Poster Paper

Vertex-Centric Graph Processing: Good, Bad, and the Ugly

Arijit Khan
Nanyang Technological University, Singapore
arijit.khan@ntu.edu.sg

ABSTRACT

We study distributed graph algorithms that adopt an iterative vertex-centric framework for graph processing, popularized by Google's Pregel system. Since then, there are several attempts to implement many graph algorithms in a vertex-centric framework, as well as efforts to design optimization techniques for improving the efficiency. However, to the best of our knowledge, there has not been any systematic study to compare these vertex-centric implementations with their sequential counterparts. Our paper addresses this gap in two ways. (1) We analyze the computational complexity of such implementations with the notion of time-processor product, and benchmark several vertex-centric graph algorithms whether they perform more work with respect to their best-known sequential solutions. (2) Employing the concept of balanced practical Pregel algorithms, we study if these implementations suffer from imbalanced workload and large number of iterations. Our findings illustrate that with the exception of Euler tour tree algorithm, all other algorithms either perform asymptotically more work than their best-known sequential approach, or suffer from imbalanced workload/ large number of iterations, or even both. We also emphasize on graph algorithms that are fundamentally difficult to be expressed in vertex-centric frameworks, and conclude by discussing the road ahead for distributed graph processing.

1. INTRODUCTION

In order to achieve low latency and high throughput over massive graph datasets, data centers and cloud operators consider scale-out solutions, in which the graph and its data are partitioned horizontally across cheap commodity servers in the cluster. The distributed programming model for large graphs has been popularized by Google's Pregel framework [4], which was inspired by the Bulk Synchronous Parallel (BSP) model [12]. It hides distribution related details such as data partitioning, communication, underlying system architecture, and fault tolerance behind an abstract API. In Pregel, also known as the *think-like-a-vertex* model, graph algorithms are expressed as a sequence of iterations called supersteps. During a superstep, Pregel executes a user-defined function for each vertex in parallel. The user-defined function specifies the operation at a single vertex v and at a single superstep S. The supersteps are globally synchronous among all vertices, and messages are usually sent along the outgoing edges from each vertex.

With the inception of the Pregel framework, vertex-centric distributed graph processing has become a hot topic in the database community (for a survey, see [13]). Although Pregel provides a high-level distributed programming abstract, it suffers from efficiency issues such as the overhead of global synchronization, large volume of messages, imbalanced workload, and straggler problem due to slower machines. Therefore, more advanced vertex-centric models (and its variants) have been proposed, e.g., asynchronous (GraphLab), asynchronous parallel (GRACE), barrierless asynchronous parallel (Giraph Unchained), data parallel (GraphX, Pregelix), gather-apply-scatter (PowerGraph), timely dataflow (Naiad), and subgraph centric frameworks (NScale, Giraph++). Various algorithmic and system-specific optimization techniques were also designed, e.g., graph partitioning and re-partitioning, combiners and aggregators, vertex scheduling, superstep sharing, message reduction, finishing computations serially, among many others.

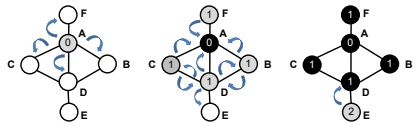
While speeding up any algorithm is always significant in its own right, there may be circumstances in which we would not benefit greatly from doing so. McSherry et. al. [5] empirically demonstrated that single-threaded implementations of many graph algorithms using a high-end 2014 laptop are often an order of magnitude faster than the published results for state-of-the-art distributed graph processing systems using multiple commodity machines and hundreds of cores over the same datasets. Surprisingly, with the exception of [14], the complexity of vertex-centric graph algorithms has never been formally analyzed. As one may realize, this is not a trivial problem — there are multiple factors involved in a distributed environment including the number of processors, computation time, network bandwidth, communication volume, and memory usage. To this end, we make the following contributions.

- We formally analyze the computational complexity of vertex-centric implementations with the notion of time-processor product [12], and benchmark several vertex-centric graph algorithms whether they perform asymptotically more work in comparison to their best-known sequential algorithms.
- We use the concept of balanced, practical Pregel algorithms [14] to investigate if these vertex-centric algorithms suffer from imbalanced workload and large number of iterations.

While the notion of balanced, practical Pregel algorithms was introduced by Yan et. al. [14], they only considered the connected component-based algorithms. On the contrary, in this paper we report as many as fifteen different graph algorithms (Table 1), whose vertex-centric algorithms were implemented in the literature. Finally, we also identify graph workloads and algorithms that are difficult to be expressed in the vertex-centric framework, and highlight some important research directions.

	Graph Workload	Vertex-Centric Algorithm	Vertex-Centric Complexity	Best Sequential Algorithm	Best Sequential Complexity	Vertex-Centric More Work?	Vertex-Centric BPPA?
1	Diameter (Unweighted)	[6]	$\mathcal{O}(mn)$	BFS [9]	$\mathcal{O}(mn)$	**No**	No
2	PageRank [1]	[4]	$\mathcal{O}(mK)$	power iteration	$\mathcal{O}(mK)$	**No**	No
3	Connected Component	Hash-Min [4]	$\mathcal{O}(m\delta)$	BFS [3]	$\mathcal{O}(m+n)$	Yes	No
4	Connected Component	S-V [14]	$\mathcal{O}((m+n)\log n)$	BFS [3]	$\mathcal{O}(m+n)$	Yes	No
5	Bi-Connected Component	[14]	$\mathcal{O}((m+n)\log n)$	DFS [3]	$\mathcal{O}(m+n)$	Yes	No
6	Weakly Connected Component	[14]	$\mathcal{O}((m+n)\log n)$	BFS [3]	$\mathcal{O}(m+n)$	Yes	No
7	Strongly Connected Component	[14]	$\mathcal{O}((m+n)\log n)$	DFS [11]	$\mathcal{O}(m+n)$	Yes	No
8	**Euler Tour of Tree**	[14]	$\mathcal{O}(n)$	DFS	$\mathcal{O}(n)$	**No**	**Yes**
9	Pre- & Post-order Tree Traversal	[14]	$\mathcal{O}(n\log n)$	DFS	$\mathcal{O}(n)$	Yes	**Yes**
10	Spanning Tree	[14]	$\mathcal{O}((m+n)\log n)$	BFS	$\mathcal{O}(m+n)$	Yes	No
11	Minimum Cost Spanning Tree [1]	[10]	$\mathcal{O}(\delta m\log n)$	Chazelle's algorithm	$\mathcal{O}(m\alpha(m,n))$	Yes	No
12	Betweenness Centrality (Unweighted)	[8]	$\mathcal{O}(mn)$	Brandes' algorithm	$\mathcal{O}(mn)$	**No**	No
13	Single-Source Shortest Path	[4]	$\mathcal{O}(mn)$	Dijkstra with Fibonacci heap	$\mathcal{O}(m+n\log n)$	Yes	No
14	All-pair Shortest Paths (Unweighted)	[6]	$\mathcal{O}(mn)$	Chan's algoithm	$\mathcal{O}(mn)$	**No**	No
15	Graph Simulation [1]	[1]	$\mathcal{O}(m^2(n_q+m_q))$	Henzinger et. al. [2]	$\mathcal{O}\left((m+n)(m_q+n_q)\right)$	Yes	No

Table 1: Efficiency analysis for vertex-centric graph algorithms: # nodes = n, # edges = m, diameter = δ



(a) Superstep 0 (b) Superstep 1 (c) Superstep 2

Figure 1: Vertex-centric algorithm for diameter computation in unweighted graphs

2. PRELIMINARIES

2.1 Time-Processor Product

Time-processor product was employed by Valiant [12] as a complexity measure of algorithms on the BSP model, defined by the following parameters. (1) Bandwidth parameter is g, that measures the permeability of the network to continuously send traffic to uniformly-random destinations. The parameter g is defined such that an h-relation will be delivered in time hg. (2) Synchronization periodicity is L, where the components at regular intervals of L time units are synchronized. In a superstep of periodicity L, L local operations and $\lfloor L/g \rfloor$-relation message patterns can be realized. (3) The number of processors is p. Let w_i be the amount of local work performed by processor i in a given superstep. Assume s_i and r_i be the number of messages sent and received, respectively, by processor i. Let $w = \max_{i=1}^{p} w_i$, and $h = \max_{i=1}^{p}(\max(s_i, r_i))$. Then, the time for a superstep is $\max(w, gh, L)$.

If we have multiple processors, we can solve a problem more quickly by dividing it into independent sub-problems and solving them at the same time, one at each processor. Given an input size n, the running time $T(n)$ is the elapsed time from when the first processor begins executing to when the last processor stops executing. A BSP algorithm for a given problem is called efficient if its processor bound $P(n)$ and time bound $T(n)$ are such that time-processor product $P(n)T(n) = \mathcal{O}(S)$, where S is the running time of the best-known sequential algorithm for the problem, provided that L and g are below certain critical values. Therefore, with this metric, we measure whether a vertex-centric algorithm performs more work, compared to the problem's best-known sequential algorithm.

2.2 Balanced, Practical Pregel Algorithms

For an undirected graph, we denote by $d(v)$ the degree of vertex v. On the other hand, let $d_{in}(v)$ and $d_{out}(v)$ denote the in-degree and out-degree, respectively, of vertex v in a directed graph. A Pregel algorithm is called a balanced, practical Pregel algorithm (BPPA) [14] if it satisfies the following. (1) Each vertex v uses $\mathcal{O}(d(v))$ (or, $\mathcal{O}(d_{in}(v)+d_{out}(v))$) storage. (2) The time complexity of the vertex-compute() function for each vertex v is $\mathcal{O}(d(v))$ (or, $\mathcal{O}(d_{in}(v)+d_{out}(v))$). (3) At each superstep, the size of the messages sent/received by each vertex v is $\mathcal{O}(d(v))$ (or, $\mathcal{O}(d_{in}(v)+d_{out}(v))$). (4) The algorithm terminates after $\mathcal{O}(\log n)$ supersteps. Properties 1-3 offer good load balancing and linear cost at each superstep, whereas property 4 impacts the total running time.

[1] K is # iterations for convergence, $\alpha()$ functional inverse of Ackermann's function. n_q and m_q the number of nodes and edges, respectively, in the query graph.

[2] For higher values of g, the time-processor product would be even higher.

3. COMPLEXITY ANALYSIS

We summarize the complexity of fifteen vertex-centric graph algorithms in Table 1. We shall discuss five of them in the following.

3.1 Diameter Computation

We consider a vertex-centric algorithm [6] that computes the exact diameter of an unweighted graph. Let us denote the eccentricity $\epsilon(v)$ of a vertex v as the largest hop-count distance from v to any other vertex in the graph. The diameter δ of the graph is defined as the maximum eccentricity over all its nodes. Instead of finding this largest vertex eccentricity one-by-one, the algorithm works by computing the eccentricity of all vertices simultaneously.

We illustrate in Figure 1 the eccentricity computation method of one vertex. Initially, each vertex adds it's own unique id to the outgoing messages (sent along the outgoing edges) and also to the history set, which resides in the local memory of that vertex. After the initial superstep, the algorithm operates by iterating through the set of received ids, which correspond to the vertices that sent the original messages. The receiving vertex then constructs a set of outgoing messages by adding each element of the incoming set which was not seen yet. The reason for keeping a history of the originating ids that were received earlier is to prevent the re-propagation of a message to the same vertices. The history set also serves to prune the set of total messages by eliminating message paths that would never result in the vertex's eccentricity.

Assuming the graph is connected, each vertex will process a message from each originating vertex exactly once. The algorithm terminates when the largest eccentricity is calculated; and therefore, the diameter of the graph is equal to the number of supersteps (minus 1, for the final, non-processing superstep).

Since each vertex generates a unique message, there are total $\Theta(n)$ messages. Each message is passed $\mathcal{O}(m)$ times, resulting in a message complexity of $\mathcal{O}(mn)$. There are total $\mathcal{O}(\delta)$ supersteps. Each vertex processes n messages; therefore, the overall computation cost is $\mathcal{O}(n^2)$. Assuming bandwidth parameter [2] $g = \mathcal{O}(1)$, the time-processor product = $\mathcal{O}(mn)$, which is equal to the complexity of the best-known sequential algorithm.

However, this vertex-centric algorithm is not BPPA because: (1) The number of messages that each vertex v relays can be asymptotically larger than $\mathcal{O}(d(v))$ at later supersteps. (2) Given that each vertex v must store a history of the messages received, each vertex stores $\mathcal{O}(n)$ vertex IDs, which is larger than $\mathcal{O}(d(v))$. (3) There are total $\mathcal{O}(\delta)$ supersteps, which could be larger than $\mathcal{O}(\log n)$.

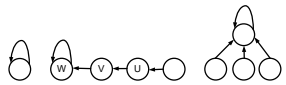


Figure 2: Forest structure of S-V algorithm [14]

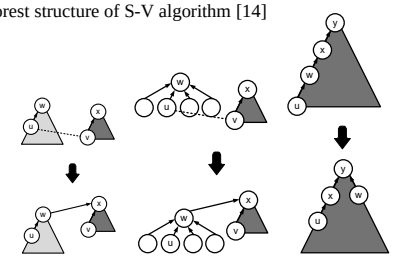


Figure 3: Tree hooking, star hooking, and shortcutting [14]

3.2 Connected Component

We study two vertex-centric algorithms: Hash-min and S-V [14].

3.2.1 Hash-Min Algorithm

We assume that each vertex in a graph G is assigned a unique ID. The color of a connected component in G is defined as the smallest vertex among all vertices in the component. In Superstep 1, each vertex v initializes $min(v)$ as the smallest vertex in the set $(\{v\} \cup neighbors(v))$, sends $min(v)$ to all v's neighbors, and votes to halt. In each subsequent superstep, a vertex v obtains the smallest vertex from the incoming messages, denoted by u. If $u < v$, v sets $min(v) = u$ and sends $min(v)$ to all its neighbors. Finally, v votes to halt. When all vertices vote to halt and there is no new message in the network, the algorithm terminates.

It takes at most $\mathcal{O}(\delta)$ supersteps for the ID of the smallest vertex to reach all the vertices in a connected component, and in each superstep, each vertex v takes at most $\mathcal{O}(d(v))$ time to compute $min(v)$ and sends/receives $\mathcal{O}(d(v))$ messages each using $\mathcal{O}(1)$ space. Therefore, it is a balanced Pregel algorithm (i.e., satisfies properties 1-3), but not BPPA since the number of supersteps can be larger than $\mathcal{O}(\log n)$, e.g., for a straight-line graph.

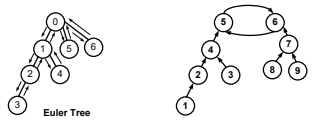
Each superstep consists of $\mathcal{O}(m)$ messages and $\mathcal{O}(m)$ computations. Assuming $g = \mathcal{O}(1)$, the time-processor product is $\mathcal{O}(m\delta)$. This is more than the complexity of the best-known sequential algorithm, which is due to BFS with complexity $\mathcal{O}(m + n)$.

3.2.2 Shiloach-Vishkin (S-V) Algorithm

In the S-V algorithm, each vertex u maintains a pointer $D[u]$. Initially, $D[u] = u$, forming a self-loop as depicted in Figure 2(a). During the algorithm, vertices are arranged by a forest such that all vertices in each tree in the forest belong to the same connected component. The tree definition is relaxed a bit to allow the tree root w to have a self-loop (see Figures 2(b) and 2(c)), i.e., $D[w] = w$; while $D[v]$ of any other vertex V in the tree points to v's parent.

The S-V algorithm proceeds in iterations, and in each iteration, the pointers are updated in three steps (Figure 3): (1) *tree hooking*: for each edge (u, v), if u's parent $w = D[u]$ is a tree root, hook w as a child of v's parent $D[v]$ (i.e., merge the tree rooted at w into v's tree); (2) *star hooking*: for each edge (u, v), if u is in a star (see Figure 2(c) for an example of star), hook the star to v's tree as Step (1) does; (3) *shortcutting*: for each vertex v, move vertex v and its descendants closer to the tree root, by hooking v to the parent of v's parent, i.e., setting $D[v] = D[D[v]]$. The algorithm terminates when every vertex is in a star. We perform tree hooking in Step (1) and star hooking in Step (2) only if $D[v] < D[u]$, which ensures that the pointer values monotonically decrease.

It was proved that the above S-V algorithm computes connected components in $\mathcal{O}(\log n)$ supersteps [14]. However, the algorithm is not a BPPA because a vertex v may become the parent of more than $d(v)$ vertices and hence receives/sends more than $d(v)$ messages in a superstep. On the other hand, the overall number of messages and computations in each superstep are bounded by $\mathcal{O}(n)$ and $\mathcal{O}(m)$, respectively. With $g = \mathcal{O}(1)$, we have the time-processor product = $\mathcal{O}((m + n) \log n)$. As earlier, this is higher than the complexity of the best-known sequential algorithm.



(a) Euler tour (b) Conjoined-tree: vertex 5 is super-vertex

Figure 4: Euler Tour and MCST construction

3.3 Euler Tour Tree Traversal

A Euler tour is a representation of a tree, where each tree edge (u, v) is considered as two directed edges (u, v) and (v, u). As shown in Figure 4(a), a Euler tour of the tree is simply a Eulerian circuit of the directed graph, that is, a trail that visits every edge exactly once, and ends at the same vertex where it starts.

We assume that the neighbors of each vertex v are sorted according to their IDs, which is usually common for an adjacency list representation of a graph. For a vertex v, let $first(v)$ and $last(v)$ be the first and last neighbor of v in that sorted order; and for each neighbor u of v, if $u \neq last(v)$, let $next_v(u)$ be the neighbor of v next to u in the sorted adjacency list. We also define $next_v(last(v)) = first(v)$. As an example, in Figure 4(a), $first(0) = 1$, $last(0) = 6$, $next_0(1) = 5$, and $next_0(6) = 1$.

Yan et. al. [14] designed a 2-superstep vertex-centric algorithm to construct the Euler tour as given below. In Superstep 1, each vertex v sends message $\langle u, next_v(u)\rangle$ to each neighbor u; in Supertep 2, each vertex u receives the message $\langle u, next_v(u)\rangle$ sent from each neighbor v, and stores $next_v(u)$ with v in u's adjacency list. Thus, for every vertex u and each of its neighbor v, the next edge of (u, v) is obtained as $(v, next_v(u))$, which is the Euler tour.

The algorithm requires a constant number of supersteps. In every superstep, each vertex v sends/receives $\mathcal{O}(d(v))$ messages, each using $\mathcal{O}(1)$ space. By implementing $next_v(.)$ as a hash table associated with v, we can obtain $next_v(u)$ in $\mathcal{O}(1)$ expected time given u. Therefore, the algorithm is BPPA. In addition, with $g = \mathcal{O}(1)$, the time-processor product = $\mathcal{O}(n)$. This matches with the time complexity of the best-known sequential algorithm.

3.4 Minimum Cost Spanning Tree

Salihoglu et. al. implemented the parallel (vertex-centric) version of Boruvka's minimum cost spanning tree (MCST) algorithm [10] for a weighted, undirected graph G. The algorithm iterates through the following phases, each time adding a set of edges to the MCST S it constructs, and removing some vertices from G until there is just one vertex, in which case the algorithm halts.

1. Min-Edge-Picking: In parallel, the edge list of each vertex is searched to find the minimum weight edge from that vertex. Ties are broken by selecting the edge with minimum destination ID. Each picked edge (v, u) is added to S. As proved in Boruvka's algorithm, the vertices and their picked edges form disjoint subgraphs $T_1, T_2, \ldots, T_k$, each of which is a *conjoined-tree*, i.e., two trees, the roots of which are joined by a cycle (Figure 4(b)). We refer to the vertex with the smaller ID in the cycle of T_i as the super-vertex of T_i. All other vertices in T_i are called its sub-vertices. The following steps merge all of the sub-vertices of every T_i into the super-vertex of T_i.

2. Super-vertex Finding: First, we find all the super-vertices. Each vertex v sets its pointer to the neighbor v picked in Min-Edge-Picking. Then, it sends a message to v.pointer. If v finds that it received a message from the same vertex to which it sent a message earlier, it is part of the cycle. The vertex with the smaller ID in the cycle is identified as the super-vertex. After this, each vertex finds the super-vertex of the conjoined-tree it belongs to using the *Simple Pointer Jumping* algorithm. The input R to the algorithm is the set of super-vertices, and the input S is the set of sub-vertices.

Simple-Pointer-Jumping-Algorithm (R, S)

repeat until every vertex in S points to a vertex in R
  for each vertex v that does not point to a vertex in R **do**
    perform a pointer jump: v.pointer $\rightarrow$ v.pointer.pointer

3. Edge-Cleaning-and-Relabeling: We shrink each conjoined tree into the super-vertex of the tree. This is performed as follows. In the set of edges of G, each vertex is renamed with the ID of the super-vertex of the conjoined tree to which it belongs. The modified graph may have self-loops and multiple edges. All self-loops are removed. Multiple edges are removed such that only the lightest edge remains between a pair of vertices.

The above operations can be implemented in $\mathcal{O}(\delta)$ supersteps, which is due to the maximum number of iterations required for the simple pointer jumping algorithm. Each superstep has message and computation complexity $\mathcal{O}(m)$. The three above phases are repeated, that is, the graph remaining after the i-th iteration is the input to the $i+1$-th iteration, unless it has just one vertex, in which case the algorithm halts. Furthermore, the number of vertices of the graph at the $i+1$-th iteration is at most half of the number of vertices at the i-th iteration. Hence, the number of iterations is at most $\mathcal{O}(\log n)$. With $g = \mathcal{O}(1)$, the time-processor product is $\mathcal{O}(m\delta \log n)$. This is higher than the complexity of the best-known sequential algorithm for MCST, which is $\mathcal{O}(m\alpha(m,n))$ by Chazelle's algorithm. Here, $\alpha()$ is the functional inverse of Ackermann's function, and it grows extremely slowly, so that for all practical purposes it may be considered a constant no greater than 4. Even if we consider widely-used Prim's algorithm (sequential), it has time complexity $\mathcal{O}(m + n \log n)$ using fibonacci heap and adjacency list. In summary, the vertex-centric algorithm for MCST performs more work than the problem's sequential solutions.

The algorithm is not in BPPA, since (1) the Edge-Cleaning-and-Relabeling step increases the number of neighbors of the super-vertices, and (2) the number of supersteps is $\mathcal{O}(\delta \log n)$.

3.5 Difficult Problems for Vertex-Centric Model

An important question is whether *all* kinds of graph analytics tasks and algorithms can be expressed *efficiently* at vertex level. (1) Vertex-centric model usually operates on the entire graph, which is often not necessary for online ad-hoc queries [15], including shortest path, reachability, and subgraph isomorphism. (2) This model is not well-suited for graph analytics that require a subgraph-centric view around vertices, e.g., local clustering coefficient, triangle and motifs counting. This is due to the communication overhead, network traffic, and the large amount of memory required to construct multi-hop neighborhood in each vertex's local state [7]. (3) Not all distributed algorithms for the same graph problem can be implemented in a vertex-centric framework. As an example, it is difficult to implement the distributed union-find algorithm for the connected component problem using a vertex-centric model [5]. However, this algorithm is useful for graph streams. (4) State-of-the-art research on vertex-centric graph processing mainly focused on a limited number of graph workloads such as PageRank and connected components, and it is largely unknown whether some other widely-used graph computations, e.g., modularity optimization for community detection, betweenness centrality (weighted graphs), influence maximization, link prediction, partitioning, and embedding can be implemented efficiently over vertex-centric systems.

4. DISCUSSION AND CONCLUSION

Our analysis shows that vertex-centric algorithms often suffer from imbalanced workload/ large number of iterations, and perform more work than their best-known sequential algorithms.

Due to such difficulties, alternate proposals exist where the entire graph is loaded on a single machine having larger memory, or on a multi-core machine with shared-memory. Nevertheless, distributed graph processing systems would still be critical due to the two following reasons. First, graph analysis is usually an intermediate step of some larger data analytics pipeline, whose previous and following steps might require distribution over several machines. In such scenarios, distributed graph processing would help to avoid expensive data transfers. Second, distributed-memory systems generally scale well, compared to their shared-memory counterparts.

However, one distributed model might not be suitable for all kinds of graph computations. Many recent distributed systems, e.g., Trinity, NScale, and Apache Flink support multiple paradigms, including vertex-centric, subgraph-centric, dataflow, and shared access. But, perhaps more importantly, we need to identify the appropriate metrics to evaluate these systems. In addition to time-processor product and BPPA that we studied in this work, one can also investigate the speedup and cost/computation. Two other critical metrics are *expressibility* and *usability*, which were mostly ignored due to their qualitative nature. The former identifies the workloads that can be efficiently implemented in a distributed framework, while the later deals with ease in programming, e.g., domain-specific languages, declarative programming, high-level abstraction to hide data partitioning, communication, system architecture, and fault tolerance, as well as availability of debugging and provenance tools. With all these exciting open problems, this research area is likely to get more attention in the near future.

5. REFERENCES

[1] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz. A Distributed Vertex-Centric Approach for Pattern Matching in Massive Graphs. In *IEEE International Conference on Big Data*, 2013.

[2] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *FOCS*, 1995.

[3] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, 1973.

[4] 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.

[5] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at What Cost? In *HOTOS*, 2015.

[6] C. Pennycuff and T. Weninger. Fast, Exact Graph Diameter Computation with Vertex Programming. In *HPGM*, 2015.

[7] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Analytics on Large Graphs. In *VLDB*, 2014.

[8] M. Redekopp, Y. Simmhan, and V. K. Prasanna. Optimizations and Analysis of BSP Graph Processing Models on Public Clouds. In *IPDPS*, 2013.

[9] L. Roditty and V. V. Williams. Fast Approximation Algorithms for the Diameter and Radius of Sparse Graphs. In *STOC*, 2013.

[10] S. Salihoglu and J. Widom. Optimizing Graph Algorithms on Pregel-like Systems. In *VLDB*, 2014.

[11] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[12] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[13] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big Graph Analytics Systems. In *SIGMOD*, 2016.

[14] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. In *VLDB*, 2014.

[15] Q. Zhang, D. Yan, and J. Cheng. Quegel: A General-Purpose System for Querying Big Graphs. In *SIGMOD*, 2016.