UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS PERES GASPAR

PROGRAMMING MODELS TO DISTRIBUTED GRAPHS

FORTALEZA

2017

LUCAS PERES GASPAR

PROGRAMMING MODELS TO DISTRIBUTED GRAPHS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Jose Antonio Fernandes de Macedo

FORTALEZA

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

G232p Gaspar, Lucas Peres.
Programming models to distributed graphs / Lucas Peres Gaspar. – 2017.
42 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Ciências, Curso de Computação, Fortaleza, 2017.
Orientação: Prof. Dr. José Antônio Fernandes de Macêdo.

1. Large graphs. 2. Distributed system. 3. Programming models. I. Título.

CDD 005

LUCAS PERES GASPAR

PROGRAMMING MODELS TO DISTRIBUTED GRAPHS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Jose Antonio Fernandes de Macedo (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Regis Pires Magalhães
Universidade Federal do Ceará (UFC)

Prof. Dr. Sabeur Aridhi
University of Lorraine (UL)

Prof. Dr. Engelbert Mephu Nguifo
Université Blaise Pascal Clermont-Ferrand (UBP)

Aos meus pais, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram a esperança para seguir. Aos meus amigos que sempre me auxiliaram em meu aprendizado. Essa conquista é de todos nós.

AGRADECIMENTOS

Primeiramente agradeço à Deus por toda esta jornada.

Ao Prof. Dr. José Antônio Fernandes de Macedo por me orientar neste início de jornada acadêmica.

Aos meus pais, que sempre me apoiaram nas escolhas da vida. Em particular minha mãe, que sempre mostrou que devemos buscar mais conhecimento e sempre continuar com os estudos.

Aos meus colegas do Insight Data Science Lab, por terem me auxiliado em minhas pesquisas, principalmente ao Prof. Regis Pires Magalhãs e David Araújo Abreu, que me motivam e auxiliam a ampliar meus conhecimentos, tanto na ciência quanto na vida.

Ao Departamento de Computação e seu corpo docente, por terem me proporcionado 4 anos de muita experiência e aprendizado.

Aos meus colegas do PET Computação UFC e da graduação, que sempre acreditaram em mim como cientista.

Por fim, a todos aqueles que participaram direta e indiretamente de toda minha graduação.

"Always pass on what you have learned."

(Master Yoda(Frank Oz), in *Star Wars Episode VI: Return of the Jedi*, 1983)

RESUMO

Processamento de grafos em larga escala é vital para muitas aplicações científica, e esse processamento está entre os sete principais métodos de análise de dados massivos. Embora alguns frameworks facilitem o desenvolvimento de processamento de grafos paralelo e distribuído, existe uma necessidade para um entendimento mais profundo de seu modelo conceitual. Este trabalho revisa alguns dos mmais conhecidos modelos de programação e detalha seu funcionamento e suas características. Além disso, nós também apresentamos frameworks baseados nesses modelos. Também realizamos comparações entre os modelos, a fim de mostrar pontos fortes e fracos para que possamos decidir quando e como usar tais modelos.

Palavras-chave: Gandes Grafos. Sistemas Distribuidos. Modelos de Programação.

ABSTRACT

Large-scale graph processing is vital for many scientific applications, and graph processing is among the seven principal computational methods of massive data analysis. Although some frameworks facilitate the development of parallel and distributed graph processing, there is a need for in-depth understanding of their conceptual model. This work reviews some of the well-known programming models and details how they work, their main features. Besides, we also present frameworks that are based on these models. We also make a comparison among them, to show its pros and cons so that we can decide when and how to use those models.

Keywords: Large Graphs. Distributed System. Programming Models.

LIST OF FIGURES

Figura 1 – Undirected graph example 16
Figura 2 – Directed graph example 17
Figura 3 – Directed graph example 17
Figura 4 – BSP Higher Value Vertex Example 22
Figura 5 – BSP Superstep Workflow 23
Figura 6 – Undirected Graph with 2 triangles 25
Figura 7 – GAS Higher Value Vertex Example 29
Figura 8 – GAS Superstep Workflow 30

LIST OF TABLES

Tabela 1 – Algorithms Study Frequency (GUO *et al.*, 2014) 18
Tabela 2 – Large Graphs Sizes 19
Tabela 3 – Models comparison 37

LIST OF ALGORITHMS

1 Bulk Synchronous Parallel (BSP) Vertex Interface 23
2 BSP Triangle Count 25
3 GAS Vertex Interface 29
4 GAS Triangle Count 31

LISTA DE ABREVIATURAS E SIGLAS

BSP	Bulk Synchronous Parallel
GAS	Gather, Apply, Scatter
GPS	Graph Processing System
RDD	Resilient Distributed Datasets
RDG	Resilient Distribute Graph

SUMÁRIO

1 INTRODUCTION 15
2 THEORETICAL FOUNDATION 16
2.1 Graphs 16
2.2 Graph's Algorithms 17
2.3 Large Graph Processing 18
2.4 Large Graph Processing Frameworks 19
2.5 Programming Models for Distributed Graph Processing 19
2.6 Conclusion 20
3 BULK SYNCHRONOUS PARALLEL (BSP) 21
3.1 Introduction 21
3.2 BSP Model 22
3.3 Primitive Functions 24
3.4 Code example 25
3.5 Execution Cost 26
3.6 Frameworks 26
3.7 Conclusion 27
4 GATHER, APPLY, SCATTER (GAS) 28
4.1 Introduction 28
4.2 GAS Model 28
4.3 Primitive Functions 30
4.4 Code example 31
4.5 Execution Cost 31
4.6 Frameworks 32
4.7 Conclusion 32
5 OTHER MODELS 33
5.1 Introduction 33
5.2 Map/Reduce 33
5.3 Functional Programming 34
5.4 Actor 35
5.5 Conclusion 36

6	**MODEL COMPARISONS**	37
6.1	**Introduction**	37
6.2	**Criteria**	37
6.3	**Analysis of models**	38
6.4	**Conclusion**	39
7	**CONCLUSIONS AND FUTURE WORKS**	40
	REFERENCES	41

1 INTRODUCTION

Thanks to the popularity of technologies and the Internet, we are handling with datasets bigger than ever. This phenomenon causes unprecedented challenges concerning data processing. The situation is not different when dealing with graphs datasets. (JORDAN, 2013) says that graph processing is among the seven principal computational methods of massive data analysis. Graphs can be used in several scenarios, like analytic, business, social networks, etc.

To deal with the large-graphs context, we have tools to manipulate them. There are a lot of frameworks that allow constructing algorithms to be applied to those graphs, but to use them we must know how they work. Before studying the frameworks, we should understand the programming models used to develop them.

The goal of this work is to describe some of those models, presenting how they work, exemplifying their functionalities, showing which components we can use to construct a distributed system with those models and showing frameworks that implement them and make some comparisons among them, allowing us to understand better how those models works and help us choose a good model to solve some problem on some giving scenarios.

The following parts of this work are organized as follows: Chapter 2 presents some definitions that are important to understanding the context of this work. On Chapter 3 and Chapter 4 we present, respectively, the BSP and Gather, Apply, Scatter (GAS) programming models, explaining how they work, a way to structure them, an algorithm example, suggestions of functions that can improve the semantics of the codes and frameworks that use those models. In Chapter 5 we present some other models that are used in the *big data* scenario that can be used to handle large-graphs, explaining their workflow and frameworks that uses them. Chapter 6 makes a comparison of the presented models. Finally, Chapter 7 concludes this work.

2 THEORETICAL FOUNDATION

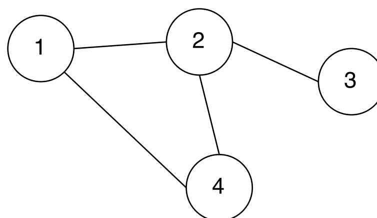
This chapter presents the core concepts used in this work.We start presenting the definition of a graph and its types. A graph is an abstract structure, and it can be used to represent many things, like roads, decision problems, production lines, street maps, network communications, social relations, and others. Sometimes, representing a problem in a graph structure can help its resolution, since there are several theorems and algorithms to extract information from graphs. this chapter also shows some types of graph algorithms that are used to retrieve information.

Sometimes we need to work with large-graphs, that cannot be stored in memory or may take a long time to apply some algorithms. The solution to that is to distribute the graph information, the processing, or both. This work also shows some concepts and challenges of processing large-graphs. We also present some frameworks that deal with them and even the programming models used by those graph frameworks.

2.1 Graphs

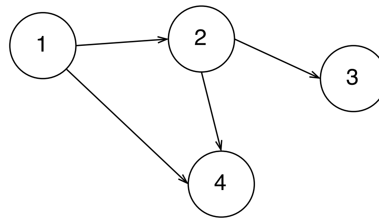
(BONDY; MURTY, 1982) brings traditional definition to a graph that is helpful to understand a lot of graph properties. In our context, we can define a **graph** as an ordered pair *G=(V,E)* composed by two sets: V, representing the set of **vertices** and E representing the **edges**. Each element from E is a pair of two elements of V, representing a relation (edge) between them. Figure 1 ilustrates a graph where $V = \{1,2,3,4\}$ and $E = \{(1,2),(1,4),(2,4),(2,3)\}$. In this case, we have an **Undirected** graph, that is, the edges of the graph represent a symmetrical relation(the relation (1,2) is the same as (2,1)).

Figure 1 – Undirected graph example



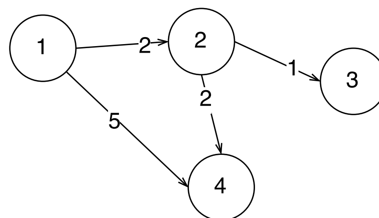
When the pair is ordered, that is, when the relationship is not symmetrical, we say that the graph is **Directed**. Figure 2 shows an example of a directed graph. The directed relations are represented with arrows pointing the direction of the relationship.

Figure 2 – Directed graph example



Suppose that one graph represents a road graph. Figure 1 says that we can go from 1 to 2 and from 2 to 1 using the same road, while Figure 2 says that we can only go from 1 to 2. But if we want to represent the distance between 1 and 2? A graph is called **weighted** when its edges have weight, that is, a value to express the relation. Figure 3 shows a directed and weighted graph where the distance(or cost) to go from 1 to 2 has the value of 2.

Figure 3 – Directed graph example



2.2 Graph's Algorithms

(HARA *et al.*, 2015) present a review of 124 papers and has classified 149 algorithms to analyze the studied frequency of those algorithms. Table 1 represents the resume of this analysis. They are classified into the following categories:

- **Statistics**: these algorithms computes values from the graph relations, like the number of relations, more related vertices, groups of relations. Those are helpful to collect useful metrics to analyze the graph.
- **Traverse**: traverse algorithms are one of the most known(and used) types: they help in finding a path between two vertices of the graph.
- **Related Components**: some graphs have subsets of vertices and edges that are not related, that means, there is no path linking those subsets to the rest of the graph. Subgroups like those are called components, and these algorithms are used to find them.
- **Community Detection**: this algorithm category can be used to answer questions like "*what are the nearest vertex of v?*", or "*how many groups of similar vertices do we have?*".

They detect groups of vertices bounded by some characteristic.

- **Evolution**: the idea of this type of algorithm is to study the *topology* of the graph to analyze how it can affect the evolution of the graph.
- **Others**: here we have some categories, like *partitioning* algorithms, that are used to divide the graph into subgraphs in a most optimized way (dividing the nodes, reducing the number of edges between two partitions, etc.) and *sampling* algorithms, that retrieve a subgraph that statistically represents a good sample of the graph(like keeping the mean degree of the vertices, the maximum shortest path, the centrality, etc.).

Table 1 – Algorithms Study Frequency (GUO *et al.*, 2014)

Class	Algorithms	Ocurrences	%
Statistics	Triangulation, Diameter, BC	24	16.1
Traverse	BFS, DFS, Shortest Path	69	46.3
Related Components	MIS, BiCC, Reachability	20	13.4
Communities Detection	Clustering, Nearest Neighbors (kNN)	8	5.4
Evolution	Forest Fire Model, Preferential Attachment Model	6	4
Others	Sampling, Partitioning	22	148
Total		149	100

2.3 Large Graph Processing

We can work with graphs of several sizes, form small graphs representing a family tree; to large graphs, like huge social networks. Such large-graphs require special techniques to be processed, since, to do so, we must use disk memory or distribute the processing. (KYROLA *et al.*, 2012) presents a table with the size of large-graphs used in experimentation on their work, in 2012. Table 2 shows some of that information.

Nowadays, there are many techniques and frameworks to handle large datasets and to collect information from them, but, when related to graphs, they cannot be applied, because, in some scenarios, the computer can not store all the data or it would take much time to process it.

That is why **Large Graph Processing** is not a simple task. To handle that amount of data in a graph requires specifically designed approaches.

Table 2 – Large Graphs Sizes

Graph	**Vertices**	**Edges**
Netflix (BENNETT *et al.*, 2007)	0.5M	99M
Domain (YAHOO. . . , 2017)	26M	0.37B
Live Journal (BACKSTROM *et al.*, 2006)	4.8M	69M
Twitter 2010 (KWAK *et al.*, 2010)	42M	1.5B
UK Web Graph (BOLDI *et al.*, 2008)	109M	3.7B
Yahoo Web (YAHOO. . . , 2017)	1.4B	6.6B

2.4 Large Graph Processing Frameworks

Many frameworks allow the processing of large-graphs in a distributed environment. Each framework has its abstraction with its particularities, like the topology element used on the distribution(vertices, edges or subgraphs).

The most common topology used is the vertex(called vertex-centric), proposed first by *Pregel*(MALEWICZ *et al.*, 2010), a Google framework. The scope of an algorithm defined on this framework is the vertex value and messages that are sent to it. This framework is also the basis for many others, where all of them try to optimize some feature or remove limitations.

(HEIDARI *et al.*, 2017) and (REZENDE, 2017) compare some of those frameworks, presenting some criteria chosen by them and explaining how they work, the scenarios to use them and their programming models.

2.5 Programming Models for Distributed Graph Processing

Programming in a distributed system is not a simple task. Many challenges can be found in the way, like synchronization between the machines (or *nodes*), the information flow, etc. There are some patterns that are used to develop such systems. These patterns help to define an execution flow, information exchange, and architecture.

When working with distributed graphs, we must define the graph element which the process will be distributed. Currently, the most common approach used is the vertex-centric, popularized by (MALEWICZ *et al.*, 2010). Every computation happens inside a vertex, using its information. This approach was based on MapReduce model(DEAN; GHEMAWAT, 2008), that proposes a local action execution schema, where they all can happen independently. All computation occurs over a vertex, and its value delimits its scope of information. However, there are other approaches like Edges-Centric, where every computation happens inside the edges,

and Block-Centrick, where every computation happens over subgraphs(or blocks) of the whole graph.

In the vertex-centric context, there are two most used programming models: the BSP and GAS, which are going to be presented in chapters 3 and 4, respectively. These models are very used due to some frameworks success.

2.6 Conclusion

This chapter presented the definition of a graph, what are the types algorithms applied to them and its importance. We also discussed the problems to work with large graph datasets and presented two programming models to deal with those problems.

Understanding those models are not just helpful to work with those frameworks, but also to understand how we can adapt a graph problem to the distributed graph context and how to construct its solution.

3 BULK SYNCHRONOUS PARALLEL (BSP)

3.1 Introduction

According to (VALIANT, 1990), in the 80's the sequential computation has brought an efficient bridge between software and hardware, allowing the compilation and execution of high-level languages. But, to deal with parallel computation, was required a model that can allow this same simplicity. That is the BSP goal.

The BSP model first appeared in 1990s (VALIANT, 1990). It was proposed at Harvard University by Leslie Valiant to deal with distributed computing in several scenarios, allowing that operations could be easily programmed to execute in multiples machines.

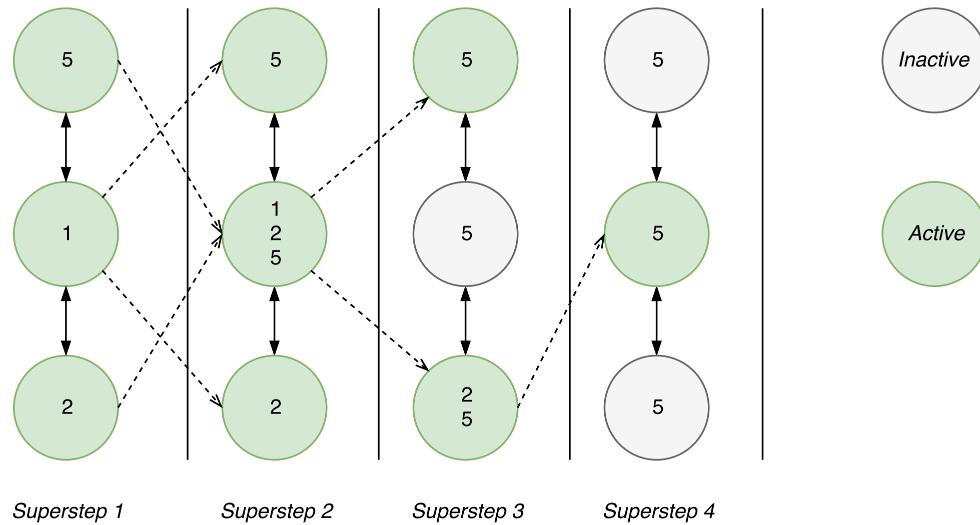
When used in the context of graphs, BSP becomes a vertex-centric programming model, working with a sequence of iterations named *superstep*. A superstep consists of a set of independent local computations, followed by a global communication phase acting as a synchronization unit. Inside a *superstep*, messages can be exchanged between vertices, passing information that will be computed on the next *superstep*.

The basic workflow of an algorithm on the BSP model is as following: on the first *superstep*, each vertex execute its *compute* function. After that, the vertex will become **inactive**. At the end of this execution, the vertex may (or not) send messages to all its neighbors. If so, the vertex that receive messages becomes **active**. After the first *superstep*, in each other *superstep S*, only the **active** vertices execute their computations and they use the messages received on the *superstep S*-1.

To understand better the BSP workflow, assume the example in Figure 4. The idea of the example is to get the vertex with the higher value. It starts with every vertex sending its value to their neighbors. On the second *superstep*, the vertices will process only the messages with a value higher than its own. That is what happens on the middle vertex: it receives 2 values that are higher than its, so it will update its value and send again messages, but now with value 5. On the third *superstep*, only the top and bottom vertices are going to be active, since only them have messages to process. The bottom vertex will update its value and send a message to the middle vertex. Then, on the last *superstep*, the middle vertex will receive a message, but will not update its value. Since no message was sent, the execution stops.

In this chapter we describe the BSP model, presenting an abstract description of its vertex model and system structure. We also present some graph manipulation primitives

Figure 4 – BSP Higher Value Vertex Example



that can improve the semantics of a program written on the model, based on the proposals of (SALIHOGLU; WIDOM, 2014). We also show frameworks that implement this model.

3.2 BSP Model

BSP is a vertex-centric model, so the main structure that we must analyze is the vertex. It computes a function in each *superstep*, and it can(or can not) change its value and then it becomes inactive. In summary, the vertex is composed of the following components:

- **Value**: the value of the vertex itself on the graph. Let's suppose the vertex represents a user on a social network. Its values can be id, name, username, etc.
- **Mutable value**: a value to the vertex that is used in the execution of an algorithm. This value is a result of the execution of some algorithm applied to the vertex, as the number of neighbors, for example.
- **Compute function**: a function that receives the messages and executes an algorithm on each *superstep*. Usually, one *compute* function is implemented for each algorithm.
- **Halt function**: function to set the vertex to *inactive* state.

Algorithm 1 presents an interface to this vertex structure. First, we have the compute function used on the *supersteps*. Then, we have a function to retrieve the vertex value from the model. After, we have functions to get and set the mutable value used on the algorithms. We also have a function to send messages to the neighbors and one to set the vertex state to inactive.

Based on that structure, we can decompose a BSP system into *building blocks* to help to identify the modules of it. Figure 4 presents the components that must be present on the

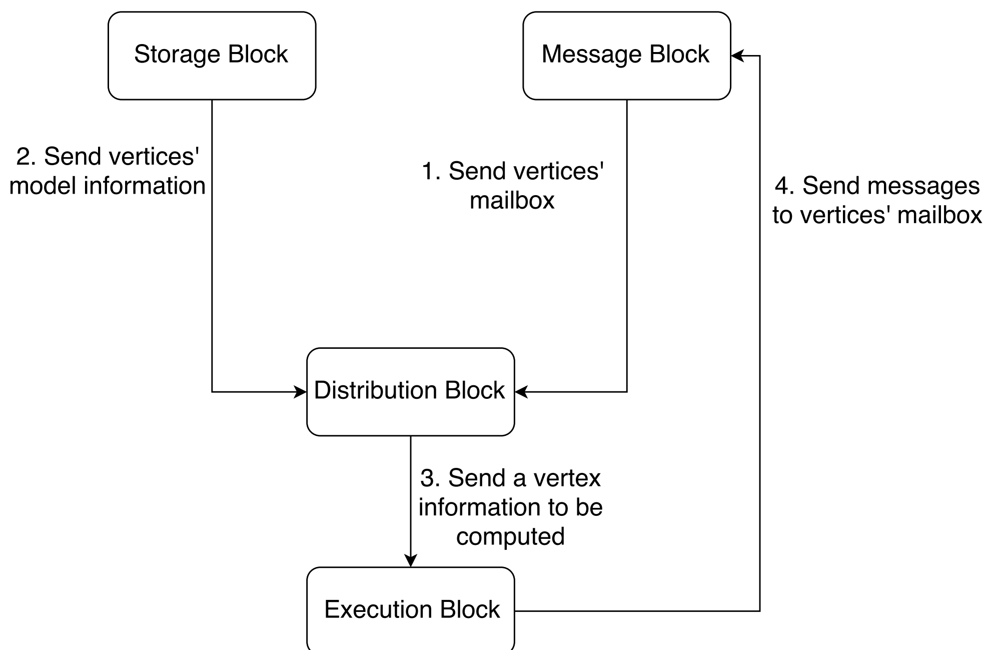
Algorithm 1 BSP Vertex Interface

public void compute*(MessageList msgs)*;
public Value getValue*()*;
public void setMutableValue*(MutableValue m)*;
public MutableValue getMutableValue*()*;
public void sendMessageTo*(Message msg)*;
public void halt*()*;

system, which are:

- **Message Block**: this block will be responsible for managing the message exchange among the vertices. It will store the *mailbox* of each vertex, controlling the insertion and retrieval of messages.
- **Storage Block**: This block is responsible for storing the graph model. It gives access to information like vertex id and its values. It can also save the *mutable values* described before.
- **Execution Block**: It executes the *compute* function of the vertex.
- **Distribution Block**: It coordinates the parallel execution of the algorithm, passing to the Execution Block its vertex information and the messages sent to it(we are going to call this set of messages as *mailbox*). This block works like the *controller* of the algorithm, and, therefore, it synchronizes the executions.

Figure 5 – BSP Superstep Workflow



The workflow of a *superstep* starts in the Distribution Block. For each vertex of the

graph that has messages on the mailbox (that means, it is active) the block will get the messages from the Message Block and the vertex data from the Storage Block and send them to the Execution Block. After that, the Execution Block takes this data and run the *compute* function. If necessary, the block can send messages to other mailboxes of vertices in the Message Block. After all the active vertices have passed over the Execution Block and finished the computing, the *superstep* ends.

This model structure uses the message trading, what let us change the number of computer nodes without concern with the access of the data, what makes the BSP model easily scalable.

3.3 Primitive Functions

Some applications may require an extense coding from the programmer. That happens because the expressiveness of the functions presented in Algorithm 1. They are the basic components of the model. The work on (SALIHOGLU; WIDOM, 2014) presents a set of primitive function to improve the algorithms semantics. From those functions, we can bring some of them to the BSP model:

Filter

This primitive is very simple: it applies some decision over each vertex and, if it returns a false value, the element will be removed from the model. This can be implemented with a call to the *compute* function over each node but is easier to understand semantically, once you do not need to create a whole vertex structure just for this.

Update Value

This one is also a standard computation in a *superstep*, but it requires that messages have been sent to the vertex from its neighbors. So, this function can work in two steps: first, the user defines the information that will be sent from the vertex to its neighbors and, after that, the user specifies what to do with the data. The system will execute two *supersteps*: one to broadcast the information defined by the user and one to handle that information.

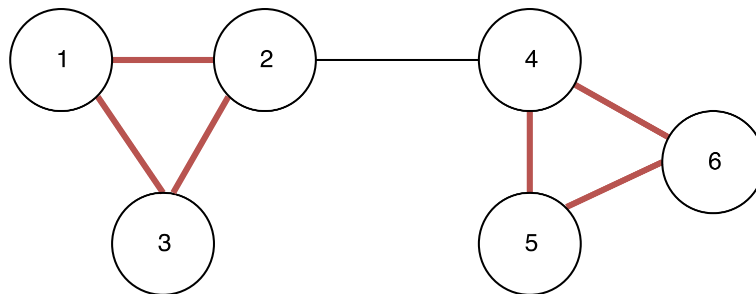
Graph Value

The idea of this primitive is to gather some value over the graph, like the mean degree of the vertices. This primitive can be seen as a *superstep* of a computation applied only over a single vertex.

3.4 Code example

We will present an abstract algorithm to *Triangle Count* on an *undirected* graph using the BSP model. Figure 5 shows an example of a graph with two triangles on its structure. Algorithm 2 shows the code of this algorithm.

Figure 6 – Undirected Graph with 2 triangles



Algorithm 2 BSP Triangle Count

```
1:  if superstep ≤ 1 then
2:      triangles ← 0
3:      sendToNeighbors(neighbors)
4:  else
5:      for message in mailbox do
6:          for vertex in message.data() do
7:              if vertex in neighbors then
8:                  triangles ← triangles + 0.5
9:              end if
10:         end for
11:     end for
12: end if
13: halt()
```

The algorithm execution is simple: when is the first *superstep*(lines 1-3), the number of triangles that the vertex is in is set to 0, and the list of neighbors of each vertex is broadcasted between their neighbors. In the others *supersteps*(lines 5-11), for each message received, the

vertex will check the neighbors' list in the messages, checking for familiar neighbors. If a common neighbor is found, the *triangle* value will be incremented in 0.5 (the graph is undirected so that the same triangle will be computed twice). After that, each vertex will have the number of triangles in which it is.

3.5 Execution Cost

The cost of execution inside a *superstep* in the BSP model can be represented as the sum of three factors:

- The cost of the longest running local computation, that is, the vertex that will take the longest time to execute;
- The higher number of messages that will be sent to a vertex plus the number of messages exchanged between the distribution block and each vertex (usually they are two: one to start the execution and another one to notify its end);
- The cost of the synchronization barrier at the end of the *superstep*.

Based on that, we can approximate the execution cost from an algorithm with S *supersteps* by the following formula, where:

$$\sum_{s=1}^{S} w_s + g \sum_{s=1}^{S} h_s + Sl$$

- S is the number of *supersteps*;
- w_s is the longest vertex execution on *superstep* s;
- g represents the number of messages exchanged between the distribution and execution block;
- h_s is the maximum number of messages sent from a vertex on *superstep* s.
- l is the cost of a *superstep*

3.6 Frameworks

One of the most popular BSP implementation is *Pregel*, from Google, and most of other BSP implementations follows its characteristics. Its API that allows the user to build his vertex implementation with its *compute* function. The framework has *generic* classes that allow computing store and send in the message any data.

Many frameworks try to optimize it. One of them is the Graph Processing System (GPS)(SALIHOGLU; WIDOM, 2013). Its major contributions to *Pregel* are:

- It allows the programmer to manipulate multiple vertices at the same time, where one vertex is the *master vertex* that can control the others.
- It optimizes the partitioning of the vertices among the nodes, allowing fewer messages to be exchanged among the machines.
- It can partition high-degree vertices over the nodes so that the computation of these vertices can also be parallelized.

3.7 Conclusion

This chapter describes features of the BSP model. We show a primary interface to its vertex, explaining it and an example of an algorithm using this model. We also show how can we have some primitive functions to improve the semantics of the model. Finally, we briefly show some frameworks that use this model.

The BSP model is simple to understand and very helpful on the distributed graph scenario, but, on situations that are expensive to exchange information between computer nodes, the BSP can lead to high execution costs.

4 GATHER, APPLY, SCATTER (GAS)

4.1 Introduction

The BSP model allows the creation of many frameworks to handle graphs in a distributed environment. The BSP requires that the information exchange occurs by message passing, and it can have a high cost. The GAS model addresses this problem.

GAS was initially presented in *PowerGraph*(GONZALEZ *et al.*, 2012) in 2012. Gonzalez proposed this new programming model based on the framework *GraphLab*(LOW *et al.*, 2012), that implements an alternative information exchanging to the BSP model.

The GAS model is an abstraction to BSP. The difference between them is how the computation occurs over the vertices in each *superstep*. This execution can be divided into three steps:

- **Gather**: the vertex gathers the information of all its neighborhood through the edges, without the need to directly receive a message from some vertex.
- **Apply**: after gathering all the information needed, the vertex compute the data to update its mutable value.
- **Scatter**: in the end, the computed data is sent to the edges, allowing that other vertices can gather that information on the next *superstep*.

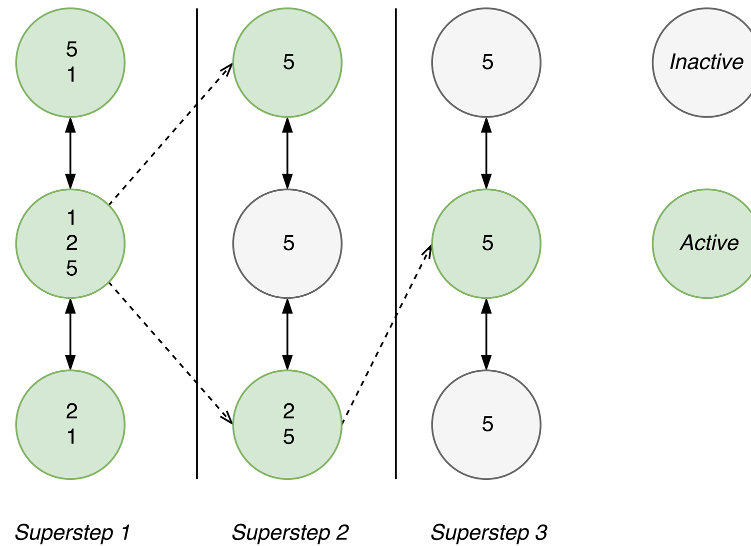
To understand better the GAS workflow, we retake the example from the BSP model. Figure 7 shows this example. The only difference is that, on the first *superstep*, the nodes already know the values of its neighbors. Then, the execution flows in the same way that on the BSP.

This chapter discusses the GAS model, following the same structure used in the last chapter. First, we present an abstract description of its vertex model and structure to the system. Then, we show some primitives to graph manipulation. For last, frameworks that implement this model.

4.2 GAS Model

The basic structure of a GAS vertex can be seen as the BSP one, but adding the three functions before mentioned(gather, apply and scatter), where the *apply* function is the *compute* function. The model in Algorithm 3 presents a model to the GAS vertex. One difference between this model and the BSP presented is the three function gather, apply and scatter, that represents

Figure 7 – GAS Higher Value Vertex Example



the steps mentioned above. They are private because, on the execution, the compute function is the one that will be called and, inside it, those three functions are going to be used.

Algorithm 3 GAS Vertex Interface

---

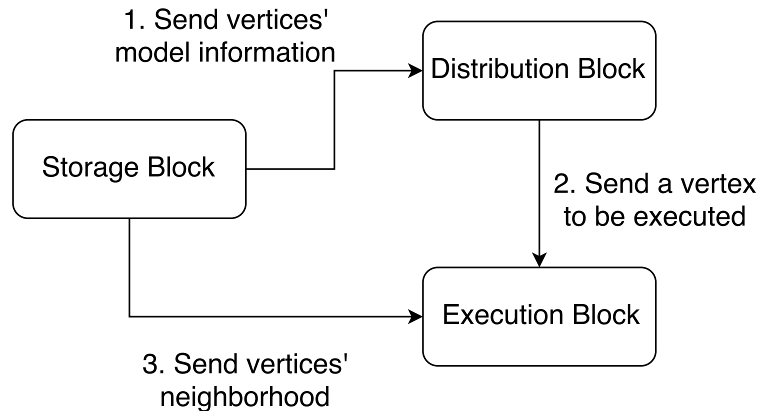
private Update gather*()*;
private Update apply*()*;
private void scatter*()*;
public void compute*()*;
public Value getValue*()*;
public void setMutableValue*(MutableValue m)*;
public MutableValue getMutableValue*()*;
public void halt*()*;

Other difference on this model is that we have an **Update** type on the gather and scatter functions. This type is responsible for representing the changes of the vertices' values over the graph. Messages on the BSP are an example of Update since they are responsible for carrying the information from the vertices.

The GAS model abstracts the way that the information will flow over the graph. This allows information to be passed by a message, file serialization, direct communication, shared memory, etc. Therefore, the block architecture is more straightforward than the BSP one, allowing the Message Block removal from the requirements. Figure 8 presents this block architecture and the workflow of a *superstep*.

The workflow on GAS is almost the same used in *BSP*: the Distribution Block takes the vertex information and send it to the Execution Block, that will execute the *gather*, *apply* and

Figure 8 – GAS Superstep Workflow



scatter functions.

Without the requirement to exchange messages, this model can become more efficient then BSP, since the information flow can be made inside a node, without the need to communicate with another, but it requires a more effort on programming since the optimization depends on the implementation of the information flow.

4.3 Primitive Functions

As presented in the last chapter, we can use some primitive function to improve the algorithm semantic on the GAS model. We can use the same 3 presented to BSP and add 2 more from those presented on (SALIHOGLU; WIDOM, 2014):

Collect Values

This primitive allows a vertex to collect information from its neighbors, like the *gather* function. So that, the vertex can group that information and store a new one inside it. This implementation can be made only using a *gather* and an *apply* functions.

Aggregate Vertices

The idea of this primitive is to merge some vertices into a *supervertex*, whose value is a representation of all the others. This function is helpful to apply some graph clustering partitioning algorithms. The user informs the criteria to join the vertices and how they are going to be merged.

4.4 Code example

On Algorithm 4 implements *Triangle Count* using the GAS model. Different from BSP model, GAS allows this algorithm to run in just one *superstep*. That happens because the common neighbors checking can be done at the beginning of the algorithm since the vertex already have information about its neighborhood.

Algorithm 4 GAS Triangle Count

```
1: for neighborsList in neighborhood do
2:     for vertex in neighborsList do
3:         if vertex in neighbors then
4:             triangles ← triangles + 0.5
5:         end if
6:     end for
7: end for
8: halt()
```

The algorithm execution is almost the same from the Algorithm 2 but is not needed more than one *superstep*: for each neighbor oh the vertex(line 1), the vertex will check for a common neighbor between them (lines 2 and 3). If so, the *triangles* value is increased by 0.5(this value is because each triangle will be counted twice). After that, the vertex enters in the inactive state.

4.5 Execution Cost

The execution cost of a *superstep* on the GAS model can be analyzed by the same BSP formula, but with some little changes. The formula and its variables are:

$$\sum_{s=1}^{S} w_s + g\sum_{s=1}^{S}(h_s + u_s) + Sl$$

- S is the number of *supersteps*;
- w_s is the longest vertex execution on *superstep* s;
- g represents the number of messages exchanged between the distribution and execution block;
- h_s is the maximum number of messages sent from a vertex on *superstep* s.
- u_s is the higher cost to update a neighborhood on *superstep* s
- l is the cost of a *superstep*

It is possible that, in a *superstep*, a node sends messages to other nodes, to update the neighborhoods (it all depends on how it will be implemented). The cost to update the neighborhoods must be counted independently if is by message passing or not, that is why u_s is used. That represents the higher cost to update the neighborhood.

4.6 Frameworks

GraphLab(LOW *et al.*, 2012) in an *open source* implementation of graphs with focus on to Data Science and Machine Learning. It has a *shared memory* feature that allows the vertices know its neighborhoods so that they can execute the *gather* and the *scatter* phases. The implementation is a little more complex than the *Pregel* one: it uses an *Update* structure to represent the information of the neighborhood. The execution of its algorithms happens over that data and the vertex data.

Power Graph(GONZALEZ *et al.*, 2012) is a framework that combines the best of *Pregel* and *GraphLab*, since they can have bad performances on high-degree vertices. It is the first framework to use the GAS officially model to factor the vertices programs over the edges, using the *shared memory* approach from *GraphLab* and the commutative and associative gather concept from *Pregel*. In other words, the information collected from the neighborhood can be gathered in any order.

4.7 Conclusion

In this chapter we described the GAS model, presenting its vertex structure, system organization, an algorithm execution, primitive functions and frameworks that use this model.

Like the BSP model, GAS can have a high cost on information exchange among the nodes, but the abstraction of this information flow can allow the implementation of some optimized techniques to do this exchange, what can reduce its cost.

5 OTHER MODELS

5.1 Introduction

There are some programming models very popular to deal with large datasets on the big data scenario, without the need to be a graph dataset. Since those models and its frameworks' are very used, some frameworks were developed over them to handle large-graphs.

(WU *et al.*, 2017) describes a set of programming models that are used in the *Big Data* scenario. It describes each model and makes comparison among them and their frameworks.

On this chapter, we briefly present the *MapReduce*, *Functional Programming* and *Actor* models, explaining how their work, how can we use them in graph context and present frameworks to them.

5.2 Map/Reduce

MapReduce is a Tuple Centric model proposed by Jeffrey Dean, from Google, on 2008(DEAN; GHEMAWAT, 2008). This model works dividing the function into two major categories:

- **Map**, where some computation is applied to each data unit. This computation returns an object containing, at least, one key and one value. This key represents the category of the resultant information.
- **Reduce** combines all the data with the same key, applying a function defined by the programmer. After mapping the data, the ones with the same key are grouped and condensed into one major information. The result or the reduce is a set of data, one for each unique key on the map stage.

A very common example of the *MapReduce* model is the word count. Assume that you want to count the number of occurrences of a word in a text. Using *MapReduce*, you could map the phrases, counting the number of times that a word appears, and them reduce it, for each word found, summing its occurrences.

When working with graphs, we can use the MapReduce model working as a data unity any element of graph's topology(an edge, a vertex or a subgraph). Each one of this units must contain all the data that they need, because no message is exchanged between them during the execution, except if some structure to broadcast the information or to access external data

is implemented, but with a computational cost. When using the vertex as the data unit, the algorithms became similar to the GAS model, but without the *scatter* phase.

Apache Hadoop(HADOOP, 2009) is the open-source implementation of Google's *MapReduce* paradigm, mainly implemented in Java. It has a *map* and a *reduce* interface so that programmers can implement their functions in order to process the data. From *Hadoop* we have *Pegasus*(KANG *et al.*, 2009): a graph mining framework that can handle graphs with *billions* of nodes and edges. It is constructed over the *Haddop* environment and uses a matrix-vector generalization to execute the algorithms.

5.3 Functional Programming

Functional Programming is a programming paradigm that has grown in the last years and is becoming very common in the programming languages. The importance of this paradigm was described since the 80's on (HUGHES, 1989) and (HENDERSON, 1980). One advantage of this paradigm is that everything is a function. We can pass a function as values and generate ones according to some input values.

This paradigm is emerging as a programming model for this generation on big data systems, due to frameworks like *Spark*(ZAHARIA *et al.*, 2010). This framework was build over *Hadoop* environment and provides functional interfaces to data manipulation, that is stored on its built-in model called Resilient Distributed Datasets (RDD). *Spark* was created to solve some limitations to the *MapReduce* model, giving a set of high-level function primitives.

When working with graphs, we can similarly use this model as *MapReduce*: we can define a topology element of the graph to represent the data unit and, over them, apply some function defined by the programmer. Here we also may not have a simple way to trade information between two data unities, so the data units should contain all the information that they need.

GraphX(XIN *et al.*, 2013) is a framework built over *Spark* to work with distributed graphs. It uses a structure of graph called Resilient Distribute Graph (RDG), that is based on *Spark*'s RDD. Because of *Spark*, this framework has some extra features like fault tolerance and easy scalability. With a few lines of code, *GraphX* implement *Pregel* API, allowing that algorithms made on this framework can be easily translated.

5.4 Actor

The *Actor* model is used for concurrent computation. It has a primitive computation unit called as *actor* which is responsible for reacting to events triggered by messages in different contexts. The following example presents a scenario to help explain this model:

"Suppose we have a family living in their home and the home need to be clean. The parents say the children to clean their respective bedrooms. When the kids finish cleaning their bedrooms, they can go out and play."

In this example, we have two kinds of *actors*: parent and child. When the parents say to the child to clean the bedroom, that is a message sent from the parents to the child, where we can see the "clean the room"as the action that will be executed, and it will be executed concurrently by each child. The "respective bedroom"represents that each child actor will have its domain, that is, a set of data to handle. When the children finish their works, they send a message to inform that and, after, they can go out and play. That is an example of a workflow using the *Actor* model.

One framework that is widely used and uses this model is *Akka*(THURAU, 2012). Typesafe developed its first release in 2009 and, although it allows the implementation of other models, it focuses on the *Actor* model. *Akka* uses asynchronous messages to communicate the *actors*, so there is no synchronization primitive. Also, its *actors* objects can run local and distributed, without the need to modify their logic.

To work with graphs, we can use the *Actor* model defining subgraphs to actors so that the operations are going to be executed inside each actor. This division allows that we can use this model to compose or create new programming models. For example, we can divide the graph into subgraphs, in each subgraph run a BSP algorithm and then apply a *reduce* function over it.

BLADYG(ARIDHI *et al.*, 2016) is a graph framework that uses the actor model to handle large dynamic graphs built over the *Akka* framework. Each *worker* actor on this framework is responsible to a *partition*(subgraph) of the graph, that will execute computations over it. There is a *master* actor that is responsible for coordinating the execution of the workers and processing the information computed by them.

5.5 Conclusion

This chapter presented some big data programming models, explaining how they operate, its features and frameworks that implement them.

We also saw how to apply those models on the context of distributed graphs and briefly presented frameworks that use those models and famous implementations of them, showing that big data models can also be used to help handle large-scale graphs.

6 MODEL COMPARISONS

6.1 Introduction

To better understand all the models presented in this work and how they resemble and diverge, we compare those models following some of the criteria presented in (REZENDE, 2017), so that we can be able to, given a scenario, chose an adequate model to use.

On this chapter, we first define the criteria, and then we will analyze the models based on them.

6.2 Criteria

Each model has its workflow with similarities in some cases. Following we present six features that we use to compare the models.

- **Partitioning Influence**: the distribution of the information among the nodes can have a critical influence on the performance of the algorithms because that can imply in more messages to be exchanged, what increases the cost. So, we may have models that the partitioning has a high or a low influence on the performance.
- **Computation Unit**: the approach to the computation unit can be vertex-centric, edge-centric or block-centric.
- **Execution**(**EX**): the execution can be synchronous(**S**), if there is a synchronization unit (like the *superstep* from the BSP) or asynchronous(**A**), otherwise.
- **Mutability**(**M**): it supports mutability if, during the execution of an algorithm, a new vertex, edge or a set of both is inserted, it will consider that without the need to restart the algorithm.

Table 3 – Models comparison

Model	**Partitioning Influence**	**Computation Unit**	**Execution**	**Mutability**
BSP	High	Vertex	Synch*	Yes
GAS	High	Vertex	Synch*	Yes
MapReduce	Low	Vertex /Edge /Block	Synch	No
Functional Programing	Low	Vertex /Edge /Block	Synch	No
Actor	High	Subgraph	Asynch	Can

6.3 Analysis of models

From Table 3 we report that the BSP and GAS are basically the same:

- The way that the vertices are distributed can imply on a significant information exchange over the nodes, increasing the cost of computation;
- Both are vertex-centric;
- They have the *superstep* as a synchronization unit. These models have some implementations like (WANG *et al.*, 2013), that make an asynchronous implementation of the BSP model to improve the convergence time of some algorithms and keep the simplicity and scalability. Also, *Graphlab*(LOW *et al.*, 2012) allows the execution in asynchronous way;
- If a change is made in the model, in the next *superstep* it will be sensed by the nodes and the new information will flow over the graph;

Based on that, if we have an environment with low cost to message exchange or the need to a keep changing the number of machines, the BSP and the GAS model can be very efficient. Otherwise, on scenarios with high message exchanging cost, those models may be costly.

About the *MapReduce* and *Functional Programming* models we can see that the distribution of the information would not have a significant impact on the execution. That happens because we need to take the information of every node to retrieve a final result and they work with an abstract data unity, allowing to use vertex and subgraphs. Also, if any information is added to the graph model, all the functions may be applied again to capture the modification. This situation exemplifies why we have not many frameworks to handle distributed graphs using these models.

MapReduce and *Functional Programming* models can be helpful if we need to compute some simple metrics over a graph, without the need of a more coordinated algorithm(like a shortest path, for example), since they do not have any communication among the computations and the operations can be applied over the vertices in any order.

The *Actor* model is the most different among the models presented here: it uses a subgraph as computation unit, it is asynchronous due to the events that are triggered by message passing and it can or not sense the mutability, that depends on how the application will be implemented.

The *Actor* model can be useful when we have a huge graph but we don't want to retrieve information from the whole graph. One example can be the roads network. If we want

the shortest path between two points in the same city in Europe, we do not need to access the information from South America. So, on this examples, we could have an actor responsible for each continent.

6.4 Conclusion

Here we made model comparisons, presenting how they behave according to some criteria. We conclude that there is a high similarity between BSP and GAS, and also between *MapReduce* and *Functional*. We also conclude that the *Actor* model is different from the others.

7 CONCLUSIONS AND FUTURE WORKS

This work presented some programming models for distributed graphs. We discussed how they work, explaining their workflow and structure, and introduced frameworks that use those models.

On the vertex-centric models, we presented BSP and GAS models. We discussed their features, explaining its workflow, how to structure a vertex on these models and wich components we may use to build a distributed system. We also show an algorithm to examplify these models and how to calculate its execution cost.

Some big data models were presented also. We briefly described *Functional Programming*, *Map/Reduce* and *Actor* models, presenting frameworks that uses those models and how to use them on graphs context.

We also compare the models, presenting some details that can help to choose a model to implement an application and exemplifying some scenarios that each model can be helpful. This shows that there is no best model to work with distributed graphs. All of them have their particularities and are useful in some contexts.

With this new knowledge, we can propose as future work an efficiency comparison among those models, to better define the scenarios to each model and also propose some optimization that must be done when implementing them. Also, this knowledge will be helpful to develop a distributed framework to handle large graphs but with one more feature: the information on the graph(vertex value, edges cost, etc.) can have different values in different moments. Those are named *time-dependent* graphs.

REFERENCES

ARIDHI, S.; MONTRESOR, A.; VELEGRAKIS, Y. Bladyg: A novel block-centric framework for the analysis of large dynamic graphs. In: ACM. **Proceedings of the ACM Workshop on High Performance Graph Processing**. [S.l.], 2016. p. 39–42.

BACKSTROM, L.; HUTTENLOCHER, D.; KLEINBERG, J.; LAN, X. Group formation in large social networks: membership, growth, and evolution. In: ACM. **Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.], 2006. p. 44–54.

BENNETT, J.; LANNING, S. *et al.* The netflix prize. In: NEW YORK, NY, USA. **Proceedings of KDD cup and workshop**. [S.l.], 2007. v. 2007, p. 35.

BOLDI, P.; SANTINI, M.; VIGNA, S. A large time-aware web graph. In: ACM. **ACM SIGIR Forum**. [S.l.], 2008. v. 42, n. 2, p. 33–38.

BONDY, J. A.; MURTY, U. S. R. **GRAPH THEORY WITH APPLICATIONS**. [S.l.: s.n.], 1982.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. **Communications of the ACM**, ACM, v. 51, n. 1, p. 107–113, 2008.

GONZALEZ, J. E.; LOW, Y.; GU, H.; BICKSON, D.; GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In: **OSDI**. [S.l.: s.n.], 2012. v. 12, n. 1, p. 2.

GUO, Y.; VARBANESCU, A. L.; IOSUP, A.; MARTELLA, C.; WILLKE, T. L. Benchmarking graph-processing platforms: a vision. In: ACM. **Proceedings of the 5th ACM/SPEC international conference on Performance engineering**. [S.l.], 2014. p. 289–292.

HADOOP, A. **Hadoop**. 2009. <http://hadoop.apache.org/>. [Online; accessed 06-December-2017].

HARA, C. S.; PORTO, F.; OGASAWARA, E. Tópicos em gerenciamento de dados e informações 2015. 2015.

HEIDARI, S.; SIMMHAN, Y.; CALHEIROS, R. N.; BUYYA, R. Scalable graph processing frameworks: A taxonomy and open challenges. 2017.

HENDERSON, P. **Functional programming: application and implementation**. [S.l.]: Prentice-Hall, 1980.

HUGHES, J. Why functional programming matters. **The computer journal**, Oxford University Press, v. 32, n. 2, p. 98–107, 1989.

JORDAN, M. Committee on the analysis of massive data, committee on applied and theoretical statistics, board on mathematical sciences and their applications, division on engineering and physical sciences, council, nr, 2013. frontiers in massive data analysis. **Frontiers in Massive Data Analysis**, 2013.

KANG, U.; TSOURAKAKIS, C. E.; FALOUTSOS, C. Pegasus: A peta-scale graph mining system implementation and observations. In: IEEE. **Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on**. [S.l.], 2009. p. 229–238.

KWAK, H.; LEE, C.; PARK, H.; MOON, S. What is twitter, a social network or a news media? In: ACM. **Proceedings of the 19th international conference on World wide web**. [S.l.], 2010. p. 591–600.

KYROLA, A.; BLELLOCH, G. E.; GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In: USENIX. [S.l.], 2012.

LOW, Y.; BICKSON, D.; GONZALEZ, J.; GUESTRIN, C.; KYROLA, A.; HELLERSTEIN, J. M. Distributed graphlab: a framework for machine learning and data mining in the cloud. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 5, n. 8, p. 716–727, 2012.

MALEWICZ, G.; AUSTERN, M. H.; BIK, A. J.; DEHNERT, J. C.; HORN, I.; LEISER, N.; CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In: ACM. **Proceedings of the 2010 ACM SIGMOD International Conference on Management of data**. [S.l.], 2010. p. 135–146.

REZENDE, C. A. d. **A component-oriented framework for large-scale parallel processing of big graphs**. Tese (Doutorado), 2017.

SALIHOGLU, S.; WIDOM, J. Gps: A graph processing system. In: ACM. **Proceedings of the 25th International Conference on Scientific and Statistical Database Management**. [S.l.], 2013. p. 22.

SALIHOGLU, S.; WIDOM, J. Help: High-level primitives for large-scale graph processing. In: ACM. **Proceedings of Workshop on GRAph Data management Experiences and Systems**. [S.l.], 2014. p. 1–6.

THURAU, M. **Akka framework**. 2012. <https://media.itm.uni-luebeck.de/teaching/ws2012/sem-sse/martin-thurau-akka.io.pdf>. [Online; accessed 06-December-2017].

VALIANT, L. G. A bridging model for parallel computation. **Communications of the ACM**, ACM, v. 33, n. 8, p. 103–111, 1990.

WANG, G.; XIE, W.; DEMERS, A. J.; GEHRKE, J. Asynchronous large-scale graph processing made easy. In: **CIDR**. [S.l.: s.n.], 2013. v. 13, p. 3–6.

WU, D.; SAKR, S.; ZHU, L. Big data programming models. In: **Handbook of Big Data Technologies**. [S.l.]: Springer, 2017. p. 31–63.

XIN, R. S.; GONZALEZ, J. E.; FRANKLIN, M. J.; STOICA, I. Graphx: A resilient distributed graph system on spark. In: ACM. **First International Workshop on Graph Data Management Experiences and Systems**. [S.l.], 2013. p. 2.

YAHOO WebScope. 2017. <https://webscope.sandbox.yahoo.com/catalog.php?datatype=g>. Accessed: 2017-12-04.

ZAHARIA, M.; CHOWDHURY, M.; FRANKLIN, M. J.; SHENKER, S.; STOICA, I. Spark: Cluster computing with working sets. **HotCloud**, v. 10, n. 10-10, p. 95, 2010.