Graph Processing: Main Concepts and Systems

Mikhail Chernoskutov
Krasovskii Institute of Mathematics and Mechanics,
Ural Federal University
Yekaterinburg, Russia
mach@imm.uran.ru

Abstract

The development of new processing and storage facilities leads to an increase in the size of the problems to be solved. One of such problems is graph processing, which characterized by their non-determinism and an irregular memory access pattern, that greatly complicates its development and debugging. In this paper, we describe the basic concepts that make it possible to simplify the development of graph algorithms and its porting to various computer architectures. In addition, a description of the most well-known graph processing systems that implement these concepts is given.

Keywords: high performance computing, graph algorithms, graph processing systems, parallel processing

1 Introduction

A graph is a mathematical abstraction for the representation of objects and the connections between them. The graph $G = (V, E)$ consists of the set of vertices V with the number of elements n and the set of edges E with the number of elements m. Using the graphs, various objects of the real world can be represented. For instance, in the study of social networks it is convenient to represent individual users as vertices, and links between them as edges [1, 2]. Graphs are actively used in the processing of natural languages for the word sense induction problem [3]. We can also model the work of the brain, denoting individual neurons (or brain areas) as vertices, and, therefore, the structural or functional connections between vertices are represented as edges [4, 5]. Another typical use case of graphs is the modeling of road networks, where intersections are vertices, and roads are edges [6]. Graphs are also used in bioinformatics, computer security, data analysis and other fields of science, industry and business.

The huge volume of accumulated data led to the growth of graphs that need to be investigated. This led to the emergence of the network science, dealing with the study of the features of the interaction between many objects. The number of algorithms necessary for domain scientists is also growing rapidly. As a result, various specialized tools for graph analysis appear on the market.

Development and debugging of graph algorithms is a non-trivial task because of the non-determinism of many such algorithms. The use of parallel computations as a tool for accelerating computation further complicates this task. Therefore, there are many different methods and technologies that make it possible to simplify the development of graph algorithms and simplify the work of various domain scientists. Below are the main concepts described in the paper:

- Shift to parallel processing in order to speedup processing of big graphs (allows to faster process graphs with big size);
- New data structures for storing graphs, allowing faster and more convenient graph mutation and navigation through the graph (allows to operate with graphs in more efficient and convenient way);
- Development of new models for describing graph algorithms, which makes it possible to effectively use the parallelism embedded in the algorithms and effectively map it to modern computing architectures (allows to construct architecture-independent algorithms with more productivity for the developer).

In this paper, we describe the above technologies

In: V. Voevodin, A. Simonov (eds.): Proceedings of the GraphHPC-2017 Conference, Moscow State University, Russia, 02-03-2017, published at http://ceur-ws.org.

used in various existing graph processing systems. The work is structured as follows. The Section 2 provides a description of the various types of parallelism that are used to speed up the processing of graphs. The Section 3 describes the data structures that are used to store graphs, together with its strengths and weaknesses. Section 4 is devoted to the description of various models of processing graphs. We concludes with final remarks in Section 5.

2 Shift to Parallel Processing

2.1 Serial Processing

Sequential execution of graph algorithms is still very common. Sequential processing can be used for prototyping of new algorithms and can be one of the steps to building highly efficient parallel algorithms that allow to process graphs with billions of vertices.

Up to now, graph processing systems based on serial execution of algorithms, such as NetworkX [7], Gephi [8] and igraph [9], are still very popular and ubiquitous. The popularity of these systems is determined by a large set of algorithms and a lot of tools for visualizing the output of the graph algorithms. These systems are actively used in the complex networks analysis.

2.2 Synchronous Parallel Processing

Sequential algorithms are not always suitable for solving real-world problems. With the rapid growth of the data size, the size of the processed graphs is also growing. Obvious way to accelerate graph processing (in addition to developing more efficient algorithms) is the parallelization of existing algorithms on graphs.

The most common parallel processing model for graph computations is the bulk synchronous model (BSP) [10]. Another name for algorithms that work in accordance with the BSP model is level-synchronized algorithms. Within the BSP model, the $N + 1$ iteration will be executed only after the N iteration will be completed. This model can be implemented by using two nested `for` loops – one for all vertices, and the other one for all adjacent edges of each vertex.

Drawback of the concept of synchronous-parallel processing connected with overheads arising in some algorithms because of the processing of inactive (not affecting the algorithm output) vertices at each iteration of the algorithm.

Among the graph processing systems based on the BSP model, the Boost Graph Library (BGL) [11], as well as its parallel implementation called Parallel BGL [12], are the most widely used. These systems implement the concept of adjacency lists (with ability to modify the graph topology) and have quite a lot of supported algorithms. Parallel BGL is capable of processing graphs consisting of billions of vertices and allows developer to parallelize calculations on hundreds of computational processes by using MPI [13].

2.3 Asynchronous Parallel Processing

Asynchronous parallel processing of graphs has not yet become as widespread and ubiquitous as the BSP model, but in some cases it allows to execute some algorithms (such as belief propagation [14]) more efficiently due to it faster convergence.

Asynchronous execution assumes an independent operation of each parallel thread or process from all other threads or processes. At the same time, when working with shared memory, such undesirable effects as data race can arise. This forces developers to use duplicate copies of graph elements, as well as use different synchronization primitives.

GraphChi [15] is one of the systems that provide the user with parallel asynchronous processing of graphs. Asynchronous processing is achieved by using the concept of Parallel Sliding Windows, which involves extracting and updating the state of a small number of vertices and edges that are simultaneously extracted from the PC's memory (SSD or hard drive). A similar concept is implemented in the CuSha system [16] and allows to achieve asynchronous processing of the graphs using GPGPU accelerators.

3 Various Graph Formats

3.1 Adjacency and Incidence Matrices

The matrix is a natural mathematical representation of the graph. The graph $G = (V, E)$ consisting of n vertices in this case can be described by the matrix $n \times n$ of the form $\mathbf{A} = (a_{ij})$, where:

$$a_{ij} = \begin{cases} 1, & e_{ij} \in E \\ 0, & e_{ij} \notin E \end{cases} \tag{1}$$

An example of an adjacency matrix of a directed graph is shown in Figure 1.

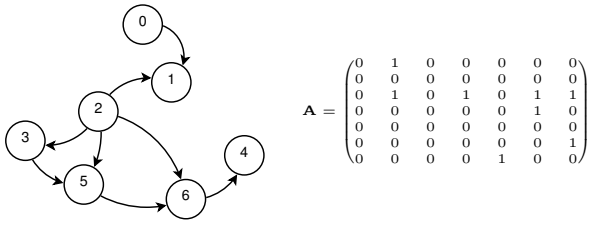

Figure 1: Representation of the directed graph (left) in the form of an adjacency matrix (right)

If the graph is non-directed, then the corresponding adjacency matrix is symmetric; $a_{ij} = a_{ji}$. If the graph

is weighted, then the elements of the adjacency matrix take the following form:

$$a_{ij} = \begin{cases} w_{ij}, & e_{ij} \in E \\ 0, & e_{ij} \notin E \end{cases} \quad (2)$$

where w_{ij} is weight of the edge e_{ij}. Another way to represent weighted graph is to add supplementary weight matrix to adjacency matrix (with a restriction on zero weight edges).

Another way of describing the graph is the incidence matrix. The incidence matrix contains data about all edges in the rows, and the columns are used to represent the vertices. Matrix has the form $B = b_{ij}$ and size $m \times n$, where m is the number of edges in the graph. Due to the fact that in this case each row must simultaneously contain data on both the beginning and the end of corresponding edge, it is most convenient to represent graph with two matrices: one serves to store the data about the beginning of the edges, and the other – about the ends (it works only for directed graphs, but edges of undirected graphs can be represented as couples of directed edges with opposite directions). The incidence matrix of the graph depicted in Figure 1 is shown in Figure 2.

$$\mathbf{B}_{\text{from}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{B}_{to} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 2: The incidence matrix for the beginnings (left) and the ends (right) of the edges

Data structures represented by adjacency and incidence matrices allows developer to quickly find the necessary elements in the graph, as well as add new elements to the graph. In the case of an adjacency matrix, the addition or removal of new edges to the graph can be performed by changing the values of the necessary matrix elements from 0 to 1 and vice versa. The incidence matrix makes it possible to deal not only with general graphs, but also with multigraphs (having parallel edges), and also with hypergraphs (having edges incident to any number of vertices). This improvement is achieved by describing each edge with two separate lines in different matrices: one line in first matrix for the beginning of hyperedge and one line in second matrix for the end of hyperedge.

However, these graph storage formats have one significant drawback – the amount of memory required to store matrices is proportional to $n \times n$ for adjacency and to $n \times m$ for incident matrix [17]. For instance, an integer adjacency matrix for a graph of 1 000 000 vertices will occupy more than 3.7 TB of memory. Thus, the use of matrices "as-is" to handle large graphs is unacceptable for many real-world applications.

3.2 Compressed Lists

Another one popular graph storage format is adjacency lists. The adjacency lists allow the graph to be stored in the form of few linear arrays.

The adjacency lists allow storing only existing graph elements (non-zero elements in the adjacency matrix), which allows to achieve a linear dependence between the amount of consumed memory and the number of vertices and edges in the graph. The Compressed Sparse Rows (CSR) format, which implements the concept of contiguity lists, is one of the most popular and useful, and consists of two arrays:

- `row pointers` – contains the offsets in the rows of the corresponding adjacency matrix;
- `column ids` – contains data about the end of each edge.

From i to $(i + 1)$ element of the `row pointers` array there are the ranges of vertex numbers in the array `column ids`, which contains outgoing edges incident to i vertex. Below is a representation of the above arrays for the graph from Figure 1:
`row pointers = [0, 1, 1, 5, 6, 6, 7, 8]`
`column ids = [1, 1, 3, 5, 6, 5, 6, 4]` For instance, vertex 2 has 4 edges (`row pointers[3] - row pointers[2] = 5 - 1 = 4`). Edges can be reconstructed by looking at `column ids` array from position `column ids[1]` to `column ids[5]` (not included). It stores numbers of vertices, that incident to opposite ends of edges outgoing from each vertex in the graph. In the example above, vertex 2 has outgoing edges to vertices 1, 3, 5, 6.

The traversal of the vertices and edges of the graph in this case is performed with two nested loops: the outer loop reads offsets from the `row pointers` array to obtain information about the number of edges outgoing from each vertex, the inner loop reads `column ids` array for following processing neighbors of each vertex in the graph.

The main advantage of the CSR format is the compactness of the graph representation, which makes it convenient for processing large graphs. For instance, the CSR format is actively used in the Knowledge Discovery Toolbox (KDT) [18] and GraphCT [19] systems to store and process large graphs obtained from analysis of biological data and social networks. In the KDT system, the CSR format is used for compact storage the incidence matrices of the graphs. In the GraphCT system, on the other hand, CSR is used to store adjacency matrices.

However, the CSR format is only suitable for processing static graphs. Processing of dynamic graphs

with the CSR format is extremely inefficient, and requires a complete rebuild of the entire data structure with every graph mutation. In addition, the parallel processing of graphs with skewed degree distribution is burdened with additional overhead costs in the form of computational workload imbalance amongst computational processes or threads [20, 21].

3.3 Custom Formats

Typically, custom formats designed to simplify development of some complex graph algorithms. For instance, some maximum flow algorithms (Edmonds-Carp algorithm [22] and push-relabel algorithm [23]), uses a special "residual conductance" network, which is repeatedly rebuilt during the execution of the algorithm. Another example is the Girvan-Newman community detection algorithm [24]. The main idea of the algorithm is to isolate communities by removing inter-community edges from the graph.

As seen, the above algorithms use not only the traversal procedure through all vertices and edges (which can be efficiently performed using the CSR format), but also the procedures of searching for individual graph elements, as well as modifying the graph topology. The detailed list of most commonly used operations is presented in Table 1. Table 2 shows memory complexity for various operations.

Table 1: Time complexity analysis of graph storing formats

Procedure	Matrix	CSR
Vertex addition	$O(N)$	$O(1)$
Vertex deletion	$O(N)$	$O(N^2)$
Edge addition	$O(1)$	$O(N^2)$
Edge deletion	$O(1)$	$O(N^2)$
Check whether vertex in graph	$O(1)$	$O(1)$
Get weight of (random) edge	$O(1)$	$O(N)$
Get list of ingoing neighbors of vertex	$O(N)$	$O(N^2)$
Get list of outgoing neighbors of vertex	$O(N)$	$O(N)$

Table 2: Space complexity analysis of graph storing formats

Procedure	Matrix	CSR
Vertex addition	$O(N)$	$O(1)$
Vertex deletion	$O(N)$	$O(1)$
Edge addition	$O(1)$	$O(1)$
Edge deletion	$O(1)$	$O(1)$

As can be seen from Tables 1 and 2, the use of each of the formats is a trade-off between processing efficiency and memory consumption. Using CSR allows developer to store graphs with a lot of vertices and edges (light weight operations of vertex and eage addition and deletion), but badly suited for graph topology modification (see vertex and edge deletion operations) and navigation through the graph (it is hard to obtain list of ingoing edges to some random vertex). Opposite, storing graph as matrix allows developer to carry out fast graph topology modifications (with complexity $O(1)$ and $O(N)$) as well as navigating through the graph (checking random vertex and edge state with complexity $O(1)$), but matrix data structure suffers from non-linear memory complexity (need $O(N^2)$ space for graph with N vertices).

At the moment, there is no standard format for storing graphs, which would allow to efficiently modify the graph, access its individual elements in linear time, and store only significant elements of the graph, and allow parallel processing at the same time. However, a number attempts are being made to develop such a format. For instance, the extended storage functionality and parallel processing of dynamic graphs is provided by the data structure called STINGER [25, 26]. This data structure is based on linked lists consisting of blocks of graph elements. Each block is a data storage for an individual subset of vertices or edges. Each block contains special metadata about the elements stored inside it (for example, the minimum and maximum number of the vertex inside the block). By separating the entire array of vertices or edges into subsets, and using metadata, the STINGER data structure allows to process dynamically changing graphs in parallel. Another example of custom graph format is Resident Distributed Graphs (RDG) from GraphX [27], that uses vertex-cut partitioning scheme to minimize data transfers during graph computations by evenly assigning edges to computational nodes. Also, RDG allows to efficiently view, filter and transform the graph by operating on three tables (implemented as Spark Resilient Distributed Datasets [28]) storing the RDG: `EdgeTable` (adjacency structure and edge data), `VertexDataTable` (vertex data) and `VertexMap` (mapping from the id of a vertex to the ids of the virtual partitions that contain adjacent edges).

4 Graph Processing Models

4.1 Vertex-Centric Paradigm

According to this paradigm, graph algorithms are executed in "think like a vertex" model. Thus, each individual vertex is considered as a separate computational element, dealing with the data available to it (for instance, internal variables) and having the ability to execute various computational algorithms (provided by developer). A vertex can receive messages from neighboring vertices along its incoming edges, and also send

messages to other vertices along the outgoing edges.

The virtue of the vertex-centric model is the ability to naturally parallelize the computations. In this case, the processing of each vertex (or, possibly, the vertex group) will be assigned to a separate computational process or thread. The implementation of these principles makes it possible to achieve high scalability of computing.

However, such a model proves to be convenient not for all algorithms. For example, the page-rank algorithm [29] or the label-propagation algorithm [30] naturally fits in this model, in contrast to, for example, complex algorithms for spectral graph analysis.

The most famous (and the very first at the same moment) implementation of this paradigm is Pregel [31], developed by Google. The computations in this system consist of several steps. In the first step, the graph is initialized and the initial values is assigned to all vertices. After initialization, a series of supersteps separated by synchronization steps are executed (the Pregel computation model follows the BSP [10] paradigm). At each step, every vertex executes its program, specified by the developer, using the data available to it, as well as data obtained from other vertices (by incoming edges). After finishing the calculations, the vertex can send data to its neighbors (by outgoing edges). One important property of Pregel is that only "active" nodes can compute and send data. Vertex can become "active" when received messages from other vertices (the vertex state machine is shown in Figure 3).

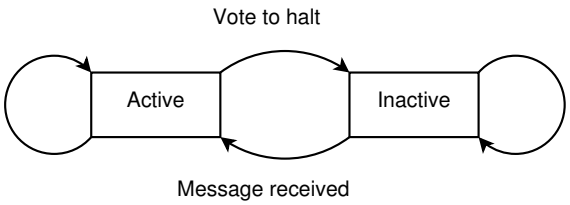

Figure 3: Pregel vertex state machine

4.2 Domain-Specific Languages

Domain-Specific Languages (DSL) is a special programming language intended for applications in a specific domain area. DSL, as a rule, contains domain-specific expressions and constructs. The program developed using the DSL is translated by the compiler into the target language (for example, C/C++, CUDA, etc.).

Advantages of using DSL to develop algorithms on graphs is increasing the productivity of development. DSL allows to express algorithmic ideas using terms of the domain area, which usually requires significantly less programming code compared to similar programs developed with low-level languages. Using DSL in conjunction with compilers that support various hardware architectures allows developer not to worry about porting the program code to more efficient and high-performance platforms. Finally, DSL allows to use highly optimized (usually parallelized) procedures that allow you to efficiently process large graphs (for example, the procedure for simultaneously updating the state of all vertices in a graph, etc.).

The drawbacks of DSL include the inability to use the programs written with it together with other code (written in C++, for example). It is allowed to use programs translated from DSL as separate ready-made computational modules, but this option may not always be convenient for the developers and domain scientists.

One of the most popular DSL for development of high-performance parallel graph algorithms is Green-Marl [32,33]. Green-Marl allows to calculate the scalar invariant for each element in the graph, as well as the property of each element in the graph (for instance, the centrality metric [34]) or extract a specific subgraph from the existing graph. Using this DSL, the developer has the ability to operate the apparatus of graph theory, while receiving, after compilation, a well-parallelized code. For example, a parallel search of all vertices in the graph happens according its breadth-first order (special traversal operations `InBFS` and `InRBFS` are used for this purpose). Also, Green-Marl has special containers for storing sets of vertices: Set, Order and Sequence, which differs in various types of access to the elements and are designed for parallel and/or sequential access to its elements. The use of such containers greatly simplifies the development of some algorithms (for example, the Dijkstra algorithm [35] or Δ-stepping [36]). An example of an implementation of the algorithm using Green-Marl is shown in Figure 4.

4.3 Parallel Processing Primitives

Parallel processing primitives serves as a small "building blocks" of more complex graph algorithms. Thus, different algorithms may consist of combinations of the same set of primitives.

The development of complex algorithms on graphs is a big problem, coupled with a number of difficulties. Debugging is one such challenge. For example, the correctness of the execution of some community detection algorithms [30] (especially when working with real data) is hard to control, because there is no clear criteria for the correctness of the detection of communities (except some theoretical metrics like modularity, that may not always be suitable for real-world applications). This complexity is further intensified

```
Procedure Compute_BC(G: Graph,
BC: Node_Prop<Float>(G)) {
  G.BC=0; // initialize BC
  Foreach(s: G.Nodes) {
    // define temporary properties
    Node_Prop<Float>(G) Sigma;
    Node_Prop<Float>(G) Delta;
    // Init. Sigma for root
    s.Sigma = 1;
    // Traverse graph
    // in BFS-order from s
    InBFS(v: G.Nodes From s)(v!=s) {
      // sum over BFS-parents
      v.Sigma=Sum(w: v.UpNbrs){w.Sigma};
    }
    // Traverse graph
    // in reverse BFS-order
    InRBFS(v!=s) {
      // sum over BFS-children
      v.Delta = Sum (w:v.DownNbrs) {
      v.Sigma / w.Sigma * (1+ w.Delta)
      };
    v.BC += v.Delta @s; //accumulate BC
} } }
```

Figure 4: Betweenness centrality algorithm described in Green-Marl

when it comes to parallel processing of graphs with a large number of vertices and edges. When using special-purpose graph processing primitives, the developer only needs to be sure of the correctness of each primitive operation separately. It is necessary to control only the high-level description of the algorithm when using primitives.

Another difficulty is porting the code to various parallel architectures. Writing efficient code for computational accelerators (GPGPU or MIC) or moving from a shared memory model to a distributed memory usually requires a lot of efforts. Considering the fact that every 3 to 5 years there are significant changes in architectures and programming models, the work of porting a large number of different algorithms can continue "infinitely". The use of primitives implemented for different architectures makes it possible to abstract from this problem and implement only primitives themselves for each architecture.

The advantage of using processing primitives against using DSL is the ability to use primitives along with other code in the program. Using DSL, on the contrary, forces the developer to work only in a special development environment.

Primitives are used in a variety of graph processing systems. However, there is no single standard for the use of primitives for parallel processing of graphs at the moment. Therefore, in many systems the set of these primitives varies, as well as the principles by which they are built. Some implementations of such primitives will be described below.

The PowerGraph [37] system is based on the Gather-Apply-Scatter (GAS) model and includes three types of primitives:

- Gather – each vertex v collects data about its adjacent vertices and edges;
- Apply – the value of each vertex v is updated (taking into account previously collected data in the Gather phase);
- Scatter – The new value of the vertex v distributes to adjacent vertices.

By combining the above primitives, the developer has the opportunity to create various set of graph algorithms.

The GAS model is not the only possible way of expressing "building blocks" of the complex graph algorithms. Another well-known example is the Advance-Filter-Compute model introduced in the Gunrock – system for developing algorithms on graphs using GPGPU accelerators [38]. This model is based on modification of the vertex frontiers during execution of each iteration of the algorithm and includes the following primitives:

- Advance – obtaining a new vertex frontier by passing along adjacent edges from the current vertex frontier;
- Filter – obtaining a new vertex frontier by selecting some subset of vertices from the current vertex frontier;
- Compute – obtaining a new vertex frontier by executing a procedure defined by the developer, that applied to the current vertex frontier.

Primitives are also used in other parallel graph processing systems, such as MapGraph [39], HelP [40], GraphPad [41], GraphBLAS [42], etc.

5 Conclusion

This paper gives an overview of the main concepts (shift to parallel computing, using complex data structures and new computational models) that are used in modern graph processing systems. Using these concepts allows:

- Simplify development of novel graph algorithms;
- Use different computer architectures, without changing the program code;
- Speedup the execution of complex graph algorithms.

Acknowledgment

The research was supported by the RFBR under the project no. 16-37-00203 mol_a and the Ministry of Education and Science of the Russian Federation Agreement no. 02.A03.21.0006.

References

[1] E. Otte and R. Rousseau, "Social network analysis: a powerful strategy, also for the information sciences," *Journal of Information Science*, vol. 28, no. 6, pp. 441–453, 2002.

[2] M. Grandjean, "A social network analysis of twitter: Mapping the digital humanities community," *Cogent Arts & Humanities*, vol. 3, no. 1, p. 1171458, 2016.

[3] D. Ustalov, A. Panchenko, and C. Biemann, "Watset: Automatic Induction of Synsets from a Graph of Synonyms," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Vancouver, Canada), pp. 1579–1590, Association for Computational Linguistics, 2017. http://aclweb.org/anthology/P17-1145 (accessed: 20.10.2017).

[4] E. Bullmore and O. Sporns, "Complex brain networks: Graph theoretical analysis of structural and functional systems," vol. 10, pp. 186–98, 03 2009. https://www.nature.com/nrn/journal/v10/n3/pdf/nrn2575.pdf (accessed: 20.10.2017).

[5] A. Horn, D. Ostwald, M. Reisert, and F. Blankenburg, "The structural–functional connectome and the default mode network of the human brain," *NeuroImage*, vol. 102, no. Part 1, pp. 142 – 151, 2014. https://www.sciencedirect.com/science/article/pii/S1053811913010057 (accessed: 20.10.2017).

[6] M. G. Bell *et al.*, *Transportation network analysis*. Wiley Online Library, 1997.

[7] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, (Pasadena, CA USA), pp. 11–15, Aug. 2008. http://aric.hagberg.org/papers/hagberg-2008-exploring.pdf (accessed: 20.10.2017).

[8] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," 2009. http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154 (accessed: 20.10.2017).

[9] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. https://pdfs.semanticscholar.org/1d27/44b83519657f5f2610698a8ddd177ced4f5c.pdf (accessed: 20.10.2017).

[10] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, Aug. 1990. http://web.mit.edu/6.976/www/handout/valiant2.pdf (accessed: 20.10.2017).

[11] *The Boost Graph Library: User Guide and Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. https://markqiu.files.wordpress.com/2009/12/boost-graph-library.pdf (accessed: 20.10.2017).

[12] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," in *In Parallel Object-Oriented Scientific Computing (POOSC*, 2005. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.2137&rep=rep1&type=pdf (accessed: 20.10.2017).

[13] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, *MPI-2: Extending the message-passing interface*, pp. 128–135. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.2841&rep=rep1&type=pdf (accessed: 20.10.2017).

[14] G. Elidan, I. McGraw, and D. Koller, "Residual belief propagation: Informed scheduling for asynchronous message passing," in *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence*, UAI'06, (Arlington, Virginia, United States), pp. 165–173, AUAI Press, 2006. https://ai.stanford.edu/~koller/Papers/Elidan+al:UAI06.pdf (accessed: 20.10.2017).

[15] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 31–46, USENIX Association, 2012. https://www.cs.cmu.edu/~guyb/papers/KBG12.pdf (accessed: 20.10.2017).

[16] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on gpus," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, (New York, NY, USA), pp. 239–252, ACM, 2014.

[17] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. 2011.

[18] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, *A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis*, pp. 930–941. http://epubs.siam.org/doi/pdf/10.1137/1.9781611972825.80 (accessed: 20.10.2017).

[19] D. Ediger, K. Jiang, E. J. Riedy, and D. A. Bader, "Graphct: Multithreaded algorithms for massive graph analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 2220–2229, Nov 2013. https://www.cc.gatech.edu/~bader/papers/GraphCT-TPDS2013.pdf (accessed: 20.10.2017).

[20] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 117–128, ACM, 2012. http://research.nvidia.com/sites/default/files/pubs/2012-02_Scalable-GPU-Graph/ppo213s-merrill.pdf (accessed: 20.10.2017).

[21] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 378–389, May 2012.

[22] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, pp. 248–264, Apr. 1972. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.610.4784&rep=rep1&type=pdf (accessed: 20.10.2017).

[23] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, pp. 921–940, Oct. 1988. https://www.cs.princeton.edu/courses/archive/fall03/cs528/handouts/a%20new%20approach.pdf (accessed: 20.10.2017).

[24] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002. http://www.pnas.org/content/99/12/7821.full.pdf (accessed: 20.10.2017).

[25] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*, pp. 1–5, Sept 2012. http://ieee-hpec.org/2012/index_htm_files/ediger.pdf (accessed: 20.10.2017).

[26] D. A. Bader, J. Berry, A. Amos-Binks, C. Hastings, K. Madduri, and S. C. Poulos, "Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation," 2009. https://pdfs.semanticscholar.org/6992/7a3b9fc25e655ce662c03deb1e9d2832585c.pdf (accessed: 20.10.2017).

[27] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, (New York, NY, USA), pp. 2:1–2:6, ACM, 2013. http://www.istc-cc.cmu.edu/publications/papers/2013/grades-graphx_with_fonts.pdf (accessed: 20.10.2017).

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf (accessed: 20.10.2017).

[29] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Seventh International World-Wide Web Conference (WWW 1998)*, 1998. http://ilpubs.stanford.edu:8090/361/1/1998-8.pdf (accessed: 20.10.2017).

[30] N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," vol. 76, p. 036106, 10 2007. https://arxiv.org/pdf/0709.2938.pdf (accessed: 20.10.2017).

[31] 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*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010. https://kowshik.github.io/JPregel/pregel_paper.pdf (accessed: 20.10.2017).

[32] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 349–362, ACM, 2012. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.220.1796&rep=rep1&type=pdf (accessed: 20.10.2017).

[33] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, "Simplifying scalable graph processing with a domain-specific language," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 208:208–208:218, ACM, 2014. https://pdfs.semanticscholar.org/2d8b/e5e1b88ac9919984b9369f7045fbb0af0d08.pdf (accessed: 20.10.2017).

[34] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.2024&rep=rep1&type=pdf (accessed: 20.10.2017).

[35] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, pp. 269–271, Dec. 1959.

[36] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, pp. 114–152, Oct. 2003. https://ac.els-cdn.com/S0196677403000762/1-s2.0-S0196677403000762-main.pdf?_tid=a3d2287c-b5a5-11e7-b112-00000aacb362&acdnat=1508511048_d998da6a4ec35491c44e069c0b48b756 (accessed: 20.10.2017).

[37] 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*, OSDI'12, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2012. https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf (accessed: 20.10.2017).

[38] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," *SIGPLAN Not.*, vol. 51, pp. 11:1–11:12, Feb. 2016. https://arxiv.org/pdf/1501.05387.pdf (accessed: 20.10.2017).

[39] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, (New York, NY, USA), pp. 2:1–2:6, ACM, 2014. https://pdfs.semanticscholar.org/3ebf/3857a60c3e224284bbbe6c7127d0a12c546d.pdf?_ga=2.86211930.371173678.1508511132-1908779532.1478770815 (accessed: 20.10.2017).

[40] S. Salihoglu and J. Widom, "Help: High-level primitives for large-scale graph processing," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, (New York, NY, USA), pp. 3:1–3:6, ACM, 2014. http://ilpubs.stanford.edu:8090/1085/2/primitives_tr_sig_alternate.pdf (accessed: 20.10.2017).

[41] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, "Graphpad: Optimized graph primitives for parallel and distributed platforms," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 313–322, May 2016.

[42] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, Sept 2016. https://arxiv.org/pdf/1606.05790.pdf (accessed: 20.10.2017).