Techniques for Graph Analytics on Big Data

M. Usman Nisar, Arash Fard and John A. Miller
Department of Computer Science
University of Georgia,
Athens, GA, USA
Email: {nisar, ar, jam}@cs.uga.edu

***Abstract*—Graphs enjoy profound importance because of their versatility and expressivity. They can be effectively used to represent social networks, web search engines and genome sequencing. The field of graph pattern matching has been of significant importance and has wide-spread applications. Conceptually, we want to find subgraphs that match a pattern in a given graph. Much work has been done in this field with solutions like Subgraph Isomorphism and Regular Expression matching. With Big Data, scientists are frequently running into massive graphs that have amplified the challenge that this area poses. We study the speedup and communication behavior of three distributed algorithms for inexact graph pattern matching. We also study the impact of different graph partitionings on runtime and network I/O. Our extensive results show that the algorithms exhibit excellent scalable behavior and min-cut partitioning can lead to improved performance under some circumstances, and can drastically reduce the network traffic as well.**

***Keywords*-graph analytics; big data; graph simulation; parallel and distributed algorithms.**

I. Introduction

Graphs are of utmost importance in Computer Science because of their expressivity and the ability to abstract a huge class of problems. They have been successfully used to study and model numerous problems in different fields. These applications vary from software plagiarism detection, web search engines, study of molecular bonds to the modeling of social networks like Facebook, LinkedIn and Twitter [1].

Graph pattern matching is one of the most important and widely studied class of problems in graphs. A considerable amount of research has been put into this area, sprouting concepts like Subgraph Isomorphism, Regular Expression matching [2] and Graph Simulation [3]. Conceptually, pattern matching algorithms seek to find subgraphs of a given graph that are similar to the given query graph [4]. Though, Subgraph Isomorphism returns the strictest matches for graph matching in terms of topology [5], the problem is NP-complete [6], and thus does not scale well.

Graph simulation, on the other hand, provides a practical alternative to subgraph isomorphism by relaxing the stringent matching conditions of subgraph isomorphism, and allowing matches to be found in polynomial time. Some researchers [7], [8] even argue that graph simulation is more appropriate than subgraph isomorphism for modern problems like social network analysis because it yields matches that are conceptually more meaningful. With the rapid advent of Big Data, graphs have transformed into huge sizes and are rapidly getting out of the grasp of conventional computational approaches. In this paper, we address the problem of graph pattern matching on such big graphs.

The outline of the paper is as follows: We discuss the background and a description of the subgraph pattern matching problem along with its types in next section. We follow-up with a brief description of three new distributed algorithms that we proposed in [4]. Then we go through the implementation of distributed algorithms on two platforms and briefly compare their pros and cons. We also give runtime and speed-up graphs of the three distributed algorithms to study their behavior. In section 6, we study the impact of graph partitioning along two lines - runtime improvement and network I/O reduction.

II. Background

In this section, we discuss the background and give motivation for the need of new approaches to deal with large scale graphs in general and query processing in particular. Today, Facebook has over 1 billion vertices and the average degree of each vertex is 140[1], Twitter has well over 200 million active users creating over 400 million tweets each day[2]. In genome sequencing, recent work [9] attempts to solve the genome assembly problem by traversing the de Brujin graph of the read sequence. The de Brujin graph can contain as many as 4^k vertices where k is atleast 20.

To handle the mammoth scale of these graphs, an obvious approach is to distribute the graphs onto multiple machines and then run them concurrently to efficiently calculate the result in parallel. Consequently some of the basic challenges are the following, (1) how to distribute the graph? (2) how to come up with an *efficient* algorithm that runs as *concurrently* as possible? (3) how to reduce the communication/traffic among different machines (in part, a side effect of the top two)? We go through these questions as we discuss the problem of Subgraph Pattern Matching in depth.

[1]http://www.facebook.com/press/info.php?statistics
[2]http://blog.twitter.com

A. Subgraph Pattern Matching

The problem of subgraph matching is defined as follows: Let *G=(V, E, l)* be a graph, where *V* is the set of vertices, *E* is the set of edges, and *l* is the labelling function that assigns a label to each vertex in *V*. Let $Q=(V_q, E_q, l_q)$ be the query (pattern) graph where V_q is the set of vertices, E_q is the set of edges and l_q is the labeling function on V_q. Intuitively, the goal of subgraph pattern matching is to find all subgraphs from the data graph *G* that match the pattern graph *Q*. Thus, $G'(V', E', l')$ is a subgraph of G if and only if (1) $V' \subseteq V$; (2) $E' \subseteq E$; and (3) $\forall u \in V' : l'(u) = l(u)$.

In this paper, without loss of generality we assume all vertices are labeled, all edges are directed, and there are no multiple edges. We also assume a query graph is a connected graph because the result of pattern matching for a disconnected query graph is equal to the union of the results for its connected components. We use the terms pattern and query graph interchangeably.

B. Types of Pattern Matching

In this section, we briefly review four different types of pattern matching. The first one is Subgraph Isomorphism, the next one is Graph Simulation proposed in [3], and the last two are Dual and Strong Simulation proposed in [10]. For a detailed survey of algorithms, see [11].

1) Subgraph Isomorphism: Arguably, subgraph isomorphism is the most widely studied problem for graph pattern matching. By definition, subgraph isomorphism describes a bijective mapping between a query graph $Q(V_q, E_q, l_q)$ and a subgraph of a data graph $G(V, E, l)$, denoted by $Q \trianglelefteq_{iso} G$. That is, assuming $G'(V', E')$ is a subgraph of G, graph Q will be subgraph isomorphic to G if there is a bijective function f from the vertices of Q to the vertices of G' such that (u, v) is an edge in Q if and only if $(f(u), f(v))$ is an edge in G' [12]. It should be noted that function f ensures that u and $f(u)$ have the same labels. Ullmann's algorithm and VF2 algorithm [13] are widely known to solve the problem of subgraph isomorphism [11]. However, this problem is NP-hard in general so exact algorithms are not practical for very large graphs.

2) Graph Simulation: Graph simulation allows a faster alternative to subgraph isomorphism by relaxing some restrictions. Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, l)$ via *graph simulation*, denoted by $Q \trianglelefteq_{sim} G$, if there is a binary relation $R \subseteq V_q \times V$ such that (1) for every $u \in V_q$ there is a $u' \in V$ such that $(u, u') \in R$; (2) $l_q(u)$ equals $l(u')$; (3) for every $v \in child(u)$, there is a $(v, v') \in R$ such that $v' \in child(u')$ (adapted from [10]). Here, $child$ returns all the direct children of a given vertex.

Intuitively, graph simulation only captures the child relationships of vertices. HHK - a quadratic algorithm, was first proposed in [3] and efficiently computes the match set on medium-sized graphs, but it is still not efficient enough for massive graphs.

3) Dual Simulation: Dual simulation extends graph simulation by also taking into account the parent relationships of the vertices, thus resulting in a stricter match set. Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, l)$ via *dual simulation*, denoted by $Q \trianglelefteq^{D}_{sim} G$, if (1) Q is a graph simulation match to G with a match relation $R_D \subseteq V_q \times V$, and (2) for every $w \in parent(u)$, there is a $(w, w') \in R$ such that $w' \in parent(w')$ (adapted from [10]). The method $parent$ returns all the direct parents of a vertex.

4) Strong Simulation: Strong simulation builds on dual simulation by introducing a locality condition. The term *ball* is coined [10] to capture this aspect of the match. The ball for a vertex *v* with radius *r* contains all the vertices V_B that are within an undirected distance of *r* from the vertex *v*, moreover it has all the edges between those vertices V_B and no more. Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, l)$ via *strong simulation*, denoted by $Q \trianglelefteq^{S}_{sim} G$, if there exists a vertex $v \in V$ such that (1) $Q \trianglelefteq^{D}_{sim} G[v, d_Q]$ with maximum dual match set R^b_D in ball *b* where d_Q is the diameter of Q, and (2) v is member of at least one of the pairs in R^b_D. The connected part of the result match graph of each ball with respect to its R^b_D which contains v is called a *maximum perfect subgraph* of G with respect to Q [4].

C. Models of Computation

With the advent of Big Data computing, computational models for graph algorithms have been re-examined. Over the years, a number of ideas have been proposed for efficient and scalable processing of graphs. Since this paper is focussed on big graphs, we only go through some of the most important distributed models.

1) MPI-like: Several libraries are developed using MPI (Message Passing Interface) during the last decade to provide platforms for distributed graph processing like Parallel BGL [14] and CGMgraph [15]. However, these libraries do not support fault tolerance or some other issues that are important for very large scale distributed systems.

2) MapReduce: Google introduced a system based on MapReduce model for the processing large data sets [16]. It provides fault tolerance and ease of programming; hence, some scientists have implemented their parallel graph processing package over the MapReduce platform like Pegasus [17] that is developed for some graph mining algorithms. However, this model is inherently unsuitable for an iterative algorithm which is the case for most of graph algorithms.

3) Vertex-Centric BSP: Bulk Synchronous Parallel (BSP) is a model proposed by Valiant [18] as a computation model for parallel processing. As it is illustrated in figure 1, computation in this model is a series of supersteps. Each superstep contains three ordered stages: (1) Concurrent computation where different processes run concurrently; (2) Communication where all processes exchange their messages; (3) Barrier Synchronization in which every process waits for others to reach to the same state before going to

the next superstep. In the vertex-centric programming model, each vertex of the data graph is a computing unit which can be conceptually mapped to a process in the BSP model.

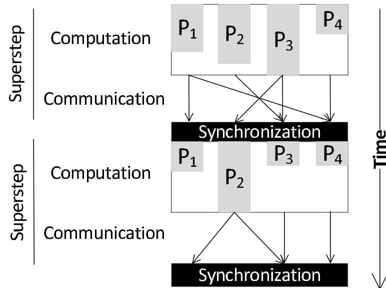


Figure 1: An example for BSP model

Google presented Pregel [19] based on Vertex-centric BSP model for processing very large graphs. A few other open source projects are also introduced that have a similar idea, namely GPS (Graph Processing System) [20], Apache Giraph[3], Apache Hama[4], and Signal/Collect in its synchronous mode [21].

4) Vertex-Centric Asynchronous: Although implementing a system based on BSP model makes the system inherently free of any deadlock, but its synchronization stage can introduce a bottleneck because in each superstep all the workers should wait for the slowest one. Therefore, there are a few other systems that follow the idea of vertex-centric asynchronous model; i.e., they enjoy benefits of a vertex-centric model, but avoid bottlenecks of BSP model. Examples are GraphLab [22] and Signal/Collect in its asynchronous mode [21].

D. Graph Partitioning

Graph partitioning has extensive applications in many areas including telephone network design, VLSI design and task scheduling. The problem is to partition the graph into p roughly equal parts, such that number of edges connecting vertices in different parts is minimized [23].

It is important in the context of distributed computing because we want to partition the graph into pieces such that each piece is mostly *self-contained*; i.e., we want to reduce its communication to other parts as much as possible, thus in theory resulting in speed-up. However, it is not a trivial problem. When we partition the graph, we need to take care of two points:

1) An effort must be made to partition the graph into equal parts, so every worker gets a fair amount of load.
2) We want to minimize the number of edges going from one partition to the other.

[3]Apache Giraph: http://incubator.apache.org/giraph/
[4]Apache Hama: http://hama.apache.org/

We term the partitioning that enforces the two conditions listed above as min-cut partitioning. It is an NP-complete problem and has been extensively studied in its own right. It has been successfully used with considerable improvement in various applications. It must be noted that graph partitioning has no guarantees to provide consistent improvements for all graph algorithms. It will result in good speed-up, if the communication is mostly between adjacent vertices, since partitioning tries to co-locate them in a single partition; however, if that is not the case then communication is inevitable. Semih [20] was able to achieve 2.2x improvement in speed by adding min-cut partitioning to the PageRank algorithm, but in the case of Highly-Connected Component, Single Source Shortest Path the speed-up was only 1.47 and 1.08, respectively.

III. Distributed Algorithms for Graph, Dual and Strong Simulation

In this section, we give an outline of distributed algorithms for graph, dual and strong simulation that are designed for a vertex-centric system. Our implementation of strong simulation is an optimized version of the original strong simulation.

Graph Simulation: In the designed distributed algorithm for graph simulation, the query graph is distributed among all vertices of the data graph, and then each vertex should find out its match set among the vertices of the query graph. Vertex u' of the data graph matches to vertex u of the query graph if the two vertices have the same label, and each child of u has at least one match among the children of u'.

In a vertex-centric system, a vertex initially knows only about its own label and the id of its children. Therefore, each vertex needs to communicate with its neighbors to learn about their labels and status in order to evaluate the child-relationship condition. A Boolean flag, called match flag, is dedicated to each vertex which indicates if the vertex has a potential match among the vertices of the query graph. This flag is initially false.

In an example displayed in figure 2, all the vertices of the data graph labeled a, b, and c make their flag true at the first superstep, and then vertices 1, 2, and 5 send messages to their children. At the second superstep only vertices 5, 6, and 7 will reply back to their parents. At the third superstep, vertices 1, 5, 6, 7, and 8 can successfully validate their match set, but vertex 2 makes its flag false, because it receives no message from any child. Therefore, vertex 2 sends a removal message to vertex 1. This message will be received by vertex 1 at superstep four. It will successfully reevaluate its match set, and the algorithm will finish at superstep five when every vertex has voted to halt (there is no further communication).

Dual Simulation: The algorithm for dual Simulation is very similar to graph Simulation. In addition to child-relationship, we extend the algorithm to check parent-relationship as well.

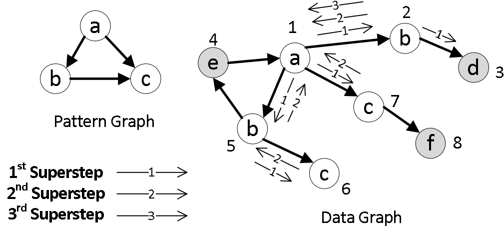


Figure 2: An example for distributed graph simulation

Strong Simulation: A version of strong Simulation is done in two phases: (1) we run dual simulation to find the match set R and then (2) for each vertex in R, we create a ball with a diameter of d_q. Once we have all the balls ready, we run dual simulation on each to compute the final output of strong simulation.

The biggest challenge we faced in strong simulation was the creation of balls. Because of the scale of graphs, we may end up creating balls for many of vertices simultaneously which could bog down the whole system. Therefore, we tried two different approaches that we go through below.

1) Depth-First Ball: As the name suggests, the ball creation process works in a depth-first fashion. In the first superstep, each vertex left after dual simulation performs as the center of a ball and sends messages to its adjacent vertices with the specified ball size. The receiving vertices reply back with their labels and forward the message to their adjacent vertices with a decremented ball size. This process is repeated and these messages are removed from the system when the ball size reaches zero. Since this approach is very slow and does not scale well, we have omitted its details, but an explanation can be found at [24].

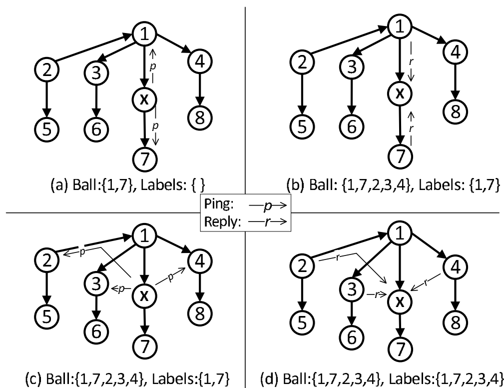


Figure 3: A breadth-first ball around vertex X with $d_q = 2$

2) Breadth-First Ball: This approach works on a simple ping-reply model. In figure 3, let us suppose we want to create a ball around vertex *X* of radius 2. Initially, X only knows its adjacent node ids 1,7 and no label information. It starts off by sending a ping message to all of its adjacent nodes. In the second superstep, all the recipient nodes reply back with their labels and the ids of their children and parents. Thus, in 3(b), X upon receiving these messages stores the gathered information about the vertices in its ball; e.g., labels and ids of its descendants and ascendants. In 3(c), it sends another ping message to all of its boundary vertices (vertices at a distance 2 of the center) which reply back with their labels and the ids of their children and parents, subsequently saved by the node X in 3(d).

The drawback of this approach is that it results in almost twice the number of supersteps, yet it is much more efficient and performs much better than the other approach, therefore we adopted this approach for our strong simulation tests.

IV. Implementation of Distributed Algorithms

In this section, we implement graph simulation on two different distributed computing infrastructures and compare their pros and cons.

A. GPS - Graph Processing System

As mentioned above, because all of the algorithms were designed with a BSP and vertex-centric model, we decided to use something akin to Pregel for the implementation. Of the numerous options available, we picked GPS for our algorithms since it offers everything that we desire of Pregel and is available as open-source. It is written in Java and also gives us an option to write a *master.compute()* method that proved to be quite handy in case of strong simulation.

As in Pregel, there are two types of main components in the system: one master node and *k* worker nodes. A GPS job starts off by partitioning the data graph over all the participating workers. Every worker reads its partition and then distributes the vertices based on a round-robin scheme, i.e., vertex *v* gets assigned to worker *W = v.id % k*. The lifecycle of a GPS job can be summarized in following steps: (a) parse the input graph files, (b) start a new superstep, and (c) terminate computation when all the vertices have voted to halt and there are no messages in transit, otherwise go to (b).

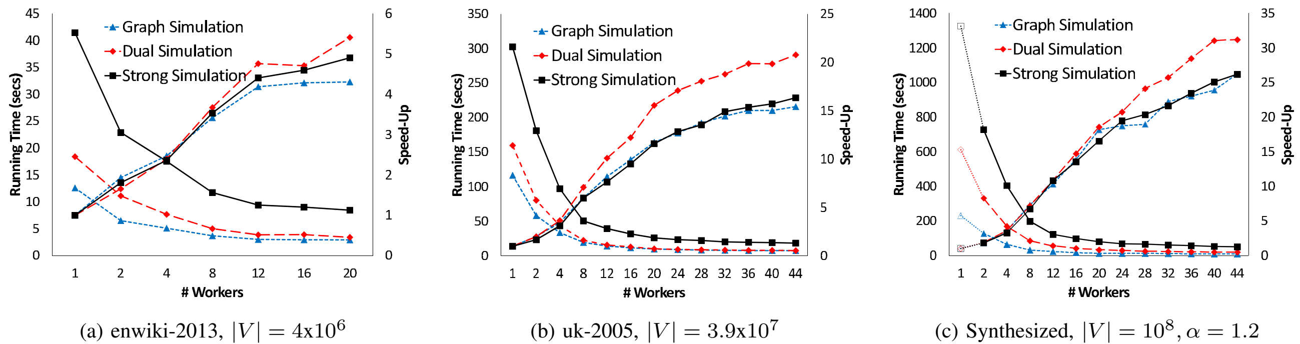
B. Akka

Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event driven applications on the JVM. It has an extended API that lets you manage service failures, load management (back-off strategies, timeouts and processing-isolation). It also scales well with increases in the number of cores and/or number of machines[5]. The API is available both in Java as well as Scala.

C. GPS vs Akka

GPS delivers on its promise of Pregel. However, to enable the message passing for the custom types requires substantial effort that is not only time-consuming, but is also prone to errors. For example, if the message contains some complex

[5]http://akka.io/



(a) enwiki-2013, $|V| = 4\text{x}10^6$ (b) uk-2005, $|V| = 3.9\text{x}10^7$ (c) Synthesized, $|V| = 10^8, \alpha = 1.2$

Figure 4: Running times and speed-up for distributed algorithms, $|V_q| = 25$, $\alpha_q = 1.2$

types, we need to be very careful with how the message is serialized at the sending vertex and then deserialized at the receiving end. Implementation detail of serializer and deserializer can be cumbersome, hard to maintain and not so readable.

Akka, unlike GPS is a general purpose toolkit for building highly concurrent and distributed applications and not something that is built ground-up only for graphs. It means there is some work that needs to be done to make Akka work in a fashion similar to BSP. Perhaps, the biggest edge that Akka has over other comparable models is its inherent ability and support for sending messages between actors. With Akka, the developers do not have to worry about serializing/deserializing of data. They can send messages wrapping complex types with extremely concise and terse syntax.

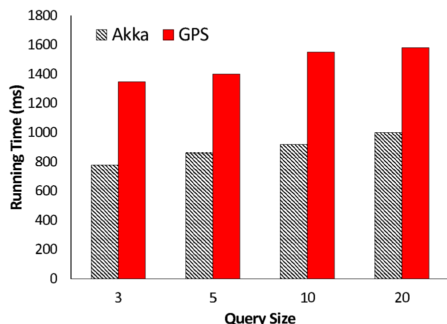


Figure 5: GPS and Akka on Graph Simulation

We implemented a prototype application for Graph Simulation using Akka. We were surprised by the power and ease provided by Akka for rapid development. Even with the most basic implementation, we were able to achieve much better times as we were getting from a system like GPS. The tests were conducted on the amazon-2008 [25] dataset and were run over a cluster of 5 machines. As can be seen in Figure 5, Akka ran almost twice as fast as GPS.

V. Evaluation of Distributed Algorithms

Just like any other distributed algorithm, speed-up and efficiency are of significant importance for the scalability of algorithms discussed above. Here, we try to examine that behavior by varying the number of workers and how the algorithms react.

A. Experimental Setup

We have used both synthesized and real-world graphs in our experiments. Having the number of vertices, the number of edges in a synthesized data graph is determined by α such that $|E| = |V|^\alpha$. To study the consistency of results, we ran the experiments for three different datasets - one synthesized ($|V| = 10^8, \alpha = 1.2$) and two real-life, uk-2005 and enwiki-2013 [25] (uk-2005 has 39459925 vertices and 936364282 edges whereas enwiki-2013 is relatively smaller but more dense, with 4206785 vertices and 101355853 edges). For query graphs, one parameter used is $|V_q|$ indicating the number of vertices and α_q, which if not mentioned otherwise is kept constant at 1.2. To retrieve a query graph for a particular data set, we took $|V_q|$ as the input which is the size of query graph. Then we randomly extract a connected subgraph from the dataset that adheres to the α_q constraint and has $|V_q|$ number of vertices.

The experiments were conducted using GPS on a cluster of 12 machines. Each one has a 128GB DDR3 RAM, two 2GHz Intel Xeon E5-2620 CPUs, each with 6 cores. The ethernet connection is 1Gb. In case of more than 11 workers, we assigned multiple workers to the same worker node in a round-robin fashion.

B. Experimental Results

We give the graphs for running times and speed-up in figure 4. It is clear that all three algorithms exhibit scalable behavior as we increase the number of workers. The speed-up with smaller datasets is less because as we increase the number of workers, each worker gets a small load - meaning less computational load and more time wasted in synchronization costs inherent in BSP. Dual simulation

always shows the best speed-up and graph simulation shows the least speed-up. The reason is because of the duality condition, each worker has to do almost twice the work for every vertex as compared to graph simulation, this increased computational load scales well as we increase the number of workers and consequently, dual simulation tends to show better speed-up.

Strong simulation keeps bouncing between the other two depending upon the distribution of the dual match - if they are well balanced across all workers, then balls are created in parallel and we get better speed-up and vice-versa. This is because ball-creation is a time-consuming process and slight imbalances can overload some workers, thus negatively affecting the overall speed-up. It must be noted that for the synthesized dataset, we could not run the tests on a single worker due to the size of dataset. Using multiple datasets and queries, we calculated an average speedup of 1.82 from one to two workers, thus in the graph we have extrapolated the values marked by dotted line in figure 4c.

VI. Impact of Graph Partitioning

We used METIS [23] for graph partitioning, which can partition an unstructured graph into k parts using either multilevel recursive bisectioning [26] or multilevel k-way [27] schemes. Both the models can provide high-quality yet different partitions so we tried both to study the relative impact. The algorithms work with a simple goal: *edge-cut* which basically tries to minimize the edges that travel between different *well-balanced* partitions. After extensive experiments, k-way partitioning performed better so we only report its results below.

A. Experimental Setup

We conducted extensive testing of min-cut partitioning on multiple datasets. We use both a real world dataset and a synthesized dataset. uk-2002 [25] was used as the real life dataset, that is a 2002 crawl of the .uk domain. We also synthesized a random-edge graph with $|V| = 10^7$ and an α of 1.2. Because of space constraints and similarity of results, we have omitted the results of synthesized [24] from the paper, however, we do make a mention with explanation where we deem necessary.

B. Experimental Results

We identify two complexity measures for our tests, (1) *runtime* which is the time taken to complete a given job and (2) *network traffic* which is the number of bytes sent among workers to complete a given job.

1) Runtime: To study the impact of min-cut partitioning on the runtime of algorithms, we compare its times against the default partitioning *round-robin*. We compare the times for both by running them against queries of different sizes. In figure 6. we see that the runtime for graph and dual simulation are always faster with min-cut. This is because both always communicate with their adjacent vertices only and with min-cut partitioning, there is a higher probability the adjacent vertices will be on the same worker.

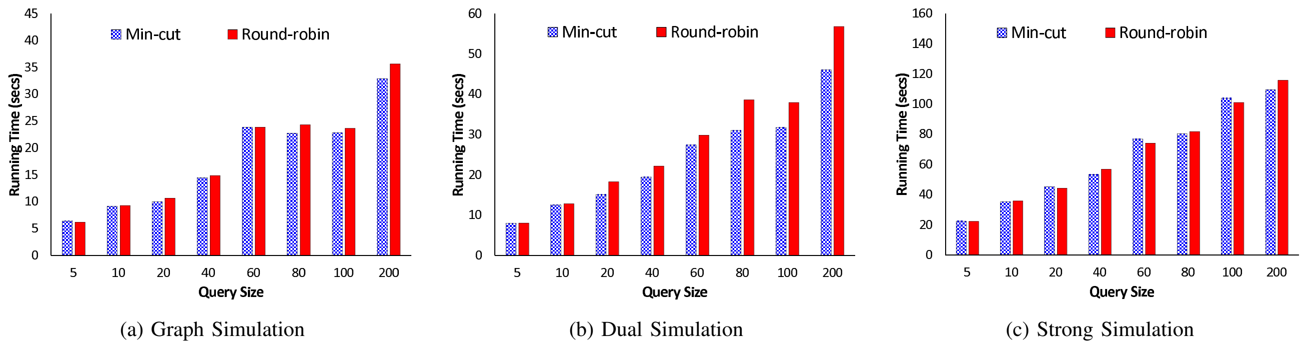
However, strong simulation is different because when it is in the process of building the ball, it needs to communicate with vertices that are further away from the center of the ball. This increases the probability that the vertex will communicate with a vertex that lies on some other partition, thus washing away any benefit that we obtained from min-cut partitioning. That is why, strong simulation appears mostly unaffected by either type of partitioning. The results for the synthesized dataset were very similar to this real life dataset.

2) Network traffic: Another important criteria is the data that needs to be moved between workers. By reducing the total network traffic, we increase our chances of reducing the runtime and any cost associated with them. The effect of reduced data traffic could be more important when there is a network-bandwidth limit in the system like geographically distributed systems. In figure 7, we observe that the network traffic for all always drop with min-cut partitioning - that is due to the primary goal of min-cut partitioning which is to reduce the inter-partition communication. Information provided in [25] shows the power-law nature of the real-world datasets that we use in our experiments. This characteristic of the datasets is a contributive factor to the drastic drop in the network traffic [28]. Usually, these graphs have a few vertices that have a much higher connectivity to other vertices, with min-cut partitioning we move these vertices along with their connections to the same worker, thus resulting in a drastic drop in the total traffic. A similar pattern is not present in the synthesized dataset, since connectivity is uniform.

VII. Related Work

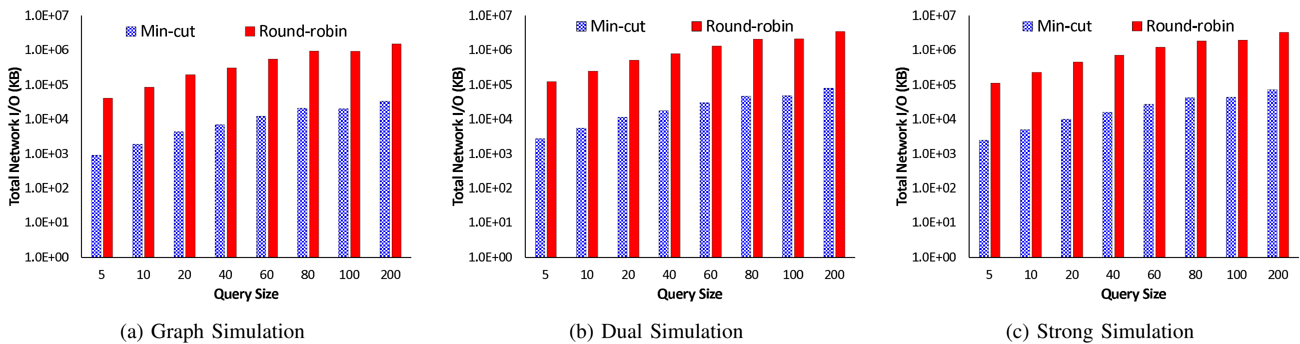
The problem of subgraph pattern matching is a widely studied topic. Over the years, many different graph pattern matching techniques have been proposed. There are two broad categories of matching, *exact* and *inexact* algorithms. Subgraph Isomorphism is one of the most well-renowned yet NP-complete problem which matches the graph based on the exact topological structure of a pattern. On the other hand, a few alternative techniques like p-homomorphism [29] and bounded simulation [30], that rely on inexact but semantic matching [5], have been proposed recently.

Although the first practical distributed approach for graph simulation was introduced in [31], their approach is not based on a vertex-centric model and uses a modified version of the HHK [3] algorithm. Also, not all of its stages are run in parallel and one is strictly serial.The idea of dual and strong simulation were first introduced in [10]. To best of our knowledge, no other distributed algorithms are implemented for these two models prior to our work.



(a) Graph Simulation (b) Dual Simulation (c) Strong Simulation

Figure 6: Partitioning effect on the runtime of dataset uk-2002-hc, $\alpha_q = 1.2$



(a) Graph Simulation (b) Dual Simulation (c) Strong Simulation

Figure 7: Partitioning effect on the network I/O of uk-2002-hc, $\alpha_q = 1.2$

Graph partitioning is another important problem that is gaining traction with the advent of distributed algorithms. These algorithms mostly try to work with the goal of minimizing the edge-cut in the graph. METIS [23] is an effective tool that can employ two techniques to generate high-quality partitions. The effects of graph partitioning on different algorithms have been studied by numerous researchers. [20] talks about the impact of partitioning on algorithms like PageRank, SSSP, etc. and report an impressive speed-up.

VIII. Conclusion

Graph pattern matching has been an important topic in the field of Computer Science and has been gaining prominence recently. It has become more challenging with the rapidly increasing size of graphs. In this paper, we study the three polynomial time pattern matching techniques in detail. This paper draws the following conclusions with regards to the three distributed algorithms we developed for inexact graph pattern matching: (1) they exhibit scalable behavior as we increase the number of workers, (2) min-cut graph partitioning improves the runtime of graph and dual simulation consistently especially with bigger queries, (3) min-cut graph partitioning always reduces the network I/O among workers and (4) we present two different techniques that can be used to build balls around a vertex in a vertex-centric setting.

In the future, we intend to look into different ways to improve the strong simulation running times on GPS and Akka. It is clear that we need to find out how we can improve the process of ball-creation. Instead of creating the ball on every dual-matched vertex, we intend to come up with techniques to reduce the total number of balls created without compromising the results. We also want to come up with new algorithms that do not suffer the synchronization bottlenecks of the BSP model, thus achieving better parallelism.

Acknowledgments

The authors would like to thank Dr. George Karypis and Semih Salihoglu for their generous help and assistance with METIS and GPS, respectively.

References

[1] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, “Measurement and analysis of online social networks,” in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 29–42.

[2] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, “Efficient subgraph matching on billion node graphs,” *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.

[3] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*. IEEE, 1995, pp. 453–462.

[4] A. Fard, U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz, "Distributed algorithms for graph pattern matching," http://www.cs.uga.edu/˜ar/abstract.pdf, Tech. Rep., 2013.

[5] B. Gallagher, "Matching structure and semantics: A survey on graph-based pattern matching," *AAAI FS*, vol. 6, pp. 45–53, 2006.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[7] J. Brynielsson, J. Hogberg, L. Kaati, C. Mårtenson, and P. Svenson, "Detecting social positions using simulation," in *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*. IEEE, 2010, pp. 48–55.

[8] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. A. Miller, "Towards efficient query processing on massive time-evolving graphs," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, oct. 2012, pp. 567 –574.

[9] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[10] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 310–321, 2011.

[11] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International journal of pattern recognition and artificial intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[12] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 10, pp. 1367–1372, 2004.

[14] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[15] A. Chan, F. Dehne, and R. Taylor, "Cgmgraph/cgmlib: Implementing and testing CGM graph algorithms on pc clusters and shared memory machines," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 81–97, 2005.

[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[17] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.

[18] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[19] 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*. ACM, 2010, pp. 135–146.

[20] S. Salihoglu and J. Widom, "GPS: A graph processing system," Tech. Rep., 2012.

[21] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: Graph algorithms for the (semantic) web," in *The Semantic Web–ISWC 2010*. Springer, 2010, pp. 764–780.

[22] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1006.4990*, 2010.

[23] G. Karypis and V. Kumar, "Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0," Tech. Rep., 1995.

[24] "Supplement for: A comparison of techniques for graph analytics on big data," http://www.cs.uga.edu/˜nisar/papersupplement.pdf.

[25] "Laboratory for Web Algorithmics," http://law.di.unimi.it/datasets.php.

[26] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[27] ——, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *SIAM Review*, vol. 41, no. 2, pp. 278–300, 1999.

[28] B. A. Huberman and L. A. Adamic, "Internet: growth dynamics of the world-wide web," *Nature*, vol. 401, no. 6749, pp. 131–131, 1999.

[29] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu, "Graph homomorphism revisited for graph matching," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1161–1172, 2010.

[30] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: from intractable to polynomial time," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 264–275, 2010.

[31] S. Ma, Y. Cao, J. Huai, and T. Wo, "Distributed graph pattern matching," in *Proceedings of the 21st international conference on World Wide Web*. ACM, 2012, pp. 949–958.