Analyzing system characteristics for graph algorithms across different graph frameworks

Aakarsh Agarwal
Dept. of Computer Sciences
University of Wisconsin-Madison
aakarsh.agarwal@wisc.edu

Mohil Patel
Dept. of Computer Sciences
University of Wisconsin-Madison
mpatel48@wisc.edu

Sweksha Shukla
Dept. of Computer Sciences
University of Wisconsin-Madison
sshukla24@wisc.edu

***Abstract*—Graphs have become increasingly common data structure. Large graphs exist as social network, web graphs, etc. Because of their many real-world applications, we have seen an increase in graph specific frameworks which can run graph specific algorithms. In this paper we want to analyze performance and system characteristics of different graph frameworks on well-known graph algorithms. We have analyzed four frameworks namely GraphChi, GraphX, GraphFrames and Spark. We have collected results for three algorithms: PageRank, Connected Components and Triangle Counting. For each of the algorithms, we have collected the following metrics: Execution Time, CPU Usage, Disk Read/Write Throughput, Memory Usage and Network Throughput. From the experiments we have found that GraphChi is a better choice for smaller graphs. But as graph size increases to the order of 100 million edges, GraphX seems to have an edge. We have also observed how specific system design philisophies across different frameworks can affect all the above mentioned metrics.**

I. Introduction

Graphs as a data structure can be represented as a set of vertices(V) and edges (E), i.e $G = (V, E)$. Where vertices are the nodes in the graphs and edges are the linking between pair of nodes. Each vertex and edge can have its own set of properties, which can be used to capture information about them. This is a versatile data structure and can be used to capture a lot of real-world data. For e.g. social networks (Facebook friends, twitter followers, etc), web graphs (hyperlinks between web pages), protein interaction graphs, etc. In the recent decade we have witnessed the graph size grow from few million edges to billions of edges. Processing such huge graphs efficiently is a new technical challenge and there has been many research work in that direction.

In the last decade there has been much work in designing graph processing frameworks to work efficiently with huge graphs. The research work spans in many directions. Ranging from single machine frameworks, like GraphChi [1] which focus processing huge graphs efficiently on a single machine using Parallel Sliding Window concept to minimize memory usage, to multi-machine frameworks like GraphX [3] on Apache Spark [2], which designs specialized operators in Spark to optimize graph processing and also uses better data partitioning schemes to minimize the communication overheads. Each of these frameworks have their own benefits and trade offs, and are designed for different settings. For e.g. GraphFrames [4] is a multi-node framework which can work effectively with Spark SQL [5] and process graphs as relational tables. Using GraphFrames in a data processing environment, which already includes Spark SQL, will be easier, and it also makes it easy for the developers as they can use their SQL knowledge directly for making graph algorithms. This reduces the development time.

In this paper, we analyzed the performance and system characteristics of different graph frameworks on well-known graph algorithms, in an attempt to answer the question: which framework is better in what settings. Graph algorithms that we analyzed in the paper are PageRank [6], Connected Components and Triangle Counting. These are all well known graph algorithms that are used in real-world applications for different tasks. We compared the performance of these algorithms against four different graph frameworks. Details about the frameworks are in Table I.

TABLE I: Graph Frameworks

Frameworks	Distributed or Single Machine	Design Philosophy
Spark [2]	Multi-Node Distributed Framework	Generalized Distributed Data Processing Framework
GraphX [3]	Multi-Node Distributed Framework	Built on top of Spark for optimized Graph Processing
GraphFrames [4]	Multi-Node Distributed Framework	View graphs as relational table & operate on them using queries
GraphChi [1]	Singe Machine Framework	Optimized for low memory usage to operate on huge graphs

The outline for the rest of the paper is as follows. Section II discusses related works and section III covers the background of frameworks shown in Table I. Section IV describes the Experimental Setup for all the different frameworks and how we have collected the performance and system characteristics data. Section V is dedicated to the results for each algorithms we analyzed, showing the Execution time, CPU usage, Disk throughput, Memory usage and Network throughput for all the frameworks. Section VI concludes with our takeaways from this analysis. Lastly, section VII describes potential future works.

II. Related Work

Existing graph-parallel abstractions, in the domain of distributed systems, such as Pregel [11] and GraphLab [12]

encode computation as vertex programs which run in parallel and interact along edges in the graph. However, real-world graphs often follow a power-law degree distribution, which implies that a few nodes are very highly connected, and many nodes are very weakly connected. This leads to inefficient partitioning in the distributed environment. PowerGraph [13] abstraction addresses this issue by exploiting the Gather-Apply-Scatter (GAS) model of computation to factor vertex-programs over edges, splitting high-degree vertices and exposing greater parallelism in natural graphs. All these distributed systems, even if optimized to any levels, suffer from minimum I/O and communication overhead among several nodes which motivates using single machine instead. DGL-KE [14] is one such approach, which uses a single machine with large CPU memory and transfers batches synchronously. This approach is limited by CPU memory capacity and slow training time due to the data movement.

Above mentioned frameworks are domain specific implementations. In this paper, we also focus on the analysis of the frameworks built on the idea of using general purpose dataflow platform in the distributed setting. We believe that exploiting the power of generic implementation methodology in the context of big data systems can prove to be revolutionary. It is also interesting to explore the possibility of single machine frameworks replacing the distributed frameworks. Our motivation for the selection of algorithms for this benchmarking project is based on these observations.

III. Background

In this section, we cover the system design details and philosophy of the frameworks (Table I) that we have analyzed in this study.

GraphChi [1]: Processing graphs using distributed computing frameworks can be inefficient due to all the communication overhead involved. GraphChi tries to overcome this challenge, by using a single machine which can process large graphs efficiently. GraphChi is a disk-based system for computing efficiently on graphs with billions of edges. Using well-known methods it breaks down a large graph into small parts and uses a novel Parallel-Sliding Windows (PSW) method to process them efficiently. It also provides theoretical guarantees on number of disk reads/writes.

GraphX [3]: With many specialized distributed graph processing frameworks available, GraphX sets out to answer the question: Can we use generalized distributed frameworks with optimizations to achieve same performance as specialized frameworks? To answer this question they designed an optimized API over Spark to do graph processing. They did two major optimizations to improve the performance: first is the vertex-cut partitioning of graphs to minimize communication across nodes and second is designing specialized join algorithms & materialized view optimization which enables faster processing for graphs.

GraphFrames [4]: There are some frameworks which are designed as graph-on-RDBMS systems. GraphFrames sets out to generalize this idea of graph-on-RDBMS systems. It is an integrated system that lets users combine graph algorithms, pattern matching and relational queries, and optimizes work across them. GraphFrames is built on top of Spark SQL. They have designed an execution strategy which generalizes the strategies of graph-on-RDMS systems to support multiple views of graphs. And they have designed a graph-aware query optimization algorithm which takes into account available views.

Spark [2]: Spark is a generalized distributed data processing framework. It introduces Resilient Distributed Datasets (RDDs), which are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators. Compared to other systems, RDDs provide an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data. This allows Spark to perform better than other systems in many tasks. We have added Spark in our analysis to have a comparison between a basic implementation and specific libraries built on top of Spark.

IV. Experimental Setup and Methodology

We have run all our experiments on Cloudlab [7]. For multi-node distributed frameworks (as detailed in Table I), we have run a three VM setup, where one VM was working as master & worker (simultaneously) and the rest two VMs were both worker nodes. For single machine framework, we have run all the experiments on a single VM setup. Each VM has the same hardware configuration: Intel Xeon E5-2630 with 5 cores, 32 GB of RAM and 96 GB of disk space. We ensured that all the frameworks were configured to utilize all the hardware resource to full extent.

PageRank [6] works on directed graphs. We have run experiments for PageRank on the datasets shown in Table II. On the other hand, Connected Components and Triangle Counting both run on undirected graphs. For both the algorithms, we used the datasets shown in Table III.

TABLE II: PageRank Datasets

Dataset	Nodes	Edges
Berkeley-Stanford Web Graph	685,230	7,600,595
Pokec social network	1,632,803	30,622,564
LiveJournal social network	4,847,571	68,993,773

All datasets are taken from SNAP [8]

While running experiments, we ensured that each framework is configured to utilize all the available hardware resources. To capture system characteristics, we have used dstat (linux tool) to collect the data. For multi-node frameworks, we have collected dstat on all the 3 nodes as shown in plots in the Results section. Before running each experiment, we have cleared the OS filesystem caches to ensure that disk read/write values are reliable. Also, for multi-node frameworks, we

TABLE III: Connected Components & Triangle Counting Datasets

Dataset	Nodes	Edges
Social circles: Facebook	4,039	88,234
Twitch Gamers Social Network	168,114	6,797,557
LiveJournal network	3,997,962	34,681,189

All datasets are taken from SNAP [8]

have run experiments using HDFS as the filesystem backend. Whereas, for single machine framework, we have the system OS (Ubuntu 18.4) as filesystem backend.

V. Results

In this section, we will be discussing the results for each algorithm that we analyzed. For each algorithm, we analyzed five metrics - Execution time, CPU Usage, Disk Throughput, Memory Usage and Network Usage.

A. *Pagerank*

The plot in figure 1 shows the time taken by different frameworks for PageRank Algorithm. We have collected data for 50 iterations, fixed for all the datasets. The general trend is that GraphChi is the fastest framework for small and medium graphs. But for the largest graph that we analyzed, GraphChi is taking same time as GraphX. In general, seeing the trend, we expect the performance of GraphX to be better for larger graphs. Another peculiar behavior we see is that GraphFrames, even after being a graph specific framework, is having the worst performance than Spark.

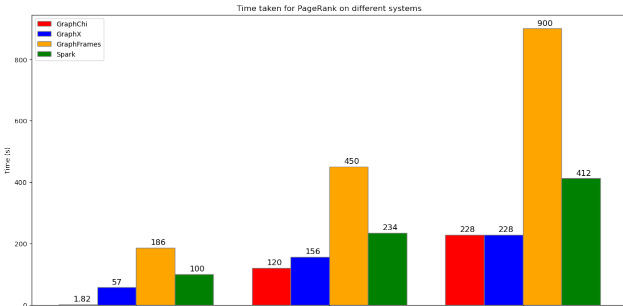

Fig. 1: Execution Time for PageRank

CPU Usage: For the CPU usage, we collected the data for largest graph in the Table II, i.e. LiveJournal Social Network. The first plot, as shown in figure 2, is plotting the CPU usage for node-0 for all the frameworks. Few important trends that we see here are, first even though GraphChi has a very good performance, it seems to be bottlenecked at 60% CPU Usage. We suspect that this is because GraphChi is designed to have low memory usage, thus it is getting bottlenecked due to disk reads/writes.

The second plot in figure 3 shows the CPU Usage for all the three nodes for multi-node frameworks. A peculiar

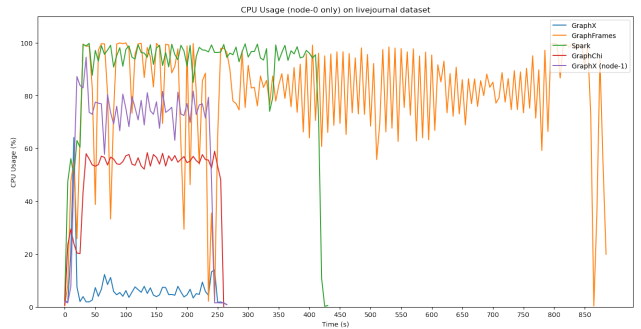

Fig. 2: PageRank CPU Usage Node-0

trend that we see here is that GraphX only has CPU usage on node-1 and node-2. Node-0 for GraphX is having very low CPU usage. In general we have seen a trend that for smaller graphs, GraphX only uses node-1 and as the graph size increases we see node-2 CPU usage. Considering the trend we expect that we will see all three nodes being used once we go for even larger graphs. This behavior could be due to GraphX's specialized vertex-cut partitioning scheme and other optimizations which are designed to minimize communication overhead.

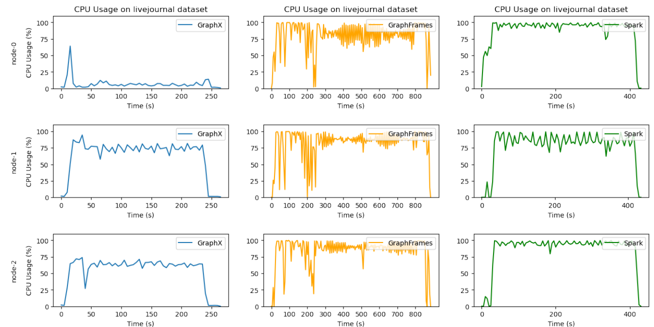

Fig. 3: PageRank CPU Usage for all 3 nodes

Disk Throughput: In disk throughput we are plotting disk reads & write throughput. Before collecting any data we have cleared the OS filesystem caches to ensure that the reads are happening from the disk. The figure 4 shows the node-0 disk reads and writes for all the frameworks (Note: We have plotted node-1 for GraphX also because of its distinct behavior of not using node-0 as seen in figure 3). In the disk reads we see that GraphChi is having a high disk read throughput, which is because it is designed to have sequential disk reads. In disk writes we observe that GraphX node-0 is having four huge disk write spikes. Other than that disk writes for Spark and GraphFrames are constantly happening at a fixed rate. Majority of these writes corresponds to temporary file writes.

Memory Usage: Similar to CPU Usage, in memory usage also we have plotted two graphs. The first plot is shown

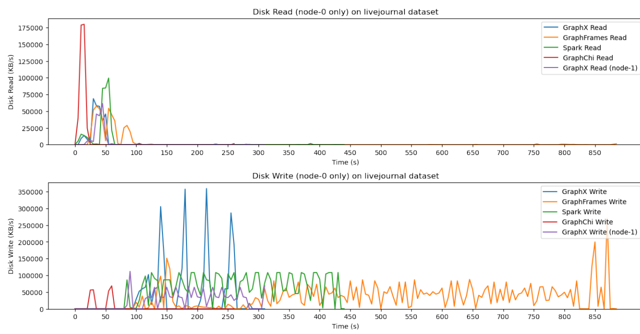

Fig. 4: PageRank Disk Throughput

in figure 5. This plot shows the memory usage for all the frameworks for node-0. We see a clear trend that GraphChi is having very low memory usage, which is expected as it is designed to have low memory usage. Another trend that we see repeatedly across all the algorithms is that GraphFrames is always having the highest memory usage across all the frameworks.

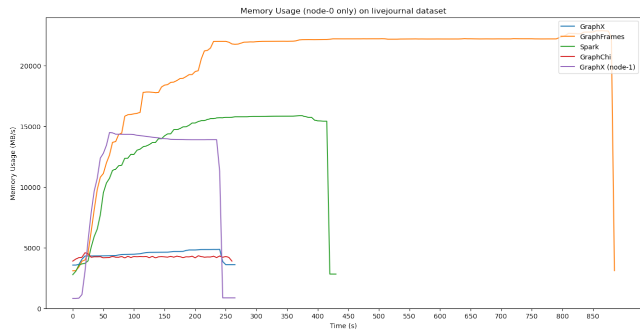

Fig. 5: PageRank Memory Usage Node-0

We have also plotted the memory usage for all the nodes as shown in figure 6. GraphFrames and Spark both show similar memory usage for all the three nodes. On the other hand, we only see memory usage for node-1 and node-2 for GraphX. This trend is same as what we have seen for CPU Usage earlier.

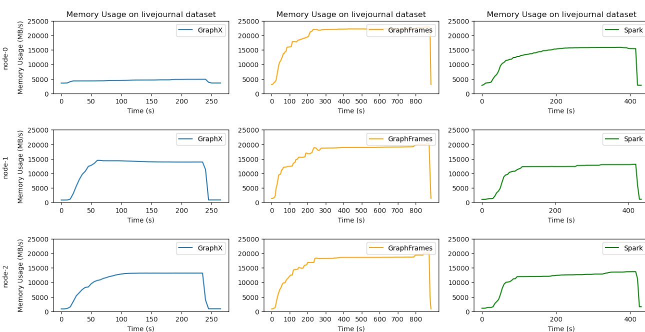

Fig. 6: PageRank Memory Usage all nodes

Network Usage: The last system characteristic that we analyzed is the network throughput. We have analyzed both the network receive and send behavior. Note that network behavior plots are only relevant for multi-node frameworks. Figures 7 and 8 show the network receive and send behavior for all the multi-node frameworks. Here we see that the network activity is very less for GraphX compared to GraphFrames or Spark. We think that this behavior is due to GraphX's (vertex-cut) partitioning scheme which is designed to minimize communication overhead. Also this property can be a major contributor to its good performance.

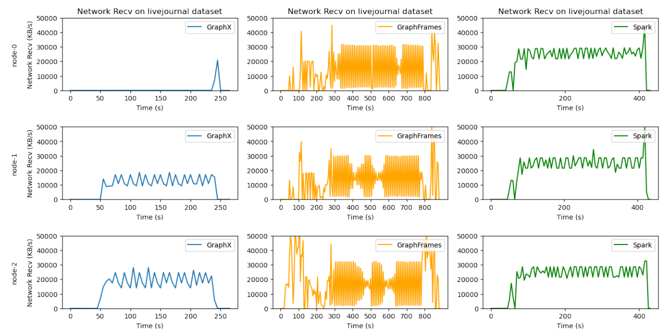

Fig. 7: PageRank Network Receive Throughput

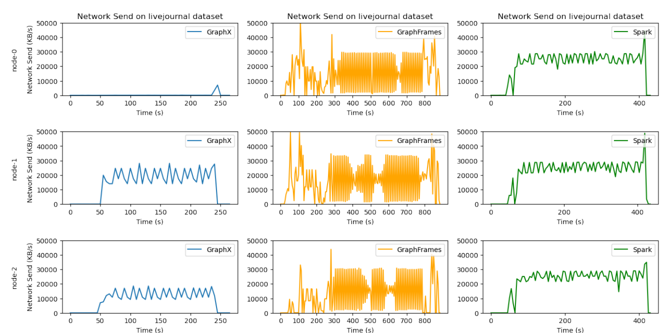

Fig. 8: PageRank Network Send Throughput

B. Connected Components

The second algorithm that we analyzed was connected components. GraphX, GraphFrames and GraphChi provide APIs to run connected components directly. For spark we implemented the large-small star algorithm [9] to collect the data.

Figure 9 shows the execution time for connected components for all the datasets from Table III. Clearly GraphChi is the best framework here in terms of execution time. Followed by GraphX, which is the best choice among multi-node frameworks. Spark's implementation of large-small star algorithm seems to performing the worst.

CPU Usage: The CPU usage for connected component is shown in figure 10. We have collected the CPU usage (and rest of the metrics) only for LiveJournal network dataset, as it

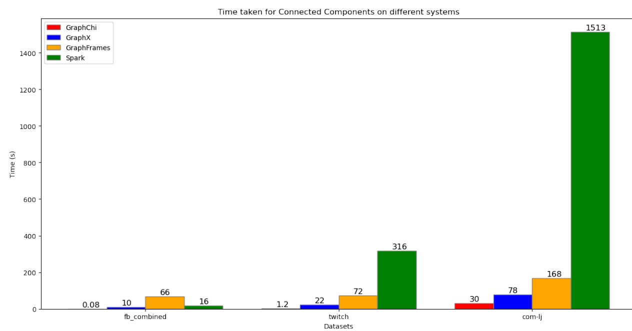

Fig. 9: Execution Time for Connected Component

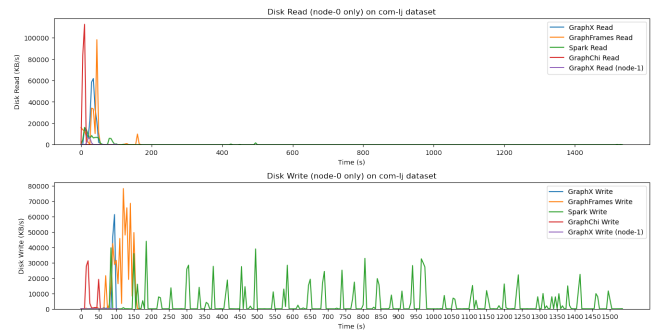

Fig. 11: Connected Components Disk Throughput

has the largest execution time. In here we see similar trends as seen in PageRank. First similar observation is that GraphChi is still bottlenecked at around 60% CPU Usage, even though it is performing the best. Second observation is that GraphX's node-0 is still showing low CPU usage, same as PageRank. We have also plotted GraphX's node-1 CPU usage which is showing good CPU usage. The behavior is similar as before, which we think is due to its partitioning scheme designed to minimize communication overhead. Last peculiar observation that we see is in Spark, we see that for roughly half of its execution time Spark is having very low CPU usage (this behavior is seen across all the nodes). This might be due to inefficiencies in large-small star algorithm implementation, and we think Spark is trying to read data during that time.

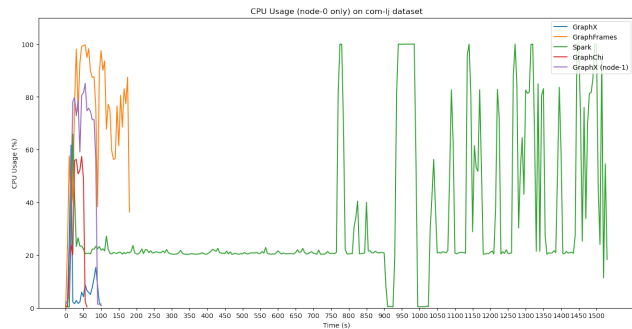

Fig. 10: Connected Components CPU Usage Node-0

Disk Throughput: The disk throughput plots for connected components is shown in figure 11. The trends here are similar as before, we see large disk read throughput in GraphChi, which is due to its sequential read behavior. We also see high disk writes in GraphFrames and Spark, which are probably corresponding to writing temporary files.

Memory Usage: Memory usage plots for all the frameworks are showing in figure 12. GraphFrames is showing the highest memory usage, same as before. Also GraphChi is showing the smallest memory usage, which is in accordance of its system design, which tries to minimize memory usage. GraphX node-1 is showing a high memory usage. Spark is not showing a high memory usage, which could be reason for its low performance as it might be trying to read/write data from disk during its execution.

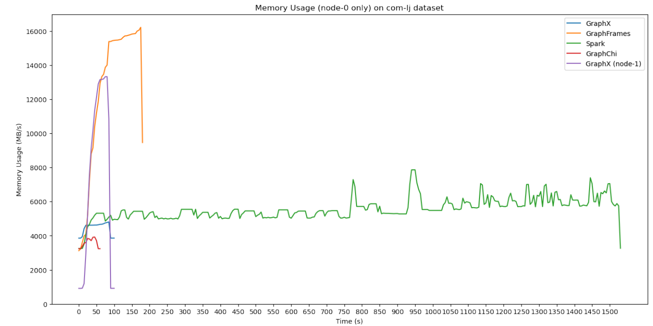

Fig. 12: Connected Components Memory Usage

Network Usage: In network usage, as shown in figure 13, we are seeing a clear trend that GraphFrames is having very high network usage. Which could be indicative of its inefficient partitioning scheme as it is requiring too much data movement across the nodes. This may reflect a broader issue in GraphFrames which could be reason for its bad performance in general. In contrast to that GraphX is having a very low network activity, which is an indication of its optimal partitioning scheme. This could be a major contributor to GraphX's better performance.

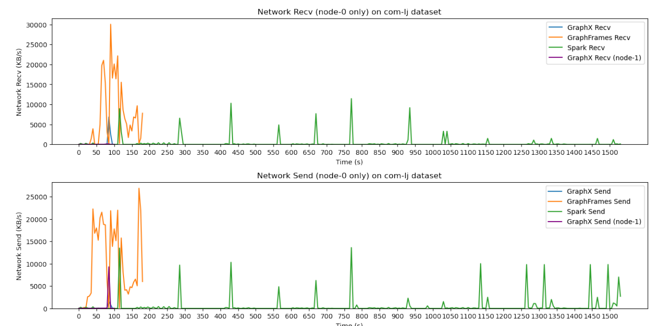

Fig. 13: Connected Components Network Usage

C. Triangle Counting

The third algorithm that we analyzed was triangle counting. GraphX, GraphFrames and GraphChi provide APIs to run triangle counting directly. For spark we used a custom implementation [10].

Figure 14 shows the execution time for triangle counting for all the datasets from Table III. Clearly GraphChi is the best framework here in terms of execution time. Followed by GraphX, which is the best choice among multi-node frameworks, the gap between GraphChi and GraphX is increasing. Spark and GraphFrames both failed with out of memory exception for medium and large graphs.

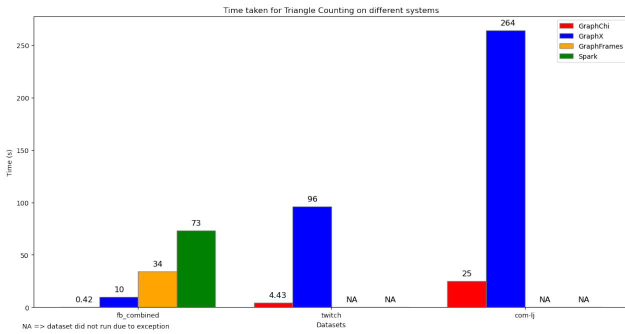

Fig. 14: Execution Time for Triangle Counting

CPU Usage: The CPU usage for triangle counting is shown in figure 15. We have collected the CPU usage (and rest of the metrics) only for fb-combined network (smallest graph in Table III) dataset as Spark and GraphFrames both failed with out of memory exception for medium and large graphs. In here we see that all the systems show similar CPU usage. This could be because the graph is small and every system has some initialization leading to high CPU usage during the start.

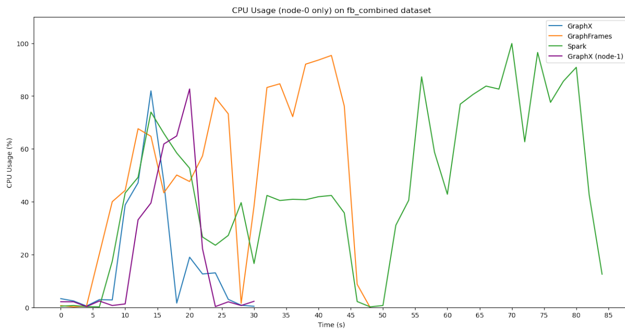

Fig. 15: Triangle Counting CPU Usage Node-0

Disk Throughput: The disk throughput plots for triangle counting are shown in figure 16. The trends are showing expected disk read behavior for all the systems. We see large disk read throughput initially to load the graph. Disk write does not show anything out of ordinary.

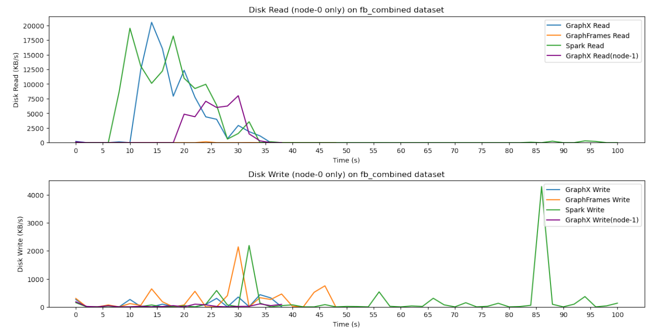

Fig. 16: Triangle Counting Disk Throughput

Memory Usage: Memory usage plots for all the frameworks are showing in figure 17. GraphFrames is showing the highest memory usage followed by Spark and GraphX. GraphX node-1 is showing the lowest memory usage. Spark is once again not showing a high memory usage, which could be reason for its low performance as it might be trying to read/write data from disk during its execution.

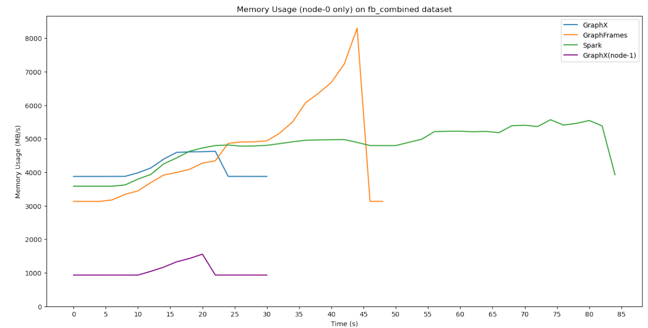

Fig. 17: Triangle Counting Memory Usage

Network Usage: In network usage, as shown in figure 18, we are seeing a clear trend that GraphFrames is having very high network usage. This as, we pointed out earlier, could be indicative of its inefficient partitioning scheme. Due to which we are seeing too much data movement across nodes. This could bottleneck GraphFrames' performance. Comparing that with GraphX, we see that it is showing very low network activity and this is due to its optimal partitioning scheme.

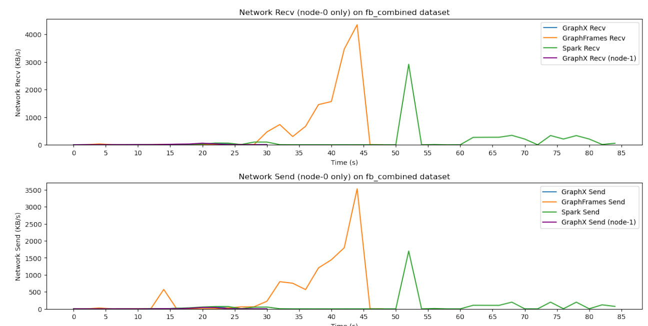

Fig. 18: Triangle Counting Network Usage

VI. Conclusion

The main objective that we wanted to answer from this study was to understand which framework to use and in what settings. To this end, we have identified that GraphChi is the best framework, as it has consistently provided the best timings for all the datasets we tested. But we do see a trend that GraphX will be a better choice for graphs larger than 100 million edges. As the graph sizes increases, GraphChi is limited by its disk read/writes which makes GraphX perform better. Also, GraphX can be easily scaled horizontally, unlike GraphChi which is a single machine Framework.

Other key observations from this study are, first we have observed that GraphChi is generally bottlenecked at 60% CPU usage even for large graphs. This can be due to its system design which tries to minimize memory usage and thus requires frequent disk read/writes. Second, we have observed that GraphX's vertex-cut partitioning scheme, which is designed to minimize communication overhead, is helping it perform better. It ensures minimal data movement across nodes and thus saves time. Lastly, we observe that GraphFrames has both high network activity and memory usage. This could be indicative of its suboptimal data partitioning scheme which requires too much data movement across nodes. This is a contributing factor to bad performance of GraphFrames. Overall, having a good partitioning scheme and communication optimization will be beneficial for performance improvements in multi-node systems.

VII. Future Work

In this study, for multi-node systems, we have run all the experiments on a three VM setup. A good future work would be to analyze the impact of increasing the number of nodes in the multi-node setup and see how it impacts the performance. Another potential direction can be to analyze the impact of random node failure in multi-node systems while experiments are running. And the last direction could be to drop memory and buffer caches during experiments to see which frameworks are more cache sensitive.

References

[1] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI) (OSDI'12). USENIX Association, Berkeley, CA, USA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

[2] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. NSDI 2012. April 2012

[3] GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. Reynold S. Xin, Daniel Crankshaw, Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica. OSDI 2014. October 2014

[4] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. 2016. GraphFrames: an integrated API for mixing graph and relational queries. In Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES '16). Association for Computing Machinery, New York, NY, USA, Article 2, 1–8. https://doi.org/10.1145/2960414.2960416

[5] Spark SQL: Relational Data Processing in Spark. Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia. SIGMOD 2015. June 2015

[6] Page, Lawrence and Brin, Sergey and Motwani, Rajeev and Winograd, Terry (1999) The PageRank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford InfoLab

[7] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The design and operation of cloudlab. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19). USENIX Association, USA, 1–14

[8] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. ACM Trans. Intell. Syst. Technol. 8, 1, Article 1 (January 2017), 20 pages. https://doi.org/10.1145/2898361

[9] Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. 2014. Connected Components in MapReduce and Beyond. In Proceedings of the ACM Symposium on Cloud Computing (SOCC '14). Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670997

[10] Ostap Orishko. Triangle Counting implementation in spark using RDD. https://github.com/orishko-py/pyspark-triangle-count

[11] 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 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 135–146, 2010.

[12] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. arXiv preprint arXiv:1204.6078, 2012.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pages 17–30, 2012.

[14] D. Zheng, X. Song, C. Ma, Z. Tan, Z. Ye, J. Dong, H. Xiong, Z. Zhang, and G. Karypis. Dgl-ke: Train- ing knowledge graph embeddings at scale. In Pro- ceedings of the 43rd International ACM SIGIR Conference on Research and Development in Informa- tion Retrieval, pages 739–748, 2020.