Sandpiper: Scaling Probabilistic Inferencing to Large Scale Graphical Models

Alexander Ulanov*[1], Manish Marwah*, Mijung Kim*, Roshan Dathathri†[2], Carlos Zubieta*[3], and Jun Li*[4]
*Hewlett Packard Labs
Palo Alto, California, 94304
Email: firstname.lastname@hpe.com
†University of Texas at Austin
Austin, Texas, 78712

***Abstract*—We design and implement a scalable version of loopy belief propagation (BP), a widely used algorithm for performing inference on probabilistic graphical models. However, implementations of BP on generic data processing platforms such as Apache Spark do not scale well to very large graphical models containing billions of vertices. To handle such large-scale graphs, we leverage a number of strategies. Our implementation is based on Apache Spark GraphX. We propose a novel graph partitioning strategy to reduce both computation and communication overhead providing a 2x speed-up. We use efficient memory management for storing the graph and shared memory for high-speed communication. To evaluate performance and demonstrate scalability of the approach, we perform a range of experiments including using real-world graphs with billions of vertices, where we achieve an overall 10x speed-up over a vanilla Spark baseline. Further, we apply our BP implementation to infer the probability of a website being malicious by performing inference on a graphical model derived from real, large-scale hyperlinked web-crawl data. We have open sourced our implementation.**

I. Introduction

Probabilistic graphical models (PGM) combine probabilistic reasoning with graph theory and are widely used for a variety of applications including fraud detection, computer vision, and recommender systems. An important step involved in the use of such models is *inference*, that is, making a probabilistic prediction of one or more variables given evidence on some other variables. The most common inference methods are variational [1] or sampling-based [2], which require iterative computations over the underlying graph until the estimations converge. Since such algorithms are usually communications bound and require random data access, their scalability with model size is poor, especially when the model does not fit in memory of a single machine.

There are several design challenges related to implementing graphical model inference that scales to large models. The first being the choice of the underlying data processing framework, and whether to use a specialized graph processing system.

In recent years, several specialized large-scale graph processing platforms have been developed, which could be used for graphical model inference. These include Apache Giraph [3], GraphLab [4], and others. Since these systems are tailored and optimized for iterative graph processing, they generally outperform generic data processing platforms such as Hadoop/Spark [5]. However, graph analytics is usually only one of several stages in an analytics workflow [6], and deploying a specialized graph processing framework results in large amounts of data movement and duplication. For this reason, we build on top of GraphX [6], which provides a tight integration with Apache Spark.

Other challenges relate to graph partitioning, representation and optimizations for large memory machines. Most implementations of graphical model inference use a factor graph representation, which although more generic, can double the memory requirements and also increase the convergence time. Further, it is important how the graphical model is partitioned. Spark/GraphX perform a random hash partitioning which result in both increased memory consumption and increased communications. Lastly, runtime performance can benefit from several optimizations for a large-memory machine.

In this paper, we design and implement a scalable version of a graphical model inference algorithm called belief propagation on Apache Spark/GraphX. While our work applies to other inference methods such as Gibbs sampling, here we focus on belief propagation. Our implementation has been open-sourced and available on Github as Project Sandpiper [7].

We evaluate the performance of our implementation on graphical models derived from synthetic and real data, including a real graph containing billions of vertices. Furthermore, we apply our method to a security use case, where we infer the probability of a web site being malicious based on a large-scale graphical model constructed from a web graph.

Specifically, we make the following main contributions:

- an open-source implementation of belief propagation that scales to very large graphical models;
- a new graph partitioning algorithm that reduces memory consumption by 2x, and improves BP run time by 2x over random partitioning (currently used by Spark GraphX);
- Combining all our optimizations, we get a 10x speed-up compared to a Vanilla Spark GraphX implementation on the same hardware;
- performance evaluation using synthetic and real graphs[1]
- a security use case involving a graphical model with billions of vertices and edges.

[1]Currently at Facebook
[2]Work done during internship at Hewlett Packard Labs
[3]Currently at Wizeline
[4]Currently at Ebay

[1]note that in this paper we use term graph in the context of a graphical model, that is, the underlying graph in a graphical model.

II. Background and Related Work

A. Graphical Model Inference

A probabilistic graphical model (PGM) [8] combines probabilistic dependency and reasoning with graph theory. PGMs provide a means to express and reason about dependencies without making the problem intractable (e.g., assuming every variable depends on all others), or making it too simplistic (e.g., assuming all variables are independent). The graph structure provides an elegant representation of the conditional independence relationships between variables.

The most common methods of doing inference on PGMs are (i) MCMC based methods [2], such as Gibbs sampling, and (ii) variational methods [1], such as loopy belief propagation. MCMC based techniques are known to be slow with high mixing time, therefore requiring an large number of samples. In this paper, we focus on a a variational method called loopy belief propagation [9], a commonly used, message-passing based algorithm for computing marginals of random variables in a graphical model, such as a Markov Random Field. It provides exact inference for trees, and approximate inference for graphs with cycles (in which case it is referred to as loopy belief propagation). Even though loopy belief propagation is an approximate algorithm with no convergence guarantees, it works well in practice for many applications [10]. Furthermore, compared to MCMC, loopy belief propagation performs well on large scale, sparse graphical models, such as those constructed from real life power law graphs including the website graph derived from hyperlinks considered in the security use case later in this paper.

B. Spark and GraphX

Apache Spark is a distributed data processing engine for clusters [5]. Spark operates similarly to Hadoop MapReduce, however Spark allows caching of intermediate results in memory for faster processing. Spark contains a number of built-in libraries for supporting additional workloads, including SQL, Machine Learning and Graph. The latter is implemented in GraphX project [6]. GraphX provides bulk synchronous parallel (BSP) scatter/gather API for message passing between the vertices in the graph. Spark's own MapReduce API is also supported. GraphX contains implementations of several graph algorithms, including triangle counting and PageRank. The BSP message passing API combined with map and reduce is useful for implementing various graph algorithms on a large scale because it allows embarrassingly parallel processing and alleviates the limitations of Pregel API. The other benefit of using GraphX is its tight integration with Spark. Graph processing in Spark is easily pipelined with the other data processing steps, such as extraction, transformation and loading of data and querying the result. Spark is being widely adopted for big data processing. The above mentioned features provide an advantage to GraphX compared to other graph processing engines and libraries such as Apache Giraph [3], Turi (ex-GraphLab) [4], and others. GraphX represents a graph using RDDs of vertices, edges and the routing table for replicated vertices. Some of the vertices are replicated because graph is randomly partitioned using vertex cut [6]. While we have developed our implementation of belief propagation on GraphX, it is easily portable to GraphFrames, or a similar graph processing system developed on top of Spark.

C. Graph partitioning

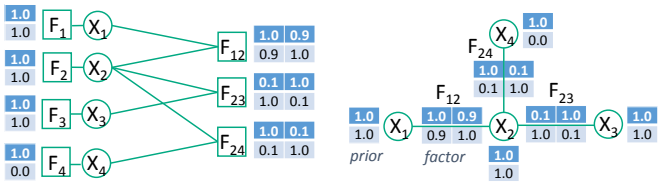
Graph partitioning has been studied widely in the literature [11]–[21] . They are broadly classified into edge-cut and vertex-cut policies. Vertex-cut policies can yield more balanced partitions than edge-cut policies, especially for power-law graphs. hMetis [12] performs vertex-cut by transforming the graph to an equivalent hypergraph (edges are converted into vertices, and vertices into hyper-edges) and performing an edge-cut on the hypergraph. PowerGraph [14] was the first to propose an efficient single pass vertex-cut partitioning policy. More recent vertex-cut policies, such as Least Incremental Cost (LIC) [17] and PowerLyra [15], have been shown to perform better than that. We propose a new vertex-cut partitioning policy motivated by them. All these policies stream edges one-by-one (one or two passes) and then assign an edge to a partition based on some heuristic. Once an edge is assigned to a partition, they maintain some runtime state that reflects the decision, which is then used to determine the assignment for the rest of the edges. The policies only differ in the heuristics used to determine the assignment of an edge to a partition. The heuristics used in our policy yields better partitions, as we shall see in Section IV-B.

D. Malicious website detection

We apply our scalable implementation of belief propagation to the problem of estimating the probability of a website being malicious. Users accessing malicious websites are exposed to several security threats such as phishing, malware, malicious links, and spam. Thus detecting malicious sites to prevent users from visiting them is an important problem which has been widely studied in the past. Broadly, there are two main approaches: 1) content-based, which uses website's host, domain, or web page information for detecting malicious sites. For example, [22] extracts lexical features from URLs and other features from host information (e.g. registration date, owner, etc.) to build a classifier to detect malicious sites; and, 2) link/graph based, which leverages the link structure of the web graph to detect malicious sites. Most of these use variants of PageRank, e.g, TrustRank [23]. In [24], a graphical model is built based on DNS requests, while in [25], both content based and link/graph based features are used to train a classifier. However, they do not address scalability of their algorithms.

III. Proposed Methodology

In our implementation of large-scale graph inferencing we address the following requirements: scalability, efficient graph representation, numerical stability and partitioning. We estimate the scalability of the algorithm by considering the complexity of computations and communications in the algorithm. Belief propagation is implemented on top of Apache Spark GraphX. Generalized graph representation is addressed by using factor graphs. We also implement a simpler, more efficient representation, which can be used for pairwise models, that is, graphical models where order of potential functions is limited to two. Numerical stability is handled by the use of logarithmic domain for computations. We propose a greedy partitioning algorithm that minimizes the communication overhead required to update the state of partitioned graph.



(a) Factor graph representation (b) Pairwise graph representation

Fig. 1: Example of a graphical model.

A. Scalability Modeling

We estimate the algorithm scalability by considering the amount of communication and computation it requires if implemented in Spark. Spark follows the bulk synchronous parallel model of data processing. The time of one iteration consists of computation and communication times. We have derived the iteration time of Belief Propagation in our recent work [26]. The computation time complexity is: $t_{cp} = max_{i\in[1,n]}(E_i)/(F \cdot n) \cdot (S + 2 \cdot (S + S^2))$, where n is the number of computationally equal workers with F FLOPS; S number of states; $max_{i\in[1,n]}(E_i)$ is the computing device (partition) with maximum number of edges E_i. The communication time complexity is: $t_{cm} = 32/B \cdot r \cdot V \cdot S$, where B is the connection bandwidth between workers; 32 is the number of bits per state; r is the replication factor; V is the total number of vertices.

According to the proposed model, the communication is proportional to the number of vertices and the replication factor. Per-vertex computation is relatively inexpensive, if the number of states in graphical model is small. The maximum scalability is reached when computation time is equal to communication time.

B. Generalized graph representation

The generic representation of a graphical model is a factor graph as shown in Figure 1a. Variables and factors are represented as vertices. Thus the number of variables is the sum of variables and factors. Priors are also factors, but for the sake of simplicity they can be merged with the vertices that represent variables. Edges represent interaction between variables and factors. A factor graph can be converted to a pairwise representation, as shown in Figure 1b. The conversion reduces the number of vertices to the number of variables if the original factors are pairwise. Therefore, such representation requires less memory and is appropriate if the original model is itself pairwise. In particular, vertex represents a variable and edge represents a factor.

C. Numerical stability

We store all values in log form and implement computations in log domain in order to prevent underflow. There are three types of computations in the original domain: multiplication, division, and normalization. Multiplication is needed for computing conditional probabilities. Division is needed to remove the values of the incoming message from the belief. These two operations are translated to summation and subtraction in log domain: $\log(x \cdot y) = \log x + \log y; \log(x/y) = \log x - \log y$.

Normalization involves summation of values in the original domain. It is possible to factor out the largest value to prevent overflow[2] (assuming that $x \geq y$): $\log(x+y) = \log x + \log(1 + e^{\log y - \log x})$.

D. Partitioning

Any large-scale graph processing requires a distributed implementation and hence partitioning of the graph. We now describe our proposed graph partitioning scheme. The input to our partitioner is an unordered list of edges and nodes. The data on the nodes and edges is ignored because our technique is algorithm and data agnostic. We also compute the degree of each vertex and determine if the vertex is low-degree or high-degree based on a tunable threshold. This information is then used by our heuristic. Motivated by Least Incremental Cost (LIC) [17] and PowerLyra [15], we design a greedy vertex-cut partitioning heuristic that minimizes the expected cost of partitioning. Specifically, these are the metrics that it tries to minimize, in the order of higher to lower priority:

- Number of vertices that are replicated.
- Total number of low-degree vertices (including replicas).
- Total number of high-degree vertices (including replicas).
- Difference between maximum and minimum load, measured in the number of edges, of all partitions.

We term our partitioning policy as Least low-Degree Incremental Cost (LDIC) since it prioritizes avoiding replication of low-degree vertices over that of high-degree vertices.

LDIC chooses the set of candidate partitions for an edge:

- Partitions that contain replicas of both incident vertices, if it exists.
- Otherwise, if one of the incident vertices is low-degree and the other high-degree, partitions that contain replicas of the low-degree vertex, if it exists.
- Otherwise, partitions that contain at least one of the vertices, if it exists.
- Otherwise, all the partitions.

Once the candidate partitions are chosen, the edge is assigned to the least loaded one among them and replicas are created for the incident vertices on that partition.

LDIC partitioning policy prioritizes reducing replication over maintaining load balance. Similar to LIC, for each edge, it first determines the candidate partitions that minimize replication the most. However, unlike LIC, it prioritizes minimizing replication of low-degree vertices over that of high-degree vertices. It then assigns the edge to the least loaded partition among the candidates. Although PowerLyra has a similar goal, it does not consider the load of a partition. Moreover, it does not handle symmetric or undirected graphs well since it explicitly considers the direction of an edge, whereas our policy is agnostic to the direction of an edge.

E. In-memory Optimization

Spark is gaining widespread adoption for large-scale workloads as an in-memory analytics platform. Today, running large-scale iterative, memory intensive workloads, such as BP, is still inefficient mainly due to unnecessary communication and memory inefficiencies in the Java heap. Given the growing availability of affordable scale-up servers, we have leveraged performance benefits from in-memory processing on shared memory of scale-up servers to data analytics applications

[2]For derivation see http://colorfulengineering.org/logmath-notes.pdf

TABLE I: Pairwise BP experiment results

Vertices	Edges	Size on TMPFS	Size in JVM memory	Iteration time (sec)	Number of iterations
0.2M	0.7M	0.03GB	0.4GB	4	8
1.6M	8.9M	0.3GB	5.2GB	16.8	8
16.2M	99.3M	5.4GB	59.7GB	25.3	8
101.7M	1.75B	64GB	840.5GB	132	20

TABLE II: Factor graph BP experiment results

Vertices	Edges	Size on TMPFS	Size in JVM memory	Iteration time (sec)	Number of iterations
0.9M	1.4M	0.05GB	1.2GB	1.3	14
10.7M	18M	0.6GB	13GB	21.8	15
115.5M	198.5M	6.9GB	154.2GB	33.5	16
1.9B	3.5B	118GB	2.5TB	600	40

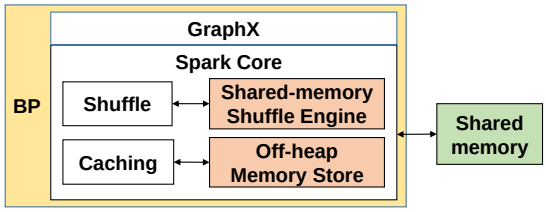


Fig. 2: BP with in-memory optimization on Sparkle

and designed and implemented an enhanced Spark, called *Sparkle* [27]–[29]. Sparkle leverages the large shared memory in scale-up systems to optimize Spark's performance for communication and memory intensive workloads. Figure 2 shows our BP implementation with in-memory optimizations using shared memory in Sparkle.

IV. Experiments

We performed a series of experiments to evaluate the scalability of our implementation of BP and to demonstrate its applicability to a real-world use case. The performance is evaluated in terms of the iteration times and space usage of our BP implementation depending on the graph size and graph representation.We also performed experiments to measure the effectiveness of our proposed partitioning algorithm, LDIC, which can be used to partition any graph, and not just graphical models that are considered here.

Our experimental platform is Superdome X with 240 cores and 12 TB DRAM across 16 NUMA nodes (sockets). Each worker process is bound to the CPU and memory of a NUMA node. We used Spark 1.6.1 with 8 up to 45 workers, and 32 GB up to 164 GB RAM for the JVM memory. For Spark shuffling, we used TMPFS bound to a NUMA node to store the shuffle data and TCP/IP communication. For the in-memory optimized Sparkle experiments, we used the shared-memory shuffle engine for shuffling and off-heap memory store for caching.

We ran our experiments on graphical models with varying number of vertices and edges with the largest model containing 1.9 billion vertices and 3.5 billion edges. Two of these, the largest one and the one with 101.7 million vertices and 1.75 billion edges are derived from real-world web graph obtained from Web Data Commons [30], [31]. It is the hyperlinked graph obtained from a web crawl conducted by Common Crawl in August 2012 [32]. All the other graphical models were generated synthetically from a real graphical model based on DNS data such that basic properties like degree distribution and factor potentials were preserved [33]. The graphs used in our experiments are shown in Tables I and II.

A. Iteration time and space usage

In this section, we compare the iteration time and space usage (for both TMPFS and JVM memory) of BP for graphs of varying sizes, using the two different representations. TMPFS is used to store shuffle data.

In Table I, which summarizes results for pairwise representation, we see the iteration time and space usage increase with the size of graph (also shown in Figure 3) Here the increase is close to linear. Note that the web graph (101.7M/1.75B) takes more number of iterations for convergence (20) than the others. While in general it is hard to say how many iterations BP will take to converge (or if it will converge at all), in practice it usually converges in a small number of iterations (10–20). Table II shows results for factor graph representation. As the graph size increases, both iteration time and space usage for factor graphs increase more steeply than for pairwise graph (see Table II and Figure I).

B. Partitioning

In this section, we first show that our LDIC policy yields better partitions than the existing state-of-the-art and then show the impact of LDIC policy on the performance of BP.

We statically analyze LDIC and compare it with existing partitioning policies: random (used in Spark), PowerGraph [14], PowerLyra [15], and LIC [17]. We partition an undirected graph with 162 million vertices and 1 billion edges into 64 partitions using the different policies. To estimate the communication volume required for the resulting partitions, we measure the replication factor (mean number of replicas per vertex) and the number of vertices that are replicated, shown in Figure 4. Among the existing policies, LIC has the lowest replication factor but it replicates more vertices than PowerLyra since it replicates more low-degree vertices. PowerLyra replicates the lowest number of vertices but has a higher replication factor than LIC since it creates more replicas for high-degree vertices. Our LDIC policy is the best in both reducing the number of replicated vertices and the replication factor. To measure the load balance, we measure the minimum and maximum number of edges assigned to a partition. PowerGraph and PowerLyra has 5% and 10% more edges in the maximum loaded partition than the minimum loaded one, while the rest, including LDIC, have less than 1%. This shows that LDIC partitioning policy minimizes communication volume while balancing the load.

We compared the iteration time for our pairwise BP with 101.7M/1.75B graph using the LDIC partitioning vs. random partitioning which is the Spark's default partitioning. As expected, the LDIC partitioning improves the iteration time almost 2x (132 vs. 247 sec, for LDIC vs. random partitioning). This gain is due to reducing shuffle data and the shuffle operation is one of the bottlenecks in Spark's BSP.

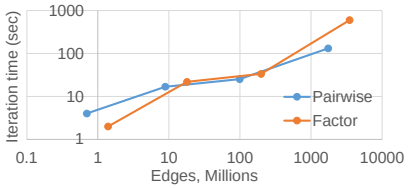


Fig. 3: BP iteration time on various graphs

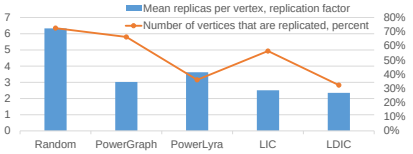


Fig. 4: Comparison of replication in various partitioning schemes

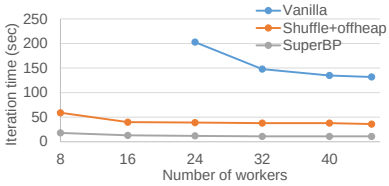


Fig. 5: BP iteration times on the 101.7M/1.75B graph

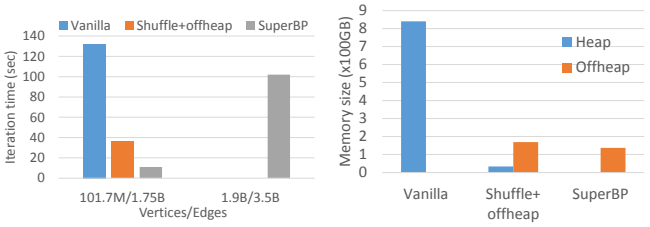


Fig. 6: (a) BP iteration times evaluated on two different graph sizes; (b) BP Memory usage on the 101.7M/1.75B graph

C. Strong scaling and memory optimization

In this section, we present how in-memory optimization in our BP implementation improves its performance. We use pairwise graph representation in these experiments. We compare vanilla BP (no modifications) with our in-memory optimized BP.

We conducted experiments with different variants of in-memory optimization. The first variant uses the shared memory shuffle engine for the shuffle operation and the off-heap memory store for data caching. We call this *Shuffle+offheap*. We further optimized this implementation (Shuffle+offheap) by taking advantage of the globally visible data structures in the off-heap memory store, by consolidating two shuffle stages into one single stage (out of the total three stages in one iteration), and replacing Scala with C++ implementation for the cpu-intensive message computation. We call this second variant *SuperBP*.

We used the two largest graphs, 101.7M/1.75B and 1.9B/3.5B for these and found that benefits from Sparkle get bigger as the graph size is larger since larger graphs have higher memory consumption and consequently more frequent calls to garbage collector.

As expected, our in-memory optimized BP implementations dramatically improve the performance for both iteration time and memory usage. BP with Shuffle+offheap is 3x and SuperBP is 10x faster than BP with vanilla Spark in iteration time (see Figure 6a). Note that only SuperBP is able to run on the 1.9B/3.5B graph; the others run out of memory.

Figure 6b shows memory usage of the three variants of BP on the 101.7M/1.75B graph. While the Vanilla BP uses only JVM heap memory, Shuffle+offheap BP and SuperBP use off-heap memory for shuffle and data caching. As shown in the figure, the memory usage in our in-memory optimized BP is dramatically reduced since using the off-heap memory allows more compact data representation eliminating the overhead of Java objects. Note that Shuffle+offheap BP uses a bit of heap memory to store the input graph while SuperBP uses the off-heap memory for input graph as well as shuffle and data caching.

Figure 5 shows how the three variants of BP scale with increasing number of workers on the 101.7M/1.75B graph. As shown in the figure, the iteration time of BP with vanilla Spark decreases with adding more workers while the in-memory optimized BP does not benefit as much. Overall it shows our in-memory optimized BP runs with much smaller resources but better performance.

D. Large Scale BP Use Case: Malicious Websites Detection

In order to determine a maliciousness score of a website, we leveraged hyperlinked web graph data together with blacklists and whitelists of domains.

We used a real, large-scale web graph for this use case. We derived it from the web graph available at Web Data Commons [30], based on the Common Crawl data [32]. The graph is from a August 2012 crawl [31] and contains web graphs at three different granularities, namely where vertices are web pages, web hosts, or web domains. For this particular use case we looked at the web hosts graph. After pre-processing the raw graph, we transformed it into a Markov random field (MRF) with 101.7 million vertices and 1.75 billion edges. The vertices in the MRF represent random variables indicating the probability of that host being malicious. The edges, determined by the hyperlinks, represent the relationships between the hosts parameterized by the edge potential.

We exploit the concept of homophily, that is, entities like to be associated with similar entities. In the context of the web graph, this implies that benign sites are likely to have hyperlinks to other benign sites, while malicious sites are likely to have hyperlinks to other malicious sites. In particular, we used high edge potentials for benign–benign links, and malicious–malicious links; very low potential for benign–malicious links, since good sites are unlikely to link to malicious ones; and, moderate potential for malicious–benign, since malicious sites may also link to good sites, in addition to other malicious ones.

Since the web graph is from August 2012, we used archive.org to obtain blacklists [34] from that time period. We used the one million most visited sites compiled by Alexa [35] as a proxy for a whitelist. Again we used archive.org to get the list from the same time period. Both these lists contained domain names while our graph had host names. After processing and matching the sites in these lists with the sites in the host web graph, there was an overlap of 17.8 M hosts in the graph with the whitelist domains, and 930 K hosts matched with domains in the blacklists. This information was used to populate the node priors of the MRF. All the vertices (hosts) that had no match were assigned a uniform prior.

We use the pairwise graph representation for the MRF with the following parameters: maximum number of iterations – 50, epsilon – $1e-3$ (used to determine convergence). Once all the parameters are determined, we run belief propagation on the graphical model to infer the marginal probability of all the vertices (hosts), which would be an estimate of a maliciousness score of the host.

Belief propagation converged in 20 iterations in about 250 seconds. We ranked the nodes based on their marginals. It is challenging to validate the results since the web graph is from several years ago (and malicious sites are quite dynamic). Even though thorough validation proved impractical, we qualitatively validated the results by having an expert manually examine a sample of the top malicious sites discovered. We were able to retrieve a good percentage of the top 100 discovered malicious sites from common crawl. Examination of these pages found that a large number of these sites either had inappropriate content, spam links, or misleading content (e.g., links did not match content).

V. Conclusion

Graphical model inference algorithms such as loopy belief propagation are widely used, but are not very scalable. We designed and implemented a scalable version of BP on Apache Spark GraphX, that scales to billions of vertices without requiring a specialized graph processing platform (and the associated data copying and migration). The main components of our approach are: efficient graph partitioning, concise representation, and leveraging in-memory optimizations for large memory machines. Our experiments show our partitioning algorithm provides a 2x speed-up, while with all optimizations combined we obtain a 10x speed-up over vanilla Spark GraphX. We present a large scale use case for estimating maliciousness of web sites using a graphical model constructed from a hyperlinked web graph. We have open sourced our implementation as project Sandpiper [7].

References

[1] M. J. Wainwright and M. I. Jordan, "Graphical models, exponential families, and variational inference," *Foundations and Trends in Machine Learning*, vol. 1, no. 12, pp. 1–305, 2008.

[2] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, "An introduction to mcmc for machine learning," *Machine Learning*, vol. 50, no. 1, pp. 5–43, Jan 2003.

[3] "Apache giraph," http://giraph.apache.org.

[4] "Turi," http://turi.com.

[5] "Apache spark," http://spark.apache.org.

[6] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." in *OSDI*, vol. 14, 2014, pp. 599–613.

[7] "Project sandpiper: Implementation of the loopy belief propagation algorithm for apache spark," https://github.com/HewlettPackard/sandpiper.

[8] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[9] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[10] K. P. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1999, pp. 467–475.

[11] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[12] ——, "Multilevel k-way hypergraph partitioning," in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, ser. DAC '99. New York, NY, USA: ACM, 1999, pp. 343–348.

[13] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 124–124.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30.

[15] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:15.

[16] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM International Conference on WSDM*, ser. WSDM '14. New York, NY, USA: ACM, 2014, pp. 333–342.

[17] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: ACM, 2014, pp. 1456–1465.

[18] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 1222–1230.

[19] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, "Data partitioning strategies for graph workloads on heterogeneous clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 56:1–56:12.

[20] U. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM J. Sci. Comput.*, vol. 32, no. 2, pp. 656–683, Feb. 2010.

[21] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International on CIKM*, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 243–252.

[22] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond blacklists: Learning to detect malicious web sites from suspicious urls," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 1245–1254.

[23] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen, "Combating web spam with trustrank," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 576–587.

[24] P. Manadhata, S. Yadav, P. Rao, and W. Horne, "Detecting malicious domains via graph inference," in *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, ser. AISec '14. New York, NY, USA: ACM, 2014, pp. 59–60.

[25] C. Castillo, D. Donato, A. Gionis, V. Murdock, and F. Silvestri, "Know your neighbors: Web spam detection using the web topology," in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '07. New York, NY, USA: ACM, 2007, pp. 423–430.

[26] A. Ulanov, A. Simanovsky, and M. Marwah, "Modeling scalability of distributed machine learning," in *Data Engineering (ICDE), 2017 IEEE 33nd International Conference on*. IEEE, 2017.

[27] "Project sparkle," https://github.com/HewlettPackard/sparkle.

[28] M. Kim, J. Li, H. Volos, M. Marwah, A. Ulanov, K. Keeton, J. Tucek, L. Xu, and P. Fernando, "Sparkle: Optimizing spark for large memory machines," in *Proceedings of the 2017 ACM Symposium on Cloud Computing*. ACM, 2017.

[29] M. Kim, J. Li, H. Volos, M. Marwah, A. Ulanov, K. Keeton, J. Tucek, L. Cherkasova, L. Xu, and P. Fernando, "Sparkle: Optimizing spark for large memory machines and analytics," *CoRR*, vol. abs/1708.05746, 2017.

[30] "Web data commons," http://webdatacommons.org.

[31] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web — revisited: A trick of the heavy tail," in *Proceedings of the 23rd International Conference on WWW*. New York, NY, USA: ACM, 2014, pp. 427–432.

[32] "Common crawl data," http://commoncrawl.org.

[33] A. Chakrabarti, M. Marwah, and M. Arlitt, "Robust anomaly detection for large-scale sensor data," in *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, ser. BuildSys '16. New York, NY, USA: ACM, 2016, pp. 31–40.

[34] "Urlblacklist," http://urlblacklist.com.

[35] "Alexa500," http://www.alexa.com/topsites.