Ligra: A Lightweight Graph Processing Framework for Shared Memory

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Abstract

There has been significant recent interest in parallel frameworks for processing graphs due to their applicability in studying social networks, the Web graph, networks in biology, and unstructured meshes in scientific simulation. Due to the desire to process large graphs, these systems have emphasized the ability to run on distributed memory machines. Today, however, a single multicore server can support more than a terabyte of memory, which can fit graphs with tens or even hundreds of billions of edges. Furthermore, for graph algorithms, shared-memory multicores are generally significantly more efficient on a per core, per dollar, and per joule basis than distributed memory systems, and shared-memory algorithms tend to be simpler than their distributed counterparts.

In this paper, we present a lightweight graph processing framework that is specific for shared-memory parallel/multicore machines, which makes graph traversal algorithms easy to write. The framework has two very simple routines, one for mapping over edges and one for mapping over vertices. Our routines can be applied to any subset of the vertices, which makes the framework useful for many graph traversal algorithms that operate on subsets of the vertices. Based on recent ideas used in a very fast algorithm for breadth-first search (BFS), our routines automatically adapt to the density of vertex sets. We implement several algorithms in this framework, including BFS, graph radii estimation, graph connectivity, betweenness centrality, PageRank and single-source shortest paths. Our algorithms expressed using this framework are very simple and concise, and perform almost as well as highly optimized code. Furthermore, they get good speedups on a 40-core machine and are significantly more efficient than previously reported results using graph frameworks on machines with many more cores.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Performance

Keywords Shared Memory, Graph Algorithms, Parallel Programming

1. Introduction

There has been significant recent interest in processing large graphs, and recently several packages have been developed for processing such large graphs on parallel machines including the parallel Boost graph library (PBGL) [19], Pregel [34], Pegasus [24], GraphLab [29, 30], PowerGraph [17], the Knowledge Discovery Toolkit [8, 31], GPS [40], Giraph [16], and Grace [39]. Motivated by the need to process very large graphs, most of these systems (with the exception of the original GraphLab [29] and Grace) have been designed to work on distributed memory parallel machines.

In this paper, we study Ligra, a lightweight interface for graph algorithms that is particularly well suited for graph traversal problems. Such problems visit possibly small subsets of the vertices on each step. The interface is lightweight in that it supplies only a few functions, the implementation is simple, and it is fast.

Our work is motivated in part by Beamer et. al.'s recent work on a very fast BFS for shared memory machines [3, 4]. They use a hybrid BFS which uses a sparse representation of the vertices when the frontier is small and a dense representation when it is large. Our interface supports hybrid graph traversal algorithms and for BFS, we achieve close to the same efficiency (time and space) as the optimized BFS of Beamer et. al., and our code is much simpler than theirs. In addition, we apply it to many other applications including betweenness centrality, graph radii estimation, graph connectivity, PageRank and single-source shortest paths.

Ligra is designed for shared memory machines. Compared to distributed memory systems, communication costs are much cheaper in shared memory systems, leading to performance benefits. Although shared memory machines cannot scale to the same size as distributed memory clusters, current commodity single unit servers can easily fit graphs with well over a hundred billion edges in memory[1], large enough for any of the graphs reported in the papers mentioned above.[2] Shared memory along with the existing support for parallel code (CilkPlus [27] in our case) on multicores allows for our lightweight implementation. Furthermore, these multicore servers have sufficient memory bandwidth to get quite good speedups over sequential codes (up to 39 fold on 40 cores in our experiments). Shared memory algorithms tend to be simpler than their distributed counterparts. Unlike in distributed memory, race conditions can occur in shared memory, but as we later show, this can be dealt with in our system with appropriate uses of the atomic compare-and-swap instruction. Compared to the distributed memory systems mentioned above, our system is over an order of magnitude faster on a per-core basis for the benchmarks

PPoPP'13, February 23–27, 2013, Shenzhen, China.

[1] For example, the Intel Sandy Bridge based Dell R910 has 32 cores (64 hyperthreads) and can be configured with up to 2 Terabytes of memory, and the AMD Opteron based Dell R815 has 64 cores and can be configured with up to 1 Terabyte of memory.

[2] The largest graph in the papers cited is a synthetic 127 billion edges in the Pregel paper [34]. The rest of the papers do not use any graphs larger than 20 billion edges. The largest non-synthetic graph described is the Yahoo graph with 6.6 billion directed edges [42].

```
1:  Parents = {−1, . . . , −1}                          ▷ initialized to all -1's
2:
3:  procedure UPDATE(s, d)
4:      return (CAS(&Parents[d], −1 , s ))
5:
6:  procedure COND(i)
7:      return (Parents[i] == −1)
8:
9:  procedure BFS(G, r)                                  ▷ r is the root
10:     Parents[r] = r
11:     Frontier = {r}          ▷ vertexSubset initialized to contain only r
12:     while (SIZE(Frontier) ≠ 0) do
13:         Frontier = EDGEMAP(G, Frontier, UPDATE, COND)
```

Figure 1. Pseudocode for Breadth-First Search in our framework. The compare-and-swap function CAS(*loc*,*oldV*,*newV*) atomically checks if the value at location *loc* is equal to *oldV* and if so it updates *loc* with *newV* and returns *true*. Otherwise it leaves *loc* unmodified and returns *false*.

we could compare with, and typically faster even on absolute terms to the largest systems run, which sometimes have two orders of magnitude more cores. Finally, commodity shared memory servers are quite reliable, often running for up to months or possibly years without a failure.

Ligra supports two data types, one representing a graph $G = (V, E)$ with vertices V and edges E, and another for representing subsets of the vertices V, which we refer to as *vertexSubset*s. Other than constructors and size queries, the interface supplies only two functions, one for mapping over vertices (VERTEXMAP) and the other for mapping over edges (EDGEMAP). Since a vertexSubset is a subset of V, the VERTEXMAP can be used to map over any subset of the original vertices, and hence its utility in traversal algorithms—or more generally in any algorithm in which only (possibly small) subsets of the graph are processed on each round. The EDGEMAP also processes a subset of the edges, which is specified using a vertexSubset to indicate the valid sources, and a Boolean function to indicate the valid targets of each edge. Abstractly, a vertexSubset is simply a set of integer labels for the included vertices and the VERTEXMAP simply applies the user supplied function to each integer. It is up to the user to maintain any vertex based data. The implementation switches between a sparse and dense representation of the integers depending on the size of the vertexSubset. In our interface, multiple vertexSubsets can be maintained and furthermore, a vertexSubset can be used for multiple graphs with different edge sets, as long as the number of vertices in the graphs are the same.

With this interface a breadth-first search (BFS), for example, can be implemented as shown in Figure 1. This version of BFS uses a Parents array (initialized all to -1, except for the root r where Parents$[r] = r$) in which each vertex will point to its parent in a BFS tree. As with standard parallel versions of BFS [28, 41], on each step i (starting at 0) the algorithm maintains a frontier of all vertices reachable from the root r in i steps. Initially a vertexSubset containing just the root vertex is created to represent the frontier (line 11). Using EDGEMAP, each step checks the neighbors of the frontier to see which have not been visited, updates those to point to their parent in the frontier, and adds them to the next frontier (line 13). The user supplied function UPDATE (lines 3–4) atomically checks to see if a vertex has been visited using a compare and swap (CAS) and returns true if not previously visited (Parents$[i] == -1$). The COND function (lines 6–7) tells EDGEMAP to consider only target vertices which have not been visited (here, this is not needed for correctness, but is used for efficiency). The EDGEMAP function returns a new vertex set containing the target vertices for which UPDATE returns true, i.e., all the vertices in the next frontier (line 13). The BFS completes when the frontier is empty and hence no more vertices are reachable.

The interface is designed to allow the edges to be processed in different orders depending on the particular situation. This is different from many of the interfaces mentioned in the first paragraph (e.g. Pregel, GraphLab, GPS and Giraph) which are vertex based and have the user hardcode how to loop over the out-edges or in-edges. Our implementation supports a few different ways to traverse the edges. One way is to loop over each vertex in a sparse representation of the active source vertices applying the function to each out-edge (this is basically the order Pregel, GPS and Giraph supports). This loop over the out-edges can either be parallel or sequential depending on the degree of the vertex (Pregel and the others do not support parallel looping over out-edges, although the most recent version of GraphLab does [17]). A dense representation of the set of source vertices could also be used. Another way to map over the edges is to loop over all destination vertices sequentially or in parallel, and for each in-edge check if the source is in the source vertex set and apply the edge function if so. Finally, we can simply apply a flat map over all edges checking which need to be applied.

In this paper, we apply the Ligra framework to a collection of problems: breadth-first search, betweenness centrality, graph radii estimation, graph-connectivity, PageRank, and Bellman-Ford single-source shortest paths. All of these applications have the property that they work in rounds and each round potentially processes only a subset of the vertices. In the case of BFS, each vertex is only processed once but in the others they can be processed multiple times. For example, in the shortest paths algorithm a vertex only needs to be added to the active vertex set if its distance has changed. Similarly in a variant of PageRank, a vertex needs to be processed only if its PageRank value has changed by more than some delta since it was last processed.

Betweenness centrality, a technique for measuring the "importance" of vertices in a graph, is basically a version of BFS that accumulates statistics along the way and propagates first in the forward direction and then backward direction. In betweenness centrality, one needs to keep around the frontiers during the forward traversal to facilitate the backward traversal. In Ligra, this is easily done by storing the vertexSubsets in each iteration during the forward traversal. In contrast, this cannot be easily expressed in Pregel and GraphLab, because although vertices can be made inactive in Pregel and GraphLab, the state is associated with the vertices as opposed to being separate.

Our contributions are threefold:

1. We provide an abstraction based on edgeMaps, vertexMaps and vertexSubsets for programming a class of parallel graph algorithms.
2. We provide an efficient and lightweight implementation of our framework, and applications using the framework.
3. We provide experimental evaluation of using the framework and timing results of our applications on various input graphs.

The Ligra code and applications can be found at `www.cs.cmu.edu/~jshun/ligra/`.

2. Related Work

Beamer et. al [3, 4] recently developed a very fast BFS for shared-memory machines.They use a hybrid BFS consisting of the conventional top-down approach, where each vertex on the current frontier explores all of its neighbors and adds unvisited neighbors to the next frontier (write-based), and a bottom-up approach, where each unvisited vertex in the graph tries to find any parent (visited vertex) among its neighbors (read-based). While the neighbor vis-

its in the top-down approach will mostly be to unvisited vertices when the frontier is small, for large frontiers many of the edges will be to neighbors already visited. The edges to visited neighbors can be avoided in the bottom-up approach because an unvisited vertex can stop checking once it has found a parent; this makes it more efficient than the top-down approach for large frontiers. The disadvantage of the bottom-up approach is that it processes all of the vertices, so is more expensive than the top-down approach for small frontiers. Beamer et. al.'s hybrid BFS switches between the two approaches based on the size of the frontier, and the representation of the active set of vertices also switches between sparse and dense accordingly. They show that for small-world and scale-free graphs, the hybrid BFS achieves a significant speedup over previous BFS implementations based on the top-down approach. We use this same idea in a more general setting.

Pegasus [24] and the Knowledge Discovery Toolbox (KDT) [15, 31] process graphs by using sparse matrix operations with generalized matrix operations. Each row/column corresponds to a vertex and each non-zero in the matrix represents an edge. Pegasus uses the Hadoop implementation of MapReduce in the distributed-computing setting, and includes implementations for PageRank, random walk with restart, graph diameter/radii, and connected components. It does not allow a sparse representation of the vertices and therefore is inefficient when only a small subset of vertices are active. Also, because it is built on top of MapReduce, it is hard to make it perform well. KDT provides a set of generalized matrix-vector building blocks for graph computations. It is built on top of the the Combinatorial BLAS [8], a lower-level generalized sparse matrix library for the distributed setting. Using the building blocks, the KDT developers implement algorithms for breadth-first search, betweenness centrality, PageRank, belief propagation and Markov Clustering. Since the abstraction allows for sparse vectors as well as sparse matrices, it is suited for the case when only a small number of vertices are active. However, it does not switch representations of the vertex sets based on its density. We give some performance comparisons with both systems in Section 6.

Pregel is an API for processing large graphs in the distributed setting [34]. It is a vertex-centric framework, where vertices can loop over their edges and send messages to all their out-neighbors. These messages are then collected at the target vertex, possibly using associative combining. The system is bulk-synchronous so the received value is not seen until the next round. The reported performance of Pregel is relatively slow, likely due to the overhead of the framework and the use of a distributed memory machine. The GPS [40] and Giraph [16] systems are public source implementations of the Pregel interface with some additional features. The GPS system allows for graph partitioning and reallocation during the computation. This improves performance over Pregel, but only marginally.

GraphLab is a framework for asynchronous parallel graph computations in machine learning. It works in both shared-memory and distributed-memory architectures [29, 30]. It differs from Pregel in that it does not work in bulk-synchronous steps, but rather allows the vertices to be processed asynchronously based on a scheduler. The vertex functions can run at any time as long as specified consistency rules are obeyed. It is therefore well-suited for the machine learning types of applications for which it is defined, where each vertex accumulates information from its neighbors states and updates its state, possibly asynchronously. The recent PowerGraph framework combines the shared-memory and asynchronous properties of GraphLab with the associative combining concept of Pregel [17]. In contrast to our vertexSubset data type, both Pregel and GraphLab assume a single graph, and do not allow for multiple vertex sets, since state is associated with the vertices.

Grace is a graph management system for shared-memory [39]. It uses graph partitioning techniques and batched updates to exploit locality. Updates to the graph are done transactionally. Their reported times are slower than that of our system for applications like BFS and PageRank, after accounting for differences in input size and machine specifications.

GraphChi is a system for handling graph computations using just a PC [26]. It uses a novel parallel sliding windows method for processing graphs from disk. Although their running times are slower than ours, their system is designed for processing graphs out of memory, whereas we assume the graphs fit in memory.

Galois is a graph system for shared-memory based on set iterators [38]. Unlike our EDGEMAP and VERTEXMAP functions, their set iterator does not abstract the internal details of the loop from the user. Their sets of active elements for each iteration must be generated directly by the user, unlike our EDGEMAP which generates a vertexSubset which can be used for the next iteration.

Green-Marl is a domain-specific language for writing graph algorithms for shared-memory [21]. Graph traversal algorithms using Green-Marl are written using built-in breadth-first search (BFS) and depth-first search (DFS) primitives whose implementations are built into the compiler. Their language does not support operations over arbitrary sets of vertices on each iteration of the traversal, and instead the user must explicitly filter out the vertices to skip. This makes it less flexible than our framework, which can operate on arbitrary vertexSubsets. In Green-Marl, for traversal algorithms which cannot be expressed using a BFS or DFS (e.g. radii estimation and Bellman-Ford shortest paths), the user has to write the for-loops themselves. On the other hand, such algorithms are naturally expressed in our framework.

Other high-performance libraries for parallel graph computations include the Parallel Boost Graph Library (PBGL) [19] and the Multithreaded Graph Library (MTGL) [5]. The former is developed for the distributed-memory setting and the latter is developed for massively multithreaded architectures. These libraries provide few higher-level abstractions beyond the graphs themselves.

3. Preliminaries

A variable *var* with type *type* is denoted as *var* : *type*. We denote a function f by $f : X \mapsto Y$ if each $x \in X$ has a unique value $y \in Y$ such that $f(x) = y$. We denote the Cartesian product of sets A and B by $A \times B$ where $A \times B = \{(a, b) : a \in A \wedge b \in B\}$. We define the Boolean value set *bool* to be the set $\{0, 1\}$ (equivalently {*false*,*true*}).

We denote a directed unweighted graph by $G = (V, E)$ where V is the set of vertices and E is the set of (directed) edges in the graph. Graphs have type *graph*, vertices have type *vertex* and edges have type *vertex* $\times$ *vertex*, where the first vertex is the source of the edge and the second the target. We will use the convention of denoting the number of vertices in a graph by $|V|$ and number of edges in a graph by $|E|$. We denote a weighted graph by $G = (V, E, w)$, where w is a function which maps an edge to a real value (w : *vertex* $\times$ *vertex* $\mapsto \mathbb{R}$), and each edge $e \in E$ is associated with the weight $w(e)$. $N^+(v)$ denotes the set of out-neighbors of vertex v in G and $deg^+(v)$ denotes the out-degree of v in G. Similarly, $N^-(v)$ and $deg^-(v)$ denote the in-neighbors and in-degree of v in G.

A ***compare-and-swap*** (CAS) is an atomic instruction that takes three arguments—a memory location (*loc*), an old value (*oldV*) and a new value (*newV*); if the value stored at *loc* is equal to *oldV* it atomically stores *newV* at *loc* and returns *true*, and otherwise it does not modify *loc* and returns *false*. In our implementations, we use CAS's both directly and as a subroutine to other atomic functions, such an as atomic increment. Throughout the paper we use the notation $\&x$ to refer to the memory location of variable x.

4. Framework

4.1 Interface

For an unweighted graph $G = (V, E)$ or weighted graph $G = (V, E, w(E))$, our framework provides a ***vertexSubset*** type, which represents a subset of vertices $U \subseteq V$. Note that V, and hence U, may be shared among graphs with different edge sets. Except for some constructor functions and some optional arguments described in Section 4.4, the following describes our interface.

1. **SIZE**(U: *vertexSubset*) : $\mathbb{N}$.

   Returns $|U|$.

2. **EDGEMAP**(G : *graph*,
   U : *vertexSubset*,
   F : (*vertex* $\times$ *vertex*) $\mapsto$ *bool*,
   C : *vertex* $\mapsto$ *bool*) : *vertexSubset*.

   For an unweighted graph $G = (V, E)$ EDGEMAP applies the function F to all edges with source vertex in U and target vertex satisfying C. More precisely, for an active edge set

   $$E_a = \{(u, v) \in E \mid u \in U \wedge C(v) = \textit{true}\},$$

   F is applied to each element in E_a, and the return value of EDGEMAP is a vertexSubset:

   $$\text{Out} = \{v \mid (u, v) \in E_a \wedge F(u, v) = \textit{true}\}.$$

   In this framework, F can run in parallel, so the user must ensure parallel correctness. F is allowed to side effect any data that it is associated with (and does so when used in the graph algorithms we discuss later), so F, C, E_a and *Out* can depend on order. The function C is useful in algorithms where a value associated with a vertex only needs to be updated once (i.e. breadth-first search). If the user does not need the this functionality, a default function $C_{\textit{true}}$ which always returns true may be supplied.

   For weighted graphs, F takes the edge weight as an additional argument.

3. **VERTEXMAP**(U : *vertexSubset*,
   F : *vertex* $\mapsto$ *bool*) : *vertexSubset*.

   Applies F to every vertex in U. Its returns a vertexSubset:

   $$\text{Out} = \{u \in U \mid F(u) = \textit{true}\}$$

   As with EDGEMAP, the function F can run in parallel.

4.2 Implementation

We index the vertices V of a graph from 0 to $|V| - 1$. A vertexSubset $U \subset V$ is therefore a set of integers in the range $0, \ldots, |V| - 1$. In our implementation this set is either represented sparsely as an array of $|U|$ integers (not necessarily sorted) or as a Boolean array of length $|V|$, $true$ in location i if and only if $i \in U$. For example, for a graph with 8 vertices the sparse representation of a vertex subset $\{0, 2, 3\}$ could be $[0, 2, 3]$ or $[3, 0, 2]$ and the corresponding dense representation would be $[1, 0, 1, 1, 0, 0, 0, 0]$. The implementation of vertexSubset contains routines for converting its sparse representation to a dense representation and vice versa. In the following pseudocode we assume unweighted graphs, but it can easily be extended to weighted graphs. Also we overload notation and use U and Out both to denote subsets of vertices and also to denote the vertexSubsets representing them.

For a given graph $G = (V, E)$, a vertexSubset representing a set of vertices $U \subseteq V$ and functions F and C, the EDGEMAP function (pseudocode shown in Algorithm 1) calls one of **EDGEMAPSPARSE** (Algorithm 2) and **EDGEMAPDENSE** (Algorithm 3) based on $|U|$ and the number of outgoing edges of U (if this quantity is greater than some threshold, it calls EDGEMAPDENSE, and otherwise it calls EDGEMAPSPARSE). EDGEMAPSPARSE loops through all vertices present in U in parallel and for a given $u \in U$ applies $F(u, \text{ngh})$ to all of u's neighbors ngh in G in parallel. It returns a vertexSubset that is represented sparsely. The work performed by EDGEMAPSPARSE is proportional to $|U|$ plus the sum of the out-degrees of U. On the other hand, EDGEMAPDENSE loops through all vertices in V in parallel and for each vertex $v \in V$ it sequentially applies the function $F(\text{ngh}, v)$ for each of v's neighbors ngh that are in U, until $C(u)$ returns *false*. It returns a dense representation of a vertexSubset. For EDGEMAPSPARSE, since a sparse representation of a vertexSubset is returned, duplicate vertex IDs in the output vertexSubset must be removed. Intuitively EDGEMAPSPARSE should be more efficient than EDGEMAPDENSE for small vertexSubsets, while for larger vertexSubsets EDGEMAPDENSE should be faster. The threshold of when to use EDGEMAPSPARSE versus EDGEMAPDENSE is set to $|E|/20$, which we found to work well across all of our applications.

Algorithm 1 EDGEMAP

```
1: procedure EDGEMAP(G, U, F, C)
2:     if (|U| + sum of out-degrees of U > threshold) then
3:         return EDGEMAPDENSE(G, U, F, C)
4:     else return EDGEMAPSPARSE(G, U, F, C)
```

Algorithm 2 EDGEMAPSPARSE

```
1: procedure EDGEMAPSPARSE(G, U, F, C)
2:     Out = {}
3:     parfor each v ∈ U do
4:         parfor ngh ∈ N+(v) do
5:             if (C(ngh) == 1 and F(v, ngh) == 1) then
6:                 Add ngh to Out
7:     Remove duplicates from Out
8:     return Out
```

Algorithm 3 EDGEMAPDENSE

```
1: procedure EDGEMAPDENSE(G, U, F, C)
2:     Out = {}
3:     parfor i ∈ {0, ..., |V| − 1} do
4:         if (C(i) == 1) then
5:             for ngh ∈ N−(i) do
6:                 if (ngh ∈ U and F(ngh, i) == 1) then
7:                     Add i to Out
8:                 if (C(i) == 0) then break
9:     return Out
```

The VERTEXMAP function (Algorithm 4) takes as inputs a vertexSubset representing the vertices U and a Boolean function F, and applies F to all vertices in U. It returns a vertexSubset representing subset Out $\subseteq U$ containing vertices u such that $F(u)$ returns *true*.

Algorithm 4 VERTEXMAP

```
1: procedure VERTEXMAP(U, F)
2:     Out = {}
3:     parfor u ∈ U do
4:         if (F(u) == 1) then Add u to Out
5:     return Out
```

4.3 Graph Representation

Our code represents in-edges and out-edges as arrays. In particular the in-edges for all vertices are kept in one array partitioned by their target vertex and storing the source vertices. Similarly, the out-edges are in an array partitioned by the source vertices and storing the target vertices. Each vertex points to the start of their in-edge and out-edge partitions and also maintains their in-degree and out-degree. Note that EDGEMAPSPARSE only uses the out-edges and

EDGEMAPDENSE only uses the in-edges. To transpose a graph (i.e. switch the direction of all edges), which is needed in betweenness centrality, we swap the roles of the in-edges and out-edges. When a graph is symmetric (or undirected) the in-neighbors and out-neighbors are the same so only one copy needs to be stored. For weighted graphs, the weights are interleaved with the edge targets in the edge array for cache efficiency.

4.4 Optimizations

Here we discuss several optimizations to our interface and implementation. These optimizations affect only performance and not correctness.

Note that EDGEMAPSPARSE applies F in parallel to target vertices (second argument), while EDGEMAPDENSE applies F sequentially given a target vertex. Therefore the F in EDGEMAPDENSE does not need to be atomic with respect to the target vertex. An optimization is for EDGEMAP to accept two version of its function F, the first of which must be correct when run in parallel with respect to both arguments, and the second of which must be correct when run in parallel only with respect to the first argument (source vertex). Both functions should behave exactly the same if EDGEMAP were run sequentially. If this optimization is used, then EDGEMAPSPARSE would use the first version of F as before, but EDGEMAPDENSE would use the second version of F (which we found to be slightly faster for some applications).

The default threshold of when to use EDGEMAPSPARSE versus EDGEMAPDENSE is $|E|/20$, but if the user discovers a better threshold, it can be passed as an optional argument to EDGEMAP.

If the user is careful in defining the F and C functions passed to EDGEMAP to guarantee that no duplicate vertices will appear in the output vertexSubset of EDGEMAP, then the remove-duplicates stage of EDGEMAPSPARSE can be bypassed. Our EDGEMAP function takes a flag indicating whether duplicate vertices need to be removed.

For EDGEMAPDENSE, the inner for-loop is sequential (see Algorithm 3) because the behavior of C may allow it to break early (e.g., in BFS, breaking after the first valid parent is found). If instead the user wants to run the inner for-loop in parallel and give up the option of breaking early, a flag can be passed to EDGEMAP to indicate this.

Since EDGEMAPDENSE is read-based, we also provide a write-based version of EDGEMAPDENSE called **EDGEMAPDENSE-WRITE** (shown in Algorithm 5). This write-based version loops through all vertices in V in parallel and for vertices contained in U it applies F (now required to correct when run in parallel with respect to both arguments) to all of its neighbors in parallel, as in EDGEMAPSPARSE. It returns a dense representation of a vertexSubset. We found EDGEMAPDENSE-WRITE to be more efficient than EDGEMAPDENSE only for two of our applications—PageRank and Bellman-Ford shortest paths. In our framework, the user may pass a flag to EDGEMAP specifying whether to use EDGEMAPDENSE (default) or EDGEMAPDENSE-WRITE when the vertexSubset is dense. The user would need to figure out experimentally which version is more efficient.

Algorithm 5 EDGEMAPDENSE-WRITE

```
1: procedure EDGEMAPDENSE-WRITE(G, U, F, C)
2:     Out = {}
3:     parfor i ∈ {0, ..., |V| − 1} do
4:         if (i ∈ U) then
5:             parfor ngh ∈ N+(i) do
6:                 if (C(ngh) == 1 and F(i, ngh) == 1) then
7:                     Add ngh to Out
8:     return Out
```

For VERTEXMAP, if the user knows that the input and output vertexSubsets are the same, an optimized version of VERTEXMAP that avoids creating a new vertexSubset can be used.

5. Applications

Here we describe six applications of our framework. In the following discussions, the "frontiers" of the algorithms are represented as vertexSubsets.

5.1 Breadth-First Search

A ***breadth-first search*** (BFS) algorithm takes a graph $G = (V, E)$ and a starting vertex $r \in V$, and computes a breadth-first search tree rooted at r containing all nodes reachable from r. A simple parallel algorithm processes each level of the BFS in parallel. The number of iterations required is equal to the (unweighted) distance of the furthest node reachable from the starting vertex, and the algorithm processes each edge at most once. There has been recent work on developing fast parallel breadth-first search algorithms for shared-memory machines [3, 4, 28, 41] and these algorithms have been shown to be practical for many real-world graphs.

In our framework, a breadth-first search implementation is very simple as we described in Section 1. To make the computation more efficient for dense frontiers for which EDGEMAPDENSE is used, we can also provide a version of UPDATE, which is not atomic with respect to d and does not use a CAS. The code for BFS is shown in Figure 1.

5.2 Betweenness Centrality

Centrality indices for graphs have been widely studied in social network analysis because they are useful indicators of the relative importance of nodes in a graph. One such index is the betweenness centrality index [13].

To precisely define the betweenness centrality index, we first introduce some additional definitions. For a graph $G = (V, E)$ and some $s, t \in V$, let σ_{st} be the number of shortest paths from s to t in G. For vertices $s, t, v \in V$, define $\sigma_{st}(v)$ to be the number of shortest paths from s to t that pass through v. Define $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ to be the ***pair-dependency*** of s, t on v. The ***betweenness centrality*** of a vertex v, denoted by $C_B(v)$ is equal to $\sum_{s \neq v \neq t \in V} \delta_{st}(v)$. A naive method to compute the betweenness centrality scores is to perform a BFS starting at each vertex to compute the pair-dependencies, and then sum the pair-dependencies for each $v \in V$. There are $O(|V|^2)$ pair-dependency terms associated with each vertex, hence this method requires $O(|V|^3)$ operations.

Brandes [6] presents an algorithm which avoids the explicit summation of pair-dependencies and runs in $(|V||E|+|V|^2 \log |V|)$ operations for weighted graphs and $O(|V||E| + |V|^2)$ operations for unweighted graphs. Brandes' defines the ***dependency*** of a vertex r on a vertex v as follows:

$$\delta_{r\bullet}(v) = \sum_{t \in V} \delta_{rt}(v) \tag{1}$$

For any given r, Brandes' algorithm computes $\delta_{r\bullet}(v)$ for all v in linear time for unweighted graphs, by using the following two equations, where $P_r(v)$ is defined to contain all immediate parents of v in the BFS tree rooted at r:

$$\sigma_{rv} = \sum_{u \in P_r(v)} \sigma_{ru} \tag{2}$$

$$\delta_{r\bullet}(v) = \sum_{w : v \in P_r(w)} \frac{\sigma_{rv}}{\sigma_{rw}} \times (1 + \delta_{r\bullet}(w)) \tag{3}$$

The algorithm works in two phases: the first phase of the algorithm computes the number of shortest paths from r to each vertex using Equation 2, and the second phase computes the dependency scores

via Equation 3. The first phase is similar to a forward BFS from vertex r and the second phase works backwards from the last frontier of the BFS. This algorithm can be parallelized in two way—(1) for each vertex, the traversal can be done in parallel, and (2) each vertex can perform their individual computations independently in parallel with other vertices' computations. Although much more efficient than the naive algorithm, Brandes' algorithm still requires at least quadratic time, and is thus prohibitive for large graphs. To address this problem, there has been work on computing approximate betweenness centrality scores based on using the pair-dependency contributions from just a sample of the vertices of the vertices and scaling the betweenness centrality scores appropriately [2, 14]. The KDT package provides a parallel implementation of batched computation of betweenness centrality scores by running multiple individual computations independently in parallel [31].

We describe the betweenness centrality computation from a single root vertex—these computations can be run independently in parallel for any sample of the vertices. The computation here is different from the BFS described in Section 5.1 in that instead of finding a parent, each vertex v needs to maintain a count of the number of shortest paths passing through it. This means the number of updates to v is equal to its number of parents in the BFS tree, instead of just one update as in BFS.

The psuedocode for our implementation is shown in Algorithm 6. The frontier is initialized to contain just r. For the first phase we use an array of integers *NumPaths*, which is initialized to all 0's except for the root vertex which has NumPaths$[r]$ set to 1. By traversing the graph in a breadth-first manner and updating the NumPaths value for each v that is traversed, we obtain the number of shortest paths passing through each v from r (NumPaths$[v]$ will remain 0 if v is unreachable from r). The PATHSUPDATE function passed to EDGEMAP is shown in lines 13–18. As there can be multiple updates to some NumPaths$[v]$ in parallel, the update attempt is repeated with a compare-and-swap until successful. Line 18 guarantees that a vertex is placed on the frontier only once, since the old NumPaths value will be 0 for at most one update. Each frontier of the search is stored in a *Levels* array for use in the second phase.

To keep track of vertices that have been visited (and avoid having to remove duplicates in EDGEMAPSPARSE), we also maintain a Boolean array *Visited*. Visited is initialized to all 0's (except for the root vertex whose entry is set to 1), and we set a vertex's entry in Visited to 1 after it is first visited in the computation. To do this, we use a VERTEXMAP and pass the VISIT function shown in lines 9–11 of Algorithm 6 to VERTEXMAP. The COND function in lines 27–28 makes EDGEMAP only consider unvisited target vertices. The psuedocode for the first phase starting at a root vertex is shown in lines 32 to 36.

For the second phase, we use a new array *Dependencies* (initialized to all 0.0) and reuse the Visited array (reinitialized to all 0). Also we transpose the graph (line 40), since edges now need to point in the reverse direction. The algorithm operates on the vertexSubsets in the Levels array returned from the first phase in reverse order, uses the same VISIT and COND functions as in the first phase, and passes the DEPUPDATE function shown in lines 20 to 25 of Algorithm 6 to EDGEMAP. Psuedocode for the second phase of the betweenness-centrality computation is shown in lines 42–46.

5.3 Graph Radii Estimation and Multiple BFS

For a graph $G = (V, E)$, the ***radius*** of a node $v \in V$ is the shortest distance to the furthest reachable node of v. The ***diameter*** of the graph is the maximum radius over all $v \in V$. For unweighted graphs, one simple method for computing the radii of all nodes (and hence the diameter of the graph) is to run $|V|$ BFS's, one starting at each vertex. However, for large graphs this method is impractical

Algorithm 6 Betweenness Centrality

```
 1: NumPaths = {0, ..., 0}                                   ▷ initialized to all 0
 2: Visited = {0, ..., 0}                                    ▷ initialized to all 0
 3: NumPaths[r] = 1
 4: Visited[r] = 1
 5: currLevel = 0
 6: Levels = [ ]
 7: Dependencies = {0.0, ..., 0.0}                           ▷ initialized to all 0.0
 8:
 9: procedure VISIT(i)
10:     Visited[i] = 1
11:     return 1
12:
13: procedure PATHSUPDATE(s, d)
14:     repeat
15:         oldV = NumPaths[d]
16:         newV = oldV + NumPaths[s]
17:     until (CAS(&NumPaths[d], oldV, newV) == 1)
18:     return (oldV == 0)
19:
20: procedure DEPUPDATE(s, d)
21:     repeat
22:         oldV = Dependencies[d]
23:         newV = oldV + NumPaths[d]/NumPaths[s] × (1 + Dependencies[s])
24:     until (CAS(&Dependencies[d], oldV, newV) == 1)
25:     return (oldV == 0.0)
26:
27: procedure COND(i)
28:     return (Visited[i] == 0)
29:
30: procedure BC(G, r)
31:     Frontier = {r}          ▷ vertexSubset initialized to contain only r
32:     while (SIZE(Frontier) ≠ 0) do                        ▷ Phase 1
33:         Frontier = EDGEMAP(G, Frontier, PATHSUPDATE, COND)
34:         Levels[currLevel] = Frontier
35:         Frontier = VERTEXMAP(Frontier, VISIT)
36:         currLevel = currLevel + 1
37:
38:     Visited = {0, ..., 0}                                ▷ reinitialize to all 0
39:     currLevel = currLevel − 1
40:     TRANSPOSE(G)                                         ▷ transpose graph
41:
42:     while (currLevel ≥ 0) do                             ▷ Phase 2
43:         Frontier = Levels[currLevel]
44:         VERTEXMAP(Frontier, VISIT)
45:         EDGEMAP(G, Frontier, DEPUPDATE, COND)
46:         currLevel = currLevel − 1
47:     return Dependencies
```

as each BFS requires $O(|V| + |E|)$ operations, leading to a total of $O(|V|^2 + |V||E|)$ operations (see [11]). This approach can be parallelized by running the BFS's independently in parallel, and also by parallelizing each individual BFS, but currently this is still impractical for large graphs.

There has been work on techniques to estimate the diameter of a graph. Magnien et. al. [33] describe several techniques for computing upper and lower bounds on the diameter of a graph, using BFS's and spanning subgraphs. They describe a method called the *double sweep lower bound*, which works by first running a BFS from some node v and then a second BFS from the furthest node from v (call it w). The radius of w is then taken to be a lower bound on the diameter of the graph. Their method can be repeated by picking more vertices to run BFS's from. Ferrez et. al. [12] perform experiments with parallel implementations of some of these methods. Another approach based on counting neighborhood sizes was described by Palmer et. al. [37]. Their algorithm approximates the neighborhood function for each vertex in a graph, which is more general than computing graph radii. Kang et. al. [23] parallelize this

algorithm using MapReduce. Cohen [10] describes an algorithm for approximating neighborhood sizes, which requires $O(|E| \log |V|)$ expected number of operations for undirected graphs.

We implement the simple method for estimating graph radii by performing BFS's from a sample of K vertices. Its accuracy can be improved by using the double sweep method of Magnien et. al. [33]. Instead of running the BFS's in parallel independently, we run multiple BFS's together. In the ***multiple-BFS*** algorithm, each vertex maintains a bit-vector of length K. Initially K vertices are chosen randomly to act as "source" vertices and each of these K vertices has exactly one unique bit in their bit-vector set to 1; all other vertices have their bit-vectors initialized to all 0's. The K sampled vertices are placed on the initial frontier of the multiple-BFS search. In each iteration, each frontier vertex bitwise-ORs its vector into each of its neighbors' vectors. Vertices whose bit-vectors changed in an iteration are placed on the frontier for the next iteration. The algorithm iterates until none of the bit-vectors change.

For a sample of size K this algorithm simulates running K BFS's in parallel, but without computing the BFS tree (which is not needed for the radii computation). Storing the iteration number in which a vertex v's bit-vector last changed is a lower-bound on the radius of v since at least one of the K sampled vertices took this many rounds to reach v. If K is set to be the number of bits in a word (32 or 64) this algorithm is more efficient than naively performing K individual BFS's in two ways: (1) the frontiers of the K BFS's could overlap in any given iteration and this algorithm stores the union of these frontiers usually leading to fewer edges traversed per iteration and (2) performing a bitwise-OR on bit-vectors can pass information from more than one of the K BFS's while only requiring one arithmetic operation. Note that this algorithm only estimates the diameter of the connected components of the graph which contain at least one of the K sampled vertices; if there are multiple connected components in the graph, one would first compute in parallel the components of the graph and then run the multiple-BFS algorithm in parallel on each component.

To implement the multiple-BFS algorithm in our framework (pseudocode shown in Algorithm 7), we maintain two bit-vectors, *Visited* and *NextVisited*, which are initialized to all 0's, except for the K sampled vertices each of which has a unique bit in their Visited bit-vector set to 1. We also maintain an array *Radii*, which for each vertex stores the iteration number in which the bit-vector of the vertex last changed. It is initialized to all ∞ except for the K sampled vertices which have a Radii entry of 0. At the end of the algorithm, Radii contains the estimated (lower-bound) radius of each vertex, the maximum of which is a lower-bound on the graph diameter. In the pseudocode, we use "|" to denote the bitwise-OR operation. The initial frontier contains the K sampled vertices. The update function RADIIUPDATE passed to EDGEMAP is shown in lines 6–12 of Algorithm 7. ATOMICOR(x, y) performs a bitwise-OR of y with the value stored at x and atomically updates x with this new value. It is implemented using a compare-and-swap. The reason we have both Visited and NextVisited is so that new bits that a vertex receives in an iteration do not get propagated to its neighbors in the same round, otherwise the values in Radii would be incorrect. The compare-and-swap on line 11 guarantees that any Radii entry is updated at most once (and returns *true*) per iteration. Therefore any vertex will be placed at most once on the next frontier, eliminating the need for removing duplicates. As in the other implementations, we can provide a version of RADIIUPDATE non-atomic with respect to d to EDGEMAP.

The ORCOPY function (lines 14–16) passed to VERTEXMAP simply takes an index i, performs a bitwise-OR of NextVisited[i] and Visited[i] and stores the result in NextVisited[i]. We use this because the roles of NextVisited and Visited are switched between iterations. The while loop in lines 22–26 is executed until the entries of the Radii array do not change (or equivalently, none of the bit-vectors change).

Algorithm 7 Radii Estimation

```
 1: Visited = {0, ..., 0}                          ▷ initialized to all 0
 2: NextVisited = {0, ..., 0}                      ▷ initialized to all 0
 3: Radii = {∞, ..., ∞}                            ▷ initialized to all ∞
 4: round = 0
 5:
 6: procedure RadiiUpdate(s, d)
 7:     if (Visited[d] ≠ Visited[s]) then
 8:         atomicOR(&NextVisited[d], Visited[d] | Visited[s])
 9:         oldRadii = Radii[d]
10:         if (Radii[d] ≠ round) then
11:             return CAS(&Radii[d], oldRadii, round)
12:     return 0
13:
14: procedure ORcopy(i)
15:     NextVisited[i] = NextVisited[i] | Visited[i]
16:     return 1
17:
18: procedure Radii(G)
19:     Sample K vertices and for each one set a unique bit in Visited to 1
20:     Initialize Frontier to contain the K sampled vertices
21:     Set the Radii entries of the sampled vertices to 0
22:     while (size(Frontier) ≠ 0) do
23:         round = round + 1
24:         Frontier = edgeMap(G, Frontier, RadiiUpdate, C_true)
25:         Frontier = vertexMap(Frontier, ORcopy)
26:         swap(Visited, NextVisited)          ▷ switch roles of bit-vectors
27:     return Radii
```

5.4 Connected Components

For an undirected graph $G = (V, E)$, a connected component $C \subseteq V$ is one in which all vertices in C can reach one another. The ***connected components*** problem is to find $C_1, \ldots, C_k$ such that each C_i is a connected component, $\bigcup_i C_i = V$, and there is no path between vertices belonging to different components.

One method of computing the connected components of a graph is to maintain an array *IDs* of size $|V|$ initialized such that IDs[i] $= i$, and iteratively have every vertex update its IDs entry to be the minimum IDs entry of all of its neighbors in G. The total number of operations performed by this algorithm is $O(d(|V|+|E|))$ where d is the diameter of G. For high-diameter graphs, this algorithm can perform much worse than standard edge-based algorithms which require only $O(|V|+|E|)$ operations [11, 22], but for low-diameter graphs it runs reasonably well. We show this algorithm as a simple application of our framework.

The pseudocode for our implementation is shown in Algorithm 8. The initial frontier contains all vertices in V. In addition to the IDs array, we maintain a second array *prevIDs* (used to check whether a vertex has been placed on the frontier in a given iteration yet), and pass the CCUPDATE function shown in lines 4–8 of Algorithm 8 to EDGEMAP. WRITEMIN(x, y) atomically updates the value at location x to be the minimum of x's old value and y, and is implemented with a compare-and-swap. It returns *true* if the value at location x was changed, and *false* otherwise. Line 7 places a vertex on the next frontier if and only if its ID changed in the iteration. To synchronize the values of prevIDs and IDs after every iteration, we pass the COPY function to VERTEXMAP. The while loop in lines 16–18 is executed until IDs remains the same as prevIDs. When the algorithm terminates, all vertices in the same component will have the same value stored in their IDs entry.

Algorithm 8 Connected Components

```
1:  IDs = {0, ..., |V| − 1}                    ▷ initialized such that IDs[i] = i
2:  prevIDs = {0, ..., |V| − 1}                ▷ initialized such that prevIDs[i] = i
3:
4:  procedure CCUPDATE(s, d)
5:      origID = IDs[d]
6:      if (WRITEMIN(&IDs[d], IDs[s])) then
7:          return (origID == prevIDs[d])
8:      return 0
9:
10: procedure COPY(i)
11:     prevIDs[i] = IDs[i]
12:     return 1
13:
14: procedure CC(G)
15:     Frontier = {0, ..., |V| − 1}           ▷ vertexSubset initialized to V
16:     while (SIZE(Frontier) ≠ 0) do
17:         Frontier = VERTEXMAP(Frontier, COPY)
18:         Frontier = EDGEMAP(G, Frontier, CCUPDATE, C_true)
19:     return IDs
```

5.5 PageRank

PageRank is an algorithm that was first used by Google to compute the relative importance of webpages [7]. It takes as input a graph $G = (V, E)$, a damping factor $0 \leq \gamma \leq 1$ and a convergence constant ϵ. It initializes a PageRank vector *PR* of length $|V|$ to have all entries set to $\frac{1}{|V|}$, and iteratively applies the following equation for all indices v, until the sum of the differences of PR values between iterations drops to below ϵ:

$$\text{PR}[v] = \frac{1-\gamma}{|V|} + \gamma \sum_{u \in N^-(v)} \frac{\text{PR}[u]}{deg^+(u)} \tag{4}$$

This leads to a very simple implementation in our framework. We also describe a variant of PageRank (PageRank-Delta) which applies Equation (4) to only a subset of V in an iteration. By choosing the subset to contain only vertices whose PageRank entry that changed by more than a certain amount, we can speed up the computation.

The pseudocode for our implementation of PageRank is shown in Algorithm 9. In every iteration, the frontier contains all vertices. Our implementation maintains two arrays p_{curr} and p_{next} each of length $|V|$. p_{curr} is initialized to $\frac{1}{|V|}$ for each entry and p_{next} is initialized to all 0.0's. The PRUPDATE function passed to EDGEMAP is shown in lines 5–7. ATOMICINCREMENT(x, y) atomically adds y to the value at location x and stores the result in location x; it can be implemented with a compare-and-swap. Each iteration of the while loop (lines 18–22) applies an EDGEMAP, uses a VERTEXMAP to process the result of the EDGEMAP, computes the error for the iteration and switches the roles of p_{next} and p_{curr}. The PRLOCALCOMPUTE function (lines 9–13) passed to VERTEXMAP normalizes the result of the EDGEMAP by γ, adds a constant, computes the absolute difference between p_{next} and p_{curr}, and resets p_{curr} to 0.0 for the next iteration (since the roles of p_{next} and p_{curr} become switched). The while loop is executed until the error drops below ϵ.

PageRank-Delta is a variant of PageRank in which vertices are active in an iteration only if they have accumulated enough change in their PR value. This idea is used in GraphLab for computing PageRank [30]. In our framework, in each EDGEMAP vertices pass their changes (deltas) in PR value to their neighbors, and all vertices accumulate a sum of delta contributions from their neighbors. Each VERTEXMAP only updates and returns vertices whose accumulated delta contributions from neighbors is more than a δ-fraction of its PR value since the last time it was active. Such an implementation allows for vertices which do not influence the PR values much to stay inactive, thereby shrinking the frontier. We can implement PageRank-Delta in our framework by modifying the function passed to EDGEMAP to pass the deltas instead of the PR values, and modifying the function passed to VERTEXMAP to only perform updates and return true for the vertices whose accumulated delta contributions from neighbors since it was last active is more than a δ-fraction of its PR value. Due to space limitations, we do not show the pseudocode for this algorithm.

Algorithm 9 PageRank

```
1:  p_curr = {1/|V|, ..., 1/|V|}               ▷ initialized to all 1/|V|
2:  p_next = {0.0, ..., 0.0}                   ▷ initialized to all 0.0
3:  diff = {}                                  ▷ array to store differences
4:
5:  procedure PRUPDATE(s, d)
6:      ATOMICINCREMENT(&p_next[d], p_curr[s] / deg+(s))
7:      return 1
8:
9:  procedure PRLOCALCOMPUTE(i)
10:     p_next[i] = (γ × p_next[i]) + (1−γ)/|V|
11:     diff[i] = |p_next[i] − p_curr[i]|
12:     p_curr[i] = 0.0
13:     return 1
14:
15: procedure PAGERANK(G, γ, ε)
16:     Frontier = {0, ..., |V| − 1}           ▷ vertexSubset initialized to V
17:     error = ∞
18:     while (error > ε) do
19:         Frontier = EDGEMAP(G, Frontier, PRUPDATE, C_true)
20:         Frontier = VERTEXMAP(Frontier, PRLOCALCOMPUTE)
21:         error = sum of diff entries
22:         SWAP(p_curr, p_next)
23:     return p_curr
```

5.6 Bellman-Ford Shortest Paths

The single-source shortest paths problem takes as input a weighted graph $G = (V, E, w(E))$ and a root vertex r, and either computes the shortest path distance from r to each vertex in V (if a vertex is unreachable from r, then the distance returned is ∞), or reports the existence of a negative cycle.

If the edge weights are all non-negative, then the single-source shortest paths problem can be solved with Dijkstra's algorithm [11]. Parallel variants of Dijkstra's algorithm have been studied [36], and have been shown to work well on real-world graphs [32]. However, Dijkstra's algorithm does not work with negative edge weights, and the Bellman-Ford algorithm can be used instead in this case. Although in the worst case the Bellman-Ford algorithm requires $O(|V||E|)$ operations, in contrast to the $O(|E| + |V| \log |V|)$ worst-case operations of Dijkstra's algorithm, in practice it can require many fewer than the worst case since on every step only some of the vertices might change distances. It is therefore important to take advantage of this fact and only process vertices when they actually change distance.

We first describe the standard Bellman-Ford algorithm [11] and then show how it can be implemented in our framework. The algorithm initializes the shortest paths array *SP* to all ∞ except for the root vertex which has an entry of 0. A **RELAX** procedure is repeatedly invoked by Bellman-Ford. RELAX takes G as an input and checks for each edge (u, v) if SP$[u]$ + $w(u, v)$ < SP$[v]$; if so, it sets SP$[v]$ to SP$[u]$ + $w(u, v)$. If a call to RELAX does not change any SP values then the algorithm terminates. If RELAX is called $|V|$ or more times, then there is a negative cycle in G and the Bellman-Ford algorithm reports the existence of one.

To implement the Bellman-Ford algorithm in our framework (pseudocode shown in Algorithm 10) we maintain a *Visited* array in addition to the SP array. Since only vertices whose SP value

has changed in an iteration need to propagate its SP value to its neighbors, the Visited array (initialized to all 0's) keeps track of which vertices had their SP value changed in an iteration. The update function passed to EDGEMAP is shown in lines 4–7 of Algorithm 10 (note that since this algorithm works on weighted graphs, the update function has the edge weight as an additional argument). It uses WRITEMIN (as described in Section 5.4) to posssibly update SP with a smaller path length. The compare-and-swap on line 6 guarantees that a vertex is placed on the frontier at most once per iteration. The initial frontier contains just the root vertex r. Each iteration of the while loop in lines 17–20 applies the EDGEMAP, which outputs a vertexSubset containing the vertices whose SP value changed. In order to reset the Visited array after an EDGEMAP, the BFRESET function (lines 9–11) is passed to VERTEXMAP. The algorithm either runs until no SP values change or runs for $|V|$ iterations and reports the existence of a negative-weight cycle. An iteration here differs from the RELAX procedure in that RELAX processes all vertices each time.

Algorithm 10 Bellman-Ford

```
 1: SP = {∞, ..., ∞}                                  ▷ initialized to all ∞
 2: Visited = {0, ..., 0}                             ▷ initialized to all 0
 3:
 4: procedure BFUPDATE(s, d, edgeWeight)
 5:     if (WRITEMIN(&SP[d], SP[s] + edgeWeight)) then
 6:         return CAS(&Visited[d], 0, 1)
 7:     else return 0
 8:
 9: procedure BFRESET(i)
10:     Visited[i] = 0
11:     return 1
12:
13: procedure BELLMAN-FORD(G, r)
14:     SP[r] = 0
15:     Frontier = {r}              ▷ vertexSubset initialized to contain just r
16:     round = 0
17:     while (SIZE(Frontier) ≠ 0 and round < |V|) do
18:         round = round + 1
19:         Frontier = EDGEMAP(G, Frontier, BF-UPDATE, C_true)
20:         Frontier = VERTEXMAP(Frontier, BF-RESET)
21:     if (round == |V|) then return "negative-weight cycle"
22:     else return SP
```

6. Experiments

All of the experiments presented in this paper were performed on a 40-core Intel machine (with hyper-threading) with 4×2.4GHz Intel 10-core E7-8870 Xeon processors, a 1066MHz bus, and 256GB of main memory. The parallel programs were compiled with Intel's `icpc` compiler (version 12.1.0) using CilkPlus [27] with the `-O3` flag. The sequential programs were compiled using `g++` 4.4.1 with the `-O2` flag. We also ran experiments on a 64-core AMD Opteron machine, but the results are slower than the ones from the Intel machine so we only report the latter.

The input graphs used in our experiments are shown in Table 1. ***3D-grid*** is a grid graph in 3-dimensional space in which every vertex has six edges—one connecting it to each of its two neighbors in each dimension. ***Random-local*** is a synthetic graph in which every vertex has edges to five randomly chosen neighbors, where the probability of an edge between two vertices is inversely correlated with their distance in the vertex array (vertices tend to have edges to other vertices that are close in memory). The ***rMat*** graphs are synthetic graphs with a power-law distribution of degrees [9]. ***RMat24*** (scale 24) contains 1.68×10^7 vertices and was generated with parameters $a = 0.5, b = c = 0.1, d = 0.3$. ***RMat27*** (scale 27) is one of the Graph500 benchmark graphs [18], and was generated with parameters $a = 0.57, b = c = 0.19, d = 0.05$. ***Twitter*** is a real-world graph of the Twitter social network containing 41.7 million vertices and 1.47 billion directed edges [25] . ***Yahoo*** is a real-world graph of the Web containing 1.4 billion vertices and 6.6 billion directed edges (12.9 billion after symmetrizing and removing duplicates) [42]. With the exception of Pregel, Yahoo is the largest real-world graph reported by other graph processing systems.

The number of edges reported is the number of directed edges in the graph with duplicate edges removed. The synthetic graphs are all symmetric, and we symmetrized the Yahoo graph for our experiments so that we have a larger graph. We used the original asymmetric Twitter graph. For the synthetic weighted graphs, the edge weights were generated randomly and were verified to contain no negative cycles. We used unit weights on the Twitter and Yahoo graphs for our Bellman-Ford experiments.

Input	Num. Vertices	Num. Directed Edges
3D-grid	10^7	6×10^7
random-local	10^7	9.8×10^7
rMat24	1.68×10^7	9.9×10^7
rMat27	1.34×10^8	2.12×10^9
Twitter	4.17×10^7	1.47×10^9
Yahoo*	1.4×10^9	12.9×10^9

Table 1. Graph inputs. *The original asymmetric graph has 6.6×10^9 edges.

Table 2 shows the running times for our implementations on each of the input graphs using a single thread and 40 cores with hyper-threading. All of the implementations used EDGEMAP-DENSE for the dense iterations with the exception of Bellman-Ford, PageRank and PageRank-Delta, which used EDGEMAPDENSE-WRITE, an optimization described in Section 4.4 (we found it to be more efficient in these cases). Figure 2 shows that all of our implementations scale well with the number of threads ("80" on the x-axis is 40 cores with hyper-threading).

For BFS, we achieve a 10–28 fold speedup. Using our framework we are able integrate the ideas of [3] to give a simple implementation of BFS, which is almost as fast as their highly optimized implementation. Our running times are better than those reported in [1, 28, 41], which do not take advantage of changes in the frontier density. Compared to the sequential BFS implementation in [41], we are faster on two or more threads.

For betweenness centrality (performing the two-phase computation for a single source) we achieve a 12–32 fold speedup on 40 cores. The KDT system [31] reports that on 256 cores (2.1 GHz AMD Opteron) their batched implementation of betweenness centrality (performs the two-phase computation for multiple sources in parallel) traverses almost 125 million edges per second on an rMat graph with 2^{18} vertices and 16×2^{18} edges. On rMat27 our implementation traverses 526 million edges per second using 40 cores on the Intel Nehalem machine, but it is difficult to directly compare because our machine is different and we do not do a batched computation. For the Twitter graph, since we transpose the graph for the second phase, the in-degree of some of the vertices increases dramatically, so we found that using a parallel inner loop in EDGEMAPDENSE, an optimization described in Section 4.4, was more efficient.

We ran our graph radii estimation implementation using a 64-bit vector for each vertex ($K = 64$) and achieve a 23–35$\times$ speedup on 40 cores. Kang et. al. [23] implement a slightly different algorithm for estimating the radii distribution using MapReduce, and run experiments on the Yahoo M45 Hadoop cluster (480 machines with 2 quad-core Intel Xeon 1.86 GHz processors per machine). Using 90 machines their reported runtime for 3 iterations on a 2 billion-edge graph is almost 30 minutes. Using a 40-core machine we are

Application	3D-grid			random-local			rMat24			rMat27			Twitter			Yahoo		
	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)
Breadth-First Search	2.9	0.28	10.4	2.11	0.073	28.9	2.83	0.104	27.2	11.8	0.423	27.9	6.92	0.321	21.6	173	8.58	20.2
Betweenness Centrality	9.15	0.765	12.0	8.53	0.265	32.2	11.3	0.37	30.5	113	4.07	27.8	47.8	2.64	18.1	634	23.1	27.4
Graph Radii	351	10.0	35.1	25.6	0.734	34.9	39.7	1.21	32.8	337	12.0	28.1	171	7.39	23.1	1280	39.6	32.3
Connected Components	51.5	1.71	30.1	14.8	0.399	37.1	14.1	0.527	26.8	204	10.2	20.0	78.7	3.86	20.4	609	29.7	20.5
PageRank (1 iteration)	4.29	0.145	29.6	6.55	0.224	29.2	8.93	0.25	35.7	243	6.13	39.6	72.9	2.91	25.1	465	15.2	30.6
Bellman-Ford	63.4	2.39	26.5	18.8	0.677	27.8	17.8	0.694	25.6	116	4.03	28.8	75.1	2.66	28.2	255	14.2	18.0

Table 2. Running times (in seconds) of algorithms over various inputs on a 40-core machine (with hyper-threading). (SU) indicates the speedup of the application (single-thread time divided by 40-core time).

able to process the rMat27 graph of similar size until completion (9 iterations) in 12 seconds.

Our connected components implementation achieves a 20–37 fold speedup on 40 cores. The Pegasus library [24] also has a connected components algorithm implemented for the MapReduce framework. For a graph with 59,000 vertices and 282 million edges, and using 90 machines of the Yahoo M45 cluster, they report a runtime of over 10 minutes for 6 iterations. In contrast, for the much larger rMat27 graph (also requiring 6 iterations) our algorithm completes in about 10 seconds on the 40-core machine.

For a single iteration, our PageRank implementation achieves a 29–39 fold speedup on 40 cores. GPS [40] reports a running time of 144 minutes for 100 iterations (1.44 minutes per iteration) of PageRank on a web graph with 3.7 billion directed edges on an Amazon EC2 cluster using 30 large instances, each with 4 virtual cores and 7.5GB of memory. In contrast, our PageRank implementation takes less than 20 seconds per iteration on the larger Yahoo graph. For PageRank on the Twitter graph [25], our system is slightly faster per iteration (2.91 seconds vs. 3.6 seconds) on 40 cores than PowerGraph [17] on 8×64 cores (processors are 2.933 GHz Intel Xeon X5570 with 3200 MHz bus). We also compared our implementations of PageRank and PageRank-Delta, run to convergence with a damping factor of $\gamma = 0.85$ and parameters $\epsilon = 10^{-7}$ and $\delta = 10^{-2}$. Figure 2(e) shows that PageRank-Delta is faster (by more than a factor of 6 on rMat24) because in any given iteration it processes only vertices whose accumulated change is above a δ-fraction of its PageRank value at the time it was previously active. We do not analyze the error (which depends on δ) of our PageRank-Delta implementation in this work—the purpose of this experiment is to show that our framework also works well for non-graph traversal problems.

Our parallel implementation of Bellman-Ford achieves a 18–28$\times$ speedup on 40 cores. In Figure 2(f) we compare this implementation with a naive one which visits all vertices and edges in each iteration, and our more efficient version is almost twice as fast. The single-source shortest paths algorithm of Pregel [34] for a binary tree with 1 billion vertices takes almost 20 seconds on a cluster of 300 multicore commodity PCs. We ran our Bellman-Ford algorithm on a larger binary tree with $2^{27} (\approx 1.68 \times 10^7)$ vertices, and it completed in under 2 seconds (time not shown in Table 2). Compared to our implementation of the standard sequential algorithm described in [11], our parallel implementation is faster on a single thread.

Since the Yahoo graph is highly disconnected, we computed the number of vertices and directed edges traversed for BFS and betweenness centrality and found it to be 701 million and 12.8 billion respectively (this is the largest connected component of the graph). The number of vertex and edge traversals for the graph radii algorithm ($K = 64$) on the Yahoo graph were 2.7 billion and 50 billion respectively. Note that doing 64 individual BFS's to compute the same thing would require many more vertex and edge traversal; our implementation of radii estimation (multiple-BFS) reduces the number of traversals (and hence running time) by combining the operations of multiple BFS's into fewer operations.

Figure 3 shows scalability plots for the various applications. The experiments were performed on random graphs of varying size with the number of directed edges being ten times the number of vertices. We see that the implementations scale quite well with increasing graph size, with some noise due to the variability in the structures of the different random graphs.

Figure 4 shows plots of the size of the frontier plus the number of outgoing edges for each iteration and each application on rMat24. The rMat24 graph is a scale-free graph and hence able to take advantage of the hybrid BFS idea of Beamer et. al. [4]. The y-axes are shown in log-scale. We also plot the threshold, above which EDGEMAP uses the dense implementation and below which EDGEMAP uses the sparse implementation. For BFS, betweenness centrality (same frontier plot as that of BFS), radii estimation and Bellman-Ford, the frontier is initially sparse, switches to dense after a few iterations and then switches back to sparse later. For connected components and PageRank-Delta, the frontier starts off as dense (the vertexSubset contains all vertices), and becomes sparser as the algorithm continues. See [4] for a more detailed analysis of frontier plots for BFS.

7. Conclusions

We have described Ligra, a simple framework for implementing graph traversal algorithms on shared-memory machines. Furthermore, our implementations of several graph algorithms using the framework are efficient and scalable, and often achieve better running times than ones reported by other graph libraries/systems. In addition to the algorithms discussed in this paper, we believe other algorithms such as maximum flow, biconnected components, belief propagation, and Markov clustering can also benefit from our framework. Currently, Ligra does not support algorithms based on modifying the input graph, and extending Ligra to support graph modification is a direction for future work. Recently, GPU systems have been explored for implementing graph traversal problems [20, 35]. It is possible that our framework can be extended to this context.

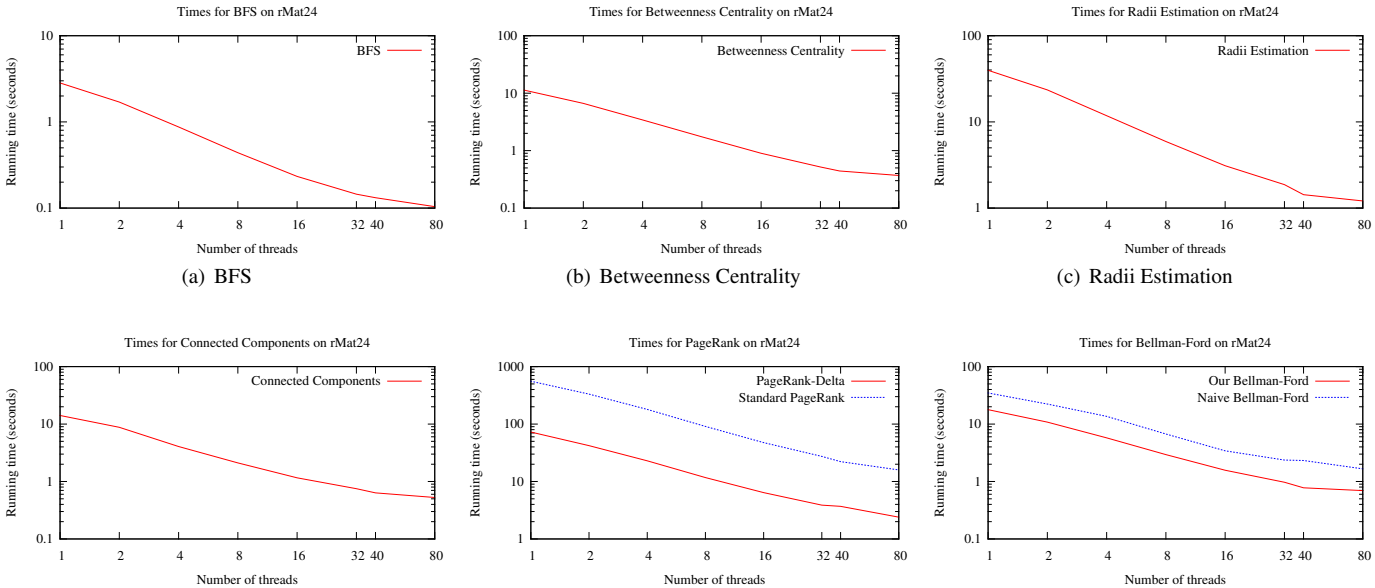
Acknowledgments. This work is partially supported by the National Science Foundation under grant number CCF-1018188, and by Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program.

References

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SC*, 2010.

[2] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, 2007.

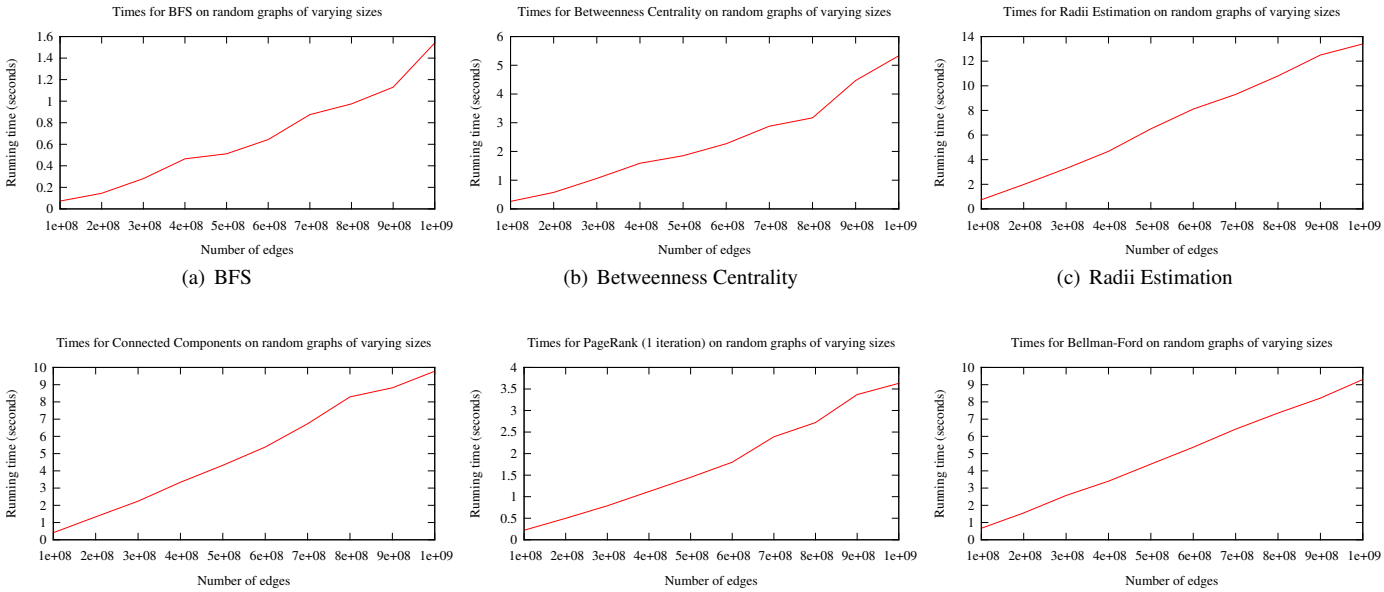
[3] S. Beamer, K. Asanović, and D. Patterson. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. *Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley*, 2011.



(a) BFS (b) Betweenness Centrality (c) Radii Estimation

(d) Connected Components (e) PageRank (f) Bellman-Ford

Figure 2. Log-log plots of running times on rMat24 on a 40-core machine (with hyper-threading).



(a) BFS (b) Betweenness Centrality (c) Radii Estimation

(d) Connected Components (e) PageRank (1 iteration) (f) Bellman-Ford

Figure 3. Plots of running times versus edge counts in random graphs on a 40-core machine (with hyper-threading).

[4] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *SC*, 2012.

[5] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *In IPDPS*, 2007.

[6] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25, 2001.

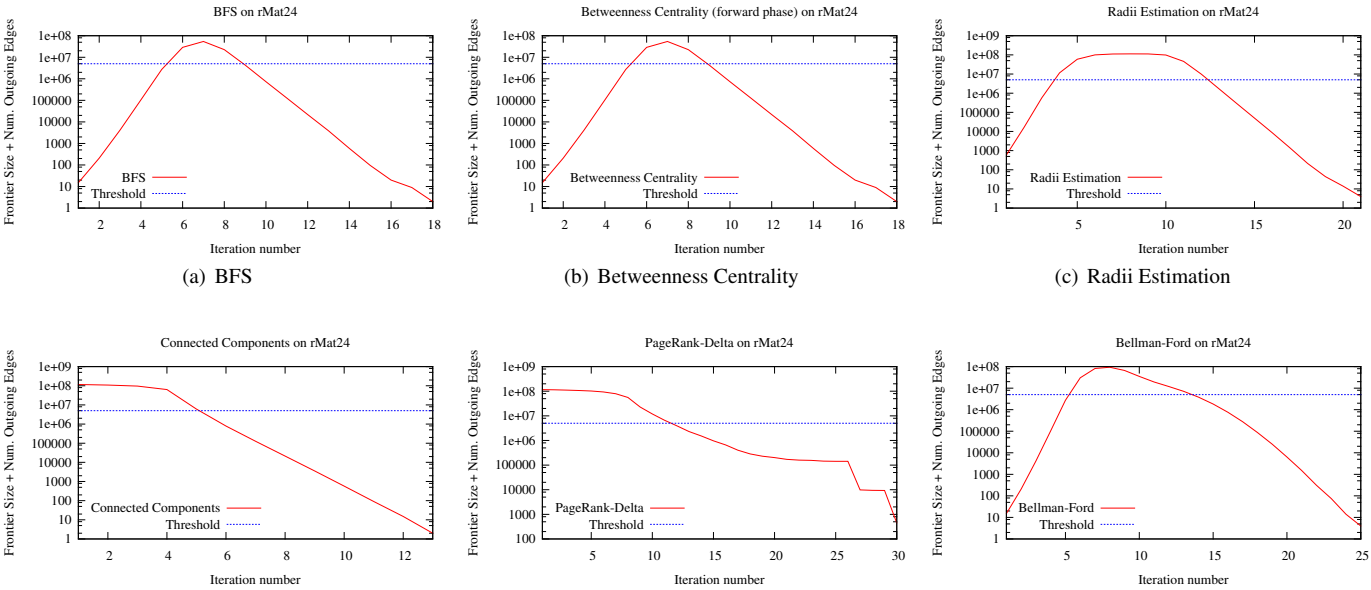
[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998.

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

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, 2004.

[10] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55, December 1997.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.



(a) BFS (b) Betweenness Centrality (c) Radii Estimation

(d) Connected Components (e) PageRank-Delta (f) Bellman-Ford

Figure 4. Plots of frontier size plus number of outgoing edges (y-axis in log scale) versus iteration number for rMat24.

[12] J.-A. Ferrez, K. Fukuda, and T. Liebling. Parallel computation of the diameter of a graph. In *HPCSA*, 1998.

[13] L. Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 1977.

[14] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[15] J. R. Gilbert, S. Reinhardt, and V. B. Shah. A unified framework for numerical and combinatorial computing. *Computing in Sciences and Engineering*, 10(2), Mar/Apr 2008.

[16] Giraph. "http://giraph.apache.org", 2012.

[17] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Power-Graph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[18] Graph500. "http://www.graph500.org", 2012.

[19] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *POOSC*, 2005.

[20] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *PACT*, 2011.

[21] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, 2012.

[22] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[23] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. In *TKDD*, 2011.

[24] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2), 2011.

[25] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[26] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[27] C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3), 2010. Springer.

[28] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, 2010.

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[30] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[31] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SDM*, 2012.

[32] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, 2007.

[33] C. Magnien, M. Latapy, and M. Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *J. Exp. Algorithmics*, 13, February 2009.

[34] 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 *SIGMOD*, 2010.

[35] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *PPoPP*, 2012.

[36] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1), 2003.

[37] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: a fast and scalable tool for data mining in massive graphs. In *ACM SIGKDD*, 2002.

[38] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, 2011.

[39] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX ATC*, 2012.

[40] S. Salihoglu and J. Widom. GPS: A graph processing system. Technical Report InfoLab 1039, Stanford University, 2012.

[41] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012.

[42] Yahoo! Altavista web page hyperlink connectivity graph, 2012. "http://webscope.sandbox.yahoo.com/catalog.php?datatype=g".