2016 IEEE International Parallel and Distributed Processing Symposium

A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization

George M. Slota
Computer Science and Engineering
The Pennsylvania State University
University Park, PA
gslota@psu.edu

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

Kamesh Madduri
Computer Science and Engineering
The Pennsylvania State University
University Park, PA
madduri@cse.psu.edu

***Abstract*—In recent years, a large number of graph processing frameworks have been introduced, with their goal to simplify analysis of real-world graphs on commodity hardware. Additionally, the Graph500 benchmark has motivated extensive optimization of fundamental graph computations such as breadth-first search and shortest paths on leading high-performance computing systems. The purpose of this current work is to bridge the gap between these two research areas: we introduce a methodology for graph processing that is simple to implement, and yet offers high performance when scaling up from a single compute node up to several thousand nodes. We develop a compact and efficient graph representation, implement several graph analytics, and describe a number of optimizations that can be applied to these analytics. We test our implementations on the 2012 Web Data Commons hyperlink graph with 3.56 billion vertices and 128.7 billion edges, and perform scalability studies up to 4096 nodes of the Blue Waters supercomputer. On 256 nodes of Blue Waters, we demonstrate execution of six graph analytics on this large hyperlink graph in about 20 minutes.**

***Index Terms*—graph analysis; hyperlink graphs; distributed-memory processing.**

I. Introduction

The growth of the Internet along with online social networks has motivated considerable interest in the study of large-scale and irregular graphs. Understanding the structure of such graphs, of any scale, has many uses. Research in the areas of social and web graph analysis has led to creation of several software tools designed to make computational analysis easier and faster, be it through simplified programming models, abstraction of parallelism, exploiting latent graph structure, or large graph processing capability on commodity hardware. Examples of software include those targeting large-scale distributed and cloud platforms, such as Pregel [17], Giraph [6], Trinity [27], PowerGraph [10], PowerLyra [5], PEGASUS [13], GraphX [11], and others, as well as those designed for highly-efficient shared or external memory processing, such as Galois [23], Ligra [28], STINGER [8], and FlashGraph [34].

In this paper, we study optimization of a collection of graph computations from a high-performance computing perspective. While our main goal is to provide an informed commentary on the efficiency, scalability, and ease of implementation of graph analytics on current high-end computing systems, we also demonstrate how the same techniques we apply to high-end systems can also accelerate analytics at a smaller scale. Focusing on one of the largest publicly-available hyperlink graphs (the 2012 Web Data Commons graph[1], which was in-turn extracted from the open Common Crawl web corpus[2]), we develop parallel implementations for the *Blue Waters* supercomputer, one of the world's most powerful computing platforms. After demonstrating scalability on up to 65,536 cores of *Blue Waters*, we then show how our implementations, without any changes, outperform state-of-the-art frameworks on a small cluster and with smaller test graph instances.

There is a lot of current work on graph algorithms, software frameworks, and graph data management systems. The following aspects motivate our 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. Instead, we want to simultaneously analyze performance of multiple analytics on the largest publicly-available real-world graph instances, and make decisions about graph data representations and decomposition strategies based on our findings.
- Synthetic graphs can never fully substitute real-world graph instances for performance evaluations. One of our goals is to identify challenges in implementing a collection of analytics running on a large real graph, while identifying optimizations are common to multiple algorithms.
- Algorithms that are commonly used for graph analytics have per-iteration operation counts, memory requirements, and communication costs that asymptotically scale linearly with the number of edges. The constant factors in the implementations are what lead to large performance gaps. Linear work computations should thus be evaluated on the largest-available graphs.

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

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.93

- Outside of work on external and semi-external memory frameworks, I/O costs are often ignored when benchmarking graph analytics. An end-to-end evaluation of multiple analytics, considering I/O, memory, and network costs, would be more representative of real-world performance.

This paper presents a methodology for in-memory, end-to-end analysis of large publicly-available data sets. We consider parallel I/O, graph construction, and running multiple useful analytics. The analytics considered determine the global connectivity of the graph, rank vertices based on two centrality measures, and identify dense clusters. We describe our implementations in detail, and discuss performance optimizations that are common to several of these analytics.

We also claim that graph analysis at large-scale concurrencies need not be daunting. All the implementations discussed in this paper amount to about 2000 C++ source lines and use only MPI and OpenMP for parallelization, with no other external library dependencies. Each analytic itself is only around 200 lines of code. If one makes informed algorithmic and data structure choices, end-to-end execution times can be quite low. Using just 256 compute nodes of Blue Waters, we are currently able to perform all six implemented analytics in about 20 minutes, and this includes graph I/O and preprocessing. Using our analytics, we are additionally able to obtain new insights into the global structure of the large hyperlink graph.

II. Experimental Setup and Data

We primarily use the NCSA *Blue Waters* supercomputer for our large scale graph analysis. *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 AMD Interlagos 6276 processors; we do not use the GPU-accelerated XK7 nodes in this work. 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 Sandia National Labs *Compton* testbed. Each node of *Compton* is a dual-socket system with 64 GB main memory and Intel Xeon E5-2670 (Sandy Bridge) CPUs at 2.60 GHz and 20 MB last-level cache running RHEL 6.1.

In Table I, we summarize the list of graphs used in this study. The 2012 Web Data Commons hyperlink data is available at three levels of aggregation: at page level (labeled Web Crawl or WC), at the granularity of subdomains or hosts (labeled Host), and at the granularity of pay-level-domain (labeled Pay). We primarily report results on the page level graph in this paper, and use the smaller graphs for performance comparisons to other software. For additional large-scale experiments, we generate R-MAT [3] and Erdös-Rényi random graphs (labeled Rand-ER) of the same size as Web Crawl. For weak scaling results, we generate R-MAT and Rand-ER graphs on different sizes, keeping the average degree constant. We also use the LiveJournal and Google graphs from SNAP [15], [32] and a Twitter crawl [2]. We do not preprocess or prune the graphs in any manner, and we use the vertex identifiers as given in the original source (or as generated for synthetic graphs).

TABLE I
Real-world and synthetic graphs used.

Graph	n	m	d_{avg}	Source
Web Crawl (WC)	3.56 B	128.7 B	36	[20]
R-MAT	3.56 B	129 B	36	[3]
Rand-ER	3.56 B	129 B	36	
R-MAT	2^{25}-2^{32}	2^{29}-2^{36}	16	[3]
Rand-ER	2^{25}-2^{32}	2^{29}-2^{36}	16	
Host	89 M	2.0 B	22	[20]
Pay	39 M	623 M	16	[20]
Twitter	53 M	2.0 B	38	[2]
LiveJournal	4.8 M	69 M	14	[15]
Google	875 K	5.1 M	5.8	[15]

III. Design, Implementation, and Optimization

We describe our end-to-end methodology in this section, using the Web Crawl (WC) as our running example of the graph being analyzed. The topological properties of WC influence our overall design choices. We first discuss the three related steps of data ingestion, vertex and edge set partitioning, and graph construction. We then describe our implementations of six graph analytics that use the in-memory distributed graph as input. To simplify presentation and to identify commonalities, we further categorize the algorithms for these six analytics to be either "BFS-like" or "PageRank-like".

A. Data ingestion

We assume the input data is available as an unsorted list of edges. Each directed edge can be represented using two 32-bit unsigned integers. Further, we assume the integers are stored on disk in a single file in binary format. We ingest edges in parallel, so that each task reads a contiguous portion of the file and approximately the same number of edges. To achieve high read bandwidth from *Blue Waters*' shared Lustre-based scratch file system, we stripe the input file across multiple storage units. In case of WC, the input file size is nearly 1 TB.

We choose a memory-efficient "one-dimensional" graph representation in this work (more details in Section III-C). Each MPI task "owns" a subset of vertices, and all of the incoming and outgoing edges from these vertices. After each MPI task reads a chunk of edges, the edges are redistributed to be in vertex identifier order using an MPI $\mathrm{Alltoallv}$ call. To store incoming edges, the order of edges is reversed and another exchange is performed. Each task will then have all of the outgoing and incoming edges for the vertices it owns, and these edge arrays are then converted to a compressed sparse row (CSR)-like representation. The data ingestion and graph creation stage is the most memory-intensive part of our implementation. To hold the outgoing edge list in memory and to use the MPI $\mathrm{Alltoallv}$ collective, we require $24m$ bytes of aggregate main memory.

B. Partitioning Strategies

With a one-dimensional partitioning, we would ideally want each task to own approximately the same number of vertices and edges, and also the ratio of internal edges (edges between owned vertices) to external edges (edges to vertices owned by another task) to be as high as possible. The aggregate number of external edges is commonly called the *edge cut*. Modern high quality partitioners, such as (Par)METIS [14] or KaFFPa [21], cannot process graphs of the size of WC [30]. We consider three simple one-dimensional partitioning strategies in this work. We implement *vertex block* partitioning, where each task gets $\frac{n}{p}$ vertices distributed in natural (or some computed) ordering, *edge block* partitioning, where each task gets approximately $\frac{m}{p}$ edges again distributed with some ordering, and *random* partitioning, where each vertex is randomly assigned to a task. In general, there can be significant edge imbalance among tasks with vertex block partitioning, and vertex imbalance with edge block partitioning. This can make the time to complete computational stages highly variable between tasks, which results in increased idle time at synchronization points for all tasks except the one with the highest work. For random partitioning, there is generally a reasonable balance among tasks (though this can be dependent on the graph's degree skew and the number of tasks). However, random partitioning suffers from an inherent lack of intra-task and inter-task locality, which increases computation load within a task and communication load among tasks.

C. Distributed Graph Representation

For our distributed graph representation, we have two primary goals: compactness in memory and fast access to any task-local graph information. Table II gives an overview of the primary structural information we store.

In several graph computations, tasks repeatedly access and update per-vertex data associated with local and *ghost vertices*. Ghost vertices are vertices adjacent to local vertices but outside a task's local domain. Having each task store this data in an n_global length array is not scalable, so a common strategy is to use a hash map. To avoid accessing a slow hash map and using arrays instead, we relabel all locally owned and ghost vertices. Local vertices are relabeled to be from 0 to (n_loc-1), while ghost vertices assume labels from n_loc to (n_loc+n_gst). This relabeling is done to the out- and in-edge arrays. Each task can thus store and access any vertex information in an (n_loc+n_gst)-length array.

To look up the local label for any global vertex identifier, such as when receiving a message from a neighboring task, we utilize a fast linear-probing hash map (map[global_id] = local_id). To map from local vertex ids to global vertex ids, such as when sending a message about per-vertex information to a neighboring task, we have the unmap array (unmap[local_id] = global_id). Finally, we also have an array of length n_gst that stores the owner task for each ghost vertex. Although with simple block partitioning, we can easily calculate this task using the global_id on-the-fly, for more complex partitioning or reordering scenarios, we are required to hold this information.

TABLE II
DISTRIBUTED GRAPH REPRESENTATION.

Data	Size	Description
n_global	1	Global vertex count
m_global	1	Global edge count
n_loc	1	Task-local vertex count
n_gst	1	Ghost vertex count
m_out	1	Task-local out-edges count
m_in	1	Task-local in-edges count
out_edges	m_out	Array of out-edges
out_indexes	n_loc	Start indices for local out-edges
in_edges	m_in	Array of in-edges
in_indexes	n_loc	Start indices for local in-edges
map	n_loc+n_gst	Global to local id hash table
unmap	n_loc+n_gst	Array for local to global id conv.
tasks	n_gst	Array storing owner of ghost vertices

D. Graph Analytics: Implementation and Optimization

We implement six graph methods that could possibly be used in real-world settings for analyzing massive graphs that are similar to WC. Since WC is a directed graph with possibly many strongly connected components, we implement a routine to extract the largest strongly connected component. This routine is labeled SCC. Another related analytic is determining the edges and vertices that belong to the largest weakly connected component (WCC). When determining the largest WCC, we ignore the edge directivity and consider vertex connectivity through both incoming and outgoing edges. We also implement two centrality measures, PageRank (labeled PR) and Harmonic Centrality [1] (labeled HC). We use the power iteration method for computing PageRank. This is an iterative method with the stopping criterion dependent on a user-defined tolerance setting on error. Note that when discussing performance results, we report running times for a single iteration of PageRank. Each PageRank iteration has an operation count that is linear in the number of graph edges. Computing the harmonic centrality of a single vertex also has an operation count that is linear in the number of edges. Determining the harmonic centrality of all the vertices is thus prohibitively expensive for large graphs. For WC, we compute the harmonic centrality scores for the top 1000 vertices ranked by their vertex degree. When reporting performance results, we only give the time taken to compute the harmonic centrality score of a single vertex. The fifth analytic we implement is *approximate k-core computation*. A k-core in a graph is a maximal connected subgraph in which all vertices have degree k or higher. If a vertex belongs to an l-core, but not an $l+1$-core, it is said to have a *coreness* of l. Using this approximate k-core analytic, we determine an upper bound for the coreness of every vertex in the graph in the following manner: we iteratively remove vertices that have degree less than 2^i, i ranging from 1 to 27, and determine the largest connected component in the pruned graph. The value 2^i thus gives a coreness upper bound for all vertices in the component. Our sixth analytic is an implementation of the Label Propagation community detection method [25]. This is also an iterative method and the stopping criterion is typically user-defined. We report the running time of a single iteration in the results section.

To simplify presentation, we categorize our implementations of the six analytics into two distinct but closely related classes. In the first class, the dominant communication phase can be viewed as all vertices in the graph propagating per-vertex information to all of their neighbors. We consider our implementation of PageRank as the prototypical example for this class. Our Label Propagation implementation, using an approach similar to Meyerhenke et al. [21], can also be considered to be belonging to the first class of computation. We ignore directivity of edges, and labels can propagate in either direction.

The second class are approaches that are based on expanding a *frontier* of vertices. Frontier expansion requires accesses to adjacency lists of vertices. These algorithms involve also exchange and propagate per-vertex information, but the frontier may be sparse and need not hold all the vertices in the graph. Algorithms of the second class can be considered to be close variants of the level-synchronous algorithm for breadth-first search (BFS). Our algorithms for the SCC (using the Forward-Backward method [9]), Harmonic Centrality, and approximate k-core analytics belong to this class. For WCC, we do a distributed-memory parallelization of the Multistep [31] algorithm. This algorithm has stages belonging to both classes.

Both of the aforementioned classes can be considered as instantiations of computations that follow a triply-nested loop structure. The outer loop is for a small number of iterations, the middle loop is over all vertices in the graph or over a smaller set of frontier vertices, and the innermost loop is to process adjacencies of vertices [29]. This structure is amenable to bulk-synchronous parallel implementations.

1) Tuning PageRank-like computational phases: We will now describe in more detail our implementations of PageRank, Label Propagation, and the second phase of WCC. This class of methods propagate some stored per-vertex values to all neighbors in every iteration. For PageRank, this value would be the current vertex PageRank score, sent to all of its neighbors through outgoing edges. For Label Propagation and WCC, this would be current vertex label or color passed to both in- and out-edge neighbors. Our implementations of these algorithms follow a similar pattern. We demonstrate this pattern using Label Propagation as the example in Algorithm 1. The algorithm is run on each MPI task, with G denoting the task-local distributed graph, and V and E denoting the local vertex and edge sets.

We initially pass per-vertex information in two queues. One queue holds the global vertex ids being passed ($vSend$) and the other queue holds the associated labels for each vertex ($lSend$). We use MPI Alltoallv collectives in lieu of explicit sends and receives for simplicity. Alltoallv requires calculating the number of items being passed to each task ($NumSend$), as well as the offsets in the send queues for where the items

Algorithm 1 Distributed Label Propagation

```
1: procedure LABELPROP-DIST($G(V, E), \delta$)
2:     $nprocs \leftarrow$ numTasksMPI()
3:     $procid \leftarrow$ localTaskNum()
4:     $NumSend[1 \dots nprocs] \leftarrow 0$
5:     for all $v \in V$ thread parallel do
6:         $ToSend[1 \dots nprocs] \leftarrow false$
7:         for all $u \in E(v) \cup E'(v)$ do
8:             $t \leftarrow$ getTask($u$)
9:             if $t \neq procid$ and $ToSend[t] \neq true$ then
10:                $ToSend[t] \leftarrow true$
11:                $NumSend[t] \leftarrow NumSend[t] + 1$
12:    $SendOffs[1 \dots nprocs] \leftarrow$ prefixSums($NumSend$)
13:    $SendOffsCpy \leftarrow SendOffs$
14:    for all $v \in V$ thread parallel do
15:        $Labels[v] \leftarrow$ initLabel($v$)
16:        $ToSend[1 \dots nprocs] \leftarrow false$
17:        for all $u \in E(v)$ do
18:            $t \leftarrow$ getTask[$u$]
19:            if $t \neq procid$ and $ToSend[t] \neq true$ then
20:                $ToSend[t] \leftarrow true$
21:                $vSend[SendOffsCpy[t]] \leftarrow$ getGlobalId($v$)
22:                $lSend[SendOffsCpy[t]] \leftarrow Labels[v]$
23:                $SendOffsCpy[t] \leftarrow SendOffsCpy[t] + 1$
24:    $vRecv \leftarrow$ Alltoallv($vSend, NumSend, SendOffs$)
25:    $lRecv \leftarrow$ Alltoallv($lSend, NumSend, SendOffs$)
26:    for $i = 1$ to $|vRecv|$ thread parallel do
27:        $vIndex \leftarrow$ getLocalId($vRecv[i]$)
28:        $vRecv[i] \leftarrow vIndex$
29:        $Labels[vIndex] \leftarrow lRecv[i]$
30:    for $i = 1$ to $niter$ do
31:        for all $v \in V$ thread parallel do
32:            $lMap \leftarrow \varnothing$
33:            for all $u \in E(v) \cup E'(v)$ do
34:                $lMap(Labels[u]) \leftarrow lMap(Labels[u]) + 1$
35:            $Labels(v) \leftarrow$ getMaxLabelCount($lMap$)
36:        for $i = 1$ to $|vSend|$ thread parallel do
37:            $lSend[i] \leftarrow Labels[vSend[i]]$
38:        $lRecv \leftarrow$ Alltoallv($lSend, NumSend, SendOffs$)
39:        for $i = 1$ to $|vRecv|$ thread parallel do
40:            $Labels[vRecv[i]] \leftarrow lRecv[i]$
41:    return $Labels$
```

for each task begin ($SendOffs$). To initialize the queues, we therefore need to loop over all local vertices and edges to first count the numbers of items being sent. We utilize a boolean array ($ToSend$) while examining the edges of each vertex ($v \in V$) to track which tasks we are already going to send our vertex information to. We then compute prefix sums on the per-task counts to create the $SendOffsCpy$ offsets array. In the second loop, we calculate the initial labels for each vertex (which would just be the global vertex identifier in this instance), and then place that label as well as the global id of the vertex into the actual send queues using a copy of the offsets array ($SendOffsCpy$) where we increment the current offset after each item placement. For PageRank and WCC, we would be initializing and sending PageRank scores and colors in lieu of labels. We perform thread-based parallelization over these loops. This increases the code complexity, and we will describe this in more detail in Section III-D3.

Once the second loop over all vertices and edges completes, we perform the initial sends and receives into buffers for vertices ($vRecv$) and labels ($lRecv$). We then update all local labels based on the vertices and associated values in the queue. When we first examine $vRecv$, the vertices are in the buffer as global ids. We first must convert them back to their local ids using the hash map in our distributed graph representation. Since this implementation passes all labels on each iteration, we perform two optimizations. We first cut the size of data being sent in half for each iteration by retaining the vertex queue and only updating and sending the label queues. Since we don't update the vertex queue, we simply replace the global ids with local ids in $vRecv$. This avoids multiple relatively costly hash map accesses on subsequent iterations. By retaining queues, we also avoid having to completely rebuild them on each iteration. Experimentally, a certain cutoff can be determined based on the number of label updates for when it would be better in terms of the computation-communication trade-off to switch from retaining the queues to rebuilding them on each iterations.

The bulk of time during the run of the algorithm is spent in the main loop, shown at line 30 of Algorithm 1 (a stopping criterion other than a fixed iteration count is also common). This loop has four main phases. First, we update the labels for each local vertex (v) by utilizing a hash map ($lMap$) that tracks the counts (as values) of each unique label (as keys) among all adjacencies of v. We update v's label to the key of whatever holds the maximum value in $lMap$ with getMaxLabelCount($lMap$), which is simply the label that appears most frequently among v's neighbors (ties are broken randomly). Second, we loop over the size of retained send queues to update the labels in $lSend$. We then exchange updated labels among all tasks. We finally loop to update the labels of the current task's ghost vertices with those received in $lRecv$ using the retained local ids in $vRecv$. All of these three primary loops are straightforward to parallelize in shared memory, and we observe consistent and high speedups when doing so. Our WCC and PageRank algorithms follow an identical pattern to this, with the primary difference being how we initialize and calculate updates for the per-vertex values.

2) Tuning BFS-like computational phases: The second class of implemented algorithms we consider do not explicitly pass per-vertex information, but instead create a global queue of vertices, with all per-vertex updates happening locally on each task. Updates might be changes to visited status, levels in a BFS tree, or some other data. We demonstrate the general outline used for our implementations in Algorithm 2 with a BFS that tracks distances from a root vertex. While a lot of recent work has focused on optimizing distributed BFS [4], [22] in the context of the Graph500 [12] benchmark, we omit BFS-specific optimizations in our current work and focus on those generalizable to all of the algorithms we are considering.

As Algorithm 2 demonstrates, we use task-local queues (Q) to track vertices in each iteration. The queue is initialized first with the root vertex for the MPI task that owns it. We use an array ($Status$) to determine whether a vertex is visited (as needed for WCC). $Status$ is also used to hold the distances from the root for each visited vertex (as needed for harmonic centrality). The $Status$ value is set to -2 initially, and updated to -1 when the vertex has been visited. This first update is done to signify that the vertex has either been added to

Algorithm 2 Distributed BFS

```
1: procedure BFS-DIST($G(V, E), root$)
2:     $nprocs \leftarrow$ numTasksMPI()
3:     $procid \leftarrow$ localTaskNum()
4:     for all $v \in V$ thread parallel do
5:         $Status[v] \leftarrow -2$
6:     $level \leftarrow 0$
7:     $globalSize \leftarrow 1$
8:     if getTask($root$) $= procid$ then
9:         $Q \leftarrow$ Insert($root$)
10:    while $globalSize \neq 0$ do
11:        $NumSend[1 \dots nprocs] \leftarrow 0$
12:        for all $v \in Q$ thread parallel do
13:            if $Status[v] \geq 0$ then
14:                continue
15:            else
16:                $Status[v] \leftarrow level$
17:            for $u \in E(v)$ do
18:                if $Status[u] = -2$ then
19:                    $Status[u] \leftarrow -1$
20:                    $t \leftarrow$ getTask($u$)
21:                    if $t = procid$ then
22:                        $Q_{next} \leftarrow$ Insert($u$)
23:                    else
24:                        $Q_{send} \leftarrow$ Insert($u$)
25:                        $NumSend[t] \leftarrow NumSend[t] + 1$
26:        $SendOffs[1 \dots nprocs] \leftarrow$ prefixSums($NumSend$)
27:        $SendOffsCpy \leftarrow SendOffs$
28:        for all $v \in Q_{send}$ thread parallel do
29:            $t \leftarrow$ getTask($v$)
30:            $vSend[SendOffsCpy[t]] \leftarrow$ getGlobalId($v$)
31:            $SendOffsCpy[t] \leftarrow SendOffsCpy[t] + 1$
32:        $vRecv \leftarrow$ Alltoallv($vSend, NumSend, SendOffs$)
33:        for $i = 1$ to $|vRecv|$ thread parallel do
34:            $Q_{next} \leftarrow$ getLocalId($i$)
35:        swap($Q, Q_{next}$)
36:        $globalSize \leftarrow$ Allreduce($|Q|$, SUM)
37:        $level \leftarrow level + 1$
```

the local queue for the next level, or the send queue for a neighboring task, so the exploration of subsequent edges incident on the vertex don't end up re-queuing that vertex. If the vertex is local, the $Status$ value will be updated to the current $level$ on the next iteration. We track the number of vertices to send to each neighboring task ($NumSend$) during exploration, and then create a send queue and perform an Alltoallv collective at the end of the iteration, similar to how we create the original queue with PageRank. We also track the cumulative size of all queues over all tasks ($globalSize$) and use it as a stopping criteria for the algorithm. Mpte that queues are reset after every iteration (not shown in pseudocode).

3) MPI+OpenMP-based Parallelism: Due to overheads associated with distributed-memory graph processing, which tend to increase with larger task counts, using all available intra-node parallelism is extremely important for maximizing performance. We use OpenMP-based shared-memory parallelism to fully utilize all available cores, while keeping MPI task counts as low as possible. The biggest concern doing so is the additional thread synchronization overhead.

All of our implementations use some form of a queue for passing updates between tasks. Having multiple threads concurrently writing to a shared queue increases contention. To reduce thread contention and use of atomics, we implement parallel thread-local queues. In this section, we discuss how

Algorithm 3 Distributed Label Propagation initialization demonstrating OpenMP thread queuing.

```
1: procedure LABELPROP-DIST-THR($G(V,E),\delta$)
2:     $nprocs \leftarrow$ numTasksMPI()
3:     $procid \leftarrow$ localTaskNum()
4:     ...
5:     begin parallel
6:     $vSend_t[QSIZE] \leftarrow \varnothing$
7:     $lSend_t[QSIZE] \leftarrow \varnothing$
8:     $NumSend_t[1 \dots nprocs] \leftarrow 0$
9:     for all $v \in V$ thread parallel do
10:        $Labels[v] \leftarrow$ initLabel($v$)
11:        $ToSend[1 \dots nprocs] \leftarrow false$
12:        for all $u \in E(v) \cup E'(v)$ do
13:            $t =$ getTask($u$)
14:            if $t \neq procid$ and $ToSend[t] \neq$ true then
15:                $ToSend[t] \leftarrow$ true
16:                $vSend_t \leftarrow$ getGlobalId($v$)
17:                $lSend_t \leftarrow Labels[v]$
18:                $NumSend_t[t] \leftarrow NumSend_t[t] + 1$
19:                $count \leftarrow count + 1$
20:                if $count > QSIZE$ then
21:                    for $i = 1$ to $nprocs$ do
22:                        atomic capture
23:                          $\mathit{off}_t \leftarrow SendOffsCpy[t] \mathrel{+}= NumSend_t[t]$
24:                        $NumSend_t[t] \leftarrow \mathit{off}_t - NumSend_t[t]$
25:                    for $i = 1 \dots count$ do
26:                        $tt \leftarrow$ getTask($vSend_t[i]$)
27:                        $vSend[NumSend_t[tt]] \leftarrow vSend_t[i]$
28:                        $lSend[NumSend_t[tt]] \leftarrow lSend_t[i]$
29:                        $NumSend_t[tt] \leftarrow NumSend_t[tt] + 1$
30:                    $NumSend_t[1 \dots nprocs] \leftarrow 0$
31:                    $count \leftarrow 0$
32:     for $i = 1$ to $nprocs$ do
33:         atomic capture
34:           $\mathit{off}_t \leftarrow SendOffsCpy[t] \mathrel{+}= NumSend_t[t]$
35:         $NumSend_t[t] \leftarrow \mathit{off}_t - NumSend_t[t]$
36:     for $i = 1$ to $count$ do
37:         $tt \leftarrow$ getTask($vSend_t[i]$)
38:         $vSend[NumSend_t[tt]] \leftarrow vSend_t[i]$
39:         $lSend[NumSend_t[tt]] \leftarrow lSend_t[i]$
40:         $NumSend_t[tt] \leftarrow NumSend_t[tt] + 1$
41:     $NumSend_t[1 \dots nprocs] \leftarrow 0$
42:     $count \leftarrow 0$
43:     barrier
44:     ...
45:     return $Labels$
```

they are used during the initialization of the $vSend$ and $lSend$ queues for Label Propagation in Algorithm 3. This method is generalizable for updating queues in all of our algorithms.

In Algorithm 3, we assume first that $NumSend$, $SendOffs$, and $SendOffsCpy$ have already been initialized. We then initialize thread-owned queues $vSend_t$ and $lSend_t$ that will be used to temporarily hold updates to the $vSend$ and $lSend$ queues. The size of these thread-owned queues ($QSIZE$) is a parameter that we can tune based on the cache hierarchy. We also have another thread-owned array $NumSend_t$ that holds per-task counts for the current contents of the threads' queues. When a thread fills its queue, it pushes updates to the global queues. First, it does the equivalent of an atomic fetch-and-add to get an offset (off_t) for the location in the send queue for each task. We can just overwrite the values in $NumSend_t$ to hold these offsets instead of creating another offsets array. A then goes through the contents of its queues, placing the vertices and per-vertex labels into the task-level queues starting at the proper offsets. When a thread finishes all of its work during the main loop, it empties whatever remains of its queue in the same way. This straightforward strategy can improve cache performance and greatly decrease synchronization costs when using multiple levels of queues.

IV. Performance Results

A. End-to-end Analytic Execution Time

We first report performance of the data ingestion and graph creation steps in Table III. The *Read* time in Table III lists the I/O read time for Web Crawl data when varying the number of compute nodes on *Blue Waters*, and using a single MPI task per node. These times correspond to read bandwidths between 20-30 GB/s. The read time is under a minute in all cases. 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. Slowdown from a single task due to network traffic and concurrent file system accesses ends up having a lower effect on total time. Table III also shows the running time in seconds for exchanging both outgoing and incoming edges (*Exch*), as well as the time to create the final distributed CSR representation (*LConv*) when varying the numbers of tasks. We note a degree of strong scaling with increasing task count. We also include in Table III a performance rate in billions of edges processed per second (GE/s), corresponding to the total number of edges processed (129 billion in- and 129 billion out-edges).

TABLE III
Parallel performance of graph construction stages.

#	Time (s)				Perf	Speedup
Nodes	*Read*	*Excg*	*LConv*	*Total*	Rate (GE/s)	
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×

Table IV gives execution times of all the analytics for the WC graph on 256 nodes of *Blue Waters*, and with the various partitioning strategies (vertex block: WC-np, edge block: WC-mp, random: WC-rand). We also include times for the R-MAT and Rand-ER graphs of the same size. We just use vertex block partitioning for the synthetic graphs. We note that the end-to-end execution times for both the vertex and edge block partitioning strategies (WC-np and WC-mp) are about 20 minutes, when including I/O and preprocessing. The k-core (27 iterations of BFS) and Label Propagation (10 iterations) routines have longer running times due to multiple iterations. Label Propagation is data access-intensive due to the repeated hash map accesses in its inner loop. However, both these analytics still finish in under 10 minutes under all three partitioning scenarios for WC. PageRank times are for 10 iterations. The synthetic graphs take longer to perform Label Propagation due to lack of good locality in either graph. Additionally, the R-MAT input suffers from load imbalance.

TABLE IV
EXECUTION TIMES ON 256 NODES OF *Blue Waters*.

Analytic	WC-np	WC-mp	WC-rand	R-MAT	Rand-ER
PageRank	87	111	227	125	121
Label Propagation	400	435	367	993	992
WCC	88	63	112	68	77
Harmonic Centrality	54	46	101	252	84
k-core	445	363	583	579	481
SCC	184	108	184	89	83

B. Weak and Strong Scaling

We run two sets of experiments to evaluate scalability of our approaches. We first examine how our implementations perform under weak scaling on synthetic graphs. Figure 1 gives the execution time of harmonic centrality and PageRank on R-MAT and Rand-ER graphs. The R-MAT graphs have an average degree of 16. The number of compute nodes is varied from 8 to 1024, with 2^{22} vertices per node in each case. We assume a simple vertex block partitioning for these tests. We observe that our harmonic centrality code scales extremely well on the Rand-ER graphs until 1024 nodes, where the communication times for collective operations begins to increase. We note that performance on the R-MAT graph does not scale quite as well due to communication and work imbalances introduced by high-degree vertices. Our PageRank implementations scales moderately well on both graph instances.

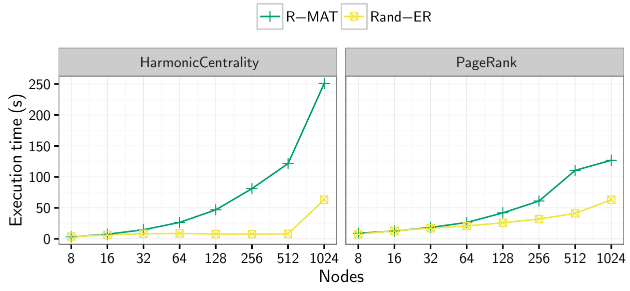

Fig. 1. Harmonic Centrality and PageRank weak scaling results with R-MAT and Rand-ER graphs on 8 to 1024 nodes of *Blue Waters*.

Figure 2 gives performance results of our Label Propagation implementation when scaling up from 256 to 4096 nodes of *Blue Waters*. We report speedup in comparison to 256-node running times. We look at the scaling of WC with various partitioning strategies, as well on as the synthetic R-MAT and Rand-ER graphs of similar size as WC. We note that Label Propagation scales well on the synthetic graphs. The best performance on WC is when using the random partitioning strategy. The loss of performance with the block partitioning strategies at high node counts is due to load imbalance. In further evaluations, we noted good scaling for our PageRank implementation, but limited strong scaling with our BFS-like algorithms, due to a greater number of global synchronizations and a lower computation to communication ratio.

To better examine impact of the partitioning scheme on scalability, we break overall execution time into three com-

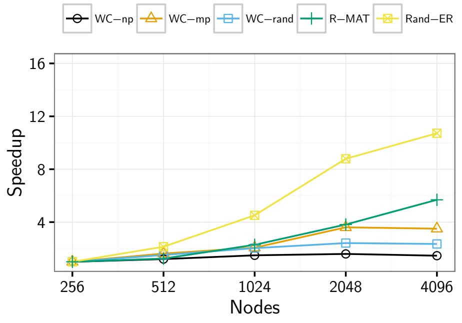

Fig. 2. Label Propagation strong scaling with WC, R-MAT, and Rand-ER graphs and 256 to 4096 nodes of *Blue Waters*.

ponents: the time that each task spends in computation, the time that a task is idle waiting for updates from other tasks, and the total time spent in communication. We track the minimum, maximum, and average values of these times across all tasks on 256 to 4096 nodes for PageRank. For consistency, we report ratios instead of actual times by normalizing each component by the total execution time. These results are shown in Figure 3.

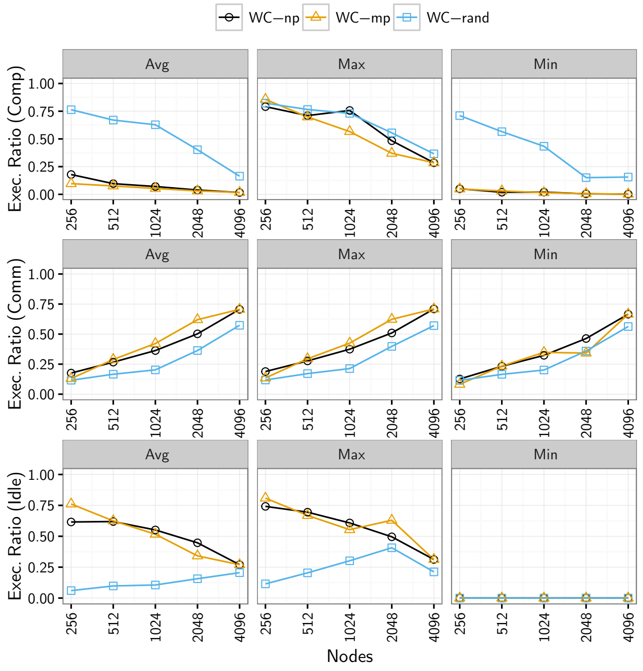

Fig. 3. PageRank per-task execution time ratios of computation, communication, and idle times when scaling from 256 to 4096 nodes on *Blue Waters*.

From Figure 3, we see that the average computational time is very high for the random partitioning input (WC-rand) in comparison to the block partitioning methods. This is due to two reasons. First, we retain native vertex ordering in the block-based strategies, which leads to better intra-node cache performance. Second, the block strategies have a lower relative number of ghost vertices, and therefore require a lower number of total global and local id lookups and hash map accesses. The maximum computation ratios are consistent among partitionings, due to the performance impact of processing high degree vertices. We also note that the proportion of communication time increases with increasing node count, as is expected when strong scaling. The idle times are mostly related to imbalances in computation, as a larger computational imbalance leads to some tasks being delayed at a synchronization or communication point, while waiting for the longest-running task to complete its portion of work. In general, we observe that random partitioning has the lowest average and max idle times, as it inherently has the best work balance. Minimum idle times are near zero, as is expected.

V. Comparison to Prior Work

We will now directly compare the performance of our analytics to the implementations available in graph analysis frameworks. In order to do direct comparisons, we focus on frameworks that use in-memory or semi-external memory graph storage. We compare to GraphX [11] (from Spark v1.5.0), PowerLyra [5] (version 1.0), and PowerGraph [10] (version 2.2). GraphX is Apache Spark API designed for in-memory parallel graph computations. PowerGraph and PowerLyra are both GraphLab derivatives designed to process graphs with skewed degree distributions, such as the web crawls and social networks we test on, efficiently. PowerLyra offers a more advanced partitioning strategy that differentiates between high and low degree vertices. In addition, we also compare to the external memory framework of FlashGraph [34], which can utilize arrays of external SDDs, as it is the only other framework known to us that has reported performance results on WC. The primary goal of our analytic codes is to balance between ease of implementation and computational efficiency, and we use execution time as the comparison metric. Note, however, that the frameworks we compare to may have other goals such as support for fault-tolerant execution, emphasis on programmability, etc.

We perform our comparisons on *Compton*. Figure 4 presents PageRank and WCC performance comparisons on five graphs. Our implementations are labeled as SRM (single node performance is SRM-1, 16-node performance SRM-16). We compare to 16-node performance on GraphX (GX), PowerGraph (PG), and PowerLyra (PL). We also compare our single node performance (SRM-1) to FlashGraph running in both external mode (FG) and main memory-only standalone (FG-SA) mode. Note that we did not use an SSD array for these experiments. Each *Compton* node only has a single SSD attached, and so the performance results for FlashGraph running in external mode may be lower than published results. The numbers might be considered as an indicator of what might typically be expected in a HPC setting without a specialized I/O subsystem. The Host, Pay, Twitter, LiveJournal, and Google graphs are distributed using random partitioning in our implementations. We compare directly to the supplied implementations of PageRank and (weakly) connected components in each of the frameworks. We set up each framework to explicitly utilize all available memory and cores on each node when possible.

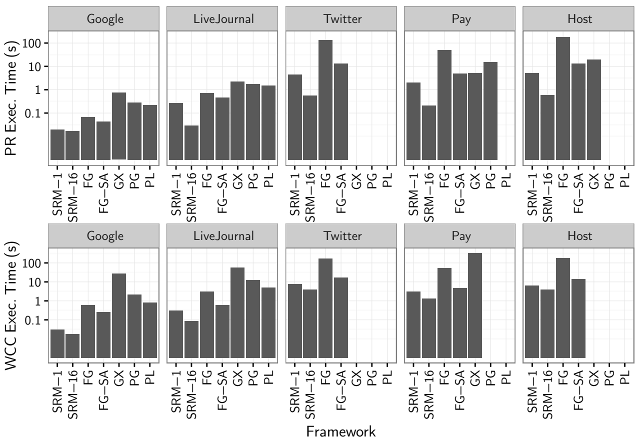

Fig. 4. Performance comparison on *Compton* to implementations from popular graph analytic frameworks.

As can be seen in Figure 4, we observe considerably higher performance with our implementations on both PageRank (top) and WCC (bottom). When running on 16 nodes, GraphX, PowerGraph, and PowerLyra all failed to process some of the larger graphs. None of them were able to process the Twitter graph for either algorithm. The causes of execution failure were memory use-related (mostly out of memory errors). Overall, our PageRank code on 16 nodes is 38× faster (geometric mean) in comparison to other codes, and our WCC code is 201× (geometric mean) faster. Running on a single node, FlashGraph in standalone mode gave results that were comparable to our results, with our code being 2.4× and 2.6× faster relative to FlashGraph-SA, and 12× and 19× relative to external memory FlashGraph, for PageRank and WCC, respectively. We point out that our speedups for WCC are larger than the speedups for PageRank, due to our use of the efficient Multistep algorithm, instead of traditional single-stage WCC approaches.

Further Comparisons: Recently, several published papers have reported performance results on large-scale graphs. The distributed clusters and software configurations used are quite different from our work. We list some notable results here for a general idea. A recent paper [7] describing Facebook's Giraph processing framework ran similar Label Propagation and PageRank implementations on several Facebook snapshots that were similar in size to WC. On a 701 M vertex, 48 B edge graph and 200 compute nodes, this work reports a per-iteration time of 9.5 minutes for a Label Propagation implementation. On a 2 B vertex, 400 B edge graph and 200 compute nodes, a per-iteration time of 5 minutes for PageRank is reported. In comparison, on the 3.56 B vertex and 129 B edge web crawl on 256 nodes, our per-iteration times are 40 seconds and 4.4 seconds, for Label Propagation and PageRank, respectively. In a recent paper on the Trinity graph processing framework [27], 8 compute node times for PageRank and BFS on an R-MAT input graph (SCALE 28, average degree 13) were reported to be 15 seconds (per iteration) and 200 seconds, respectively. Re-running the same experiments on 8 nodes of *Compton*, we observe execution times of 1.5 seconds per iteration for PageRank and a total time of about 32 seconds for BFS. We could not perform a direct comparison to FlashGraph on WC, but the authors report impressive execution times of 461 seconds for WCC, and 68 seconds (per iteration) for PageRank on a test machine with 32 cores and an attached array of 15 SSDs [34]. Our times on 256 nodes were 88 seconds for WCC and 4.4 seconds per iteration for PageRank with vertex block partitioning. Also, other recent work has shown that a considerable performance gap exists between the implementations in popular frameworks and custom, tuned implementations [18], [26].

VI. Web Crawl Analysis

Using our implementations, we were able to explore the structure of WC in detail. We present some new insights obtained in this section.

Finding densely-connected vertices or communities in large networks has been the focus of a lot of recent research [16], [24], [25], [30]. Most notably, Wang et al. [33] utilize FlashGraph to implement a spectral clustering algorithm for analyzing communities of WC. Recently, the Label Propagation algorithm [25] has received considerable attention due to the fact that it gives high-quality and stable communities, is very scalable, and also easy to implement and parallelize. Table V lists some statistics about the largest communities obtained after running our Label Propagation implementation for a fixed number of iterations. We report the number of vertices in the community (n_{in}), the number of intra-community edges (m_{in}), as well as the number of cut edges (m_{out}). We also give the label of one representative vertex in each community. The 10-iteration and 30-iteration results were produced from separate runs, but we observe similar large-scale communities in both the two lists. The biggest difference we see when increasing the number of iterations is that the communities become denser, and the intra-community to inter-community edge ratio increases. Additionally, it is possible that smaller communities end up merging. It appears the two largest communities from the 10-iteration run eventually combined in the subsequent iterations, and this is possibly a result of the large number of outgoing edges from the www.google.com/intl/en/.. community to vertices in the www.youtube.com community.

We plot the size distribution for the communities produced after 30 iterations of Label Propagation on WC in Figure 5. This plot has a striking similarity to the frequency plots of in-degree, out-degree, WCC, and SCC given in Meusel et al. [19]. There are a large number of communities of size 1 and 2.

We also use our approximate k-core implementation to obtain and analyze the coreness values of vertices. In Figure 6, we plot the cumulative fraction of vertices that have a coreness values less than or equal to k. Observe that at least 75% of the vertices have coreness value less than 32. If we remove all vertices of degree 1024 or less, then only 20 million vertices (or about 0.5% of total vertex count) remain connected. The

TABLE V
The top 10 communities ordered by vertex count, as given by our community detection algorithm.

Results after 10 Label Prop. iterations			
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/
Results after 30 Label Prop. iterations			
n_{in}	m_{in}	m_{cut}	Representative vertex
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

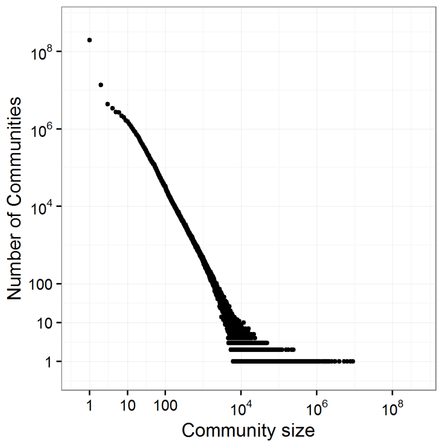

Fig. 5. Frequency plot of community sizes in WC.

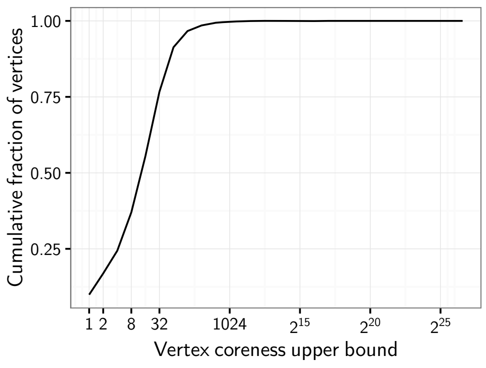

Fig. 6. Vertex coreness distribution in WC.

coreness upper bounds can be refined, if required, to compute exact coreness values for each vertex.

VII. Conclusions

In this paper, we focus on the implementation and optimization of popular graph analytics on HPC systems. We discuss our approaches in detail and describe how they may be extended and applied to other graph problems. We demonstrate that our codes can scale to over 65K cores of a large supercomputer, while also delivering high performance on a single compute node. When compared to existing implementations, we show that some of our codes are up to several orders of magnitude faster. We can also process larger graphs than several other software frameworks, as we use a relatively compact graph representation.

This work lays the foundation for future research in three different directions. We are investigating a performance-portable graph compression method that will allow us to execute graph analytics with an even smaller memory footprint. We are exploring better partitioning strategies to improve load balance and overall scalability. We also plan to extend this collection of analytics with other implementations.

Acknowledgment

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 FAST-Math 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.

References

[1] P. Boldi and S. Vigna, "Axioms for centrality," *Internet Mathematics*, vol. 10, no. 3-4, pp. 222–262, 2014.
[2] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. ICWSM*, 2010.
[3] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SDM*, 2004.
[4] F. Checconi and F. Petrini, "Traversing trillions of edges in real-time: Graph exploration on large-scale parallel machines," in *Proc. IPDPS*, 2014.
[5] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. EuroSys*, 2015.
[6] A. Ching and C. Kunz, "Giraph: Large-scale graph processing infrastructure on Hadoop," *Hadoop Summit*, vol. 6, no. 29, p. 2011, 2011.
[7] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: graph processing at Facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.
[8] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *Proc. HPEC*, 2012.
[9] L. Fleischer, B. Hendrickson, and A. Pinar, "On identifying strongly connected components in parallel," in *Parallel and Distributed Processing*, ser. LNCS. Springer Berlin Heidelberg, 2000, vol. 1800, pp. 505–511.
[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. OSDI*, 2012.
[11] 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 *Proc. OSDI*, 2014.
[12] "The Graph 500 list," http://www.graph500.org/, last accessed Feb 2016.
[13] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. ICDM*, 2009.
[14] G. Karypis and V. Kumar, "Parallel multilevel K-way partitioning scheme for irregular graphs," in *Proc. SC*, 1996.
[15] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.
[16] H. Lu, M. Halappanavar, A. Kalyanaraman, and S. Choudhury, "Parallel heuristics for scalable community detection," in *Proc. IPDPSW*, 2014.
[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. SIGMOD*, 2010.
[18] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST?" in *Proc. HotOS*, 2015.
[19] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web — revisited: A trick of the heavy tail," in *Proc. WWW*, 2014.
[20] ——, "The graph structure in the web analyzed on different aggregation levels," *J. Web Sci.*, vol. 1, no. 1, pp. 33–47, 2015.
[21] H. Meyerhenke, P. Sanders, and C. Schulz, "Partitioning complex networks via size-constrained clustering," in *Proc. ALENEX*, 2014.
[22] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *Proc. IPDPS*, 2013.
[23] 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*, vol. 46, no. 6, pp. 12–25, 2011.
[24] X. Que, F. Checconi, F. Petrini, T. Wang, and W. Yu, "Lightning-fast community detection in social media: A scalable implementation of the Louvain algorithm," Auburn University, Tech. Rep. AU-CSSE-PASL/13-TR01, 2013.
[25] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.
[26] 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 *Proc. SIGMOD*, 2014.
[27] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. SIGMOD*, 2013.
[28] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 135–146, 2013.
[29] G. M. Slota, S. Rajamanickam, and K. Madduri, "High-performance graph analytics on manycore processors," in *Proc. IPDPS*, 2015.
[30] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *Proc. BigData*, 2014.
[31] G. M. Slota, S. Rajamanickam, and K. Madduri, "BFS and coloring-based parallel algorithms for strongly connected components and related problems," in *Proc. IPDPS*, 2014.
[32] "Stanford large network dataset collection," http://snap.stanford.edu/data/index.html, last accessed Feb 2016.
[33] H. Wang, D. Zheng, R. Burns, and C. Priebe, "Active community detection in massive graphs," in *Proc. SDM-Networks*, 2015.
[34] D. Zheng, D. Mhembere, R. C. Burns, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proc. FAST*, 2015.