Using RAPIDS AI to Accelerate Graph Data Science Workflows

Todd Hricik
Ying Wu College of Computing
New Jersey Institutue of Technology
Newark, NJ USA
th99@njit.edu

David Bader
Ying Wu College of Computing
New Jersey Institute of Technology
Newark, NJ USA
bader@njit.edu

Oded Green
NVIDIA Corporation
Santa Clara, CA USA
ogreen@nvidia.com

Abstract*—Scale free networks are abundant in many natural, social, and engineering phenomena for which there exists a substantial corpus of theory able to elucidate many of their underlying properties. In this paper we study the scalability of some widely available Python-based tools for the empirical investigation of scale free network data in a typical early stage analysis pipeline. We demonstrate how porting serial implementations of commonly used pipeline data structures and methods to parallel hardware via the NVIDIA RAPIDS AI API requires minimal rewriting of code. As a utility for each pipeline we recorded the time required to complete the analysis for both the serial and parallelized workflows on a task-wise basis. Furthermore, we review a statistically based methodology for fitting a power-law to empirical data. Maximum likelihood estimations for scale were inferred after using Kolmogorov-Smirnov based methods to determine location estimates. Our serial implementation of a typical early stage network analysis workflow uses a combination of widely used data structures and algorithms provided by the NumPy, Pandas and NetworkX frameworks. We then parallelized our workflow using the APIs provided by NVIDIA's RAPIDS AI open data science libraries and measured the relative time to completion for the tasks of ingesting raw data, creating a graph representation of the data and finally fitting a power-law distribution to the empirical observations. The results of our experiments, run on graphs ranging in size from 1 million to 20 million edges, demonstrate that significantly less time is required to complete the tasks of generating a graph from an edge list, computing the degree of all nodes in the graph and fitting the scale and location parameters to the observed data.

Keywords—graph analytics, GPU computing, data science

I. Introduction

Systems that can be represented by a graph having degree distributions following a power-law are commonly found in our everyday lives. Examples can be seen in the evolution of the World-Wide Web [1][2][3], how we interact on social network applications [4][5], and even in the melodic and rhythmic qualities found in compositions of classical music [6]. These phenomena and many others which exist in a vast array of fields of study are of intense interest among the public and private research communities. As empirical network datasets become increasingly large, it is becoming ever more important for accessible and scalable analysis workflows to become available. While numerous tools are available for the implementation of network analytics pipelines, those that provide the highest degree of ease of use are based upon sequential programming models. For example the NetworkX graph analysis framework is well documented and widely used among network analysts as it provides the necessary tools, among many more, to convert raw data read into Pandas or NumPy data structures into a graph object and compute the degree of each vertex within it. Tooling that provides methods for the assessment of whether the graph is indeed scale-free however is far less diverse and abundant. This in part may be due to the fact that ad-hoc methods for analyzing scale-free nature of empirical data are often employed due to their ease of use. A consequence of using ad-hoc methods is a decrease in the reliability of the results obtained and a lack of statistical theory that supports their usage.

A scale free network has a degree distribution *k* described by a power-law

$$p(k) = P(K{=}k) = Cx^{-\alpha} \tag{1}$$

which can be mathematically characterized for either continuous or discrete random variables representing the degree, by weight in the former case or edge count in the latter, of a vertex in the graph. In this paper we focus solely on the distribution of the discrete instances for the sake of brevity and for the purpose of minimizing the amount of necessary background and consideration of the technical caveats that result when also considering the continuous case (see [7][8][9][10][11] for a detailed discussion). In the discrete case the discrete power-law distribution is parameterized by its scale (α) and its location (x_{min}) where the normalization factor

$$C = \zeta(\alpha, x_{min})^{-1} \tag{2}$$

is the inverse of the Hurwitz zeta function which is itself a generalization of the Riemann zeta function. The Riemann zeta form was first introduced by Euler and numerical solutions for these infinite sums have been developed [12]. See [13][14][15][16] for details of the numerical analysis involved with solving such sums. The python code provided by Clauset [17] computes (3) via an implementation based upon the GNU Scientific Library (GSL) [18] and whose analytic details can be visited in [15].

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE
978-1-7281-9219-2/20/$31.00 ©2020 IEEE

$$\zeta(\alpha, x_{min}) = \sum(n + x_{min})^{-\alpha} \tag{3}$$

Evident by its inclusion in many research reports, it should be noted that, as alluded to previously, one common ad-hoc practice is to assess whether the degree distribution of a graph follows a power-law by fitting a linear regression line, via ordinary least squares, to a log-log scaled plot of the degree histogram and estimate the scaling parameter via the slope of the fitted line and the location parameter by noting the degree at which the fitted line first intersects a point on the histogram. Unfortunately, such practice subjects the estimated scale and location parameters to both bias and error for several reasons. Among them are the lack of normality of the linear model residuals and the presence of correlation among them [8][11]. Clauset *et al.* provide a Python implementation [12] of power-law based empirical data [7] based upon a statistical framework that provides maximum likelihood and Kolmogorov Smirnov based methods for fitting empirical data to a power-law distribution as was described by Goldstein [10]. Briefly, the location parameter is estimated by iterating over all values of location within the search space and choosing the location that minimizes the distance D between the cumulative distribution functions of the observed data S(x) and the best fit power-law model P(x) where

$$D = max_{(x \geq xmin)}|S(x)-P(x)| \tag{4}$$

The MLE estimator for the scaling parameter is then computed

$$MLE(\alpha) = max_{(\alpha)}(-n \ln \zeta(\alpha, x_{min}) - \alpha \sum \ln x_i) \tag{5}$$

Summary of Results: To illustrate a typical early stage network analysis workflow we performed the ingestion of raw tabular data representing a network, then generate a graph representation of that data. Next we compute the degree distribution of graph and finally using slightly modified code based upon [17], we fit a power-law distribution by computing estimates of the scale and location parameters while recording the amount of time required to complete each step.

Scalable end to end network analytic pipelines that make use of data structures and tooling components which are easily integrated into complex workflows and which are experientially similar to those used across other genres of data science and engineering activities have much potential for adding value to current network analysis endeavors. To illustrate, we ported our serial pipeline implementation which uses NetworkX, NumPy and Pandas to a parallel architecture based workflow using some of the available RAPIDS AI APIs provided by NVIDIA, namely the cuGraph and cuDF Python packages. RAPIDS AI is built upon an in-memory data structure using Apache Arrow and provides several GPU data and network analysis capabilities that are implemented using functions that are syntactically very similar to the workflow tools provided by NetworkX, NumPy and Pandas. For example, the cuDF package provides data structures and methods which are based upon Pandas while cuGraph, in the same spirit, provides several graph analysis methods structured similarly to those seen in NetworkX. As a result, using cuDF and cuGraph to accelerate our workflow to GPU hardware accelerators required minimal code rewriting and to a large extent preserved the cross-interpretability of the code between the serial and parallel workflows. More specifically we ported our serial Python 3.6 pipeline to a CUDA GPU appropriate syntax by replacing NumPy and Pandas objects to the CuPy and cuDF objects provided by the RAPIDS AI. Similarly, we ported all NetworkX based syntax to a GPU appropriate RAPIDS AI cuGraph syntax.

II. Methods

A. Data

We used as our experimental dataset the social network data consisting of a series Google+ web crawls performed by Gong et. al. [4][5]. All source network data is in the form of a space delimited edge list containing 462,994,069 edges and contains 28,942,911 vertices. We sampled this dataset by taking the first 1 million, 5 million, 10 million and 20 million edges and passing each sample to the pipeline described below. Our key motivation for taking a subset of the graph is that the execution times are long.

B. Network Analysis Workflow

In our serial implementation, the edge list was first ingested using the *pandas.read_csv()* function, creation of the graph was carried out using the NetworkX function *from_pandas_edgelist()* and the degree of each node was computed by iterating over the list of values provided by the *graph.degree()* method of NetworkX's graph data structure. Next, our modified serial Python 3.x version of Ornstein's original code, serial_ptyhon3_plfit.py, was used to estimate the scale and location parameters given the degrees computed in the previous step.

The GPU code using RAPIDS AI is very similar to our serial Python implementation. The first difference is in the input ingestion where we use *cudf.read_csv()* in place of *pandas.read_csv()*. Second, creation of the graph was carried out using the cuGraph method *graph.from_cudf_edgelist()*, and the degree of each node was computed via the *graph.degrees*() method provided by the cuGraph package. Each of the serial and parallel implementation calls used for each step of the workflow can be seen in serial_gplus_exp.py and parallel_gplus_exp.py respectively. Up to this point, most of the CPU and GPU implementations were identical for all but the package used to call the basic functionality. We reimplemented the parameter estimation code (previously serial_python3_plfit.py) to use GPU based libraries and used parallel_ptyhon3_plfit.py, to estimate the scale and location parameters given the degrees computed in the previous step. Code profiling of compute times required by the original multi-threaded purely python code (Ornstein) and our serial_python3_plfit.py show two major bottlenecks that occur during the parameter estimation process. First the MLE estimator (eq. 5) is obtained from a vector of likelihoods computed for each possible scale parameter in the search space. This vector is constructed via appending each likelihood vector element via iteration within a for loop which accounts for more than 25% of the required compute time in our serial implementation. Our parallel_python3_plfit.py code uses a CuPy broadcasting based computation that resolves the entire likelihood vector in one step without the need to iterate within a for loop. Secondly, (eq. 4) must be computed for each unique degree observation obtained from the graph. The maximum of the resulting vector establishes the appropriate location parameter. This vector of the resulting fit is constructed via iteration of append operations within a for loop and is responsible over 49% of the compute time required in our serial implementation. Our parallel_python3_plfit.py implementation uses CuPy broadcasting to compute the entire vector of fits in a single step without requiring element-wise iteration within a for loop.

III. Empirical Performance Analysis

Experimental Setup: Serial experiments were run on an Intel Xeon E5-2630-v4 processor, and parallel experiments were run on NVIDIA TITAN V GPU. As mentioned, we use a slightly modified Python 3.x version of Ornstein's code, serial_plfit.py, from [4] which was written in Python 2.x. These modifications include usage of the functions that are used for estimating discrete distributions and replacing the Python 2.x list and math module based syntax with Python 3.6 NumPy and Pandas syntax. Note, GPU execution times include data transfer times from the CPU to GPU.

Benchmarks: We analyze the full end-to-end execution of the workflow. We report time for the four key execution phases: 1) data ingestion from the file system and into memory, 2) conversion of the data frame to a graph data structure, 3) computing the degree of the nodes and 4) parameter estimation. Typically, the first two phases are part of the ETL phase.

Performance Analysis:.

Time to completion for each workflow task using our serial and parallel implementations are provided (Fig. 1) along with the performance speedup achieved using the parallelized workflow. While the parallelized version of the data ingestion step did not break even until given a graph containing more than 10 million edges, we observed a 2x sped up in the time required to ingest 20 million edges. We are investigating the source of compute/memory allocation burden that is incurred during the edge ingestion process so that we can achieve additional time to completion benefits during this step. The task of computing each graph using our parallel implementation was achieved between 26x-90x less time than that taken by our serial implementation. Furthermore, our parallel code reduced the time to completion of the degree count task by 11x-28x when compared to our serial implementation. Finally, we observe that the original pure python approach (Ornstein), which uses CPU multithreading, can decrease the time to completion of the parameter estimation task by 8x-11x required by our serial implementation. Moving the computations of computing the required fits and MLE estimators to the GPU via CuPy broadcasting reduced the required time to completion of the parameter estimation task by 46x-116x.

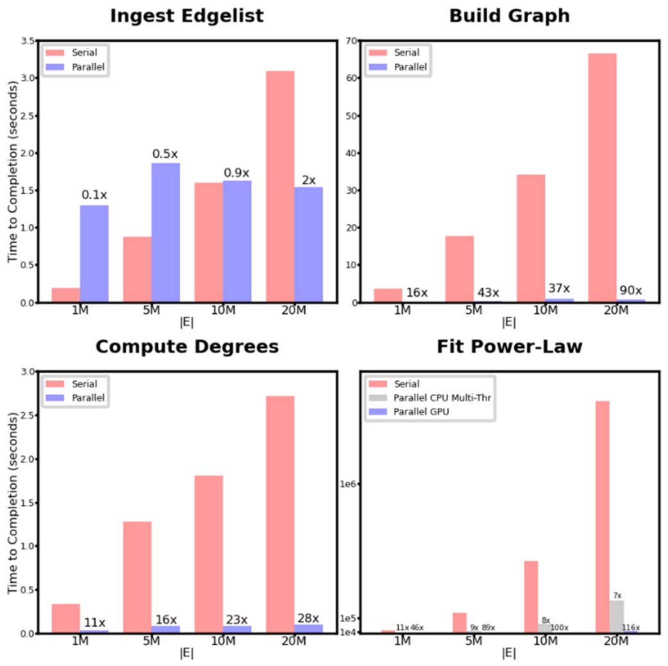

Fig. 1. Time (in seconds) for each step of the network analysis workflow using serial (red), multi-thread CPU (grey) and parallel gpu (blue) implementations.

IV. Conclusion

Replacing the Pandas and NumPy based methods within our workflow with their syntactically similar RAPIDS AI and CuPy counterparts resulted in significant reductions in the time to completion of all tasks in the network analysis workflow applied to graphs of all sizes in the experiments with the exception of the data ingestion step where graphs having less than 5 million edges were used. This is most likely caused by overhead associated with the cuDF method for reading the raw CSV data from file and we expect that the ingestion of raw data containing edges numbering on the order of 100 million and more would begin to show significant reduction in the time to completion. In future studies, we plan to run similar experiments using graphs of such size and greater to investigate how NVIDIA's RAPIDS AI and CuPy based workflows scale. In addition, we will investigate the parameter estimation step and determine if there are other optimizations that can further decrease the amount time to completion. It is likely that applying parallel computation of all cumulative distribution functions for the data and best-fit power-law models can provide additional performance gains.

References

[1] L. A. Adamic, B. Huberman, A. L. Barabasi, R. Albert, H. Jeong, G. Bianconi, "Power-law distribution of the world wide web," Science, vol. 287, 5461, p. 2115, 2000.

[2] B. Huberman, L. Adamic, "Growth dynamics of the world-wide web," Nature, vol. 2, 1999, p.131.

[3] M. Faloutsos, P. Falousos, C. Faloutsos, "On power-law relationships of the internet topology," ACM SIGCOMM Computer Communication Review, 1999, pp. 251-262.

[4] N. Gong, W. Xu, "Reciprocal versus parasocial relationships in online social networks," Springer Social Network Analysis and Minging (SNAM), 4(1), 2014.

[5] Gong, N.Z., Xu, W., Huang, L., Mittal, P., Stefanov, E., Skar, V. Song, D., "Evolution of social-atrribute networks: measurements, modeling, and impllications using Google+",

ACM Workshop on Social Network Mining and Analysis (SNA-KDD), co-located with KDD, 2012.

[6] D. Levitin, P. Chordia, V. Menon, “Musical rhythm spectra from Bach to Joplin obey a 1/f power law,” Proceedings of The National Academy of Sciencs (PNAS), vol. 109, no. 10, 2012, pp. 3716-3720.

[7] Clauset A., Shalizi, C.R., Newman, M.E.J., "Power-law distributions in empirical data.", SIAM Review, 51, 2009, pp. 661-703.

[8] Bauke, H., "Parameter estimation for power-law distributions by maximum likelihood methods", Eropean Physical Journal B 58, 167, 2007, pp. 167-173.

[9] Clauset, A., Young, M., Gleditsch, K. S. , ”On the frequency of severe terrorist events”, The Journal of Conflict Resolution, Vol. 51. No.1, Feb., 2007, pp. 58-87.

[10] Goldstein, Michel L., Steven A. Morris, and Gary G. Yen. 2004. Problems with fitting to the power-law distribution. European Physical Journal B: Condensed Matter and Complex Systems 41 (2): 255-8.

[11] M. Kutner, C. Nachtsheim, J. Neter, Applied Linear Regression Models, 4th ed., McGraw-Hill Irwin: New York, 2004, pp. 100-118.

[12] I. Blagouchine, “The history of the functional equation of the zeta-function,” Seminary on the History of Mathematics, Stedklov Inst. Of Mathematics, St. Petersberg, March 1, 2018, unpublished.

[13] H. Haas “Calculation of Riemann’s zeta function,” Mathematische Zeitschrift, Vol. 32(1), 1930, pp. 458-464.

[14] L. Vepstas, “An efficient algorithm for accelerating the convergence of oscillatory series, useful for computing the polylogarithm and Hurwitz zeta functions,” Numerical Algorithms, Vol. 47(3), 2007, pp. 211-252.

[15] Handbook of Mathematical Functions: with Formulas, Grpahs and Mathematical Tables, Revised ed., Dover, 1965, pp. 803-818.

[16] M. Coffey, “On some series representations of the Hurwitz zeta function,” Journal of Computational and Applied Mathematics, Vol. 216(1), 2008, pp. 297-305.

[17] Original Python 2.x implementations of plfit.py and plpva.py are available at http://tuvalu.santafe.edu/~aaronc/powerlaws.

[18] GNU Scientific Library (GSL) available at https://www.gnu.org/software/gsl.