-degree vertices. They demonstrated 40 GTEPS on 1024 nodes with a graph of 63M vertices and 1.8B edges.

6. CONCLUSION

In this paper, we focused on an in-memory, end-to-end analysis of the largest publicly available web crawl graph. We were able to run a multitude of analytics and report their results for the first time on some of the analytics, such as PageRank on the page level graph and label propagation-based community identification. We were also able to match the results some of the other analytics with the previous results (from external memory frameworks). We will make the entire results of community structure and connected components publicly available to the community to best understand this massive dataset.

We further narrowed our focus on end-to-end analytics with the fewest available nodes that are required to hold the graph in memory to demonstrate the relevance of small clusters or few supercomputer nodes on webscale data analytics. We believe we have successfully demonstrated that, with a turnaround time in the order of a few minutes with 256 compute nodes. This work lays the foundation for future research in three different direction. A performance portable compression method will allow us to run the analytics in-memory with a much smaller footprint. We could also scale each step of this end-to-end solution using algorithmic techniques for each one of the analytics, as mentioned in Section 5, to further improve upon the reported baseline performance. We could also utilize the technologies such custom communication runtimes and algorithmic ideas from the Graph 500 improvements to gain another constant factor improvement. Beyond all that, the key message is one can read, construct and analyze a web-scale graph for multiple meaningful analytics with few compute nodes in the time it takes to get a pizza delivered!

7. ACKNOWLEDGMENTS

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also supported by NSF grants ACI-1253881, CCF-1439057, and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

8. REFERENCES

[1] The graph 500 list. *Graph500. http://www.graph500.org/results_nov_2014*, 2015.

[2] J. Barnat and P. Moravec. Parallel algorithms for finding SCCs in implicitly given graphs. *Formal Methods: Applications and Technology*, 4346:316–330, 2006.

[3] S. Beamer, A. Buluc, K. Asanovic, and D. Patterson. Distributed memory breadth-first search revisited: enabling bottom-up search. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1618–1627. IEEE, 2013.

[4] S. Bhowmick and S. Srinivasan. A template for parallelizing the louvain method for modularity maximization. In *Dynamics On and Of Complex Networks, Volume 2*, pages 111–124. Springer, 2013.

[5] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *WWW'04*, pages 595–602. ACM, 2004.

[6] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 50. ACM, 2013.

[7] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer networks*, 33(1):309–320, 2000.

[8] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, page 1094342011403516, 2011.

[9] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[10] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[11] F. Checconi and F. Petrini. Massive data analytics: The graph 500 on ibm blue gene/q. *IBM Journal of Research and Development*, 57(1/2):10–1, 2013.

[12] F. Checconi and F. Petrini. Traversing trillions of edges in real-time: Graph exploration on large-scale parallel machines. In *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[13] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. Choudhury, and Y. Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12, Nov 2012.

[14] A. Ching and C. Kunz. Giraph: Large-scale graph processing infrastructure on Hadoop. *Hadoop Summit*, 6(29):2011, 2011.

[15] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[16] P. Erdös and A. Rényi. On random graphs. i. *Publicationes Mathematicae*, pages 290–297, 1959.

[17] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *Parallel and Distributed Processing*, volume 1800 of *LNCS*, pages 505–511. Springer Berlin Heidelberg, 2000.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2012.

[19] 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 Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[20] O. Lehmberg, R. Meusel, and C. Bizer. Graph structure in the web: aggregated by pay-level domain. In *Proceedings of the 2014 ACM conference on Web science*, pages 119–128. ACM, 2014.

[21] D. Lemire, L. Boytsov, and N. Kurz. Simd compression and the intersection of sorted integers. *arXiv: 1401.6399*, 2014.

[22] 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. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[23] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1408.2041, 2014.

[24] H. Lu, M. Halappanavar, A. Kalyanaraman, and S. Choudhury. Parallel heuristics for scalable community detection. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1374–1385. IEEE, 2014.

[25] S. Mandala, S. Kumara, and T. Yao. Detecting alternative graph clusterings. *Physical Review E*, 86(1):016111, 2012.

[26] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. Graph structure in the web — revisited: A trick of the heavy tail. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, WWW Companion '14, pages 427–432, 2014.

[27] H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning complex networks via size-constrained clustering. *CoRR*, abs/1402.3281, 2014.

[28] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[29] R. Pearce, M. Gokhale, and N. M. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2013.

[30] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The tao of

parallelism in algorithms. *ACM Sigplan Notices*, 46(6):12–25, 2011.

[31] A. Que, F. Checconi, F. Petrini, T. Wang, and W. Yu. Lightning-fast community detection in social media: A scalable implementation of the louvain algorithm. Technical report, 2013.

[32] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[33] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader. Parallel community detection for massive graphs. In *Parallel Processing and Applied Mathematics*, pages 286–296. Springer, 2012.

[34] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 979–990. ACM, 2014.

[35] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 278–289. IEEE, 2013.

[36] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146. ACM, 2013.

[37] G. M. Slota, K. Madduri, and S. Rajamanickam. Pulp: Scalable multi-objective multi-constraint partitioning for small-world networks. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 481–490. IEEE, 2014.

[38] G. M. Slota, S. Rajamanickam, and K. Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 550–559. IEEE, 2014.

[39] G. M. Slota, S. Rajamanickam, and K. Madduri. High-performance graph analytics on manycore processors. In *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2015 (To Appear).

[40] H. Wang, D. Zheng, R. C. Burns, and C. E. Priebe. Active community detection in massive graphs. *CoRR*, abs/1412.8576, 2014.

[41] A. Yoo, A. H. Baker, R. Pearce, et al. A scalable eigensolver for large scale-free graphs using 2d graph partitioning. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 63. ACM, 2011.

[42] D. Zheng, D. Mhembere, R. C. Burns, and A. S. Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proc. USENIX Conf. on File and Storage Technologies (FAST'15)*, 2015.