A Brief Survey of Algorithms, Architectures, and Challenges toward Extreme-scale Graph Analytics

Ananth Kalyanaraman
School of EECS
Washington State University
Pullman, WA, USA
ananth@wsu.edu

Partha Pratim Pande
School of EECS
Washington State University
Pullman, WA, USA
pande@wsu.edu

***Abstract*—The notion of networks is inherent in the structure, function and behavior of the natural and engineered world that surround us. Consequently, graph models and methods have assumed a prominent role to play in this modern era of Big Data, and are taking a center stage in the discovery pipelines of various data-driven scientific domains. In this paper, we present a brief review of the state-of-the-art in parallel graph analytics, particularly focusing on iterative graph algorithms and their implementation on modern day multicore/manycore architectures. The class of iterative graph algorithms covers a broad class of graph operations of varying complexities, from simpler routines such as Breadth-First Search (BFS), to polynomially-solvable problems such as shortest path computations, to NP-Hard problems such as community detection and graph coloring. We cover a set of common algorithmic abstractions used in implementing such iterative graph algorithms, state the challenges around parallelization on contemporary parallel platforms (including commodity multicores and emerging manycore platforms), and describe a set of approaches that have led to efficient implementations. We also report on advances in manycore architectural frameworks that have found application in parallel graph analytics. We conclude the paper identifying potential research directions, opportunities, and challenges that lay ahead in the path toward enabling graph analytics at exascale.**

***Index Terms*—Parallel graph algorithms, parallel architectures, irregular applications, extreme-scale computing.**

I. Introduction

Graph analytics has taken a centerstage in the data analytical workflows of numerous contemporary scientific and industrial computing applications. With rapid advancements in sensing and experimental technologies across the application spectrum, our ability to observe vast collections of data has grown significantly. This ability has in turn led to the proliferation of graph-theoretical modeling and representations, where entities are modeled as nodes (aka. vertices) and pairwise interactions between the entities are modeled as edges. Consequently, numerous operations on graphs to identify a host of structural patterns, motifs, anomalies, and paths have become ubiquitous in the discovery toolkit of the application data scientist.

In fact, graph-theoretic modeling and subsequent analysis have led to numerous fundamental discoveries and insights in many domains of science and engineering applications. For instance, at a molecular scale, analyses of protein networks have led us to an understanding of how proteins interact in groups and complexes in order to affect and implement core molecular functions. At a macro-scale, the engineering and analyses of social networks have led to a paradigm shift in the ways people organize into groups or companies or how retailers advertise their products. A growing base in application use-cases coupled with an explosion in data generated from several data-intensive domains, have generated a need to develop scalable solutions for graph analytics. In fact, it has now become routine to find real-world networks that have millions of nodes and billions of edges [22], while some networks (particularly from social computing) have already breached the trillion-scale [6]. Loading such “extreme-scale” graphs into memory and processing them at a fast response rate impose significant challenges on algorithm design *and* architecture design.

In this paper, we present a concise review of some of the key advances in algorithms and architectures in parallel graph analytics. The field of graph algorithms is vast with decades of results and development. To allow for better focus, we cover a class of graph algorithms that are iterative in nature. This class covers a large body of graph operations in itself, such as (but not limited to) breadth-first search (BFS), shortest path computations, minimum spanning tree, PageRank, community detection, and graph coloring. In Section II, we summarize the current state of art in parallel graph analytics and relate to outstanding challenges. In Section III, we present a review of the algorithmic landscape in parallel graph analytics, placing emphasis on generic abstractions and architecture-agnostic design techniques. In Section IV, we present some of the recent architectural frameworks for graph analytics. Section V concludes the paper with an outlook toward the future, stating key challenges and opportunities to contribute in this active research space.

II. Recent Advances and Challenges

Implementing graph analytics at extreme scales, where the size (as measured in the number of edges) exceeds the billions, is presently an active area of research. Although there is no one ideal metric for capturing the performance of graph implementations, an effective indicator of processing speed used by practitioners in the field, is the number of traversed edges per second (TEPS). Using this measure, benchmarking lists such

978-3-9819263-2-3/DATE19/©2019 EDAA

as Graph500.org have been maintaining a graph-centric ranking of supercomputing platforms based on their ability to execute breadth-first search (BFS). The leading platform on the latest version (November 2018) of this list is the Fujitsu K-supercomputer demonstrating 3.8×10^{13} TEPS on its 648K cores; here the network interconnect of the distributed platform is a six-dimensional torus topology. While this is impressive, such peak performances do not often extend to more complex operations on real-world inputs.

For this reason, the graph algorithms community has also been hosting challenges for identifying the fastest and/or most precise parallel implementations for more advanced operations and on real world inputs. In 2011, the 10th DIMACS graph challenge [2] evaluated multiple implementations, using a collection of real world inputs, on the quality and work performed (a Pareto function) for graph partitioning and clustering. More recently, the 2017 IEEE HPEC Graph Challenge competition [17] introduced a Static Graph Challenge (for triangle counting and k-truss searches) and a Streaming Graph Challenge (for community detection). For triangle and k-truss counting, four champion entries [33, 41, 44] report achieving over 10^9 TEPS[1] performance for a graph with over billion edges. For community detection, the champion entry [14] was achieved using a multithreaded software we had developed [27]; this code achieved 2.2×10^6 TEPS on a social network with 1.8B edges. Noteworthy in all these results is the orders of magnitude decline in performance, from the tens of trillions of TEPS for simpler operations such as BFS, to billions of TEPS for triangle counting and k-truss computations, and to millions of TEPS for the more advanced operations such as clustering. This reduction is an indicator of the increasing complexity of the underlying operation.

From an algorithmic standpoint, BFS performs a single sweep of the entire graph and therefore needing fast and rapid ways to perform arbitrary (unstructured) walks on the graph. Counting subgraph motifs such as triangles and k-trusses rely on the ability to perform arbitrary edge queries (e.g., if edges (i, j) and (i, k) exist, is there also an edge between j and k?), which lead to irregular memory access. Operations such as community detection and partitioning represent the next level of complexity, as these methods involve not only performing multiple sweeps of the graph (both vertices and edges) but also frequent local aggregation of information and multi-level graph transformations. These generate a need for more advanced algorithmic strategies in preprocessing (e.g., ordering), indexing and graph transformations, in order to maximize locality and reduce/mask data movement overheads.

From an architectural standpoint, shared memory multicore architectures have been the default target platform of choice for many graph frameworks (e.g., [1, 15, 20, 39]), partly because of the easiness in accessing graph structures on shared memory. There have also been graph frameworks developed for distributed memory parallel computers (e.g., [36]). A larger set of parallel graph libraries is available for distributed cloud computing platforms [13, 25, 28]; these use both distributed memory and secondary storage (distributed file systems) during computation. These libraries have demonstrated scaling to the trillion-scale edge graphs [5, 6, 45], while they are typically slower compared to running on tightly coupled parallel computers. Few recent works have focused on developing GPU-based manycore accelerations [31, 42, 46]. However, the main challenge with such monolithic acceleration frameworks is that their problem size reach is typically limited by the device memory (in GB), beyond which the latency to partition and redistribute the workload, and exchange computed information on-the-fly, override the compute power benefits of the processing units.

[1]Even though the TEPS measure is defined traditionally only for BFS-type of operations that perform a single pass over the graph, we adapt it here to other operations which do one or more passes, by simply dividing the number of edges in the input by the time taken; this way, the TEPS measure can be used to compare two parallel methods for the same problem on the basis of their solution time (as opposed to number of passes they do on the graph).

III. Parallel Graph Algorithms and Techniques

A. Graph-theoretic Modeling

Graphs are abstract representations that model entities as vertices and the pairwise relationships between entities as edges. Depending on the application use-case, edges may be directed or undirected, and weighted or unweighted. General graphs capture pairwise relationships between entities; there are broadly two types of such graphs—*interaction networks*, where interactions are typically observed or inferred (e.g., protein-protein interaction networks), and *homology networks*, where typically some form of pairwise homology function is applied to infer an edge between two entities (e.g., gene or protein sequence homology networks). Dynamic or time-varying graphs capture the evolution of such relationships (incremental additions and removals, disruptions) over a period of time [16] (e.g., interactions between a community of microbes over time). A bipartite network can be used to model the interactions or pairwise relationships between two different data types [34, 35, 43] (e.g., genes vs. diseases, host vs. pathogens). More generally, heterogeneous graphs can be used to model multiple modalities in entity types (multiple types of vertices) or in relation types (multiple types of edges) [38, 40].

B. Abstractions and Algorithms

A number of graph operations can be implemented as iterative graph codes that perform one or more passes of the input graph, visiting the graph either in parts or as a whole at each step. Within each pass, a fine-granular function is executed typically at a subset of vertices or edges. This has come to be known more widely as the vertex/edge-centric model of computation and has emerged as a preferred mode to write large-scale graph codes in a number of parallel, distributed and multithreaded graph libraries (e.g., [13, 25, 28, 29, 46]). Expressing computation tasks that need to occur at each vertex or edge helps expose parallelism as the graph grows in size.

a) Data-driven and Topology-driven methods: Independent of the graph operation being implemented, many iterative

graph codes can be grouped into one of the two categories—data-driven methods and topology-driven methods—a nomenclature introduced by Lenharth *et al.* [21]. In Algorithms 1 and 2, we provide a simple algorithmic pseudocode to express the main steps that occur in each of these two categories. For ease of exposition, let us consider the vertex-centric paradigm.

Algorithm 1: Data-driven Algorithms

```
Input: Graph G(V, E), seed working set S ⊆ V
Output: An assignment Π: V → ℝ
Let Γ(v) ← influence set for v
Preprocess G(V, E)
repeat
    v ← dequeue(S)
    Π(v) ← Compute a local function f(v, Γ(v))
    for each unvisited v ∈ Γ(v) do
        enqueue(S)
until S = ∅;
return Π
```

Algorithm 2: Topology-driven Algorithms

```
Input: Graph G(V, E)
Output: An assignment Π: V → ℤ
Let Γ(v) ← influence set for v
Preprocess G(V, E)
repeat
    repeat
        for each v ∈ V do
            Π(v) ← Compute a local function f(v, Γ(v))
        Compute a global quality function Q based on Π
    until convergence based on Q;
    G(V, E) ← Perform graph transformation
until convergence based on Q;
return Π
```

Under the data-driven model (Algorithm 1), an implementation performs one or more passes (or iterations) on the input graph; within each such pass, a *subset* (aka. working set) of vertices is processed/visited. These algorithms calculate a local state variable or a local function that represents an assignment at every vertex during each visit (denoted by Π); for instance, this can be a label assignment like a connected component id or a numerical variable such as the shortest path distance from a source. Typically, this assignment is locally determined by examining available values from the immediate neighbors of a vertex (denoted by $\Gamma(v)$). After a value is calculated at a vertex, it activates a subset of its neighbors (from $\Gamma(v)$) for the next iteration. Notable examples of such data-driven methods include topological sort, BFS, the Dijkstra's algorithm for single source shortest path (SSSP), connected component detection (using a BFS-based implementation), and minimum spanning tree algorithms.

Data-driven algorithms benefit from their selective activation of vertices at every iteration, as such an approach is likely to be work-optimal (i.e., visiting every vertex no more than once). However, the size of working set also limits parallelism and potentially creates load imbalanced scenarios of variable workloads across iterations. In comparison, topology-driven algorithms (shown in Algorithm 2) provide a contrasting approach in which *all* vertices within each iteration, which increases parallelism within each iteration and also keeps the workload uniform across the iteration. However, the trade-off lies in sacrificing work optimality as vertices need to be visited multiple times before reaching a state of convergence. However, this is not necessarily a performance bottleneck for low diameter networks, as the convergence rate for most graph operations is a function of the graph diameter (i.e., for information to percolate from one end of the network to another). Notable examples of topology-driven methods include the Bellman-Ford SSSP algorithm [3, 10, 30], PageRank [32], and the `Louvain` heuristic for community detection [4].

b) Heuristics and Techniques: Despite its advantages in ease of programming, both data- and topology-driven algorithmic abstractions pose several challenges when applied at large-scale.

Locality: First, the decisions taken at every vertex are typically influenced by the vertex's immediate neighborhood (Γ). This necessitates access to the changing local neighborhood data, which may be expensive to obtain due to irregularity in data access patterns and costs associated with frequent data movement (memory to chip or remote memory to local memory). This ties back to how the input graph is stored in the shared or distributed memory. Specifically, it is desirable to partition the graph in such a way that neighbors of a vertex are expected to be co-located (i.e., locality-preserving partitioning). This can be obtained by running any of the graph partitioners [18] or simpler heuristics that use vertex properties such as degrees to partition, in a preprocessing step; however the cost of running such partitioners cannot be ignored.

Ordering: Secondly, the *order* in which vertices are processed, could have an impact (one way or another) on the degree of concurrency at every step (for data-driven topology), and on the convergence rate of the solution (for topology-driven topology). In sequential codes, an "ordering" represents a linear permutation of vertices in which the vertices are to be processed at every iteration. However, in parallel codes, such an ordering may not be possible if the underlying parallel algorithm is not guaranteed to be sequentially consistent []. In most parallel implementations, the vertex set is partitioned in some way across the processing elements before processing (e.g., think of a static or dynamic scheduled `parallel for` loop, for the `for` loop in Algorithm 2). Therefore the parallel order reflects at best a partial ordering of vertices. Heuristic solutions to determine a prior, perhaps more informed partial ordering can be constructed as part of a preprocessing step. For instance, the RCM ordering [23] creates a form of degree-ordered ordering of vertices which can be distributed in parallel; whereas a distance-1 coloring of vertices could help separate the vertices into maximally independent sets, such that vertices belonging to each color set are all mutually independent and therefore can be processed concurrently [26].

Approximate computing: Furthermore, in many practical applications of topology-driven methods, the quality function (Q in Algorithm 2) typically grows more rapidly in the initial stages of the iterations than in the later stages (i.e., effectively exhibiting a diminishing returns in quality property). This introduces interesting opportunities in negotiating the resulting performance vs. quality trade-offs. For instance, it is possi-

ble to use various types of approximate computing schemes [11, 32] that adaptively "terminate" vertices spatially across the graph or temporally across iterations, so as to save the time spent per iteration while potentially trading off quality. Such techniques can also help reduce data movement as the algorithm progresses as there is no need to fetch data from terminated vertex neighbors.

IV. Architectural Frameworks for Graph Analytics

Achieving parallel scalability in graph applications remains a significant challenge for architecture design as well, due to the inherent irregularity in computation workload and data movement/access patterns. Over the years, supercomputing architectures have been optimized for large-scale simulation-based applications with floating point-intensive arithmetic. Graph applications on the other hand, epitomize a highly irregular class of applications which is data-bound (and thereby, memory/storage-bound) that perform combinatorial explorations with a mixture of integer and floating point arithmetic.

Modern day platforms support multicores (for the latency-bound parts of the workload) and manycores such as GPUs and other vector architectures (for the throughput-bound parts of the workload). While exploration of such heterogeneous capabilities is important for scaling up graph applications, other considerations such as energy efficiency and low latency in memory access become equally important factors. To this end, the Network-on-Chip (NoC) paradigm provides a conduit. In particular, for an efficient manycore platform, it is essential to align the NoC architecture with the on-chip traffic patterns exhibited by the underlying data-intensive applications. The overall on-chip traffic pattern exhibited by an application is usually a collection of i) inter-thread data exchanges for sharing application data, and ii) on-chip data exchanges that arise from the cache-coherence protocol.

To discuss the challenges associated with designing NoC-based manycore architectures for graph analytics, we consider two graph operations, for detailed study and characterization—community detection [27] and balanced coloring [26]. The computation within community detection is multi-level (based on the pseudocode in Algorithm 2), with alternating parallelized (clustering) and serialized (compaction) characteristics. In addition, community detection involves migration of vertices between the communities during each clustering phase. In contrast, balanced coloring consists of two major parallel phases, initial coloring and redistribution, with bulk of data movement and locking related traffic occurring during redistribution. Due to these characteristics, the two applications generate substantial long-range traffic patterns when run on a multicore, with significant amount of data exchanges involving physically far apart cores [8]. Furthermore, these applications show the presence of one or more hotspot nodes whose traffic injection rates are much higher than that of the average traffic injection rate.

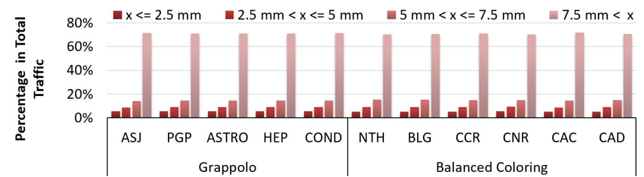

Fig. 1: Distribution of on-chip traffic as measured by the number of hops between source and destination cores for two graph applications: a community detection tool (Grappolo) and a balanced coloring tool. The figure was taken from [8].

In addition, the adopted cache coherence protocols also dictate the nature of the on-chip traffic patterns. To elaborate on this, as examples, we consider the traffic patterns associated with two different cache coherence protocols, Directory and Hammer protocols [24, 37]. Both Directory and Hammer protocols give rise to traffic hotspots and generate significant amount of long-range traffic patterns (as shown in Figure 1) warranting low latency NoC links, even among the physically far apart processing cores.

Due to the long-range and skewed traffic patterns, design of an on-chip interconnect that enables low-latency data exchange, even among physically distant cores will be critical to achieve performance at scale in these graph applications. To achieve low-latency long-range data transfers, we can use emerging NoC architectures like Wireless NoC (WiNoC), 3D NoC etc. In a recent work [8], we explored the design-space for various Network-on-Chip (NoC)-enabled multicore platforms for efficient implementation of graph analytics. We considered three types of NoC architectures, viz., traditional mesh, small-world and high-radix networks, and demonstrated the small-world network-enabled wireless NoC (WiNoC) is the most suitable platform for executing the two targeted graph applications.

The performance of WiNoC for graph analytics can be improved even further by integrating on-chip wireless links with recently proposed SMART (Single-Cycle Multi-hop Asynchronous Repeated Traversal) control mechanism [19]. The NoC latency can be attributed to two factors, i) router stage latency and ii) link traversal latency. We can lower both these factors of the NoC latency for running graph analytics by designing wireless-enabled SMART (WiSMART) NoC [9]. In such an integration, SMART enables efficient router bypass mechanism (lowers the router-stage delay), while wireless links enable single-cycle long-range data transfers (lowers link latency). 3D Network-on-Chip (NoC) is another emerging architecture capable of achieving better performance and lower energy consumption compared to their planar counterparts. Like the wireless links in WiNoC, the vertical links in 3D NoC facilitate establishing long-range short cuts necessary for a small-world networks. By integrating SMART control mechanism with 3D small-world NoC we can design efficient manycore architectures suitable for graph analytics [9].

V. Future Research Directions and Opportunities

Research and development in graph analytics are at the crossroads. On the application side, an explosion in data

has created a need to analyze very large-scale graph inputs (from billions to trillions of edges). On the computing side, we have been experiencing an explosive growth in parallel computing hardware and architectural capabilities. However, the primary question is in how we can generate an ability to harness and harvest the developments in hardware toward supporting extreme-scale graph analytics. The answer in part, lies in our ability to design innovative algorithmic strategies that can be implemented into efficient parallel software with scaling and qualitative guarantees. However, a software that is agnostic to the underlying hardware is limited in its capacity to generate optimal speedup and to fully realize the potential of the architectural features. To this end, a systematic *co-design* is necessary between hardware and software. This is particularly essential for supporting extreme-scale graph analytics as the underlying application landscape is continuously evolving alongside technological developments in architecture. In addition, the field has the advantage of being an emerging field with the reduced burden of legacy tools.

One of the more promising emerging architectures for graph applications is the processing-in-memory (PIM) platform. Although the original development of PIM dates back to the '90s [12], over the last few years it has seen a resurgence. PIM architectures have a unique potential particularly toward graph applications as it allows for moving processing closer to the data (as opposed to the traditional paradigm of moving data to processing). There has been some early work on using PIMs for graph applications [1]. However, further research is necessary in the direction of designing algorithms that are better suited to exploit the features of a PIM architecture.

Other emerging hardware-software co-design opportunities arise in graph applications. Techniques such as approximate computing could prove to be a resourceful way of exploiting trade-offs that arise between performance, energy and precision of computing. As an example, in our recent works, we explored the use of approximate computing techniques for graph operations [7, 32], which yielded significant reductions in runtime, data movement, and on-chip energy consumption. Such techniques that can deliver cross-cutting benefits have an integral role to play in future co-design methodologies, and collectively can aid in supporting scalable graph analytics. With exascale architectures looming in the horizon, this is a unique opportunity that awaits us to innovate at the intersection of algorithms, architectures and applications such as graph analytics.

Acknowledgment

We gratefully acknowledge research support from various funding agencies such as the U.S. National Science Foundation (NSF) and U.S. Department of Energy (DOE), including the following recent grants: CCF 1815467 and DOE award DE-SC-0006516.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 105–117. IEEE, 2015.

[2] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph partitioning and graph clustering, 10th DIMACS implementation challenge workshop. *Contemporary Mathematics*, 588, 2013.

[3] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[5] Fabio Checconi and Fabrizio Petrini. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 425–434. IEEE, 2014.

[6] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[7] Karthi Duraisamy, Hao Lu, Partha Pratim Pande, and Aananth Kalyanaraman. Accelerating graph community detection with approximate updates via an energy-efficient NoC. In *Design Automation Conference (DAC), 2017 54th ACM/IEEE*, pages 1–6. IEEE, 2017.

[8] Karthi Duraisamy, Hao Lu, Partha Pratim Pande, and Ananth Kalyanaraman. High-performance and energy-efficient network-on-chip architectures for graph analytics. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(4):66, 2016.

[9] Karthi Duraisamy and Partha Pratim Pande. Enabling High-Performance SMART NoC Architectures Using On-Chip Wireless Links. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[10] Lester Randolph Ford Jr and Delbert Ray Fulkerson. *Flows in networks*. Princeton university press, 2015.

[11] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed Louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895. IEEE, 2018.

[12] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The Terasys massively parallel PIM array. *Computer*, 28(4):23–31, 1995.

[13] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*, volume 14, pages 599–613, 2014.

[14] Mahantesh Halappanavar, Hao Lu, Ananth Kalyanaraman, and Antonino Tumeo. Scalable static and dynamic community detection using Grappolo. In *IEEE High Performance Extreme Computing Conference (HPEC), 2017*, pages p.6, in press. IEEE, 2017.

[15] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 77–85. ACM, 2013.

[16] Petter Holme and Jari Saramäki. *Temporal networks*. Springer, 2013.

[17] HPEC. IEEE HPEC graph challenge 2017. http://graphchallenge.mit.edu/, September 2017.

[18] George Karypis and Vipin Kumar. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[19] Tushar Krishna, Chia-Hsin Owen Chen, Woo Cheol Kwon, and Li-Shiuan Peh. Breaking the on-chip latency barrier using SMART. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 378–389. IEEE, 2013.

[20] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

[21] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[22] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[23] Wai-Hung Liu and Andrew H Sherman. Comparative analysis of the Cuthill–McKee and the reverse Cuthill–McKee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, 1976.

[24] Mario Lodde, Jose Flich, and Manuel E Acacio. Heterogeneous NoC design for efficient broadcast-based coherence protocol support.

In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 59–66. IEEE, 2012.
[25] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.
[26] Hao Lu, Mahantesh Halappanavar, Daniel Chavarria-Miranda, Assefaw H Gebremedhin, Ajay Panyala, and Ananth Kalyanaraman. Algorithms for balanced graph colorings with applications in parallel computing. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1240–1256, 2017.
[27] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.
[28] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010.
[29] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.
[30] Edward F Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292, 1959.
[31] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.
[32] Ajay Panyala, Omar Subasi, Mahantesh Halappanavar, Ananth Kalyanaraman, Daniel Chavarria-Miranda, and Sriram Krishnamoorthy. Approximate computing techniques for iterative graph algorithms. In *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), 2017*, pages p.10, in press. IEEE, 2017.
[33] Roger Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In *IEEE High Performance Extreme Computing Conference (HPEC), 2017*, 2017.
[34] Paola Pesantez-Cabrera and Ananth Kalyanaraman. Detecting communities in biological bipartite networks. In *ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM-BCB)*, pages 98–107, 2016.
[35] Janet Piñero, Núria Queralt-Rosinach, Àlex Bravo, Jordi Deu-Pons, Anna Bauer-Mehren, Martin Baron, Ferran Sanz, and Laura I Furlong. Disgenet: a discovery platform for the dynamical exploration of human diseases and their genes. *Database*, 2015, 2015.
[36] Steven J Plimpton and Karen D Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.
[37] Alberto Ros, Manuel E Acacio, and José M García. Dealing with traffic-area trade-off in direct coherence protocols for many-core CMPs. In *International Workshop on Advanced Parallel Processing Technologies*, pages 11–27. Springer, 2009.
[38] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and S Yu Philip. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):17–37, 2017.
[39] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.
[40] Yizhou Sun and Jiawei Han. Mining heterogeneous information networks: a structural analysis approach. *ACM SIGKDD Explorations Newsletter*, 14(2):20–28, 2013.
[41] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *IEEE High Performance Extreme Computing Conference (HPEC), 2017*, 2017.
[42] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 11. ACM, 2016.
[43] Maya Wardeh, Claire Risley, Marie Kirsty McIntyre, Christian Setzkorn, and Matthew Baylis. Database of host-pathogen and related species interactions, and their global distribution. *Scientific data*, 2, 2015.
[44] Michael Wolf, Mehmet Deveci, Jonathan Berry, Simon Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *IEEE High Performance Extreme Computing Conference (HPEC), 2017*, 2017.
[45] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. G ra M: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421. ACM, 2015.
[46] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.