DOI: 10.15514/ISPRAS-2022-34(1)-7

The Software Implementation of a Metagraph Processing System Based on the Big Data Approach

V.M. Chernenkiy, ORCID: 0000-0002-7684-3114 <chernen@bmstu.ru>
I.V. Dunin, ORCID: 0000-0003-1153-4561 <johnmoony@yandex.ru>
Yu.E. Gapanyuk, ORCID: 0000-0001-9005-8174 <gapyu@bmstu.ru>

Bauman Moscow State Technical University,
Baumanskaya 2nd, 5, Moscow, 105005, Russia

Abstract. The paper discusses the approach to solving the problem of processing metagraphs using Big Data technology. The formal definition of the metagraph data model and the metagraph agent model are given. The metagraph representation using the flat graph model is discussed. The flat graph and metagraph Big Data processing are described. The architecture of the system for processing data in metagraph is discussed. The metagraph processing using metagraph agents based on Big Data technology is discussed. The experiments result for parallel metagraph processing are given.

Keywords: Big Data; Big Graph; Flat Graph; Metagraph; Pregel; Signal-Collect; Gather Sum Apply

For citation: Chernenkiy V.M., Dunin I.V., Gapanyuk Yu.E. The Software Implementation of a Metagraph Processing System Based on the Big Data Approach. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 1, 2022, pp. 87-100. DOI: 10.15514/ISPRAS–2022–34(1)–7

Acknowledgements. The article is based on the materials of the report at the Seventh International Conference «Actual Problems of System and Software Engineering» (APSSE 2021).

Программная реализация системы обработки метаграфов на основе подхода Больших Данных

В.М. Черненький, ORCID: 0000-0002-7684-3114 <chernen@bmstu.ru>
И.В. Дунин, ORCID: 0000-0003-1153-4561 <johnmoony@yandex.ru>
Ю.Е. Гапанюк, ORCID: 0000-0001-9005-8174 <gapyu@bmstu.ru>

Московский государственный технический университет им. Н.Э. Баумана,
105005, Россия, Москва, 2-ая Бауманская, 5

Аннотация. В работе обсуждается подход к решению проблем обработки метаграфов с использованием технологий больших данных. Дается формальное определение метаграфовой модели данных и метаграфового агента. Обсуждается представление метаграфа через простой граф. Описывается обработка простых графов и метаграфов. Обсуждается архитектура системы по обработке метаграфов. Обсуждается обработка с использованием метаграфовых агентов на основе технологий Больших Данных. Демонстрируются результаты экспериментов.

Ключевые слова: большие данные; большие графы; графы; метаграфы; Pregel; Signal-Collect; Gather Sum Apply

Для цитирования: Черненький В.М., Дунин И.В., Гапанюк Ю.Е. Программная реализация системы обработки метаграфов на основе подхода Больших Данных. Труды ИСП РАН, том 34, вып. 1, 2022 г., стр. 87-100. DOI: 10.15514/ISPRAS–2022–34(1)–7

Благодарности. Статья подготовлена по материалам доклада на Седьмой Международной конференции «Актуальные проблемы системной и программной инженерии» (АПСПИ 2021).

1. Introduction

Presently, Big Data frameworks are widely used to process information in various domains, from modern industrial enterprise [1] to social networks analysis [2]. Big data includes big datasets with numeric and categorical data, text information, images, and video data. Information in the form of graphs is traditionally used in computer science. In particular, thesauri and ontologies are used to process knowledge and texts, but such models cannot yet be classified as big data. Probably, it is appropriate to speak about the appearance of problems of processing Big Graphs, first of all, in connection with the appearance of knowledge graphs.

The problem of processing Big Graphs can be solved in various ways, for example, by creating specialized high-performance hardware for processing graphs [3, 4]. However, the most common way for Big Graphs processing is to use specialized Big Data frameworks for Big Graphs processing.

The dominant graph model in such frameworks is usually a flat graph model or property graph model (which is, in fact, the multigraph model). Here, by a flat graph, we mean a graph in which a directed or undirected edge connects exactly two vertices. However, the flat graph model is not flexible enough and may not be a convenient solution for modeling complex data domains with hierarchical relationships. For example, complex technical products' assembly sequence problem requires not a flat graph but a hypergraph model [5].

One of the existing extensions of the traditional graph model is the metagraph model. The metagraph model consists of a metagraph data model and a metagraph agent model aimed to process metagraph data. In this article, we present the implementation of the metagraph agent model using Big Data processing capabilities.

The article is organized as follows. In sections 2 and 3, the metagraph data model and the metagraph agent model are formally defined. In section 4, the metagraph representation using the flat graph model is discussed. In section 5, the flat graph and metagraph Big Data processing are described. In section 6, the metagraph processing using metagraph agents is discussed, including the experimental subsection.

2. The Metagraph Data Model

Metagraph is a kind of complex network model proposed by A. Basu and R. Blanning in their book [6] and then adapted for information systems description in our paper [7]. According to [7]:

$$MG = \langle V, MV, E \rangle, \tag{1}$$

where *MG* – metagraph; *V* – set of metagraph vertices; *MV* – set of metagraph metavertices; *E* – set of metagraph edges.

Metagraph vertex is described by a set of attributes:

$$v_i = \{atr_k\}, v_i \in V, \tag{2}$$

where v_i – metagraph vertex; atr_k – attribute.

Metagraph edge is described by a set of attributes, the source and destination vertices, and edge direction flag:

$$e_i = \langle v_S, v_E, eo, \{atr_k\} \rangle, e_i \in E, eo = true \mid false, \tag{3}$$

where e_i – metagraph edge; v_S – source vertex (metavertex) of the edge; v_E – destination vertex (metavertex) of the edge; *eo* – edge direction flag (*eo=true* – directed edge, *eo=false* – undirected edge); atr_k – attribute.

The metagraph fragment:

$$MG_i = \{ev_j\}, ev_j \in (V \cup E \cup MV), \tag{4}$$

where MG_i – metagraph fragment; ev_j – an element that belongs to the union of vertices, edges, and metavertices.

The metagraph metavertex:

$$mv_i = \langle \{atr_k\}, MG_j \rangle, mv_i \in MV, \tag{5}$$

where mv_i – metagraph metavertex belongs to a set of metagraph metavertices MV; atr_k – attribute, MG_j – metagraph fragment.

Thus, metavertex, in addition to the attributes, includes a fragment of the metagraph. The presence of private attributes and connections for metavertex is a distinguishing feature of metagraph. It makes the definition of metagraph holonic – metavertex may include a number of lower-level elements and, in turn, may be included in a number of higher-level elements.

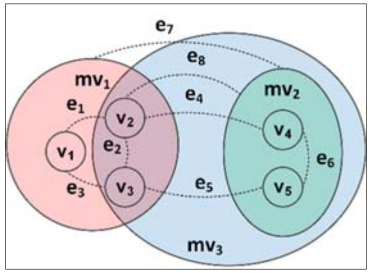

Fig. 1. The example of the metagraph data model

The example of the data metagraph (shown in fig. 1) contains three metavertices: mv_1, mv_2, and mv_3. Metavertex mv_1 contains vertices v_1, v_2, v_3 and connecting them edges e_1, e_2, e_3. Metavertex mv_2 contains vertices v_4, v_5, and connecting them edge e_6. Edges e_4, e_5 are examples of edges connecting vertices v_2-v_4 and v_3-v_5 that are contained in different metavertices mv_1 and mv_2. Edge e_7 is an example of the edge connecting metavertices mv_1 and mv_2. Edge e_8 is an example of the edge connecting vertex v_2 and metavertex mv_2. Metavertex mv_3 contains metavertex mv_2, vertices v_2, v_3, and edge e_2 from metavertex mv_1 and also edges e_4, e_5, e_8, showing the holonic nature of the metagraph structure.

3. The Metagraph Agent Model

The metagraph model is aimed for complex data description. But it is not aimed for data transformation. To solve this issue, the metagraph agent (ag^{MG}) aimed for data transformation is proposed. There are two kinds of metagraph agents: the metagraph function agent (ag^F) and the metagraph rule agent (ag^R). Thus $ag^{MG} = ag^F \mid ag^R$.

The metagraph function agent serves as a function with input and output parameters in the form of metagraph:

$$ag^F = \langle MG_{IN}, MG_{OUT}, AST \rangle, \tag{6}$$

where ag^F – metagraph function agent; MG_{IN} – input parameter metagraph; MG_{OUT} – output parameter metagraph; AST – abstract syntax tree of metagraph function agent in the form of metagraph.

The metagraph rule agent is rule-based:

$$ag^R = \langle MG, R, AG^{ST} \rangle, R = \{r_i\}, r_i : MG_j \to OP^{MG}, \tag{7}$$

where ag^R – metagraph rule agent; MG – working metagraph, a metagraph on the basis of which the rules of the agent are performed; R – set of rules r_i; AG^{ST} – start condition (metagraph fragment for start rule check or start rule); MG_j – a metagraph fragment on the basis of which the rule is performed; OP^{MG} – set of actions performed on metagraph.

The antecedent of the rule is a condition over a metagraph fragment. The consequent of a rule is a set of actions performed on metagraph. Rules can be divided into open and closed.

The consequent of the open rule is not permitted to change the metagraph fragment occurring in the rule antecedent. In this case, the input and output metagraph fragments may be separated. The open rule is similar to the template that generates the output metagraph based on the input metagraph.

The consequent of the closed rule is permitted to change the metagraph fragment occurring in the rule antecedent. The metagraph fragment changing in rule consequent causes to trigger the antecedents of other rules bound to the same metagraph fragment. But incorrectly designed closed rules system can cause an infinite loop of metagraph rule agents.

Thus, the metagraph rule agent can generate the output metagraph based on the input metagraph (using open rules) or can modify the single metagraph (using closed rules).

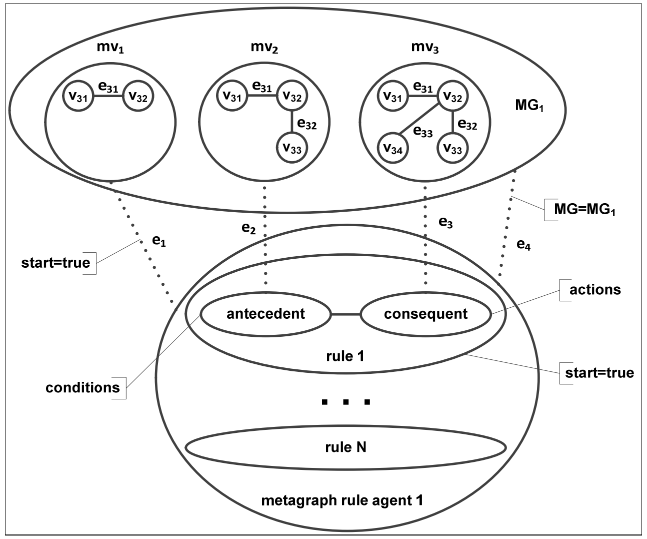

Fig. 2. The example of the metagraph rule agent

The example of a metagraph rule agent is shown in fig. 2. The metagraph rule agent «metagraph rule agent 1» is represented as metagraph metavertex. According to the definition, it is bound to the working metagraph MG_1 – a metagraph on the basis of which the rules of the agent are performed. This binding is shown with edge e_4.

The metagraph rule agent description contains inner metavertices corresponds to agent rules (rule 1 … rule N). Each rule metavertex contains antecedent and consequent inner vertices. For the given an example, mv_2 metavertex bound with the antecedent, which is shown with edge e_2, and mv_3 metavertex bound with consequent, which is shown with edge e_3. Antecedent conditions and consequent actions are defined in the form of attributes bound to antecedent and corresponding consequent vertices.

The start condition is given in the form of the attribute *«start=true.»* If the start condition is defined as a start metagraph fragment, then the edge bound start metagraph fragment to agent metavertex (edge e_1 in given an example) is annotated with the attribute *«start=true.»* If the start condition is defined as a start rule, then the rule metavertex is annotated with the attribute *«start=true»* (rule 1 in given an example). Fig. 2 shows both cases corresponding to the start metagraph fragment and to the start rule.

The distinguishing feature of a metagraph agent is its homoiconicity which means that it can be a data structure for itself. This is due to the fact that according to the definition, a metagraph agent may be represented as a set of metagraph fragments, and this set can be combined in a single metagraph. Thus, a metagraph agent can change the structure of other metagraph agents.

4. The Metagraph Representation Using Flat Graph Model

To build systems using the metagraph model, it is necessary to determine the method of physical representation of the metagraph for persistent storage and for processing in Big Data platforms. We considered various options: relational representation, representation with multiple documents (nested or separate), representation via a flat graph. The term "flat graph" is also used for planar graphs. By flat graph we mean simple graph, as an opposite to graph with nested vertices (metagraph). We conducted a comparative analysis of different representation options. Features of different options and experiments to compare performance are described in our paper [8]. Experiments have shown that the representation of the metagraph via a flat graph is the most preferable.

In such representation, each vertex, metavertex, and the edge of the graph are represented with a separate vertex of the flat graph. The example is shown in fig. 3.

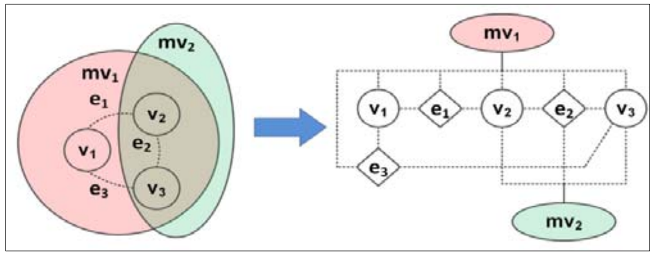

Fig. 3. The metagraph representation via a flat graph

Such a representation is, on the one hand, more intuitive. On the other hand, it allows us to efficiently process the metagraph using methods for working with flat graphs from graph DBMS (optimized graph queries) and Big Data graph platforms (distributed graph processing models).

5. The Flat Graph and Metagraph Big Data Processing

5.1 The Overview of a Metagraph Processing System

Consider a possible structure of a system for processing data in metagraph form. The system consists of two main components (shown in fig. 4): a storage module and a processing module.

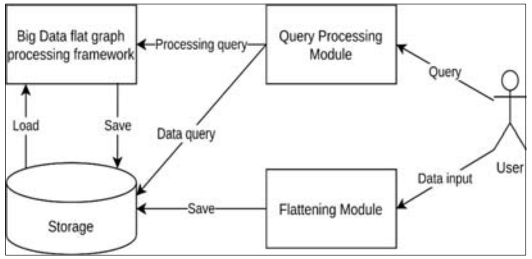

Fig. 4. The structure of the metagraph processing framework

Since we are using a graph representation, it is advisable to use a graph DBMS for storage. We assume that the conversion of the metagraph to a flat graph occurs at the writing stage to the storage in the flattening module. The processing is conducted on a Big Data platform running distributed across multiple machines.

The system works as follows. Firstly, a request comes from the user in the form of calling the system API or in some formal language. The query processing module determines if the query is a simple data query, and in this case, it returns data from storage. If the query requires calculation on metagraph, it occurs in the Big Data graph framework (which is shown in fig. 5). Metagraph is loaded from persistent storage (e.g., the Neo4j graph DBMS) into the Big Data processing platform (Apache Spark) into the RAM of the cluster nodes. Next, the request is processed with the tools of the graph processing platform. Then the problem is solved on a graph representation of the metagraph, and the result of a solution is displayed to the user.

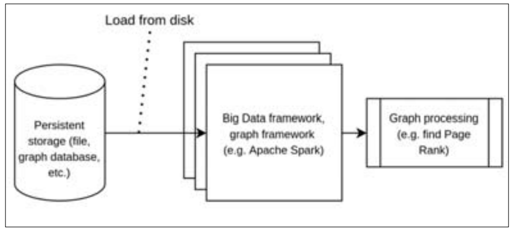
Load from disk
Persistent storage (file, graph database, etc.)
Big Data framework, graph framework (e.g. Apache Spark)
Graph processing (e.g. find Page Rank)

Fig. 5. The operation of the metagraph processing framework

5.2 Graph Processing Models

Modern Graph Big Data frameworks use different approaches to graph processing. Though these approaches share basic principles (like vertex-centric processing), but there are some differences between them. Below we briefly describe some popular models and consider their usage for metagraph processing.

5.2.1 Pregel model

Pregel model [9] was originally introduced by Google and is implemented in multiple graph frameworks (Apache Spark GraphX [10], Apache Giraph [11], Flink Gelly [12]). Pregel model (and other vertex-oriented approaches) were designed specifically to deal with graph data. Standard Big Data approaches, like Map Reduce, are not well suited for graph data structure and graph problems. Map Reduse requires passing the entire state of the graph from one stage to the next - in general requiring much more communication and associated serialization overhead. Abstraction from graph physical distribution (given by messaging system) also simplifies the algorithms. As we use graph representation of metagraph, it seems reasonable to use graph-specific approaches.

Graph processing in Pregel consists of a sequence of synchronous iterations. During each iteration, for each vertex of the graph, a user-defined function is executed, which modifies the value of the vertex in a certain way. Then the vertex sends messages to neighbors' vertices. The next iteration begins only when all vertices have been processed.

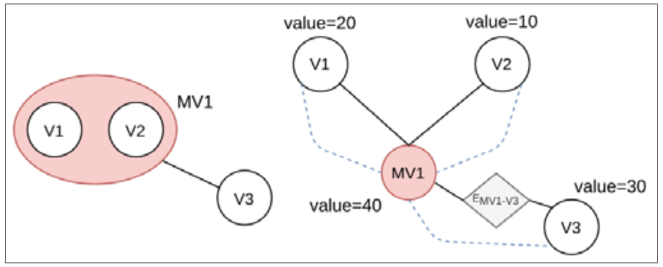

Fig. 6. The example metagraph for the Pregel model

This model is the most general and flexible, but it has a significant drawback in the restrictions on the synchronism of iterations. In real-world graphs (like social graphs), vertex degrees often follow power-law distribution [13], which means that most of the graph vertices have a low degree, and there are a few vertices with a much higher degree. For graph representation of real-world metagraphs, we may expect the same distribution because graph metavertices have not only neighbors but also sub-vertices. If the graph has a significant number of high-degree vertices (which are metavertices of metagraph), Pregel iterations will be filled unevenly since the metavertices will receive and send more messages. For example, consider the simple metagraph in fig. 6 (on the left side, we show the original metagraph, on the right side corresponding flat graph representation).

During the Pregel step of some algorithms, vertices send values to each other. Note that we presume that edge-vertices like $E_{MV1\text{-}V3}$ are not processed as separate vertices and blindly transfer messages. Still, in general, it is up to the implementation of a specific algorithm. Metavertex MV_1 sends its value to neighbor V_3 and subvertices V_1 and V_2 and receives their values. It sends and receives three times more messages than other vertices. As the next iteration will start only when metavertex is processed, all other vertices stay idle. If a metagraph has relatively large metavertices, it may become a performance issue.

5.2.2 Signal-Collect / Scatter-Gather model

This model used by Apache Flink Gelly also assumes vertex-level iterative processing [14]. It consists of two phases: signal (sending messages to other vertices) and collect (receiving messages and updating the vertex).

Two modes of operation are possible. In synchronous mode, this model works similarly to the Pregel. In asynchronous mode, there is no barrier between iterations, and vertices function as independent actors. In this case, we do not get downtime for processing metavertices. However, such a model (depending on the problem being solved) may require local locks, which will complicate the algorithms. Consider the example in fig. 7.

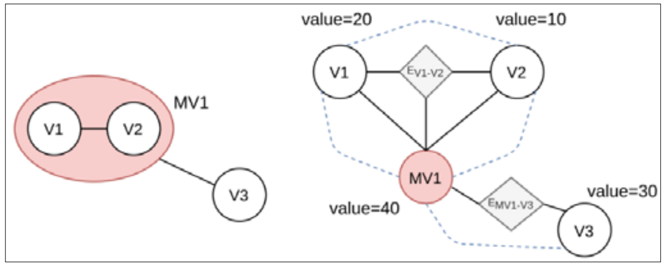

Fig. 7. The example metagraph for the Signal-Collect model

Vertex V_1 receives value from neighbor V_2 (through intermediate vertex representing edge V1-V2). With synchronous Pregel-like approach V_1 will process this value and send it further only in the next iteration, after MV_1 processes all its messages. In asynchronous mode, it will happen before MV_1 is processed. This approach is more error-prone to race condition-related problems, so algorithms should be adapted accordingly.

5.2.3 Gather Sum Apply and closely related Gather-Apply-Scatter [15] models

This model [15] used by PowerGraph [16], GraphLab [17], Apache Flink Gelly is also an iterative one. In this model, user code is launched not over the vertices but over the edges. It includes three phases (Gather, Sum, and Apply) which are shown in fig 8.

The Gather phase – execution of the algorithm relative to an edge and sending messages to the vertices. The Gather phase produces the partial result. Sum phase – each vertex performs aggregation of partial results from received messages. Apply phase – updating the vertex with the result value. For simplicity, we show gather messages only in one direction. In the case of an undirected graph, there are also reverse messages.

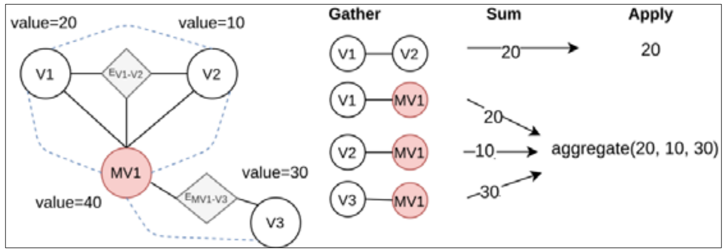

Fig. 8. The example metagraph for the Gather-Sum-Apply model

Since edges, unlike vertices, all have the same degree (equal to 2), they are processed equally. Thus, in this case, there is no downtime for processing metavertices. However, this model has limitations that can complicate the implementation of algorithms:

1) Updating a vertex requires the aggregation of messages from different edges. The update function must be associative and commutative.
2) There is no way to transfer messages between arbitrary vertices because messages are transmitted from edge to vertex.

Thus, the described models have both advantages and disadvantages. Pregel model is the most flexible, but it may not be most efficient if the metagraph contains some very high-degree metavertices. The Signal-Collect model can be more performant but requires more complicated concurrent algorithms. The Gather Sum Apply model also deals with high-degree metavertices, but it has limitations on program architecture. We can also expect that the model's effectiveness may depend on the metagraph which is being processed and the problem being solved.

How these three models are suitable for processing metagraphs is the subject of further detailed research. This article's experimental part uses the Pregel model as the most flexible of the three models.

6. The Metagraph Processing Using Metagraph Agents

6.1 Metagraph Agents in the Big Data Framework

In our experiments, we use the Apache GraphX framework, which is an extension of Apache Spark for graph processing. Below we will briefly describe how the metagraph agent approach can be applied while using frameworks like GraphX.

Metagraph agent is an instrument for transforming metagraphs. It includes information about whether it should apply to a metagraph fragment and information about how to transform a fragment.

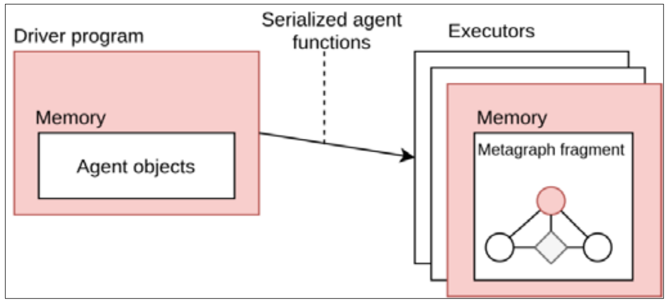

Fig. 9. Metagraph agents in processing framework

As was described in the previous section, in modern graph-oriented Big Data frameworks, usual execution models are vertex-oriented models. There are a driver program and multiple distributed executors. In Apache Spark, the driver program transfers the required code as serialized objects or functions to executors. Executors run this code over parts of distributed datasets. In the case of graph framework, this code is run relatively to vertices of a graph. For example, in the Pregel model Spark GraphX driver program would distribute two functions: «vertex program» to update vertices and «message program» to send vertices.

From the software implementation point of view, the metagraph agent would be an object in the main memory of the driver program, with methods (functions) for updating the vertex based on received Pregel messages and for sending messages from vertex (which is shown in fig. 9). Agent functions also will include rule (condition) determining whether they should be applied to a specific vertex. Graph processing framework (like Spark GraphX) will serialize agent functions and transfer them to executors. Functions of metagraph agents will be executed (if allowed by agent condition) over vertices of a fragment of metagraph graph representation. As metagraph agents are objects, the driver program can organize them in a hierarchy and update them if necessary.

6.2 The Metagraph Single Source Shortest Path Problem

To illustrate metagraph processing, we conducted an experiment where we choose a Single Source Shortest Path (SSSP) problem as a typical graph processing algorithm. The experiment was conducted on the Spark GraphX framework, which implements the Pregel model.

When we apply the SSSP algorithm to a graph representation of metagraph, two main features occur. Firstly, we have a relation «sub vertex to metavertex.» For SSSP-problem, we can assume that the «parent» and «child» vertex can be accessed with some user-defined constant distance. For same-level vertices, distance is defined by edge length, as in a simple graph case. Secondly, the graph representation of the metagraph contains technical vertices representing edges (see vertices e_1, e_2, e_3 in fig. 3). They are to be excluded from the SSSP calculation.

The algorithm goes as follows. We initialize all vertices with an infinite distance value, except for starting vertex, which has zero distance. After that, we perform multiple iterations (Pregel steps). At each iteration, we run the same code over each vertex. Vertex sends messages over its outgoing edges unless finding an edge from a vertex representing the edge to its parent metavertex (see vertices e_1 and mv_1 in fig. 3). If the edge source vertex is a normal vertex, it sends the current distance value. If the edge source vertex is an edge-vertex, it adds the length. If the edge is an edge to parent metavertex, it adds user-defined distance between child and parent vertices.

At the beginning of each iteration, vertex updates its value with a minimum of received messages and current value. When no messages are sent during iteration, the algorithm converges.

In general, this algorithm resembles Pregel SSSP-algorithm for simple graphs with a few adaptations to metagraph physical representation.

Testing program is written is Scala, algorithm of determining Pregel messages at each iteration is presented below in Python-style pseudo-code:

```
messages = []
for edge in vertex.outgoingEdges:
  if edge.isEdgeToParent and vertex.isEdgeVertex:
    continue
  if edge.isEdgeToParent:
    distanceValue = distanceToParentVertex
  else if vertex.isEdgeVertex:
    distanceValue = vertex.distance + vertex.length
  else:
    distanceValue = vertex.distance
  messages.append((edge.dst.id, distanceValue)
```

In terms of metagraph agents, we can say that graph is processed by a single metagraph agent with a closed rule. Condition of the rule's antecedent allows any fragment to be processed (as the SSSP algorithm has to visit all edges of the graph). Rule consequent includes sending messages to neighbors and updating vertex with the minimum of received distances.

In our experiments we use synthetic generated datasets. Some real world examples for this problem may include processing social graph with hierarchy, where we need to know «distance» between people, but we have not only direct social connections, but also indirect connections based on hierarchy. This hierarchy would be represented by assigning people to nested metavertices of a metagraph, thus describing, for example, connections between people based on some organizational structure (possibly with overlapping metavertices). We could traverse such graph not only by direct «friendship» relation, but also up and down organizational structure.

6.3 Experiments

We generated multiple metagraphs and graphs with the same number of vertices and similar topology for the experiments. We use normal graph processing time as a baseline to show that processing of metagraph can be performed in comparable time (not varying by orders of magnitude, which would neglect benefits of the model). Metagraph consisted of N metavertices with two vertices in each, with each vertex also having one outgoing edge. A simple graph consisted of 3N vertices with three outgoing edges each. Each vertex has three relations in both cases, and a number of vertices were the same (not considering edge-vertices for metagraph). Graphs were generated by Python scripts as CSV files and imported into the Spark GraphX framework.

Experiments were conducted on Amazon EMR infrastructure with Spark 2.4.7 and Hadoop YARN 2.10.1. We used Amazon EC2 m5.xlarge instances with the following configuration (per instance): CPU - Intel Xeon Platinum 8175M 2.5 GHz, 16 GiB memory, 64 GiB EBS Storage. CSV files with initial data were imported into the HDFS filesystem.

The size of initial data (in Megabytes) for graph and metagraph with 100 000 to 10 000 000 vertices is shown in fig. 10.

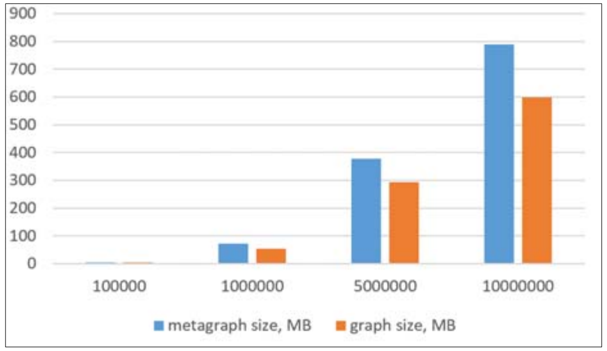

Fig. 10. Initial datasets size (MB) for different vertices counts

At first, we conducted an SSSP calculation with one master node and two worker nodes. SSSP calculation time (in seconds) for graphs and metagraphs of different sizes is shown in fig. 11.

As we can see, SSSP calculation time for metagraph is comparable to the time of processing a flat graph of the same size. Metagraph processing is obviously slower because graph representation of metagraph with N vertices and M edges will have N+M vertices in the flat graph.

We also conducted experiments to see how SSSP execution time in the case of metagraph processing is affected by the level of parallelism. At first, we conducted SSSP execution with a single worker node and a different number of Apache Spark executors. Results for processing metagraphs with

1 000 000 vertices are shown in fig. 12. Simple graph processing time is also included as the reference point.

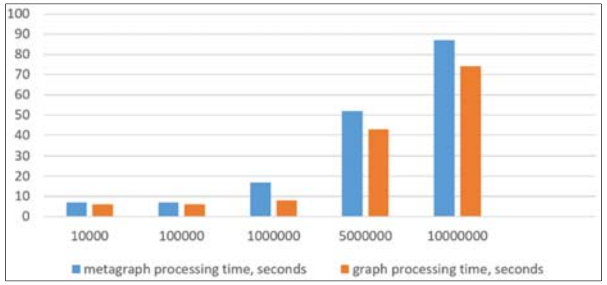

Fig. 11. SSSP calculation time (seconds) for a metagraph and a flat graph depending on the number of vertices in the graph (metagraph)

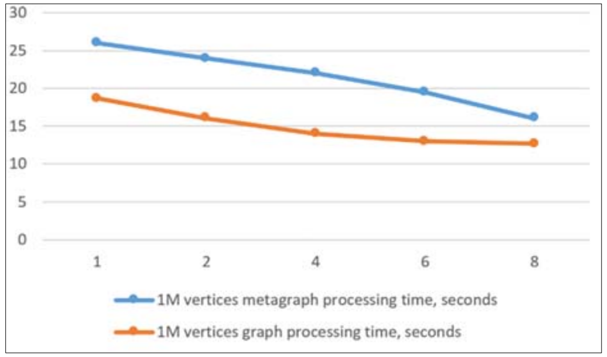

Fig. 12. SSSP calculation time (seconds) for metagraph and graph with 1 000 000 vertices depending on the number of executors

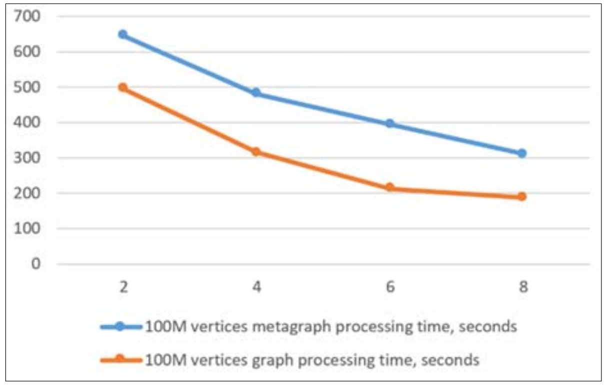

Fig. 13. SSSP calculation time (seconds) for metagraph and graph with 100 000 000 vertices depending on the number of worker nodes

Then we conducted an experiment with a larger dataset (metargraph and graph with 100 000 000 vertices) and with two to eight worker nodes in a cluster (see Fig. 13). As we can see from the charts, with the addition of new Spark executors and with the addition of worker nodes to the cluster, SSSP calculation time decreases similarly for metagraph and simple graph. Thus, metagraph processing may be effectively parallelized using Apache Spark.

7. Future Work

At the moment, we have implementation of persistent metagraph storage in graph database (Neo4j, ArangoDB). We also processed graph representation of metagraph in Big Data graph framework (Apache GraphX). Our next step will be creating a system which includes both persistent storage and processing framework and provides API for working with metagraph without manual processing of its graph representation.

8. Conclusions

Nowadays, Big Data frameworks are widely used for processing numeric and categorical data, text information, images, video data, and graph information.

The dominant graph model in such frameworks is usually a flat graph model or property graph model (which is, in fact, the multigraph model). However, to describe complex situations, flat graph models may not be enough, and we propose using a metagraph model.

The critical element of the metagraph model is metavertex. From the general system theory point of view, a metavertex is a particular case of the manifestation of the emergence principle, which means that a metavertex with its private attributes and connections becomes a whole that cannot be separated into its component parts.

The metagraph may be represented as a multipartite flat graph.

The system for processing data in metagraph form consists of two main components: the storage module and the processing module, where the processing module is based on the Big Data processing platform.

The three most popular graph processing models are the Pregel model, Signal-Collect / Scatter-Gather model, and Gather Sum Apply model. How these three models are suitable for processing metagraphs is the subject of further detailed research. In the experimental part, we use the Pregel model as the most flexible of the three models.

The experimental part is based on the Single Source Shortest Path (SSSP) problem. The SSSP calculation time for the metagraph is comparable to processing a flat graph of the same size. The metagraph processing may be effectively parallelized using Apache Spark.

The described metagraph approach with flat graph representation in graph processing frameworks seems to be an effective instrument for solving problems with processing complex data.

References

[1] Reut D., Falko S., Postnikova E. About scaling of controlling information system of industrial complex by streamlining of big data arrays in compliance with hierarchy of the present lifeworlds. International Journal of Mathematical, Engineering and Management Sciences, vol. 4, issue 5, 2019, pp. 1127-1139.

[2] Chesnokov V. Overlapping community detection in social networks with node attributes by neighborhood influence. In Proc. of the 6th International Conference on Network Analysis, Springer Proceedings in Mathematics & Statistics, vol. 197, 2017, pp. 187-203.

[3] Rasheed B., Popov A.Yu. Network graph datastore using DiSc processor. In Proc. of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, 2019, pp. 1582-1587.

[4] Abdymanapov C., Popov A.Yu. Motion planning algorithms using DISC. In: Proc. of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, 2019, pp. 1844-1847.

[5] Bozhko A.N. Hypergraph model for assembly sequence problem. In IOP Conference Series: Materials Science and Engineering, vol. 560, Issue 1, 2019.

[6] Basu A., Blanning R. Metagraphs and Their Applications. Springer, 2007, 174 p.

[7] Chernenkiy V.M., Gapanyuk Yu.E. et al. The Hybrid Multidimensional-Ontological Data Model Based on Metagraph Approach. Lecture Notes in Computer Science, vol. 10742, 2018, pp. 72-87.

[8] Chernenkiy V.M., Gapanyuk Yu.E. et al. The Principles and the Conceptual Architecture of the Metagraph Storage System. Communications in Computer and Information Science, vol. 1003, 2018, pp. 73-87.

[9] Malewicz G., Austern M.H. et al. Pregel: A System for Large-scale Graph Processing. In Proc. of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 135-146.

[10] Xin R.S., Crankshaw D. et al. GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. arXiv preprint arXiv:1402.2394, 2014.
[11] Shaposhnik R., Martella C., Logothetis D. Practical Graph Analytics with Apache Giraph. Apress, 2015, 334 p.
[12] Introducing Gelly: Graph Processing with Apache Flink, URL:https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html, accessed 2021/04/09.
[13] Faloutsos M., Faloutsos P., Faloutsos C. On power-law relationships of the internet topology. ACM SIGCOMM Computer Communication Review, vol. 29, issue 4, 1999, pp. 251-262.
[14] Signal-Collect programming model, URL: https://uzh.github.io/signal-collect/documentation.html, accessed 2021/04/09.
[15] Han M., Daudjee K. et al. An Experimental Comparison of Pregel-like Graph Processing Systems. Proceedings of the VLDB Endowment, vol. 7, issue 12, 2014, pp. 1047-1058.
[16] Low Y., Gonzalez J. et al. Distributed GraphLab: A Framework for Machine Learning in the Cloud. arXiv preprint arXiv:1204.6078, 2012.
[17] Gonzalez J.E., Low Y. et al. PowerGraph: Distributed graph-parallel computation on natural graphs. In Proc. of the 10th USENIX conference on Operating Systems Design and Implementation, 2012, pp. 17-30.

Information about authors / Информация об авторах

Valeriy Mikhailovich CHERNENKIY – doctor of technical sciences, professor. Research interests: parallel-sequential process theory, imitation modelling, automated organization management system design.

Валерий Михайлович ЧЕРНЕНЬКИЙ – доктор технический наук, профессор. Сфера научных интересов: теория описания параллельно-последовательных процессов, имитационное моделирование информационных систем, проектирование автоматизированных систем организационного управления.

Ivan Vladimirovich DUNIN – postgraduate student. Research interests: graph processing, graph databases, distributed processing.

Иван Владимирович ДУНИН – аспирант. Сфера научных интересов: обработка графов, графовые базы данных, распределенная обработка.

Yuriy Evegenievich GAPANYUK – candidate of technical sciences, associate professor. Research interests: automated systems design, hybrid intelligent information systems, complex graph models.

Юрий Евгеньевич ГАПАНЮК – кандидат технических наук, доцент. Сфера научных интересов: проектирование автоматизированных систем, проектирование гибридных интеллектуальных информационных систем, сложные графовые модели.