HAUPTSEMINAR NEUROENGINEERING

DISTRIBUTED GRAPH ALGORITHMS

January 18, 2016

Edvarts Berzs

Contents

Chapter 1

Introduction

With the immense growth of data volume that is handled in modern data processing systems, simple single-machine systems do not provide sufficient performance for many practical applications. One of the most notorious real-world examples of big data processing is Google with it's ability to handle more than a trillion search queries per year or around 3.5 billion searches per day [1].

For such setup, **distributed** systems are essential, where the data is spread out over many machines, which allows the utilization of vast computing resources, such as from off-site data processing centers.

According to [3], one of the main advantages of using a distributed system is *implementation flexibility*. For example, we can have a multitude of processing units working at different speeds on the same problem or additional processing units can be easily added or existing ones removed.

In the first part of this report some well-known concepts and algorithms of distributed graph processing are investigated, while in the second part the focus is on new developments in the field of distributed graph algorithms.

Chapter 2

Overview of important distributed graph problems

In order to investigate the main principles of distributed algorithms, a short review of some fundamentally important graph problems, for which distributed algorithms exist, is given. In this report, unless stated otherwise, when mentioning distributed algorithms, we speak of graphs whose nodes are distributed over geographically far-apart processing units. In other words, a clear distinction between parallel and distributed systems is made [16] [3].

It is important to note, that not all problems for which a serial algorithm exists, can be designed in a distributed manner. One class of algorithms, that cannot be distributed are iterative algorithms, which require data from a previous calculations to acquire the current result. Examples include iterative numerical algorithms like Newton's method or the Three-Body-Problem.

Additionally, even though it might be possible to distribute some algorithm over many processing units, it does not guarantee a performance increase of the whole program, for example, due to distribution, control or inter-process communication overhead. Although it is nearly impossible to exactly calculate the performance gain when adding additional processing units, some theoretical results have been proposed, for example, Amdahl's Law [2].

2.1 Maximum-Flow Minimum-Cut

Maximum-Flow Minimum-Cut (max-flow min-cut) is one of the graph theory problems, that has been extensively studied due to it's practical importance, for example in many computer vision applications [24].

After the invention of the Ford-Fulkerson algorithm for finding the maximum flow in

a graph in 1956, a multitude of methods with various time and memory complexity have been devised to tackle this problem. A list of some distributed max-flow min-cut algorithms with their respective messaging and time complexities is given in [18]. However, all these algorithms assume a shared memory that can be accessed by all processing units simultaneously [22] [7]. This means, that these algorithms do not scale well for truly distributed systems, where signal delays between far-away locations play an important role.

In [4] the graph is split up so that each processing unit holds all nodes, however, the edges are distributed over all processors. On the Facebook example, this would mean that every processor holds an entry for each user's information, while the interconnections between them are split up. For large graphs saving all nodes on each processing unit might still be impracticable due to excessive memory consumption. Additionally, for such a setup it might be very difficult to keep the representation up-to-date for highly dynamic systems, where the graph is constantly changing.

A truly distributed graph processing framework is presented in [24], where the graph is split over multiple machines and solved separately on each of them. The authors of [24] mention, that not all graphs converge to a solution when split up, if the whole graph has multiple solutions. The research of this effect can be considered as an open problem for the max-flow min-cut problem.

2.2 GRAPH COLORING

In it's most common setting the problem of graph coloring is to assign a color to each node in a graph such that no two nodes, that share an edge, have the same color. A typical goal is to find a color assignment with the least number of different colors.

Graph coloring is widely used in practice, for example in various systems that require user scheduling, such as various multiple access schemes in telecommunications [13].

Graph coloring is an NP-Complete problem. The lowest achievable complexity for a deterministic graph coloring algorithm for distributed systems is still unknown [14] and is subject to on-going research. For practical applications, in state-of-the-art distributed graph coloring algorithms, often some form of distributed **symmetry breaking** is applied [21], which in a way randomizes the algorithm and makes it non-deterministic. Surprisingly, despite the non-determinism of symmetry breaking, this method improves the performance of graph coloring algorithms.

In [13] a very extensive review of recent trends in distributed graph coloring algorithms is given. One of the conclusions of [13] is, that symmetry breaking or randomization, which

proves to be useful, is still poorly understood and that additional research in the field is necessary.

2.3 Minimum Spanning Tree

The aim of minimum spanning tree or MST algorithm is to choose some edges from a graph in order to connect all nodes with the lowest total weight of the edges.

The classical serial MST algorithms are Borůvka's, Prim's and Kruskal's algorithms. Since Prim's algorithm is iterative, every update step changes the global state of the graph, so iteration cannot be performed in parallel. However, according to [16], the graph can be spread out over many processors, by assigning each unit some columns V_i of the adjacency matrix, that correspond to the nodes that are assigned to node i. Some processor P_0 is assigned the coordinator role, per iteration it gathers the local result from each processor P_i and decides, which edge is to be added to the MST. This result is then broadcast back to all processors. The communication overhead for such a system is very large and it depends on the number of disjoint sets - the larger the amount, the more communication takes place in the system. An additional problem is, that the coordinating processor must wait for all processing units to deliver their local results, before it can decide for the "winner" in the respective iteration. Especially if the sets are not well-balanced, it can happen, that some processors are in idle state while waiting for the slower processors to finish their calculations.

For this reason, an asynchronous approach is desirable. An asynchronous MST algorithm can be found in [23], unfortunately it suffers from large messaging overhead, which most likely makes it infeasible for large graphs.

Minimum spanning tree problem is a prominent example of a problem, which is difficult or maybe even impossible to distribute appropriately.

2.4 Shortest path

Shortest path finding problems are among the most important ones in graph theory, due to their wide range of applications in practice, such as in routing algorithms in computer networks, path search between cities for logistics and many more. Additionally, many complicated problems include or can be reduced to a shortest path finding problem, for example natural language processing NLP and video game engine design problems. Here, the focus is on a specific subset of shortest path problems, namely on All-Pairs Shortest Path. The goal here is to find the shortest paths between any pairs of nodes in a graph.

On a single machine, the problem can be solved by applying the Floyd-Warshall algorithm. Alternatively, it is possible to apply the single-source shortest-path Dijkstra or Bellmann-Ford algorithms for each node in a graph. Dijkstra is to be chosen, if all edge weights are positive and the graph is sparse, whereas Bellman-Ford is preferred, if some edges can also have negative weights.

Floyd's algorithm can be distributed by assigning each processor p a square sub-matrix of the computing matrix [16]. Effectively, it is possible to reduce the run-time of the distributed version by a factor of p when compared with the serial version. Naturally, some additional overhead will be present in the distributed version.

There are two distinct ways to parallelize the **Dijkstra** algorithm for the all-pairs shortest path problem. These are *source partitioned formulation* and *source parallel formulation* [16].

In Dijkstra's *source partitioned formulation* each processor has *global* knowledge of the whole graph and some set of nodes assigned to it for processing. This enables each processor to calculate the shortest paths for the vertices assigned to it independently of the others. The major downside of this approach is that each processing unit has to save the whole graph locally, which is infeasible for truly distributed systems when handling huge graphs.

In Dijkstra's *source parallel formulation*, however, each processor handles only some sub-graph of the whole graph, which allows the algorithm to save a lot of memory, since now only one full copy of the graph is distributed in the system, as opposed to p copies of the graph as in source partitioned case with p being the number of processors [16] [5] [25]. A well-illustrated example can be found in [25]. It also clearly illustrates one of the biggest problems with this approach, namely, the number of exchanged messages between the processing units grows very fast as the algorithm progresses. This number depends on the number of clusters, as well as on the number of interconnections between these clusters, which corresponds to the number of boundary nodes. Parallel Dijkstra algorithm has potential to yield considerable performance improvement, however, the performance can be severely deprecated, if the partitioning is poor. Note, that in the previously mentioned implementations a planar graph is implicitly assumed, otherwise it would not be possible to partition the graph properly. It is apparent, that none of the parallel Dijkstra implementations are practically applicable as geographically distributed systems, due to the large latency of these systems.

Interestingly, [17] conveys the idea, that using the all-pairs shortest paths algorithm is impractical for large graphs, due to the need for $\mathcal{O}(|V^2|)$ memory.

2.5 DISCUSSION OF DISTRIBUTED GRAPH ALGORITHMS

It is evident, that distributed graph algorithms offer a huge potential performance gain over serial algorithms due to a much larger available processing power, however their implementation is often cumbersome. Some of the main problems are:

- As discussed in [16], one major issue with any distributed graph algorithm is, how to evenly spread the data between all processing units. This is called the "*load-balancing*" or "*graph partitioning*" problem. In [16] it is noted, that partitioning an adjacency matrix, which is commonly used for dense graphs, is rather trivial, while partitioning an adjacency list, which is memory-efficient for sparse graphs, is non-trivial and a well-performing parallelization is difficult to achieve. One of the possible solutions for partitioning is to use some clustering or community detection algorithm [6]. Despite many years of research there is still no fully satisfactory *go-to* solution for clustering problems.
- Maintaining appropriate communication between the processing units can be challenging due to necessary time synchronization, collision avoidance, link reliability, simultaneous shared memory access and signal propagation delay.
- Some algorithms, for example, ones that require the knowledge of the global state of the system or iterative algorithms that base their results on the previous steps are inherently not parallelizable and thus not distributable.

2.6 PRACTICAL EXAMPLES OF DISTRIBUTED ALGORITHMS

To convince the reader, that distributed graph algorithms are also of practical importance, some practical implementation examples are given.

MapReduce is a programming model for processing large data sets in distributed systems [10]. One of the best-known implementations of MapReduce is the Apache Hadoop BigData processing framework. MapReduce program contains two parts with the typical tasks being:

- **MAP**: Convert data to tuples and sort them [9].
- **REDUCE**: Distribute data to processing units

The main advantages of MapReduce are the ability to hide the underlying distributed data structure and the support for highly scalable computations [19].

Despite it's wide-spread use, MapReduce framework has some intrinsic problems. Most prominently, MapReduce is *not* suitable for iterative graph processing with constantly changing data [10] [19]. Additionally, MapReduce is very disk read and write intensive, because it copies the complete data set on each iteration [10], which makes it inefficient for repetitive actions with small changes on each step, which is the behavior of many iterative algorithms. For this reason, a partially asynchronous MapReduce version is proposed in [11] which reduces the number of global synchronization steps. Although the performance is reported to increase up to 8 times for some applications, the programming effort is increased, because the implemented algorithms must also work in asynchronous mode and the recovery time from an error in calculations is significantly increased.

Chapter 3

Modern approach - Vertex-centered thinking

Due to the problems mentioned in the previous chapter, we need a more sophisticated method to process very large graph structures which cannot even be stored in one single location due to their size and hardware storage limitations.

Recently, a new framework has been introduced, which is called "Think Like A Vertex" **TLAV**. The rest of this chapter is mainly based on [19].

In *vertex-centered frameworks* each node is regarded as a separate entity, which is different from traditional graph algorithms, where per step the whole graph or some sub-graph with multiple nodes is considered. Per iteration step, which is called a *superstep* in the TLAV framework, every vertex receives input data from it's adjacent vertices and then executes some user-defined program. The result is again communicated to the surrounding vertices. Each superstep is subject to strong synchronization so that at the beginning of each superstep the system is in a defined state. One downside of using the superstep concept is, that each superstep takes *as long as the slowest vertex* [17], so some balance between the node computations should be achieved or, alternatively, all the computations should be kept small and short.

One example of an algorithm, that can be implemented as a vertex-centric algorithm is the PageRank algorithm [17].

The four major cornerstones of TLAV, that have to be decided during design phase are:

1. Timing / Scheduling
2. Communication / Data access
3. Execution model / Data flow

4. Partitioning / Data storage

The main advantages of TLAV are:

- improved locality
- linear scalability
- programming simplicity
- widely applicability

Whereas the disadvantages are:

- the need for a thorough understanding of the framework
- superstep synchronization might be difficult to achieve for geographically distributed systems
- a lot of inter-process communication
- some problems do not fit to the TLAV framework, like graph coloring

3.1 TIMING

Timing describes, how the active vertices are chosen by a scheduler for execution.

3.1.1 Synchronous algorithms

Effectively, the TLAV framework reduces the computational complexity of the algorithms by trading it for significantly increased communication overhead between the nodes. Synchronous systems have been shown [17] to exhibit a linear increase in run-time complexity with the number of vertices. A significant problem is, that some synchronous algorithms like graph coloring might not converge in some cases [17].

3.1.2 Asynchronous algorithms

An important advantage of asynchronous algorithms is that they can significantly decrease the time needed for inter-process communication, because a stringent synchronization is not needed.

The state of the system, when a few computations in a superstep take much longer time than most others, is known as the ”straggler”problem. It can be avoided by adapting the vertex execution order dynamically with a scheduler. Subsequently, such dynamic, asynchronous models yield improved performance, although they have increased complexity.

To summarize, *synchronous algorithms* are:

- an appropriate choice for systems, that require a lot of disk reads/writes such as PageRank
- rather easy to validate

whereas *asynchronous algorithms*

- are more suited for CPU-bound algorithms with variable workloads
- can handle large computation delays

3.1.3 Hybrid algorithms

A possibility to actively switch between synchronous and asynchronous algorithms has also been investigated [27]. An improved computational performance is achieved by gathering information about each iteration and dynamically changing the execution mode. This method has a large scheduling overhead, which most likely renders such systems infeasible for practice.

3.2 COMMUNICATION

The communication includes the inter-process communication, as well as data access control.

The prevalent method is called *Message Passing*. Local vertex data is sent via a message, usually, to some or all of it's neighbors. Message passing is commonly used with synchronized timing.

Another option is to use *Shared memory*. It's upside is the small messaging overhead, however there are major disadvantages when using shared memory, like data races, the need for a very large memory bandwidth and extremely low latency of the memory unit, to support concurrent memory access. Shared memory solutions are particularly inefficient for geographically distributed systems due to the latency of data transfer.

One less common method is called *active messaging*, where the messages, that are transferred over the network, contain both the data to be manipulated, as well as the function, that is to be applied to it at each node. It provides more flexibility, but also incurs an increased messaging overhead.

3.3 EXECUTION MODEL

Execution model describes, how data moves in the system during computation.

Push stands for updating of neighbors of the current vertex and **pull** for receiving data from the neighboring vertices. Multiple versions of the execution models can be found in literature, with the two-phase model being the most common choice.
One-Phase: A single computation function receives some input data, does the necessary calculations and sends the results to incident edges. An example is the Pregel framework [17].
Two-Phase: **Scatter-Gather** Model. Other names: **Signal-Collect** and **Scatter-Combine** models. A wide range of various implementations exists [19].

1. **Scatter** distributes vertex value to neighbors.
2. **Gather** collects input data and performs update

Most one-phase frameworks can be converted to two-phase systems.
Three-Phase: **Gather-Apply-Scatter**. Example: PowerGraph [8]

1. **Gather** performs a generic summation of input data
2. **Apply** updates center vertex value
3. **Scatter** distributes updates to neighbors

Edge-Centric Iterate over edges instead of vertices. Similar to two-phase Scatter-Gather model.

3.4 PARTITIONING

The main goal when dividing a large graph to fit in distributed memory is to approximately evenly distribute the vertices over all processing units, while minimizing the number of interconnecting edges between them. For a more formal investigation of the k-way graph partitioning problem refer to [12]. It has been shown that exact edge partitioning problem is

NP-Hard and for this reason, commonly a sub-optimal solution is acceptable.
Standard approach is recursive bisection, however, other approaches like the multilevel k-way partitioning algorithm exist [12]. All the standard approaches have the problem of being too computationally expensive for practical implementations even for medium size graphs. An additional problem is that these algorithms require random memory access when executed. For this reason, alternative methods are sought after.

3.4.1 Distributed heuristics

Distributed heuristic algorithms require no or little coordination. The main approach prevalent in literature is *label propagation*, which is based on community detection which is the finding of groups of nodes with similar properties [20]. There are, however, two important differences for the graph partitioning case:

- partitions cannot overlap with each other
- the number of partitions has to be preset and must agree with the number of processing units, although a single processing unit might have more than one sub-graph assigned to it

The approach with label propagation has some problems:

- some centralized coordination is necessary in order to obtain nearly equally sized, balanced partitions
- it can happen, that one of the labels dominates the others. This is called *densification*. [19]

3.4.2 Streaming

The main idea is to look at each part of the input graph and decide, where to place it, in our case, to which processing unit to send the specific part of the graph. The whole graph is parsed in a single-pass, so this partitioning method can be implemented at the same time with reading the graph from disk, thus providing an online solution with little amount of overhead. The *streaming partitioner* is aware of the current vertex, that is to be assigned to some partition, as well as of the already created partitioned graph.[19].

The best-performing practical implementations use greedy heuristics to decide the placement of the vertices. In this context, greediness means assigning the current vertex to the

partition with which it shares the most edges. Calculation of the optimal partition to assign the vertex to gets increasingly computationally complex with the number of already added nodes. A problem with this approach is that it rarely finds the optimal partition, in fact any optimality guarantees are hard to derive due to the strongly iterative characteristics of the algorithms. More precisely, once a node has been assigned to some partition, at least in the classical streaming partitioner, it can never be reassigned to some other partition thus limiting the achievable efficiency of the algorithm.

3.4.3 Vertex cuts

Partitioning a graph based on vertex cuts is equivalent to partitioning a graph by edges instead of vertices. With this approach, each edge is uniquely assigned to one processing unit, while vertices can be split across multiple units. Especially for graphs with high-degree vertices, the data flow between processing units is severely decreased, because now only changes to cut vertices are propagated through the network instead of propagating the changes of edges [19].

3.4.4 Dynamic repartitioning

Even if vertices are equally distributed among the processing units in an effective manner, we might have a strong imbalance in the amount of actually active vertices in a superstep due to specifics of the algorithm that is running in the system, thus impairing the performance of our system. As a solution, *active repartitioning* of the vertex sets is allowed. For this method to be reasonable, the overhead from repartitioning the graph must be smaller than the performance loss due to partition imbalance. It is important to note, that certain constraints must be implemented in order to circumvent the previously mentioned problem of densification, which is likely to occur, if a greedy strategy is used.

Most empirical experiments do not yield a satisfying performance increase when applying dynamic partitioning, however, we note, that this method could significantly improve performance for very fast changing, dynamic vertex sets.

3.4.5 Conclusions about TLAV

As mentioned before, the TLAV framework can be applied to practical graph algorithms like the PageRank algorithm. An upside of the TLAV framework is the granularity of the system -

it is easy to add new nodes, which, for example, would correspond to new pages to PageRank algorithm.

One downside of TLAV is that it completely ignores any global structure that might be present in the graph that is being processed, that is, it ignores the partitioning and global structure of the graph. As a consequence, the computation result from one node might take a long time to propagate to a node that is many hops away.

A proposed improvement to TLAV is presented in [26] as ”Think like a graph” framework, which, instead of considering separate nodes, considers sub-graphs with multiple nodes. This provides more flexibility in local data handling, as well as significantly improves inter-process communication quality and latency significantly.

Chapter 4

Conclusions

The distributed graph algorithms is a very broad topic with many aspects that need to be considered when designing the algorithms. In many cases, for a general distributed graph algorithm, one will be able to find some data sets, where it fails to yield adequate performance or even correct results. For this reason, in practice, it is the job of the engineers to tailor the particular algorithm to the application at hand to ensure a proper execution with a reasonable performance. Clearly, this has been done successfully in practice, as can be seen in such Big Data usage examples like Google's ability to handle a trillion search queries per year or Facebook's ability to maintain the network of a billion interacting people.

A plethora of algorithms with shared memory access on the back-end of their inner workings are available for distributed graph processing. This is, however, not feasible for truly geographically distributed systems in most practical use cases, due to the signal propagation delay incurred by the large distances between the processing units.

It is evident, that additional research is needed in the field of distributed graph algorithms, that would potentially enable us to create graph algorithms with small amount of inter-process communication and no need for shared memory access.

Bibliography

[1] Google search statistics. http://www.internetlivestats.com/google-search-statistics/.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30. *Solid-State Circuits Society Newsletter, IEEE*, 12(3):19–20, Summer 2007.

[3] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

[4] Benoıt Dancoisne, Emilien Dupont, and William Zhang. Distributed max-flow in spark. 2015.

[5] Gabriele Di Stefano, Alberto Petricola, and Christos Zaroliagis. On the implementation of parallel shortest path algorithms on a supercomputer. In *Parallel and Distributed Processing and Applications*, pages 406–417. Springer, 2006.

[6] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[7] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[8] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[9] IBM. What is mapreduce? https://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/.

[10] Tomasz Kajdanowicz, Przemyslaw Kazienko, and Wojciech Indyk. Parallel processing of large graphs. *Future Generation Computer Systems*, 32:324 – 337, 2014.

[11] Karthik Kambatla, Naresh Rapolu, Suresh Jagannathan, and Ananth Grama. Asynchronous algorithms in mapreduce. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 245–254. IEEE, 2010.

[12] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[13] Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 138–144. ACM, 2009.

[14] Fabian Kuhn and Rogert Wattenhofer. On the complexity of distributed graph coloring. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 7–15. ACM, 2006.

[15] Mi Lu and H. Lin. Parallel algorithms for the longest common subsequence problem. *Parallel and Distributed Systems, IEEE Transactions on*, 5(8):835–848, 1994.

[16] Aditi Majumder. Parallel and distributed graph algorithms and their applications.

[17] 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.

[18] John Michael Marberg. *An o (n2m1/2) distributed max-flow algorithm*. University of California at Los Angeles, 1987.

[19] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing.

[20] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[21] Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 257–266. ACM, 2010.

[22] Yossi Shiloach and Uzi Vishkin. An o (n 2 log n) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[23] Gurdip Singh and Arthur J Bernstein. A highly asynchronous minimum spanning tree protocol. *Distributed Computing*, 8(3):151–161, 1995.

[24] Petter Strandmark and Fredrik Kahl. Parallel and distributed graph cuts by dual decomposition. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2085–2092. IEEE, 2010.

[25] Yuxin Tang, Yunquan Zhang, and Hu Chen. A parallel shortest path algorithm based on graph-partitioning and iterative correcting. In *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, pages 155–161, Sept 2008.

[26] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[27] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 194–204. ACM, 2015.