

Static Graph Challenge: Subgraph Isomorphism

Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra,
Paul Monticciolo, Albert Reuther, Steven Smith, William Song, Diane Staheli, Jeremy Kepner
MIT Lincoln Laboratory, Lexington, MA

Abstract—The rise of graph analytic systems has created a need for ways to measure and compare the capabilities of these systems. Graph analytics present unique scalability difficulties. The machine learning, high performance computing, and visual analytics communities have wrestled with these difficulties for decades and developed methodologies for creating challenges to move these communities forward. The proposed Subgraph Isomorphism Graph Challenge draws upon prior challenges from machine learning, high performance computing, and visual analytics to create a graph challenge that is reflective of many real-world graph analytics processing systems. The Subgraph Isomorphism Graph Challenge is a holistic specification with multiple integrated kernels that can be run together or independently. Each kernel is well defined mathematically and can be implemented in any programming environment. Subgraph isomorphism is amenable to both vertex-centric implementations and array-based implementations (e.g., using the Graph-BLAS.org standard). The computations are simple enough that performance predictions can be made based on simple computing hardware models. The surrounding kernels provide the context for each kernel that allows rigorous definition of both the input and the output for each kernel. Furthermore, since the proposed graph challenge is scalable in both problem size and hardware, it can be used to measure and quantitatively compare a wide range of present day and future systems. Serial implementations in C++, Python, Python with Pandas, Matlab, Octave, and Julia have been implemented and their single threaded performance have been measured. Specifications, data, and software are publicly available at GraphChallenge.org.

I. INTRODUCTION

Increasingly, large amounts of data are collected from social media, sensor feeds (e.g. cameras), and scientific instruments and are being analyzed with graph analytics to reveal the complex relationships between different data feeds [1]. Many graph analytics are executed in large data centers on large cached or static data sets. The processing required is a function of both the size of the graph and the type of data being processed. There is also an increasing need to make decisions in real-time to understanding how relationships represented in a graph evolve. Previous research on streaming graph analytics has been limited by the amount of processing required. Graph analytic updates must be performed at the speed of the incoming data. The sparseness of graph data can make the application of graph analytics on current processors extremely inefficient. This inefficiency has either limited the size of the graph that can be addressed to only what can be held

in main memory or requires an extremely large cluster of computers to make up for this inefficiency. The development of a novel graph analytics system has the potential to enable the discovery of relationships as they unfold in the field rather than relying on forensic analysis in data centers. Furthermore, data scientists can explore associations previously thought impractical due to the amount of processing required. The Subgraph Isomorphism Graph Challenge and the Stochastic Block Partition Challenge [2] (see <http://GraphChallenge.org>) seek to enable a new generation of graph analysis systems by highlighting the benefits of novel innovations in these systems.

Challenges such as YOHO [3], MNIST [4], HPC Challenge [5], ImageNet [6] and VAST [7], [8] have played important roles in driving progress in fields as diverse as machine learning, high performance computing and visual analytics. YOHO is the Linguistic Data Consortium database for voice verification systems and has been a critical enabler of speech research. The MNIST database of handwritten letters has been a bedrock of the computer vision research community for two decades. HPC Challenge has been used by the supercomputing community to benchmark and acceptance test the largest systems in the world as well as stimulate research on the new parallel programming environments. ImageNet populated an image dataset according to the WordNet hierarchy consisting of over 100,000 meaningful concepts (called synonym sets or synsets) [6] with an average of 1000 images per synset and has become a critical enabler of vision research. The VAST Challenge is an annual visual analytics challenge that has been held every year since 2006; each year, VAST offers a new topic and submissions are processed like conference papers. The Subgraph Isomorphism Graph Challenge seeks to draw on the best of these challenges, but particularly the VAST Challenge in order to highlight innovations across the algorithms, software, hardware, and systems spectrum.

The focus on graph analytics allows the Subgraph Isomorphism Graph Challenge to also draw upon significant work from the graph benchmarking community. The Graph500 (Graph500.org) benchmark (based on [9]) provides a scalable power-law graph generator [10] (used to build the world's largest synthetic graphs) with the goal of optimizing the rate of building a tree of the graph. The Firehose benchmark (see <http://firehose.sandia.gov>) simulates computer network traffic for performing real-time analytics on network traffic. The PageRank Pipeline benchmark [11], [12] uses the Graph500 generator (or any other graph) and provides reference implementations in multiple programming languages to allow users to optimize the rate of computing PageRank (1st eigenvector)

This material is based upon work supported by the Defense Advanced Research Projects Agency under Air Force Contract No. FA8721-05-C-0002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Department of Defense.

on a graph. Finally, miniTri (see mantevo.org) [13], [14] takes an arbitrary graph as input and optimizes the time to count triangles.

The organization of the rest of this paper is as follows. Section II describes examples of data sets that are relevant to the Graph Challenge. Section III provides details on the specifics of the subgraph isomorphism problem. Section IV gives example algorithms and implementations that can be used to solve the specific subgraph isomorphism problem. Section V presents metrics and preliminary serial performance results of the example implementation over a range of graphs. Section VI summarizes the work and describes future directions.

II. DATA SETS

Scale is an important driver of the Graph Challenge and graphs with billions to trillions of edges are of keen interest. The Graph Challenge is designed to work on arbitrary graphs drawn from both real-world data sets as well as simulated data sets. Examples of real-world data sets include the Stanford Large Network Dataset Collection (see <http://snap.stanford.edu/data>), the AWS Public Data Sets (see aws.amazon.com/public-data-sets), and the Yahoo! Webscope Datasets (see webscope.sandbox.yahoo.com). These real-world data sets cover a wide range of applications and data sizes. While real-world data sets have many contextual benefits, synthetic data sets allow the largest possible graphs to be readily generated. Examples of synthetic data sets include Graph500, Block Two-level Erdos-Renyi graph model (BTER) [15], Pure Kronecker Graphs [16], and Perfect Power Law graphs [17], [18].

The focus of the Graph Challenge is on graph analytics. While parsing and formatting complex graph data is necessary in any graph analysis system, these data sets are made available to the community in a variety of pre-parsed formats to minimize the amount of parsing and formatting required by Graph Challenge participants. The public data are available in a variety of formats such as linked list, tab separated, and labeled/unlabeled. The Graph Challenge will provide data in two formats: tab separated value (TSV) triples in an ASCII file and MMIO ASCII format (see math.nist.gov/MatrixMarket) [19]. In each case, the data have been parsed so that all vertices are integers from 1 to the total number of vertices n in the graphs and all edges weights are set to a value of 1. In addition to this, all self loops were removed from the original datasets. For directed graphs, additional edges in the opposite direction were added to make the graphs un-directed. The edges are stored in TSV files as triples of tab separated numeric strings with a newline between each edge. For example, let all the starting and ending vertices be stored in the m element vectors \mathbf{u} and \mathbf{v} . The edges of the graph are stored in the TSV file as shown here:

$$\begin{array}{ccc} \mathbf{u}(1) & \mathbf{v}(1) & 1 \\ & \vdots & \vdots \\ \mathbf{u}(m) & \mathbf{v}(m) & 1 \end{array}$$

where $i = 1, \dots, m$, $\mathbf{u}(i) \in \{1, \dots, n\}$, and $\mathbf{v}(i) \in \{1, \dots, n\}$.

Filtering on edge/vertex labels is often used when available to reduce the search space of many graph analytics. Filtering can be applied at initialization, during intermediate steps, or at the vertex level. Such filtering can be invaluable and is highly problem specific. Some Graph Challenge data sets in their original forms have labels and some data sets are unlabeled. Some users will want to filter on labels and these innovations are encouraged. The provided example implementations will all work without labels.

III. STATIC GRAPH ISOMORPHISM CHALLENGE

A. Triangle counting

Triangles are the most basic, trivial sub-graph. A triangle can be defined as a set of three mutually adjacent vertices in a graph. As shown in Figure 1, the graph \mathbf{G} contains two triangles comprising of nodes $\{a,b,c\}$ and $\{b,c,d\}$. The number of triangles in a graph is an important metric used in applications such social network mining, link classification and recommendation, cyber security, functional biology and spam detection [20].

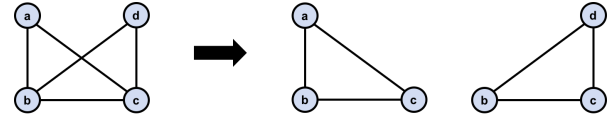


Fig. 1: The graph shown in this example contains two triangles consisting of nodes $\{a,b,c\}$ and $\{b,c,d\}$.

The number of triangles in a given graph \mathbf{G} can be calculated in several ways. We highlight two algorithms based on linear algebra primitives. The first algorithm proposed by Wolf, et. al. [13] uses an overloaded matrix multiplication approach on the adjacency and incidence matrices of the graph and is shown in Listing 1. The second approach proposed by Burkhardt, et. al. [21] uses only the adjacency matrix of the given graph and is shown in Listing 2.

Another algorithm for triangle counting based on a masked matrix multiplication approach has been proposed by Azad et al [22]. The serial version of this algorithm based on the MapReduce implementation by Cohen et al [23] is shown in Listing 3. Finally, a comparison of triangle counting algorithms based on subgraph matching, programmable graph analytics and a matrix formulation based on sparse matrix multiplication can be found in [24].

The number of triangles in a given graph \mathbf{G} can be calculated in several ways. We highlight two algorithms based

Algorithm 1: Array based implementation of triangle counting algorithm using the adjacency and incidence matrix of a graph [13].

Data: Adjacency matrix \mathbf{A} and incidence matrix \mathbf{E}
Result: Number of triangles in graph \mathbf{G}
 initialization;

$$\mathbf{C} = \mathbf{A}\mathbf{E}$$

$$n_T = nnz(\mathbf{C})/3$$

Multiplication is overloaded such that

$$\mathbf{C}(i, j) = \{i, x, y\} \text{ iff}$$

$$\mathbf{A}(i, x) = \mathbf{A}(i, y) = 1 \text{ \& } \mathbf{E}(x, j) = \mathbf{E}(y, j) = 1$$

Algorithm 2: Array based implementation of triangle counting algorithm using only the adjacency matrix of a graph [21].

Data: Adjacency matrix \mathbf{A}
Result: Number of triangles in graph \mathbf{G}
 initialization;

$$\mathbf{C} = \mathbf{A}^2 \circ \mathbf{A}$$

$$n_T = \sum_{ij} (\mathbf{C})/6$$

Here, \circ denotes element-wise multiplication

on linear algebra primitives. The first algorithm proposed by Wolf, et. al. [13] uses an overloaded matrix multiplication approach on the adjacency and incidence matrices of the graph and is shown in Algorithm 1. The second approach proposed by Burkhardt, et. al. [21] uses only the adjacency matrix of the given graph and is shown in Algorithm 2.

Another algorithm for triangle counting based on a masked matrix multiplication approach has been proposed by Azad, et. al. [22]. The serial version of this algorithm based on the MapReduce implementation by Cohen, et. al. [23] is shown in Algorithm 3. Finally, a comparison of triangle counting algorithms based on subgraph matching, programmable graph analytics and a matrix formulation based on sparse matrix multiplication can be found in [24].

Algorithm 3: Serial version of triangle counting algorithm based on MapReduce version by Cohen, et. al. [23] and [22].

Data: Adjacency matrix \mathbf{A}
Result: Number of triangles in graph \mathbf{G}
 initialization;

$$(\mathbf{L}, \mathbf{U}) \leftarrow \mathbf{A}$$

$$\mathbf{B} = \mathbf{L}\mathbf{U}$$

$$\mathbf{C} = \mathbf{A} \circ \mathbf{B}$$

$$n_T = \sum_{ij} (\mathbf{C})/2$$

Here, \circ denotes element-wise multiplication

B. k-Truss

Given a graph \mathbf{G} , a k -truss is a subgraph such that each edge is contained in at least $(k - 2)$ triangles in the same subgraph [25], [26]. Figure 2 shows graph \mathbf{G} and the 3-truss of this graph represented by the graph \mathbf{H} . As seen in this figure, each edge of \mathbf{H} is part of only one triangle. The edge labelled 1 violates the k -truss condition and is removed.

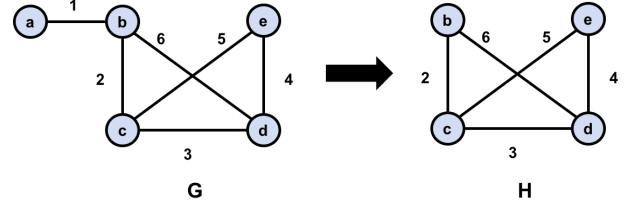


Fig. 2: Graph \mathbf{G} and its 3-truss: Each edge in Graph \mathbf{H} is part of at-least one triangle, where $k = 3$.

Algorithm 4: Array based implementation of k -Truss algorithm.

Data: Unoriented incidence matrix \mathbf{E} and integer k
Result: Incidence matrix of k -truss subgraph E_k
 initialization;

$$d = \text{sum}(\mathbf{E})$$

$$\mathbf{A} = \mathbf{E}^T \mathbf{E} - \text{diag}(d)$$

$\mathbf{R} = \mathbf{E}\mathbf{A}$

$$s = (\mathbf{R} == 2)\mathbf{1}$$

$$x = \text{find}(s < k - 2)$$

while x is not empty **do**

$$\mathbf{E}_x = \mathbf{E}(x, :)$$

$$\mathbf{E} = \mathbf{E}(x_c, :)$$

$$d_x = \text{sum}(\mathbf{E}_x)$$

$$\mathbf{R} = \mathbf{R}(x_c, :)$$

$$\mathbf{R} = \mathbf{R} - \mathbf{E}[\mathbf{E}_x^T \mathbf{E}_x - \text{diag}(d_x)]$$

$$s = (\mathbf{R} == 2)\mathbf{1}$$

$$x = \text{find}(s < k - 2)$$

end

Computing the truss decomposition of a graph involves finding the maximal k -truss for all $k \geq 2$ [27] and is summarized in Algorithm 4. Let \mathbf{E} be the unoriented incidence matrix and \mathbf{A} be the adjacency matrix of graph \mathbf{G} . Each row of \mathbf{E} has a 1 in the column of each associated vertex. To get the support for this edge, we need the overlap of the neighborhoods of these vertices. If the rows of the adjacency matrix \mathbf{A} associated with the two vertices are summed, this corresponds to the entries that are equal to 2. Summing these rows is equivalent to multiplying \mathbf{A} on the left by the edges row in \mathbf{E} . Therefore, to get the support for each edge, we can compute $\mathbf{E}\mathbf{A}$, apply to each entry a function that maps 2 to 1 and all other values to 0, and sum each row of the resulting matrix. Note also that

$$\mathbf{A} = \mathbf{E}^T \mathbf{E} - \text{diag}(\mathbf{E}^T \mathbf{E})$$

which allows us to recompute **EA** after edge removal without performing the full matrix multiplication. Pseudocode for the array based implementation of the k-truss algorithm is shown in Algorithm 4. Within the pseudocode, x_c refers to the complement of x in the set of row indices. Semantics of the MATLAB functions `find`, `sum` and `diag` can be found in the MATLAB documentation [28]. This algorithm can return the full truss decomposition by computing the truss with $k = 3$ on the full graph, then passing the resulting incidence matrix to the algorithm with an incremented k . This procedure will continue until the resulting incidence matrix is empty.

IV. COMPUTATIONAL METRICS

Submissions to the static graph isomorphism challenge will be evaluated on the basis of two metrics: Correctness and Performance.

A. Correctness

For the triangle counting kernel, correctness is evaluated by comparing the reported triangle count with the ground truth since the number of triangles for a given graph is exact. Similarly, for the k-truss kernel, correctness is based on enumerating all k-trusses for a given graph and comparing with the exact enumeration for said graph.

B. Performance

The performance of the algorithm implementation should be reported in terms of the following metrics:

- Total number of edges in the given graph: This measures the amount of data processed
- Execution time: Total time required to count the triangles or compute the k-truss of the given graph. Time required for reading graph data from a file is not included in this time.
- Rate: Measures the throughput of the implementation as the ratio of the number of edges in the graph to the execution time.
- Energy: Total amount of energy consumption in watts for the computation.
- Rate per energy (edges/second/Watt): Measures the throughput achieved per unit of energy consumed.
- Memory: Specifies the amount of memory required for the computation.
- Processor: Number and type of processors used in the computation.

C. Preliminary Benchmarking Results

Serial implementations of the Graph Challenge benchmarks were developed and tested on the MIT SuperCloud and the Lincoln Laboratory TX-Green supercomputer. Serial versions of the two graph benchmarks (triangle counting and k-truss) were implemented in MATLAB, python and Julia. Both benchmarks described in the earlier section were tested on Intel Xeon E5-2683 based compute nodes with 256 GB of RAM. The compute nodes were scheduled for exclusive access so that the benchmark process had exclusive access to all hardware

Language	Count
MATLAB	38
Octave	38
python	55
Julia	34

(a) Triangle counting

Language	Count
MATLAB	40
Octave	40
python	110
Julia	45

(b) k-Truss

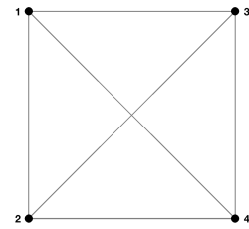
Table I: Source lines of code [29] for implementations of triangle counting and k-truss algorithms in MATLAB, GNU Octave, python and Julia.

n	Node count (M)	Edge Count	Triangles
8	65536	260610	520200
9	262144	1045506	2088968
10	1048576	4188162	8372232
11	4194304	16764930	33521672
12	16777216	67084290	134152200
13	67108864	268386306	536739848

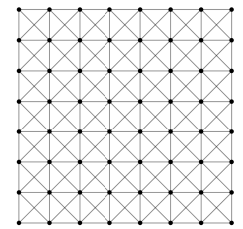
Table II: Node, edge and triangle counts for synthetic graphs used for testing initial implementations of the triangle counting and k-truss algorithms.

resources. Table I shows the source lines of code for the k-truss and triangle counting benchmarks.

1) *Benchmarking on Synthetic Graph Data*: The multi-language implementations of the triangle counting and k-truss algorithms were benchmarked on synthetic graphs. The graphs were generated as $M \times M$ images where $M = 2^n$, $n = 8, 9, 10, 11, 12, 13$. Each pixel in the image was treated as a node in the graph. A pixel was connected to its 8-neighbors by an undirected edge. Table II shows the numbers of edges, nodes and triangles for $n = 8$ to 13. Examples of two graphs generated in this manner are shown in Figure 3. These graphs were used for testing since the number of nodes, edges and triangles for a given M can be analytically calculated when M is a power of 2. Figure 4 and Figure 5 show the performance of the triangle counting and k-truss implementations in MATLAB, GNU Octave, python and Julia.



(a) Synthetic graph with 4 nodes



(b) Synthetic graph with 64 nodes

Fig. 3: Examples of synthetic graphs: Graphs are generated using $M \times M$ images with each pixel in the image being a node in the graph. Each pixel is connected to its 8 neighbors by an undirected edge. Pixels on the boundary only have 3 neighbors.

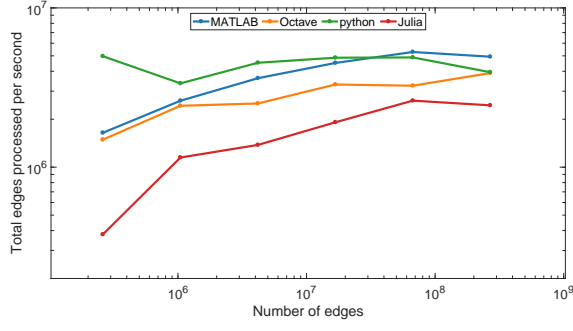


Fig. 4: Triangle-counting single core performance on synthetic data.

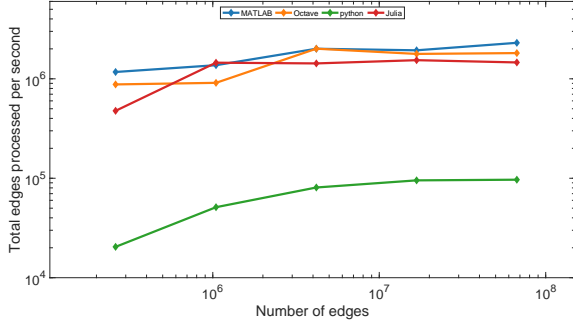


Fig. 5: k-truss single core performance on synthetic data.

2) *Benchmarking on Real World Data:* Each benchmark was run on a variety of datasets from SNAP [30]. Graphs with edge counts ranging from 25,000 to 4.6 million were used for benchmarking purposes. Figure 6 shows the performance of the triangle counting benchmark as the ratio of the number of edges in the graph to the total compute time for MATLAB, GNU Octave, python and Julia. Similarly, Figure 7 shows the performance of the k-truss implementation in the same languages for $k = 3$. The datasets shown in the figures are listed in Table III.

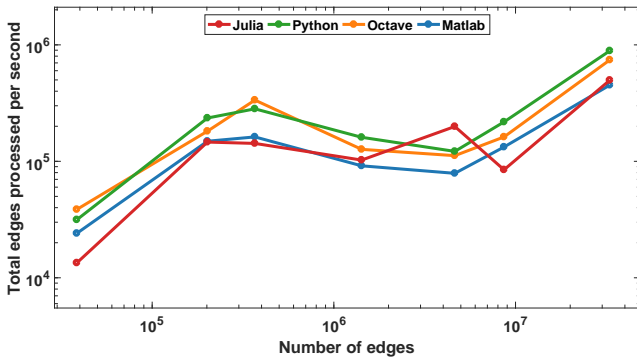


Fig. 6: Triangle counting single-core performance for a subset of SNAP datasets listed in Table III. The average run time of 100 runs for each dataset is shown.

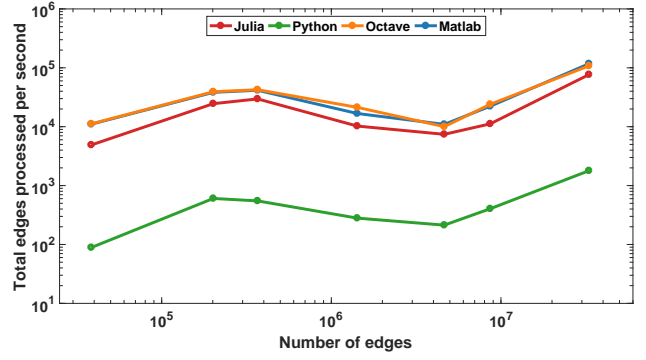


Fig. 7: k-truss single-core performance for $k=3$ on a subset of SNAP datasets listed in Table III. The average run time of 100 runs for each dataset is shown.

Name	Number of Edges	Number of Triangles
cit-HepTh-dates	38488	1418
wiki-Vote	201524	608389
email-Enron	367662	727044
soc-sign-epinions	1422420	4910076
flickrEdges	4633896	107987357
web-Google	8644102	13391903
cit-Patents	33037894	7515023

Table III: Sample SNAP datasets used to generate performance numbers in Figures 6 and 7

V. SUMMARY

The rise of graph analytic systems has created a need for ways to measure and compare the capabilities of these systems. Graph analytics present unique scalability difficulties. The machine learning, high performance computing, and visual analytics communities have wrestled with these difficulties for decades and have developed methodologies for creating challenges to move these communities forward. The proposed Subgraph Isomorphism Graph Challenge draws upon prior challenges from machine learning, high performance computing, and visual analytics to create a graph challenge that is reflective of many real-world graph analytics processing systems. The Subgraph Isomorphism Graph Challenge is a mathematically well defined specification and can be implemented in any programming environment. Subgraph isomorphism is amenable to both vertex-centric implementations and array-based implementations (e.g., using the GraphBLAS.org standard). The computations are simple enough that performance predictions can be made based on simple computing hardware models. Furthermore, since the proposed graph challenge is scalable in both problem size and hardware, it can be used to measure and quantitatively compare a wide range of present day and future systems. Serial implementations in Python, Python with Pandas, Matlab, Octave, and Julia have been implemented and their single threaded performance have been measured. Specifications, data, and software are publicly available at GraphChallenge.org.

ACKNOWLEDGMENTS

The authors would like to thank Trung Tran, Tom Salter, David Bader, Jon Berry, Paul Burkhardt, Justin Brukardt, Chris Clarke, Kris Cook, John Feo, Peter Kogge, Chris Long, Jure Leskovec, Richard Murphy, Steve Pritchard, Michael Wolfe, Michael Wright, and the entire GraphBLAS.org community for their support and helpful suggestions.

REFERENCES

- [1] DARPA. Hierarchical Identify Verify Exploit. <https://www.fbo.gov/index?s=opportunity&mode=form&id=e3d5ebb6da9795cd0697cb24293f9302>, 2017. [Online; accessed 01-January-2017].
- [2] Edward Kao, Vijay Gadepally, Michael Hurley, Michael Jones, Jeremy Kepner, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Siddharth Samsi, William Song, Diane Staheli, and Steven Smith. Streaming Graph Challenge - Stochastic Block Partition. Sept 2017.
- [3] J. P. Campbell. Testing with the yoho cd-rom voice verification corpus. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 341–344 vol.1, May 1995.
- [4] C. Cortes Y. LeCun and C. J.C. Burges. The MNIST Database. <http://www.hpcchallenge.org>, 2017. [Online; accessed 01-January-2017].
- [5] HPC Challenge. <http://yann.lecun.com/exdb/mnist/>, 2017. [Online; accessed 01-January-2017].
- [6] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.
- [7] Kristin A. Cook, Georges Grinstein, and Mark A. Whiting. The VAST Challenge: History, Scope, and Outcomes: An introduction to the Special Issue. *Information Visualization*, 13(4):301–312, Oct 2014.
- [8] Jean Scholtz, Mark A. Whiting, Catherine Plaisant, and Georges Grinstein. A Reflection on Seven Years of the VAST Challenge. In *Proceedings of the 2012 BELIV Workshop: Beyond Time and Errors - Novel Evaluation Methods for Visualization*, BELIV '12, pages 13:1–13:8. ACM, 2012.
- [9] D Bader, Kamesh Madduri, John Gilbert, Viral Shah, Jeremy Kepner, Theresa Meuse, and Ashok Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, 2:1–10, 2006.
- [10] Juri Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 133–145. Springer, 2005.
- [11] Patrick Dreher, Chansup Byun, Chris Hill, Vijay Gadepally, Bradley Kuszmaul, and Jeremy Kepner. Pagerank pipeline benchmark: Proposal for a holistic system benchmark for big-data platforms. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 929–937. IEEE, 2016.
- [12] Mauro Bisson, Everett Phillips, and Massimiliano Fatica. A cuda implementation of the pagerank pipeline benchmark. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–7. IEEE, 2016.
- [13] M. M. Wolf, J. W. Berry, and D. T. Stark. A task-based linear algebra building blocks approach for scalable graph analytics. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2015.
- [14] M. M. Wolf, H. C. Edwards, and S. L. Olivier. Kokkos/qthreads task-parallel approach to linear algebra based graph analytics. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2016.
- [15] Comandur Seshadhri, Tamara G Kolda, and Ali Pinar. Community structure and scale-free collections of erdős-rényi graphs. *Physical Review E*, 85(5):056109, 2012.
- [16] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.
- [17] Jeremy Kepner. Perfect power law graphs: Generation, sampling, construction and fitting. In *SIAM Annual Meeting*, 2012.
- [18] Vijay Gadepally and Jeremy Kepner. Using a power law distribution to describe big data. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–5. IEEE, 2015.
- [19] Ronald F. Boisvert, Roldan Pozo, and Karin A Remington. The matrix market exchange formats: Initial design. *National Institute of Standards and Technology Internal Report, NISTIR*, 5935, 1996.
- [20] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proc. VLDB Endow.*, 6(14):1870–1881, September 2013.
- [21] Paul Burkhardt. Graphing trillions of triangles. *Information Visualization*, 0(0):1473871616666393, 2016.
- [22] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPSW '15*, pages 804–811, Washington, DC, USA, 2015. IEEE Computer Society.
- [23] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engg.*, 11(4):29–41, July 2009.
- [24] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing, HPGP '16*, pages 1–8, New York, NY, USA, 2016. ACM.
- [25] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, page 16, 2008.
- [26] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, May 2012.
- [27] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner. Graphulo: Linear algebra graph kernels for nosql databases. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 822–830, May 2015.
- [28] MathWorks Inc. MATLAB Documentation. <https://www.mathworks.com/help>, 2017. [Online; accessed August-2017].
- [29] Brad Appleton. sclc – Source-code line counter. <http://www.bradapp.com/clearperl/sclc.html>, 2003. [Online; accessed March-2017].
- [30] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.