Computing methods for parallel processing and analysis on complex networks

Luis Andrés Vázquez Benítez

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya - BarcelonaTech

A thesis submitted for the degree of

Master in Innovation and Research in Informatics (MIRI-HPC)

5 May 2015

Advisor: Eduard Ayguadé Parra, Computer Sciences Department Associate Director at BSC-CNS

Advidor: Dario García Gasulla, Postdoctoral Researcher at BSC-CNS

Acknowledgements

I want to thank to my family, friends and who were near to me in this period in Barcelona and for their advises that help me to face some barriers. It is important stand out the help of IPN and CONACyT and all my colleagues at MICROSE and AI lab. They did possible this thesis, also it would not have been possible without the support of the BSC and UPC institutions.

Special Thanks

- Eduard Ayguadé Parra
- Dario García Gasulla
- Ricardo Barrón Fernández
- José Luis Vázquez Moreno
- Graciela Benítez López
- Marco Antonio Ramírez Salinas

Abstract

Nowadays to solve some problems is required to model complex systems to simulate and understand its behavior.

A good example of one of those complex systems is the Facebook Social Network, this system represents people and their relationships, Other example, the Internet composed by a vast number of servers, computers, modems and routers, All Science field (physics, economics political, and so on) have complex systems which are complex because of the big volume of data required to represent them and their fast change on their structure

Analyze the behavior of these complex systems is important to create simulations or discover dynamics over it with main goal of understand how it works.

Some complex systems cannot be easily modeled; We can begin by analyzing their structure, this is possible creating a network model, Mapping the problem´s entities and the relations between them.

Some popular analysis over the structure of a network are:

- The Community Detection – discover how their entities are grouped

- Identify the most important entities – measure the node´s influence over the network
- Features over whole network like – the diameter, number of triangles, clustering coefficient, and the shortest path between two entities.

Multiple algorithms have been created to give a result to these analyses over the network model although if they are executed by one machine take a lot of time to complete the task or may not be executed due to machine limitation resources.

As more demanding applications have been appearing to process the algorithms of these type of analysis, several parallel programming models and different kind of hardware architecture have been created to deal with the big input of data, reduce the time execution, save power consumption and enhance the efficiency in the computation in each machine also taking in mine the application requirements.

Parallelize these algorithms are a challenge due to:

- We need to analyze data dependence to implement a parallel version of the algorithm always taking in mine the scalability and the performance of the code.
- Create a implementation of the algorithm for one parallel programming model like MapReduce (Apache Hadoop), RDD (Apache Spark), Pregel(Apache Giraph) these oriented to bigdata or HPC models how MPI + OpenMP , OmpSS or CUDA.
- Distribute the data input over the processing platform for each node or offload it into accelerators such as GPU or FPGA and so on.
- Store the data input and store the result of the processing requires techniques of Distribute file systems(HDFS), distribute NoSQL Data Bases (Object Data Bases, Graph Data Bases, Document Data Bases) or traditional relational Data Bases(oracle, SQL server).

In this Master Thesis, we decided create Graph processing using Apache bigdata Tools mainly creating testing over MareNostrum III and the Amazon cloud for some Community Detection Algorithms using SNAP Graphs with ground-truth communities.

Creating a comparative between their parallel computational time execution and scalability

Contents

Figures

Equations

Tables

Pseudo-code

1 Introduction

1.1 Motivation

The motivation of this thesis is to explore the graph processing challenge in large network models, the Networks models are in different areas from companies to research and the size of the Networks models increase rapidly. The necessity of parallel programming models, new cluster architectures, memories to store the Network´s data and algorithms are important to deal with this type of analysis.

1.2 Objectives

Community Detection is one common problem in graph processing where the entities are classified in groups.

The objectives of this Master Thesis are:

The execution of parallel Communities detection algorithms in large graphs, where mainly is important:

- Evaluate the SPARK and Giraph platforms performance using Apache TinkerPop with community detection algorithms on large graphs
- Implement DEMON, FluidC and LPA community detection algorithms with a parallel programming model.
- Evaluate the quality of the community detection algorithms.
- Detect the bottle-necks of different parallel community detection algorithms in large graphs.
- Create optimizations into the parallel community detection algorithm implementations to reduce the overhead in the execution time.

1.3 Contributions

The contributions of this thesis are:

- TinkerPop Community Algorithms
    - The algorithms were developed in JAVA using the TinkerPop API
- TinkerPop Configuration on Amazon EMR

    - The configuration of TinkerPop to use it with SPARK and Giraph
- Performance report of the Community Detection Algorithms
    - Comparative Performance between algorithms
    - Comparative quality in the Community Detection

2 Theoretical framework

2.1 Network Science

The **Network Science** is a field that is viewed as the study of the collection management, analysis, interpretation and presentation of relational data who is applied in many areas from Political Science, Psychology, Engineering, Sociology, Statistics, Communication, Management.

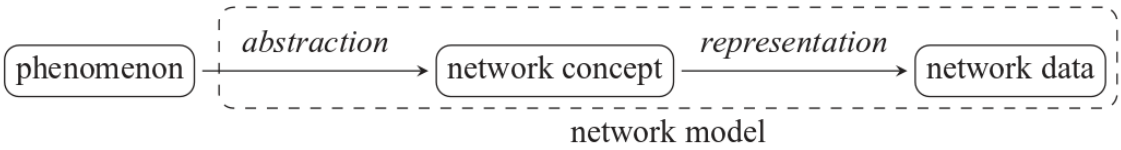


Figure 1 - The Elements of a network model by [1]

The network science uses **network theory** to analyze and model different phenomena; this is possible by diverse phenomena abstractions to Network models. The illustration 1 show a general view of the components to create a network model

The Network Model is viewed as a network representation of something where the network theory helps to create the network model or can deal with formalized aspects of network representation such as degree distribution, closure, communities, etc. [1]

2.1.1 Network Communities

Many model networks present nodes that are organized in densely linked groups that are called communities [3]. This groups represent entities that have some properties in common.

Into the graph theory one of the most popular analysis is the community detection who is to find how the vertices are grouped, also is called graph or network clustering that is an ill-defined problem because it does not have a universal definition of community and there is any clear-cut guideline on how to assess the performance of different algorithms and how

to compare them with each other, such ambiguity leaves a lot of freedom to propose diverse approaches to the problem [2].

There are many definition of graph communities one of them is [2]:

We have a subgraph C of a graph G, The number of vertices and edges are n, m for G and n_c, m_c for C respectively. The adjacency matrix of G is A, its element A_{ij} Equals 1 if vertices i and j are neighbors, otherwise it equals 0. They assume that the subgraph is connected because the communities usually are. Other types of group structures do not require connectedness.

The internal and external degree k_i^{int} and k_i^{ext} of a vertex i of the network with respect to subgraph C are the number of edges connecting i to vertices of C and the rest of the graph. Both definitions can be expressed in compact form via the adjacency matrix A

$k_i^{int} = \sum_{j \in C} A_{ij}$, $k_i^{ext} = \sum_{j \notin C} A_{ij}$

The degree of $k_i = k_i^{int} + k_i^{ext} = \sum_j A_{ij}$

- If $k_i^{ext} = 0$ and $k_i^{int} > 0$, i has neighbors only within C and is an internal vertex of C (dark green dots in the figure).
- If $k_i^{ext} > 0$ and $k_i^{int} > 0$, i has neighbors outside C and is a boundary vertex of C (bright green dots in the figure).
- If $k_i^{int} = 0$, i instead, the vertex is disjoint from C and is a boundary vertex of C (bright pink dots in the figure).

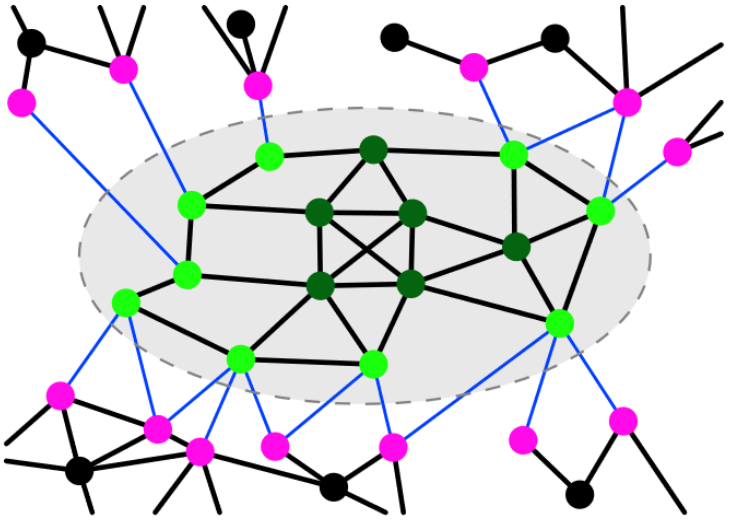


Figure 2 - Community in the network

The embeddedness ε_i is the ratio between the internal and the degree of vertex i :

$$\varepsilon_i = k_i^{int} / k_i$$

The larger ε_i the stronger the relationship between the vertex and its community.

The mixing parameter μ_i is the ratio between the external degree and the degree of vertex i : $\mu_i = k_i^{ext} / k_i$ = $\mu_i = 1 - \varepsilon_i$

The next table show variables for any subgraph:

Table 1 - Metrics for Communities

Unweighted networks			Weighted networks		
Name	Symbol	Definition	Name	Symbol	Definition
Internal degree	k_i^{int}	$\sum_{j \in C} A_{ij}$	Internal strength	w_i^{int}	$\sum_{j \in C} W_{ij}$
External degree	k_i^{ext}	$\sum_{j \notin C} A_{ij}$	External strength	w_i^{ext}	$\sum_{j \notin C} W_{ij}$
Degree	k_i	$\sum_j A_{ij}$	Strength	w_i	$\sum_j W_{ij}$
Embeddedness	ξ_i	$\frac{k_i^{int}}{k_i}$	Weighted embeddedness	ξ_i^w	$\frac{w_i^{int}}{w_i}$
Mixing parameter	μ_i	$\frac{k_i^{ext}}{k_i}$	Weighted mixing parameter	μ_i^w	$\frac{w_i^{ext}}{w_i}$

Table 2 - Metrics for Communities

	Unweighted networks			Weighted networks		
	Name	Symbol	Definition	Name	Symbol	Definition
Internal	Internal degree	k_C^{int}	$\sum_{i,j \in C} A_{ij}$	Internal strength	w_C^{int}	$\sum_{i,j \in C} W_{ij}$
	Average internal degree	$k_C^{avg\text{-}int}$	$\frac{k_C^{int}}{n_C}$	Average internal strength	$w_C^{avg\text{-}int}$	$\frac{w_C^{int}}{n_C}$
	Internal edge density	δ_C^{int}	$\frac{k_C^{int}}{n_C(n_C-1)}$	Internal weight density	$\delta_{w,C}^{int}$	$\frac{w_C^{int}}{\bar{w} n_C(n_C-1)}$
External	External degree	k_C^{ext}	$\sum_{i \in C, j \notin C} A_{ij}$	External strength	w_C^{ext}	$\sum_{i \in C, j \notin C} W_{ij}$
	Average external degree	$k_C^{avg\text{-}ext}$	$\frac{k_C^{ext}}{n_C}$	Average external strength	$w_C^{avg\text{-}ext}$	$\frac{w_C^{ext}}{n_C}$
	External edge density	δ_C^{ext}	$\frac{k_C^{ext}}{n_C(n-n_C)}$	External weight density	$\delta_{w,C}^{ext}$	$\frac{w_C^{ext}}{\bar{w} n_C(n-n_C)}$
Total	Total degree	k_C	$\sum_{i \in C, j} A_{ij}$	Total strength	w_C	$\sum_{i \in C, j} W_{ij}$
	Average degree	k_C^{avg}	$\frac{k_C}{n_C}$	Average strength	w_C^{avg}	$\frac{w_C}{n_C}$
	Conductance	C_C	$\frac{k_C^{ext}}{k_C}$	Weighted conductance	$C_{w,C}$	$\frac{w_C^{ext}}{w_C}$

The communities are dense subgraphs which are well separated from each other, this view have been challenged, due to communities may overlap as well, sharing some of the vertices. For instance, in social networks individuals can belong to different circles at the same time, like family, friends, work colleagues.

Many definitions of communities have been defined counting edges (internal, external) but other definitions have been focused on the probability that vertices share edges with a subgraph. The existence of communities implies that vertices interact more strongly with the other members of their community than they do with vertices of the other communities.

I only introduce to partition and overlapping communities. The partition is a community well defined that does not have nodes in common with other community, and the overlapping community has one or more nodes in common with other communities, the overlapping communities has more sense in social networks.

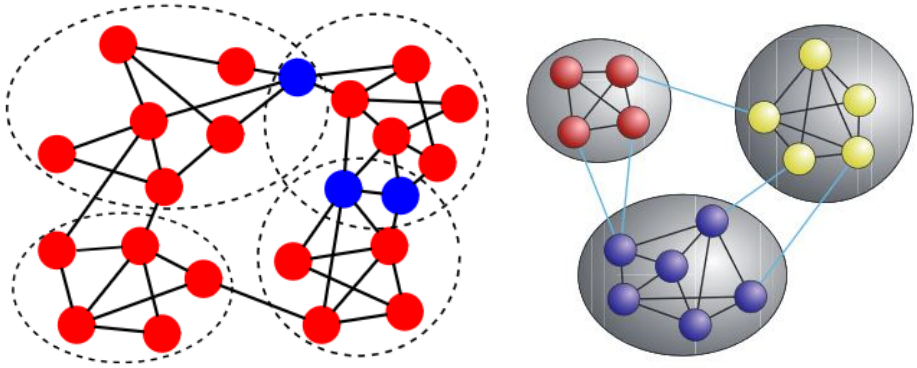


Figure 3 - Overlapping and partition communities

2.2 Parallel Computing

The parallel computing is important because we can simultaneously use multiple compute resources to solve a computation problem. Where the problem is broken into discrete parts that can be solved concurrently or parallel form [8].

There are two important reasons for using a parallel computer: to have access to more memory or to obtain higher performance. It is easy to characterize the gain in memory, as the total memory is the sum of the individual memories. The speed of a parallel computer is harder to characterize [46].

Exist a variety of parallel programming models to implement parallel algorithms. those models help as guide to implement efficient parallel code.

From [8] exists classifications of parallel computing depending of the context, and the propose of it:

Distribute computing: it is any computation that involves multiple computers, remote from each other, which have a role each in a computation problem.

High Performance computing(HPC) or Supercomputing: this type of computing executes a job as fastest as possible, in a supercomputing scenario the computer resources are an arbitrary number of computers connected by a fast network where each single computer has multiple processors and each processor has many cores.

Cloud computing: this type of computing, we can get access to resources for parallel computing, those resources are provided by an entity like Amazon EC2, Microsoft Azure and others, which give access to compute resources per hour and we can scale them as we needed. For example Amazon provide HPC, EMR, EC2, GPUs.[9]

Grid computing: this type of computing is popular at scientific community and the grid resources are provided by lots of different institutions.

Volunteer computing: this computing is a type of distribute computing in which computers owners donate their computer resources

To use those resources for parallel computing we need to implement the parallel algorithm over one parallel programming models.

2.3 Theoretical measures for Parallel Performance

From [46] there are some measures for expressing and judging the gain in execution speed from going to a parallel architecture.

2.3.1 Speedup and Efficiency

Speedup: let the same program run on a single processor, and on a parallel machine with p processors, and compare runtimes. With T_1 the execution time on a single processor and T_p the time on p processors, the speedup is defined as $S_p = T_1/T_p$ (sometimes T_1 is defined as 'The best time to solve the problem on a single processor'). The ideal case $T_p = T_1/p$.

Efficiency: To measure how far we are from the ideal speedup, they introduce the efficiency $E_p = S_p/p$; $0 < E_p < 1$

2.3.2 Amdahl´s law and Gustafson´s law

Amdahl´s law: One reason for less than perfect speedup is that parts of a code can be inherently sequential. This limit the parallel efficiency. Let F_s be the sequential fraction and F_p be the parallel fraction of a code. Respectively. Then $F_p + F_s = 1$. The parallel execution time $T_p$on p processors is the sum of the part that is sequential $T_1 F_s$ and the part that can be parallelized $T_1 F_p/p$:

$S_p = 1/(F_s + F_p/p)$.

Gustafson´s law: the speedup is now a function that decrease from p , linearly with p.

$S_p = p(p-1)F_s$.

2.3.3 Scalability

Scalability: splitting a given problem over more and more processors does not make sense: at a certain point, there is just not enough work for each processor to operate efficiently. Instead, in practice, users of a parallel code will either choose the number of processors to match the problem size, or they will solve a series of increasingly large problem on correspondingly growing numbers of processors, in both cases is hard to talk about speedup. Instead, the concept of scalability is used.

Strong scalability: the program shows strong scalability if, partitioned over more and more processors, it shows perfect or near perfect speedup.

Weak scalability: It describes that, as problem size and number of processors grow in such a way that the amount of data per processor stays constant, the speed in operations per second of each processor also stay constant.

3 State of the Art

3.1 Graph processing in real world

With the presence of companies like Linked-In, Facebook, Twitter, Google. The processing of large networks is fundamental to create analysis, this impulses to Facebook at 2015 to process a one Trillion of Edges [4] using Apache Giraph, in that paper they commented that Apache Giraph also is used to solve some iterative algorithms of Machine Learning in Facebook. Other example are network models used to detect frauds, Neo4j propose their Graph Data Base to detect frauds, searching into the graphs some patterns like rings [6]. Google created a model programming called PREGEL to parallel graph processing [7]. nowadays exist a strong trend to parallel graph processing.

3.2 Parallel Programming Models

3.2.1 PREGEL

PREGEL from [7] is a computational model created by google to deal with large graph Datasets due to the challenge that represent their processing in large scale. This model is a distribute execution and it is inspired by Valiant´s Bulk Synchronous Parallel Model where the computation is composed by sequence of iterations called supersets.

Each superset executes a function, this function is defined for each vertex at some superset.

At superset S each vertex can send messages to communicate with other vertices at Superset $S + 1$, and exists also aggregator and combiners.

The aggregators are a mechanism for global communication where each vertex can set a value that will be aggregate by one function and the result value will be able at the superset $S + 1$.

The combiners are functions to reduce messages before the messages will be send through the network.

Between supersets there is a barrier, these barriers ensures that all vertices finish their computation at superset S and receive all messages for it from other vertices.

Each vertex can have one of two states: “Active” or “Inactive”

The next diagram of state shows the vertex estate machine where each vertex changes of state when exist an action.

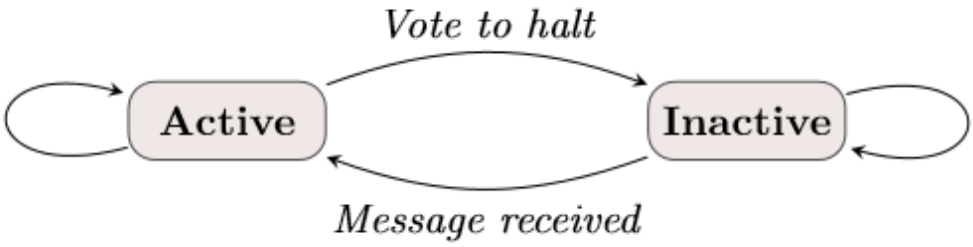


Figure 4 - Diagram of vertex states in PREGEL

The algorithm converges when all vertices in the computation at some superset are in Inactive state.

The next illustration represents the iteration of supersets where the dotted lines are messages and shaded vertices have voted to halt

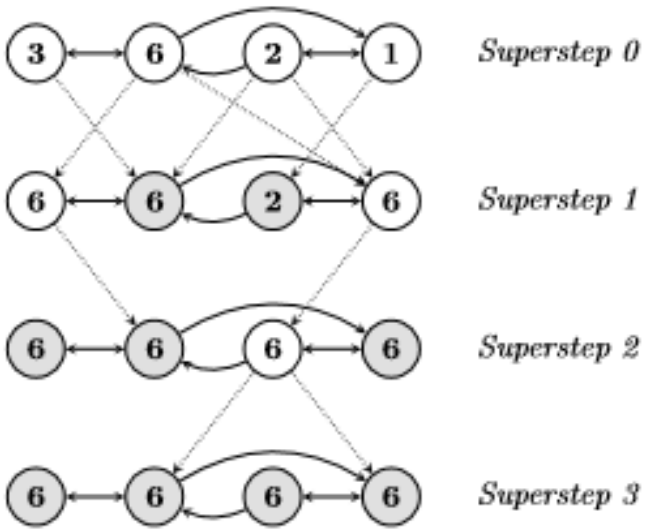


Figure 5 - PREGEL Supersets

3.2.2 Map Reduce

Map Reduce from [10] is a parallel model programming that can be having different implementations to get better performance. Google created MapReduce model programming in 2004 to execute their process with big amount of data.

This model programming run on large clusters of commodity machines and is highly scalable. This model is composed by two functions: the “mapper” and the “reducer” functions, also we can have an optional “combiner” function to reduce the number of messages and data that the mapper function send to the reducer function through the network.

In MapReduce model, we have a process called the master who has the responsibility to coordinate the execution and workers; The workers execute the mapper, (optional)combiner and reducer functions.

First, we need to split the input data among all workers using a partition function, the partition function can be given by the programmer, usually is a hash(key) mod R where R is the number of partition.

The image shows a general overview of the MapReduce model execution.

The input files are split, those chunks are sending to the workers entities, whose execute the “mapper” function then the partial result will be write to local disk to afterwards the workers whose execute the “reducer” function use the partial result to create the final result.

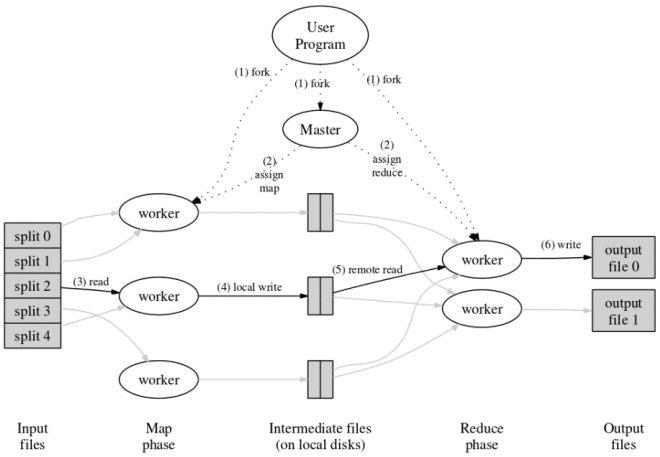


Figure 6 - overview MapReduce execution

For MapReduce, we need to map de information in pair key-value data to the mapper function and return a list of pair key-value to send to reduce function

The reduce function receive the list of key-values to give a result in form of list of values.

3.2.3 RDD

The Resilient distribute Dataset from [11] is a data model which perform the memory computation for iterative programs, this model has an advantage over MapReduce model due to the data reuse and let the programmer write parallel computation using a set of high-level operators.

The next illustration shows the transformations and actions over RDDs.

Table 3 - RDD´s transformations and actions

Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

RDD, transformations and actions let model different programming model like: Iterative version of MapReduce, PREGEL, PowerGraph and so on.

For Graph Computing RDDs were extended to create a Resilient Distribute Graphs a new data model to help create parallel graph processing. Which can implement the PREGEL or PowerGraph programming model over it. [12]

3.2.4 MPI

The message Passing interface from [11] is a programming model to distribute memory architectures, this model share information with messages that are sent through the network.

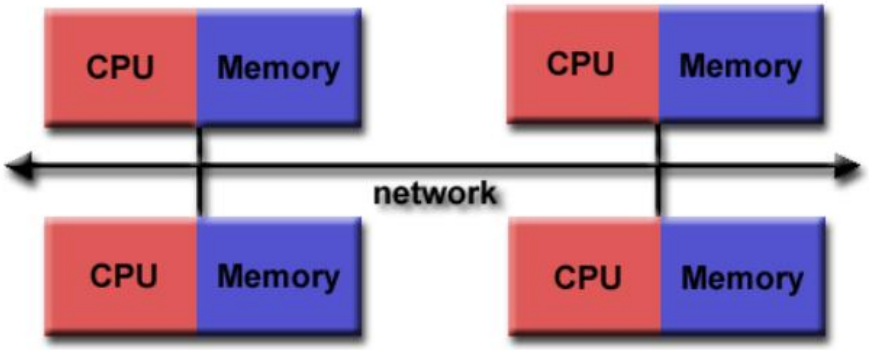


Figure 7 - MPI Architecture

3.2.5 MPI + OpenMP

This programming model from [11] is hybrid combination to exploit parallelism into a node with multiple cores or threads, each node shares local memory with their internal process and send information through passing messages to remote nodes.

This model increases the performance for locality memory

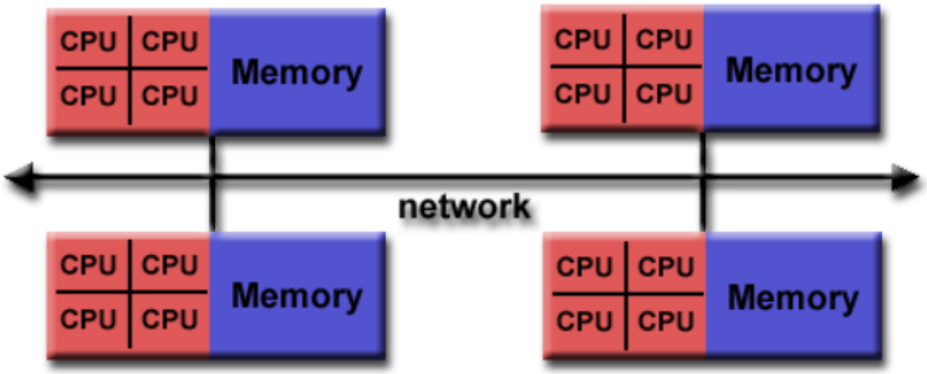


Figure 8 - MPI + OpenMP Architecture

3.2.6 CUDA

CUDA exploit the parallelism of many threads that are executed by cores into GPU. We can obtain extreme performance due to the number of cores able into GPU. CUDA programming only is able for NVIDIA architecture GPUs, like Volta, Pascal, Maxwell or Kepler Architecture.

CUDA programming model has gained area the last years, CUDA 8 lets us use Mixed precision into the program because of improves the performance on the NVIDIA Pascal Architecture, before it was able on GPUs but it was no able for the programmer just for internal process.[14]

The next illustration shows mixed precision datapath

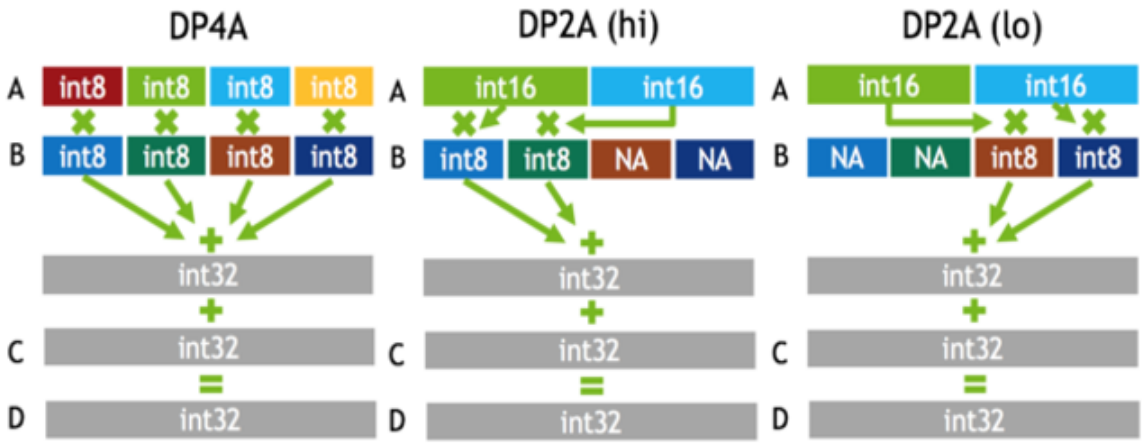


Figure 9 - Datapaths for CUDA 8 with NVIDIA Pascal Architecture

Other important feature in CUDA 8 is the unified memory that facility the code implementation extending the virtual address space to cover whole memory access of the host system. [15]

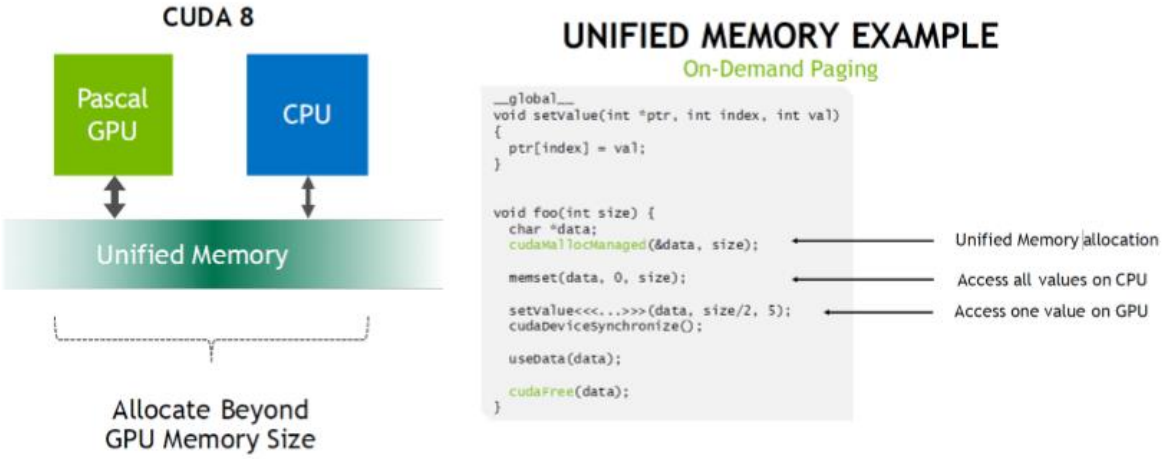


Figure 10 - Unified Memory in CUDA 8 and NVIDIA Pascal Architecture

3.3 Graph Storage

There are diverse ways to store graphs, we mainly found Graph Data Bases and Files with different formats that can be distributed in the cluster by Distributed File System like HDFS.

3.3.1 Graph Data Bases

The graph data base its fundamental for those applications that require to add or remove vertices, relations, vertex properties or edge properties into the storage graph, also create searches to retrieve small part of the graph.

Graph Data Bases are like Relational Data base in sense that they have a query language, transactions, some of them supports ACID operations, but the graph data base should have search engine dedicated to graph models, this search engine can be native or not native. this means that the graph search engine can be created using relational search engine or the search engine was specifically created for graph searching.

The Graph Data Base uses vertices and edges like entities, creating labels for each one and using transaction to add/remove vertices or edges from the graph[16]

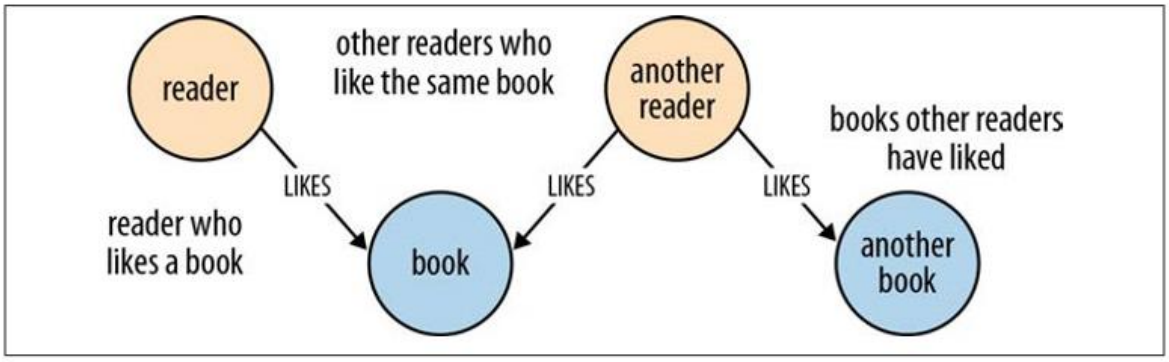


Figure 11 - Graph Data Base Data Model

3.3.2 Files

Different format files have been used to represent graphs, some of them are json(GraphSON)[19], xml(GraphML)[18], plain text that represent the graph as adjacent matrix or adjacent list, anothers is the serializarion of java objects like kryo files[17].

3.4 Challenge in parallel graph processing

The challenges in parallel graph processing from [20] mainly are produced by

- The data in graph problems are typically unstructured and highly irregular, the irregular structure of graph data makes it difficult to extract parallelism by partitioning the problem data. Scalability can be quite limited by unbalanced computational loads from poorly partitioned data.
- The relationships between entities may be irregular and unstructured, the computation and data access patterns tend not to have very much locality. The Performance in contemporary processors is predicated upon exploiting locality. Thus, high performance can be hard to obtain for graph algorithms even on serial machines.
- Graph algorithms are often based on exploring the structure of a graph in preference to performing large numbers of computations on the graph data. As a result, there is a higher ratio of data access to computation. Since these accesses tend to have a low amount of exploitable locality, runtime can be dominated by the wait for memory fetches.

The parallel applications scale well and have better performance when the problem to be solved, the algorithm, the software to express the algorithm and the hardware on which the software is run, are all well-matched [20].

In the case of graph processing the memory hierarchy creates an overhead due to the low locality of the neighbors in cache memory. This create high ratio of Fetch operations. Large graphs could use the memory on the HardDisk , this create a higher overhear therefore many applications use the process only in Memory RAM due to the random access pattern.

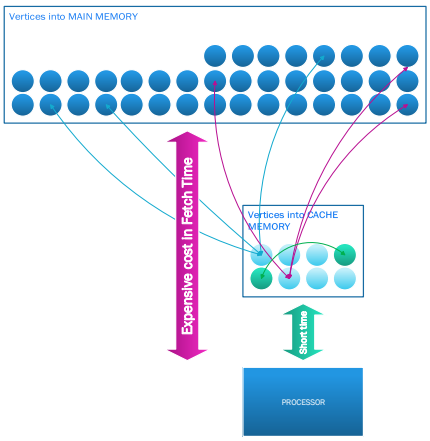


Figure 12 - low locality in cache memory in graph processing

For Architecture with many cores and share memory adds overhead to the memory hierarchy with the cache coherence protocol

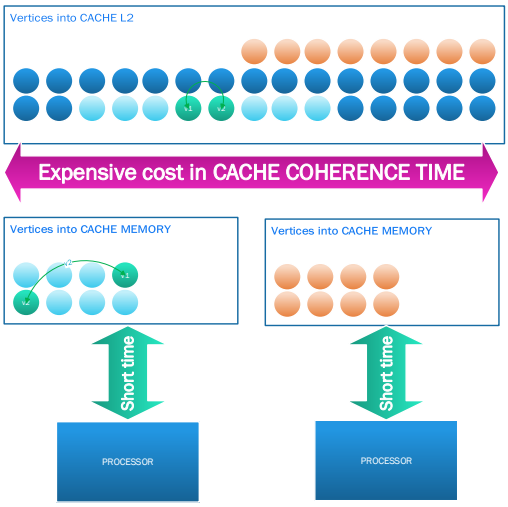


Figure 13 - Expensive cost in time due to cache coherence protocol in graph processing

The graph processing for distribute architectures, this architecture splits the data graph between a set of workers, it could produce unbalance of workload, depending of the algorithm to execute on this architecture, in the state of the art , there is not an optimal way to create partitions of the graph, each partition needs to know the context of the whole graph with adjacent list of neighbors vertices for each vertex, this produce a unbalance of memory consumption on the workers also to communicate information between vertices, if the two vertices are in different workers, it is necessary send messages ,this adds overhead for the communication Networks.

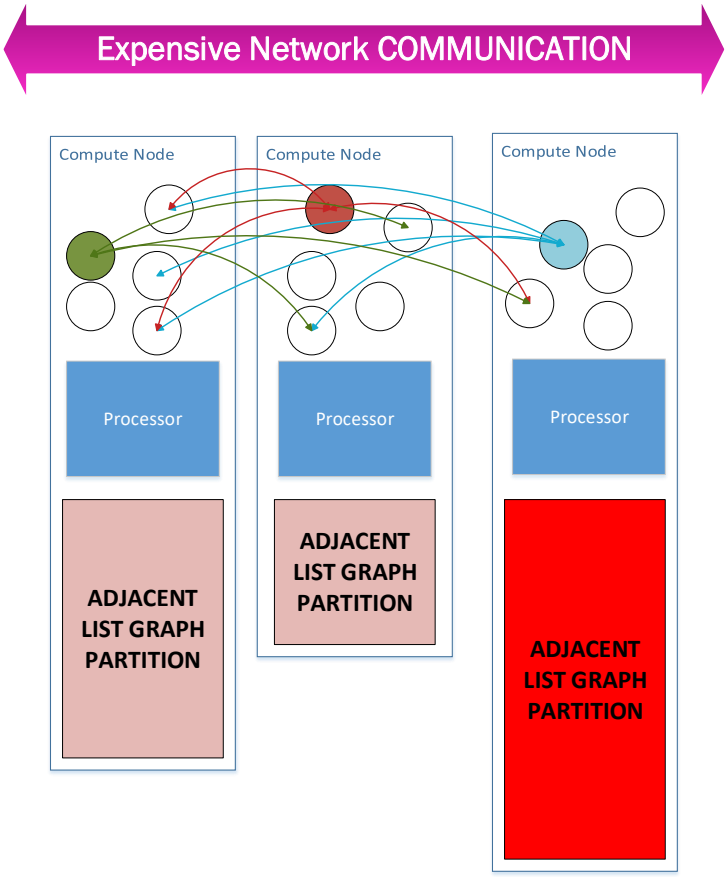


Figure 14 - expensive cost in time for low locality in graph partition

3.5 Benchmark for Large-Scale Graphs

In graph processing, the performance depends not only on the processing platform, also the workload, the algorithm being executed and the graph data itself [22].

The benchmarks help to measure the performance of different platforms

Benchmarking big-data graph processing platforms take in mind:

- The algorithm should be meaningful for real-world processing.
- Stress the choke points of the system under test
  The data set should be representative of real-world graphs also be suitable for processing on systems of different scales.
- A system that can accommodate new graph-processing platforms or environments

The next sections show some benchmarks for different platforms and systems.

3.5.1 Graphalytics

Graphalytics from [22] is a graph benchmark for graph processing platforms,

The Graphalytics data generator is:

- the LDBC Social Network Benchmark(SNB) data generator(Datagen)[25]

Graphalytics supports five graph algorithms:

- Breadth-first search.
- The connected components
- Community Detection
- Graph Evolution
- General statistics

Graphalytics supports graph-processing platforms:

- Hadoop MapReduce
- MapReduce Version 2 which runs on top of the Hadoop YARN
- Giraph
- GraphX
- Neo4j

The metric to measure the performance is Edges per Second(EPS)

3.5.2 LDBC Graphalytics

LDBC Graphalytics from [21] is an industrial-grade benchmark for graph analysis platforms.

The LDBC Graphalytics data generator is built from:

- LDBC SNB
- Graph500

LDBC Graphalytics supports six graph algorithms:

- Breadth-first search
- PageRank

- Weakly Connected Components
- Community Detection using LPA
- Local Clustering Coefficient
- Single-source shortest path

LDBC Graphalytics supports:

- Platforms
    - Different programming models and models, including vertex-centric, gather-apply-scatter and sparse matrix operations.
- Systems
    - Distributed systems
    - Multi-core single node systems
    - Many core GPU systems
    - Hybrid CPU – GPU systems
    - Distributed hybrid systems.

The Graph scale in LDBC Graphalytics is given by the equation 1 as follow:

Equation 1 - scale of a graph in LDBC Graphalytics

$$s(V, E) = \log_{10}(|V| + |E|)$$

Where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges in the graph.

LDBC Graphalytics groups datasets scales into classes, the classes are labeled according to the familiar system of "T-shirt sizes": small(S), medium(M), and large(L), with extra(X) prepended to indicate smaller and larger classes to make extremes.

The next table show the scales:

Table 4 - LDBC Graphalytics groups datset scales into classes

Scale	<7	[7.,7.5)	[7.5.,8)	[8.,8.5)	[8.5.,9)	[9.,9.5)	≥ 9.5
Label	2XS	XS	S	M	L	XL	2XL

The metrics to measure the performance are Edges per Second(EPS) or Edges plus Vertices per Second(EVPS)

3.5.3 Graph500 Benchmark

The Graph500 Benchmark from [23] is a large-scale benchmark for data-intensive supercomputers applications.

The Graph500 data generator is a Kronecker generator and receive two parameters:

- The $SCALE$: The logarithm base two of the number of vertices
- and the $edgefactor$: The ratio of the graph´s edge count to its vertex count

Equation 2 - Graph500 number of vertices for data generator

$$N = 2^{SCALE}$$

Where N is the total number of vertices.

Equation 3 - Graph500 number of edges for data generator

$$M = edgefactor * N$$

Where M is the number of edges

The next table shows some problem classes in graph500:

Table 5 - Graph500 problem classes

Problem class	Scale	Edge factor	Approx. storage size in TB
Toy (level 10)	26	16	0.0172
Mini (level 11)	29	16	0.1374
Small (level 12)	32	16	1.0995
Medium (level 13)	36	16	17.5922
Large (level 14)	39	16	140.7375
Huge (level 15)	42	16	1125.8999

The algorithm for graph500 benchmark

- Breadth-search first

The metric to measure the performance is Traversed edged per second (TEPS)

The next table show the three firsts positions on the graph500 challenge at November 2016.

Table 6 - The Graph500 List, November 2016

The Graph 500 List

November 2016

No.	Rank	Machine	Installation Site	Number of nodes	Number of cores	Problem scale	GTEPS
1	1	K computer (Fujitsu - Custom)	RIKEN Advanced Institute for Computational Science (AICS)	82944	663552	40	38621.4
2	2	Sunway TaihuLight (NRCPC - Sunway MPP)	National Supercomputing Center in Wuxi	40768	10599680	40	23755.7
3	3	DOE/NNSA/LLNL Sequoia (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)	Lawrence Livermore National Laboratory	98304	1572864	41	23751

3.6 Graph Computing tools

Nowadays we have many tools that we can use to large-scale graph processing, some of them are for High Performance Computing and others for Big-Data Analytics

We can group these tools in two classes

- On-Line Analytical Processing (OLAP) for those tools that can process the whole graph
    - Apache Giraph [5]:
        - This tool implements a PREGEL programming model.
        - It is an open source tool
        - It Works over YARN or HADOOP V1 cluster
        - Use internal Zookeper [35] service for message administration
    - Apache Hadoop [31]:
        - This tool implements MapReduce Model
        - There are two versions: HADOOP V1 and HADOOP V2; HADOOP 2 improve some features for scaling and performance.
        - It is an open source tool
    - Apache Spark [34]:
        - This tool implements RDD model
        - It can use a spark cluster or YARN cluster for distribute computing, also local machine executions

        - it uses different language implementations like JAVA, R, SCALA, and others.
    - NVIDIA NvGraph [36]
        - It is a NVIDIA library that use NVIDIA GPUs to graph processing
        - It includes three algorithms (Page Rank, Single Source Shortest Path and Single Source Widest Path)
    - Apache TinkerPop [26]:
        - This tools is an integrator of tools
        - It uses PREGEL model for graph processing, using YARN server for Spark , Giraph or MapReduce executions.
        - It uses HDFS [31] as distribute file system
        - Also it can integrate different graph data bases
    - ScaleGraph [32]:
        - This tool is built with PGAS programming language called X10
        - It implements PREGEL programming model
- On-Line Transaction Processing (OLTP) for those tools that can retrieve, search, aggregate, update information in some part of the graph
    - Apache Titan[28]
        - This tool is a distribute graph Data Base that can be configured with different tools for example: ElasticSearch[27] for indices and Apache Cassandra[30] for graph storage.
    - Neo4j [29]:
        - This tool is a graph Data Base with native graph search engine.
    - IBM SystemG [33]:
        - This tool is a suite of tools for graph storage, graph processing and graph visualization.

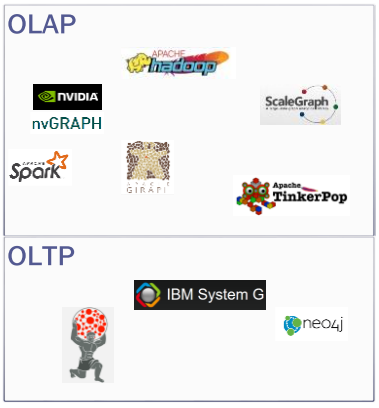


Figure 15 - Graph processing tools

3.7 Graph Processing Trends on parallel platforms

Many research has been created using different graph processing tools for measure the performance with different Hardware platforms and graph algorithms.

One of them uses k-Computer supercomputer for execute the Hybrid Breadth-First Search Algorithm with different Graph Matrix Formats to measure the performance using a graph generated by Kronecker, they obtained a performance of 38,621.4 GTEPS using 82,944 nodes and 663,552 cores with a Scale 40 problem in 2016[37].

Another used TSUBAME 2.0 supercomputer to analyze the Twitter Network data using 469.9 million users and 28.7 billion relationships, where this analysis reported that the value of degree of separation is 4.59 [38].

ScaleGraph has been tested on many scale graphs as Scale 29 for Synthetic and realistic graphs with PageRank, Breadth-First Search and other algorithms, also ScaleGraph shows better performance in comparison with similar tools like Apache Giraph or PBGL due to it is more efficient in memory utilization [41].

The next table shows the Strong-scaling performance on RMAT SCALE 25, the algorithm is no specified in the paper:

Table 7 - Strong-Scaling Performance on RMAT SCALE 25 for ScaleGraph,Giraph and PBGL

Nodes	ScaleGraph (s)	Giraph (s)	PBGL (s)
1	158.9	-	-
2	85.0	-	966.8
4	44.9	2885.1	470.3
8	23.4	443.1	309.5
16	13.3	125.3	290.9

The performance study for Graph Storage from [40] shows that the best performance is for native graph data bases (Neo4j and SystemG), the next are NoSQL data bases like Titan configured with Berkeley and HBase storage and in the last position are for Relational data bases.

The next table show the execution time of a query to traverse the graph implemented in different graph storage platforms are the next:

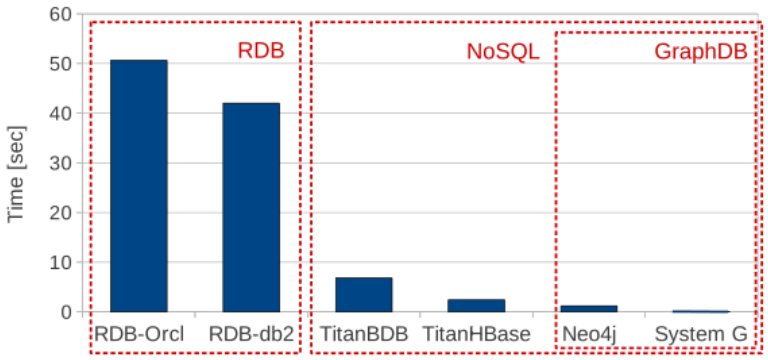


Figure 16 - Graph Storage performance

LDBC Graphalytics Benchmark study from [21] shows that Spark and Giraph are the slowest platforms for graph computing executing different graph algorithms as Breadth-First Search(BFS), Page Rank(PR), Weakly connected components(WCC), Community Detection using Label Propagation(CDLP), Local Clustering Coefficient(LCC) and Single-source shortest paths(SSSP). This was executed in one node of DAS-5 supercomputer.

The next image shows the result of the test with *S* and *L* Scale graph of LDBC Graphalytics

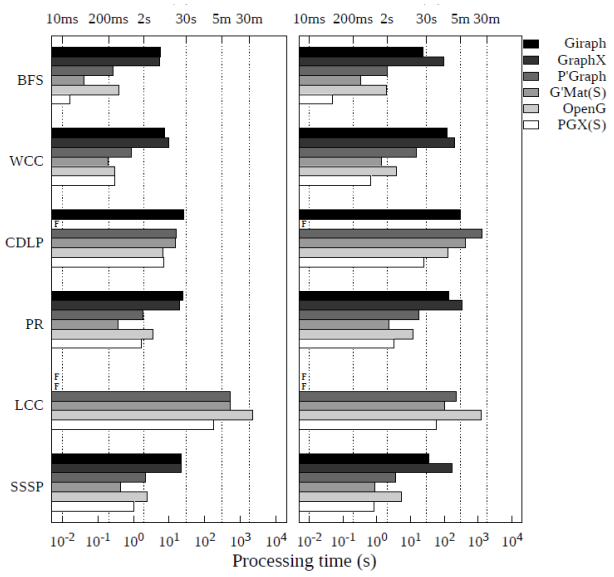


Figure 17 - Parallel platforms performance with 1 node

Also, Spark and Giraph have more expensive execution time than others graph platforms in vertical scalability and horizontal scalability.

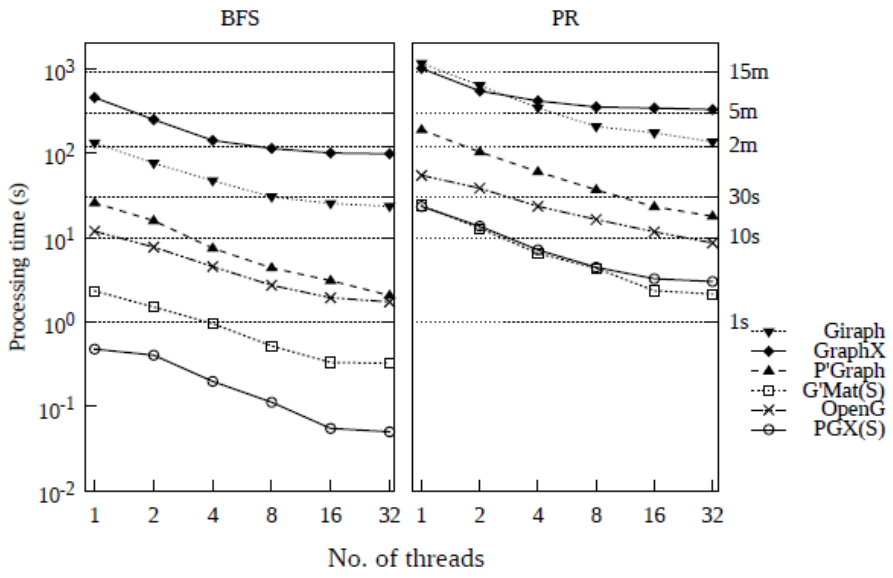


Figure 18 - Vertical scale for graph processing tools

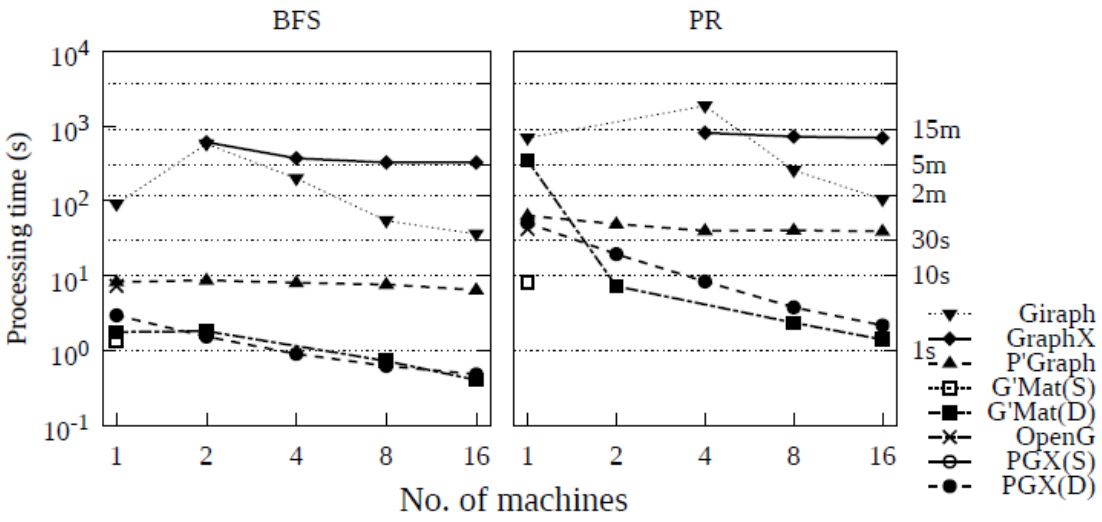


Figure 19 - Horizontal scale for graph processing tools

An interesting discover was that Most platforms fail on graph500 graph, but success on a Datagen graph of comparable scale. This indicates sensitivity to graph characteristics other than graph size.

The next table show the smallest scale graph where fail the platform to complete the execution, Platforms successfully process D1000 with scale 9.0 but fail on G26 of the same scale

Table 8 - smallest scale graph where fail the platform

Platform	Giraph	GraphX	P'graph	G'Mat(S)	OpenG	PGX(S)
Dataset	G26(XL)	G25(L)	R5(XL)	G26(XL)	R5(XL)	G25(L)
Scale	9.0	8.7	9.3	9.0	9.3	8.7

3.8 Community Detection on graphs

To extract communities from given undirected network, one typically chooses a scoring functions that quantifies the intuition that communities correspond to densely linked sets of nodes.

There exist multiple structural definitions of networks communities, the formalization of community detection lead to NP-hard problems, and the lack of reliable ground-truth makes evaluation extremely difficult [3].

From [42], a study was created where there is a comparison of different community detection algorithms using ground-truth graphs to validate the quality of the detection in different scale graphs,

The quality of the community detection is measured using the recall and precision methods, the next table show the average recall and precision for different community detection algorithms and graphs with ground-truth.

Table 9 - Average Recall and Average precision for different Community detection algorithms and graphs

	Clique P		Conclude		Copra		Demon		Ganxis		Grd CE		Info		InfoS		LinkC		Louvain		Oslom	
	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P
lfr	0.92	0.47	0.91	0.82	0.95	0.98	0.28	0.36	0.05	0.45	1.00	1.00	1.00	1.00	1.00	1.00	0.52	0.20	0.98	0.90	1.00	1.00
karate	0.48	0.34	0.80	0.50	0.64	0.51	0.71	0.71	0.94	0.63	0.89	0.89	0.80	0.63	0.80	0.63	0.50	0.30	0.68	0.48	0.97	0.97
football	0.91	0.76	0.86	0.77	0.87	0.94	0.30	0.40	0.81	0.87	0.95	0.90	0.95	0.90	0.95	0.90	0.89	0.35	0.95	0.92	0.97	0.86
polbooks	0.60	0.31	0.46	0.25	0.55	0.71	0.37	0.67	0.67	0.62	0.52	0.66	0.60	0.49	0.60	0.49	0.64	0.13	0.64	0.41	0.63	0.63
polblogs	0.67	0.10	0.80	0.24	0.63	0.94	0.57	0.43	0.97	0.96	0.94	0.94	0.91	0.15	0.91	0.15	0.59	0.01	0.87	0.29	0.91	0.29
dpd	0.24	0.15	0.19	0.26	0.05	0.35	0.02	0.24	0.16	0.33	0.04	0.27	0.30	0.24	0.24	0.24	0.53	0.10	0.26	0.27	0.18	0.27
as-caida	0.08	0.04	0.37	0.19	0.01	0.10	0.01	0.17	0.17	0.12	0.11	0.32	0.58	0.11	0.58	0.12	0.55	0.02	0.50	0.09	0.49	0.20
pgp	0.58	0.41	0.43	0.50	0.34	0.57	0.23	0.61	0.47	0.51	0.24	0.49	0.57	0.46	0.45	0.51	0.67	0.31	0.57	0.47	0.37	0.55
anobii	0.37	0.35			0.07	0.50	0.04	0.26	0.02	0.62	0.23	0.35	0.19	0.26	0.17	0.27	0.44	0.28	0.09	0.36	0.20	0.28
dblp	0.57	0.23			0.32	0.26	0.13	0.21			0.34	0.25	0.38	0.20	0.28	0.23	0.52	0.15	0.41	0.18	0.34	0.24
amazon	0.52	0.46			0.38	0.51	0.33	0.52			0.38	0.50	0.48	0.47	0.37	0.53	0.61	0.34	0.49	0.53	0.37	0.69
flickr													0.16	0.28	0.15	0.27			0.12	0.42	0.19	0.29
orkut															0.01	0.16			0.00	0.23		
lj-backstrom													0.14	0.26	0.10	0.27			0.09	0.38	0.18	0.26
lj-mislove													0.10	0.25	0.06	0.24			0.07	0.30	0.14	0.25

3.9 Graph Processing Research Direction

Dr. Toyotaro Suzumura say that one of the research directions in graph processing is to consider how we can design and implement a unified system that tightly couples a distributed graph data management platform together with distributed graph processing, and also programming models for this coupled system [39].

4 Solution

The problem to solve in this Thesis are:

- Evaluate the performance of different platforms for graph processing with community detection:

  In this thesis, I evaluated the performance of big data tools Apache SPARK and Apache Giraph using the Apache TinkerPop integrator of tools for Graph Processing on the cloud of Amazon

- Implement some community Detection Algorithms in one parallel programming model.

  The algorithms implemented in this Thesis are: DEMON, FluidC and Label propagation

- Detect the bottle-necks of different parallel community detection algorithms in large graphs.

  I am presenting a profiling using Apache Ganglia to get the performance on the whole cluster also instrument the code to measure the execution time of the code.

- Create optimizations into the parallel community detection algorithm implementations to reduce the overhead in the execution time.

  I create some modification to increase the performance and identifying the principal parameters of Hadoop.

- Evaluate the quality of the community detection.

  I compared the quality of the community detection algorithm using graphs with ground-truth and getting the precision and recall like [42]

4.1 Architecture application for graph processing

In the state of the art to process large graph in commodity cluster and most popular tools are Giraph and Spark, some version of Apache SPARK and Apache Giraph work with YARN cluster. Apache SPARK implements the RDD data model and Apache Giraph implements the BSP programming model, this thesis proposes evaluate these tools for big data processing.

TinkerPop 3 is an integrator of tools for graph processing, Tinkerpop has multiple graph data bases implementations to store the graph. Tinker Graph, Hadoop Graph, Apache Titan and Neo4j.

Also, Apache TinkerPop 3 can integrate analytics big data tools to process the graph like Apache Giraph, Apache Hadoop and Apache SPARK. TinkerPop API implement the vertex centric programming model and BSP programming model to implement algorithms.

In this thesis, I implement DEMON, LPA and FluidC in TinkerPop API to evaluate the performance of Apache Giraph and Apache SPARK for big data processing.

In this case we propose use the next Stack API for graph processing.

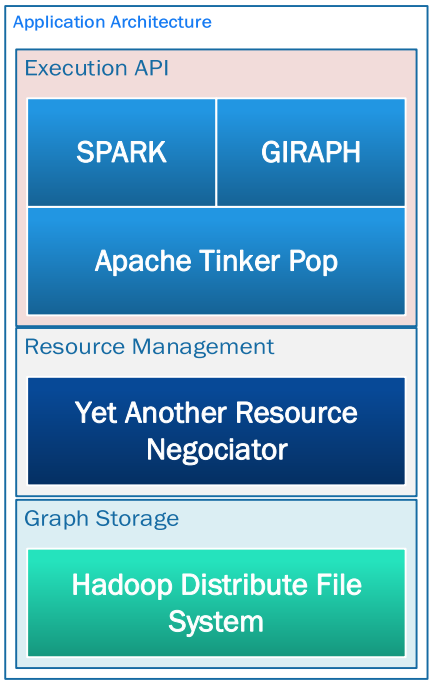


Figure 20 - Stack API

The previous stack API has the next propose:

- Apache SPARK and Apache Giraph are tools for graph processing that implement different parallel programming models.
- Apache TinkerPop is the API to implement some community detection algorithms in Vertex Centric programming model and this API implement interfaces for Apache SPARK and Apache Giraph, this advantage let us execute the same code in both platforms.
- YARN is the resource manager in the Hadoop cluster.
- And HDFS is the distribute file system to store the large graph.

4.1.1 Hadoop Distribute File System, TinkerPop 3 and graph storage

HDFS:

From [47] HDFS is a filesystem designed for strong very large files with data access patterns, running on cluster of commodity hardware.

Very Large files: in this context means files that are hundreds of megabytes, gigabytes or terabytes or petabytes in size.

HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time.

Hadoop does not require expensive, highly reliable hardware. It is designed to run on cluster of commodity hardware for which the chance of node failure across the cluster is high. At least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

TinkerPop 3:

TinkerPop 3 implements a Hadoop Graph on HDFS where use a file in adjacent list format to store the graph. This file can be a java serialization format kryo or a custom file [26].

To use Hadoop Graph is necessary install the next plugin in the machine where the Tinkerpop 3 console is running.

Hadoop-Gremlin

```
<dependency>
   <groupId>org.apache.tinkerpop</groupId>
   <artifactId>hadoop-gremlin</artifactId>
   <version>3.2.4</version>
</dependency>
```

4.1.2 Yet Another Resource Negotiator and TinkerPop 3

From [47] YARN is Hadoop´s cluster resource manager system. YARN was introduced in Hadoop 2 to improve the MapReduce implementation, but it is general enough to support other distributed computing paradigms as well.

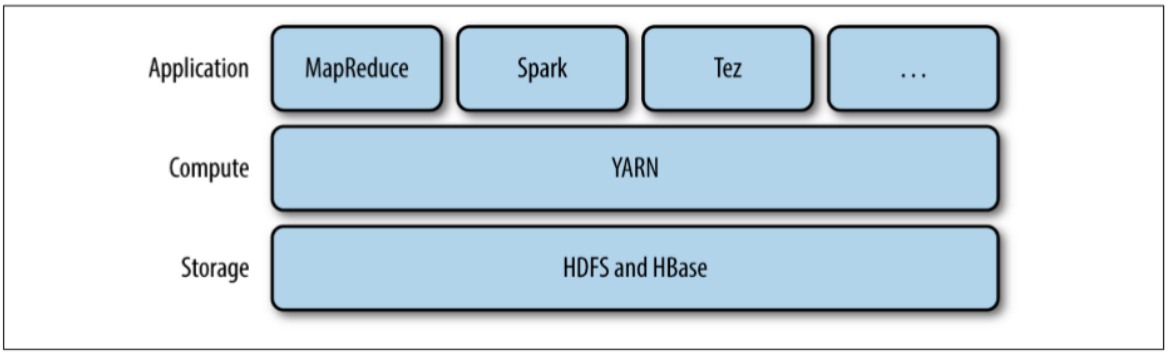


Figure 21 - YARN

YARN provide its core services via two types of long-running daemon: a resource manager to manage the use of resources across the cluster, and node manager running on all the nodes in the cluster to launch and monitor containers. A container executes an application-specific process with a constrained set of resources.

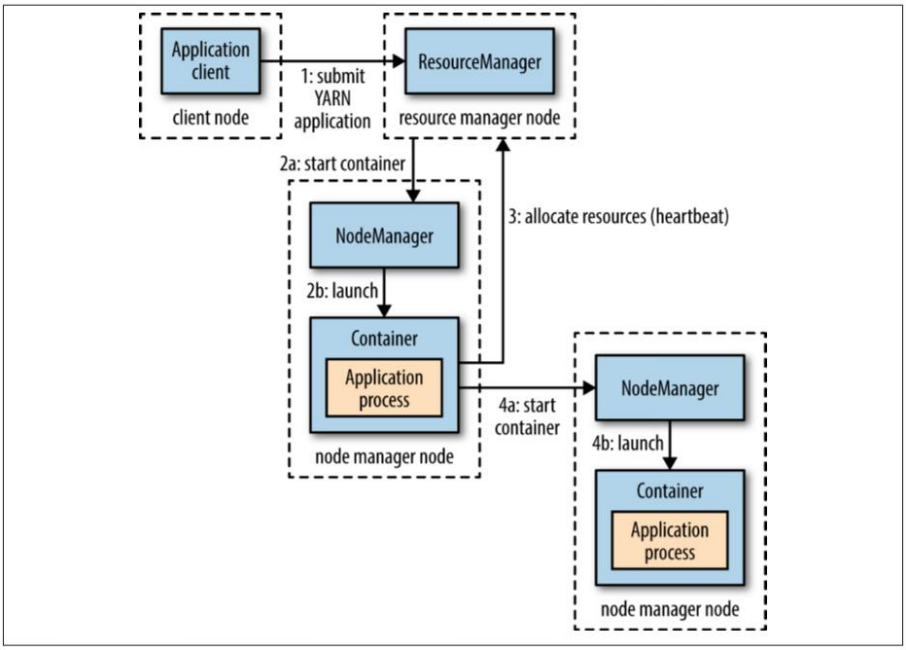


Figure 22 - YARN Architecture

From [26] Tinkerpop 3 uses a YARN server to submit the application. This application support apache SPARK and Apache Giraph models for algorithms implemented on Tinkerpop 3 due to provides interfaces to switch the platform.

To use Spark or Giraph interfaces is necessary install the next plugins in the machine where the Tinkerpop 3 console is running:

SparkGraphComputer

```
<dependency>
   <groupId>org.apache.tinkerpop</groupId>
   <artifactId>spark-gremlin</artifactId>
   <version>3.2.4</version>
</dependency>
```

GiraphGraphComputer

```
<dependency>
   <groupId>org.apache.tinkerpop</groupId>
   <artifactId>giraph-gremlin</artifactId>
   <version>3.2.4</version>
</dependency>
```

4.1.3 Apache SPARK, Giraph and Tinkerpop 3

From [26] TinkerPop 3 provides interfaces to implement a VertexProgram: A VertexProgram can be thought of as a piece of code that is executed at each vertex in logically parallel manner until some termination condition is met.

A submitted VertexProgram is copied to all the workers in the graph, A worker is not an explicit concept in the API, but is assumed of all GraphComputer implementations (SPARK, Giraph), At minimum each vertex is a worker.

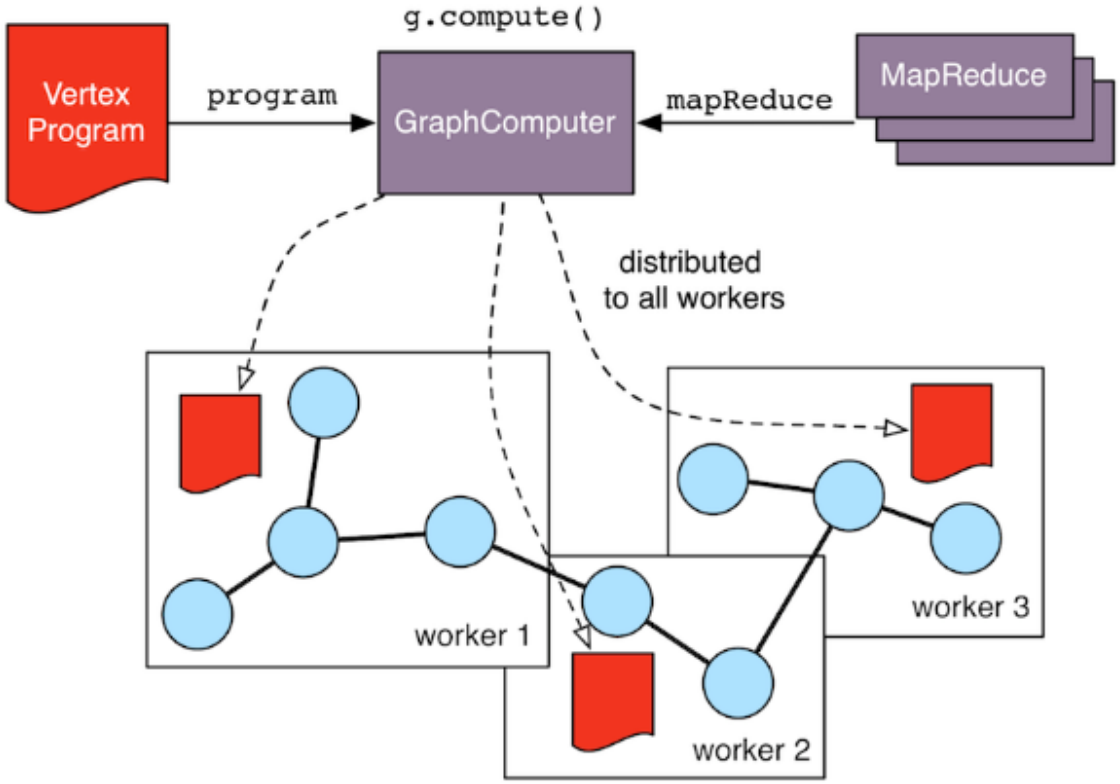


Figure 23 - TinkerPop Graph Computer

The program is executed in BSP fashion, where vertex can communicate information through messages at each iteration before the barrier Synchronization.

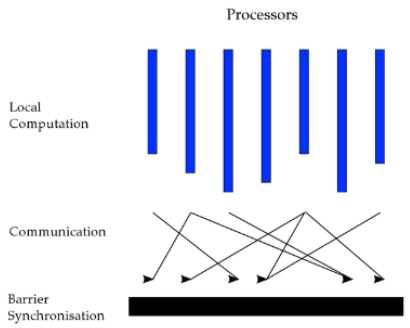


Figure 24 - Parallel execution in the TinkerPop Graph Computer

4.1.4 Software

The next version Software is used to this Final Master Thesis

Table 10 - Software

Software	Version	License
Hadoop	2.7.1	Apache
TinkerPop 3	3.2.4	Apache 2
Spark	1.6.1	Apache
Giraph	1.1.0	Apache
Gephi	0.9.1	Open
NetbeansIDE	8.2	Open
Java	8	Open

In marenostrum III, there is a software that manages the compute resources, also it is consider

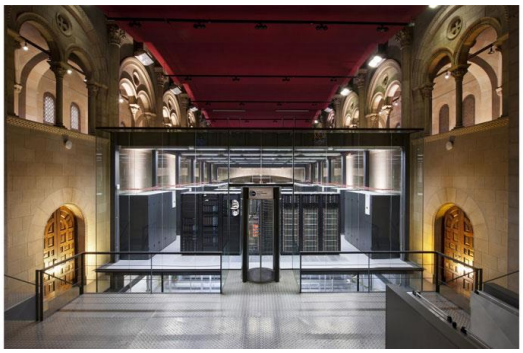
Software	Version	License
LSF	9.1.2	IBM

4.2 Platforms

To evaluate the scalability of the parallel algorithms, I will implement the stack API in two platforms:

- The Marenostrum III
- The Cloud of Amazon EMR.

4.2.1 Marenostrum III



Marenostrum III is a supercomputer with Performance of 1.1 Petaflop.

from [8] generally speaking a supercomputer is a single system that is typically comprised of tens of thousands of CPUs, coupled together with some high-performance interconnection network that allow them to communicate to each other very fast. Those thousands of CPUs can all be used in some combined way to work on a single problem.

The central components of the hardware part of a supercomputer are the compute nodes, including a Master (log-in) nodes and Worker (compute) nodes. Some nodes compose the storage system, shared by all the compute nodes. All these nodes are interconnected by different interconnection networks, including computation traffic, file system traffic, administration traffic, etc.

The general view of Marenostrum III: share disk architecture (parallel file system), all computes nodes has local disk of 500GB and 2 sockets.

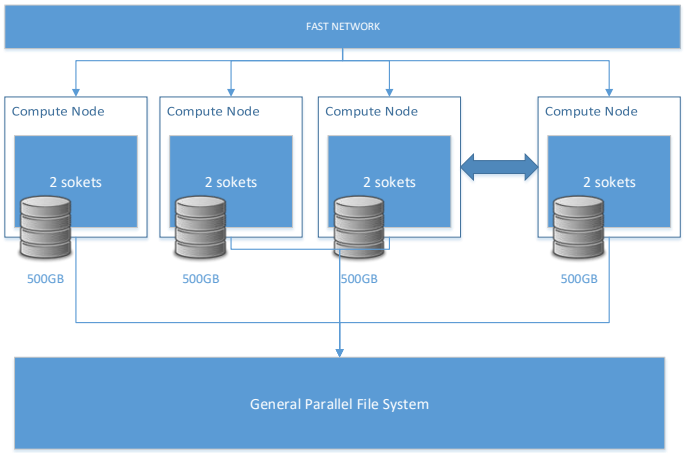


Figure 25 - general architecture of Marenostrum III without the log-in nodes

From [8] the platform LSF (Load Sharing Facility) software is leading enterprise class software that distributes work across existing heterogeneous IT resources creating a shared, scalable, and fault-tolerant infrastructure, delivering faster, more reliable workload performance while reducing cost. LSF balances load and allocates resources, while providing access to those resources. LSF provides a resource management framework that takes your job requirements, finds the best resources to run the job, and monitors its progress. Jobs always run according to host load and site policies.

4.2.2 Amazon EMR

Amazon EMR Architecture with M4.Large is a share nothing architecture, this service is provided by the Amazon Cloud, Each compute node has an Elastic Block Storage.

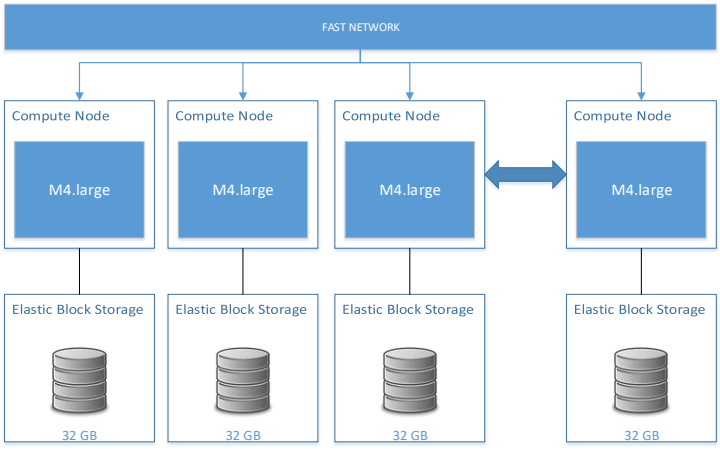


Figure 26 - Amazon EMR General architecture

4.3 Configuration

This Section is for the configuration of the Stack Software in the platforms previously described.

4.3.1 Configuration Hadoop and TinkerPop

From [47] , the next files are necessary for the configuration of hadoop

- hadoop-env.sh - Environment variables that are used in the scripts to run Hadoop
- mapred-env.sh - Environment variables that are used in the scripts to run MapReduce
- yarn-env.sh - Environment variables that are used in the scripts to run YARN
- core-site.xml - Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS, MapReduce, and YARN
- hdfs-site.xml - Configuration settings for HDFS daemons: the namenode, the secondary namenode, and the datanodes
- mapred-site.xml - Configuration settings for MapReduce daemons: the job history server
- yarn-site.xml - Configuration settings for YARN daemons: the resource manager, the web app proxy server, and the node managers
- slaves - A list of machines (one per line) that each run a datanode and a node manager
- hadoop-metrics2 .properties - Properties for controlling how metrics are published in Hadoop
- log4j.properties - Properties for system logfiles, the namenode audit log, and the task log for the task JVM process

for TinkerPop [26], we need create a file with the properties where we can set the configuration of Hadoop graph and Spark or Giraph platform.

For example:

```
# hadoop-grateful-gryo.properties

#
# Hadoop Graph Configuration
#
gremlin.graph=org.apache.tinkerpop.gremlin.hadoop.structure.HadoopGraph
gremlin.hadoop.graphReader=org.apache.tinkerpop.gremlin.hadoop.structure.io.gryo.GryoInputFormat
gremlin.hadoop.inputLocation=grateful-dead.kryo
gremlin.hadoop.outputLocation=output
gremlin.hadoop.jarsInDistributedCache=true

#
# SparkGraphComputer Configuration
#
spark.master=local[1]
spark.executor.memory=1g
spark.serializer=org.apache.tinkerpop.gremlin.spark.structure.io.gryo.GryoSerializer
```

It is important that the CLASSPATH environmental variable references to HADOOP_CONF_DIR and create the environment variable HADOOP_GREMLIN_LIBS which

it has to reference to the locations of the libraries of SPARK, Giraph and java programs, for example:

```
export HADOOP_GREMLIN_LIBS=/usr/local/gremlin-console/ext/giraph-gremlin/lib
```

4.3.2 Marenostrum III, Hadoop and TinkerPop

To configure Hadoop in a supercomputer as Marenostrum III, I face some challenge due to Marenostrum LSF does not have an integration with Hadoop jobs.

From [49], it has proven hard for Hadoop to co-exist with HPC resources management systems, since Hadoop provides its own scheduling and manages its own job and tack submissions and tracking. Since both systems are designed to have complete control over the resources that they manage. It is a challenge to enable Hadoop to co-exist with traditional batch systems such that users may run Hadoop jobs on these resources.

Hadoop uses a share nothing architecture whereas traditional HPC resources typically use a share disk architecture, with the help of high performance parallel file system.

in the marenostrum III and Hadoop we need configure a dynamic Hadoop configuration for each job on LSF due to this, we obtain a multi-level scheduling- LSF and YARN.

The next image shows the Stack of software for Marenostrum III and Hadoop integration.

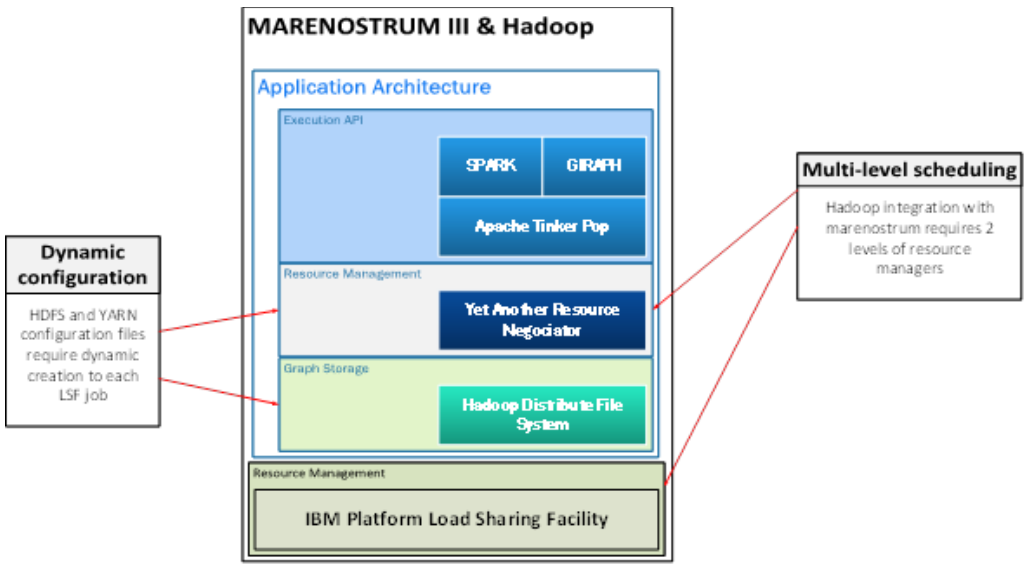


Figure 27 - Stack API integration with Marenostrum III

4.3.3 Marenostrum III, Hadoop, TinkerPop, Giraph

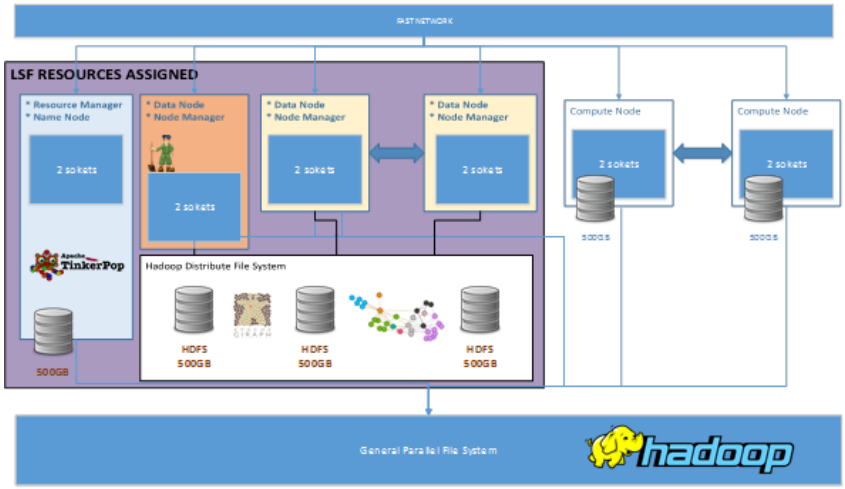
This configuration requires
1 compute node for:

- the Resource Manager
- Name Node
- Execute the TinkerPop console

n number of nodes for:

- Data Node
- Node Manager
- One of the n nodes has to run Zookeeper

The configuration is the next:



4.3.4 Marenostrum III, Hadoop, TinkerPop, SPARK

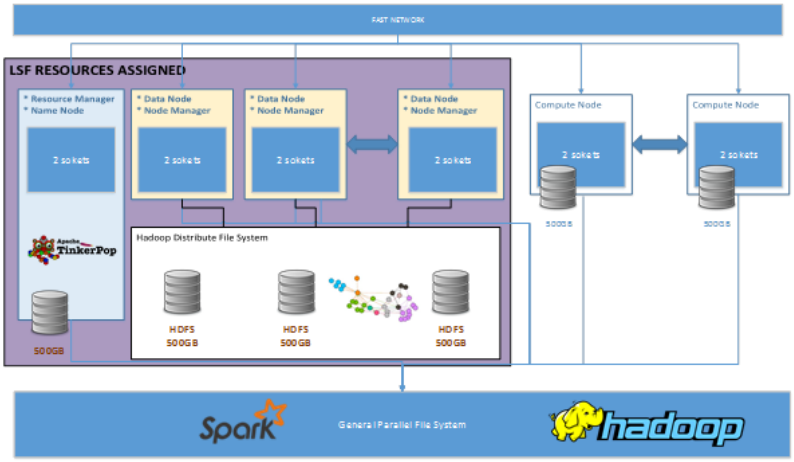
This configuration requires
1 compute node for:

- the Resource Manager
- Name Node
- Execute the TinkerPop console

n number of nodes for:

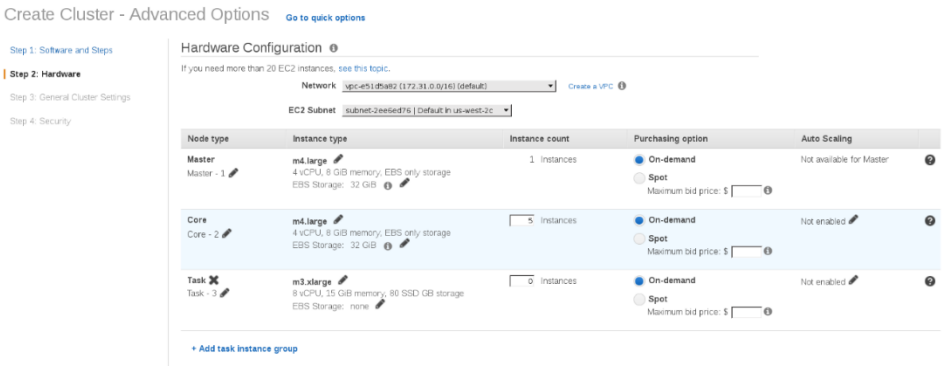
- Data Node
- Node Manager

The configuration is the next:

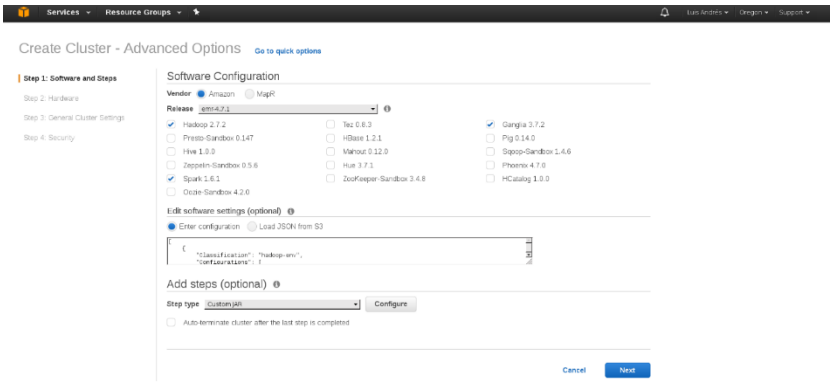


4.3.5 Amazon EMR, Hadoop and TinkerPop

In Amazon EMR we can select the hardware that we need to the computation of the problem. This hardware cloud scale in horizontal form, and we can select from a specific configuration on the Amazon cloud, for this thesis I select the m4.large instance type.

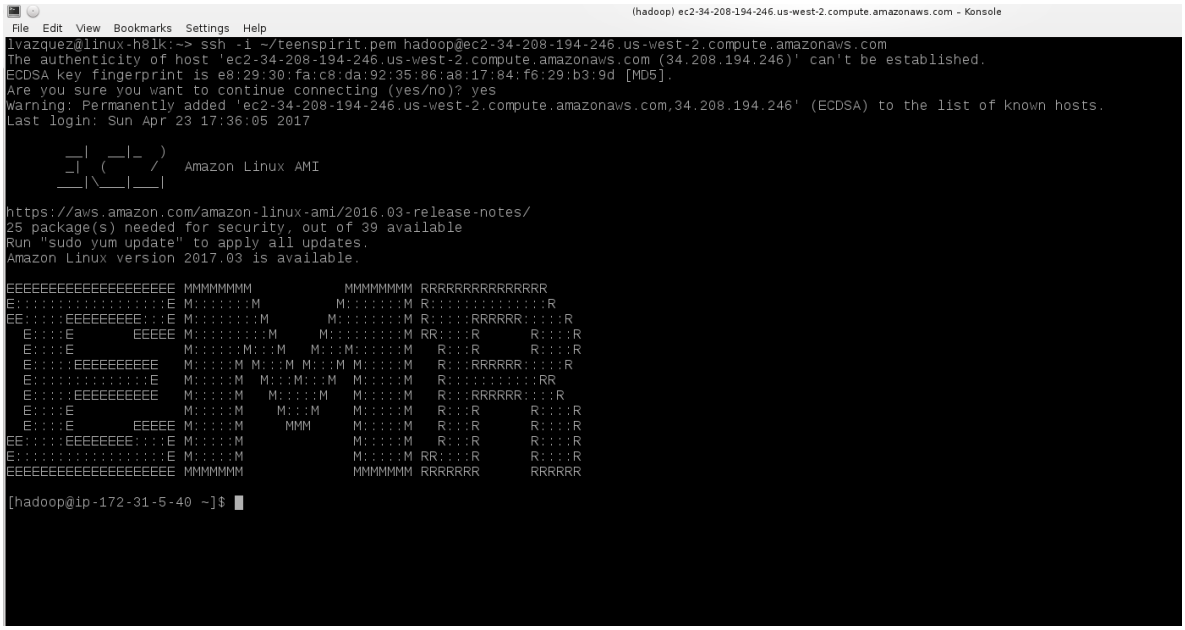


Also, we can select the Stack API to install in the cluster from a predefined compatible versions List.

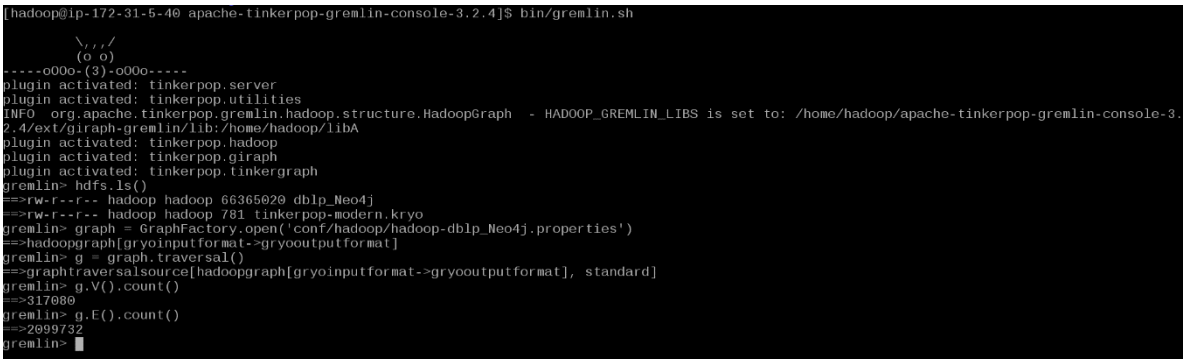


To access to the cluster, we have a public IP and remote connection using SSH , also we can upload file using SCP command.

Example of log-in in Amazon EMR



Tinkerpop console in Amazon EMR



In Amazon EMR the Software Stack is the next:

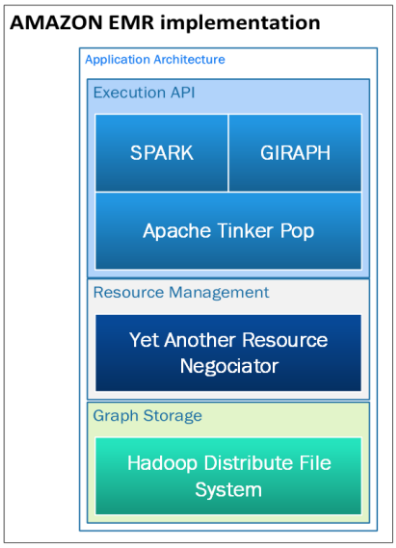


Figure 28 - Stack API in Amazon EMR

4.3.6 Amazon EMR, Hadoop, TinkerPop, SPAR

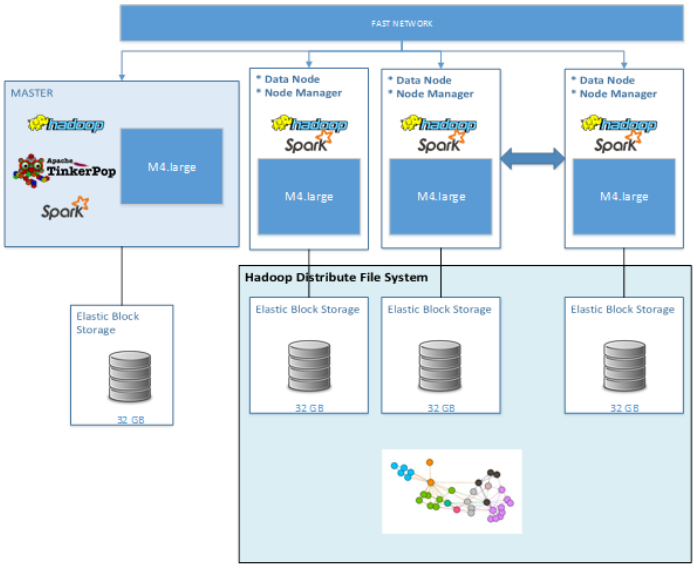
This configuration requires
1 m4.large node for:

- the Resource Manager
- Name Node
- Execute the TinkerPop console

n number of m4.large nodes for:

- Data Node
- Node Manager

The configuration is the next:



4.3.7 Amazon EMR, Hadoop, TinkerPop, Giraph

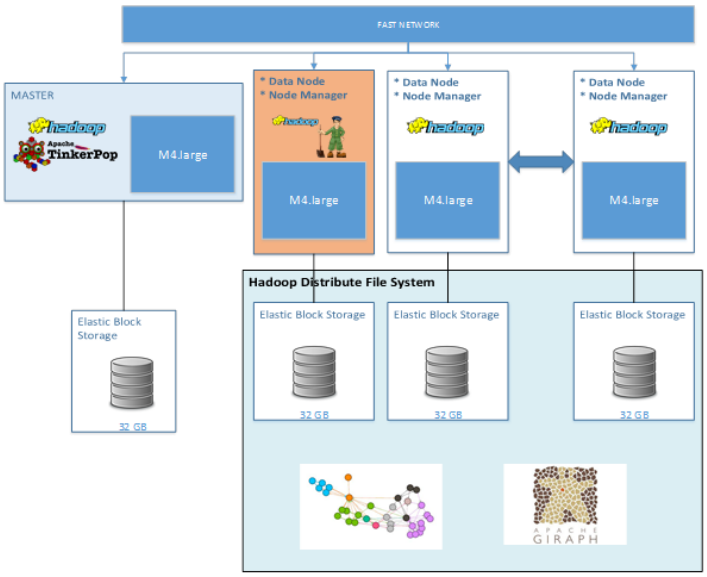
This configuration requires
1 m4.large node for:

- the Resource Manager
- Name Node
- Execute the TinkerPop console

n number of m4.large nodes for:

- Data Node
- Node Manager
- One of the n nodes has to run Zookeeper

The configuration is the next:



4.4 Graphs with ground-truth

To evaluate the parallel community detection algorithms in this master thesis I use the SNAP large graphs with ground truth. The next table show different SNAP graphs with ground-truth

Table 11 - Graphs with Ground-truth

Networks with ground-truth communities

Name	Type	Nodes	Edges	Communities	Description
com-LiveJournal	Undirected, Communities	3,997,962	34,681,189	287,512	LiveJournal online social network
com-Friendster	Undirected, Communities	65,608,366	1,806,067,135	957,154	Friendster online social network
com-Orkut	Undirected, Communities	3,072,441	117,185,083	6,288,363	Orkut online social network
com-Youtube	Undirected, Communities	1,134,890	2,987,624	8,385	Youtube online social network
com-DBLP	Undirected, Communities	317,080	1,049,866	13,477	DBLP collaboration network
com-Amazon	Undirected, Communities	334,863	925,872	75,149	Amazon product network

4.5 Community Detection Algorithms

In this thesis, we propose implement three community detection algorithms:

- DEMON
- Label Propagation
- FluidC

4.5.1 TinkerPop Implementation for Community Detection Algorithms

The community detection programs were developed in JAVA language.

From [51], The TinkerPop org.apache.tinkerpop.gremlin.process.computer.VertexProgram interface would be implemented by all java classes that need to execute the computation in parallel way. The Vertex Program represents one component of a distributed graph

computation. Each Vertex in the graph logically execute the VertexProgram instance in parallel.

The VertexProgram interface mainly has 3 methods to implement:

- **Execute method**: for each iteration, the vertices execute this method
- **Setup method**: this method is executed at the beginning of the computation
- **Terminate method**: at the end of each iteration, this method is executed to validate if the program has finished.

This VertexProgram offer a share memory of type "org.apache.tinkerpop.gremlin.process.computer.Memory" that is global memory in the whole computation where the vertices can communicate information with one another and also global information of the computation.

The next image shows the execution flow.: All methods into each iteration represent the methods that are executed at each iteration. The "setup" method in the iteration zero is a global method and it is executed at the beginning of the computation, the "execute" method is executed by each vertex in the graph, then the "terminate" method is a global method where que can validate if the algorithm has converged, on other case the algorithm executes the next iteration until the algorithm converge.

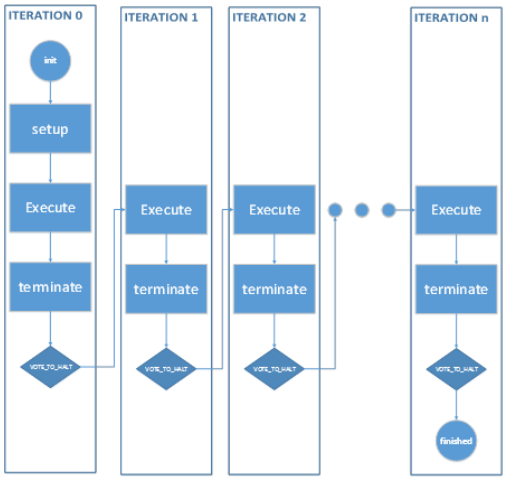


Figure 29 - General TinkerPop execution methods

StaticVertexProgram is a TinkerPop Class that implements the VertexProgram interface, The general view of the three implementations for community detection algorithms is the next:

FluidC, LabelPropagationAlgorithmBreakingTiesAsynchronous and DEMON Classes extends from StaticVertexProgram class

All classes that extends from StaticVertexProgram could use the TinkerPop VertexProgram interface and the algorithms can be executed in parallel with BSP form for any platform: Spark, Giraph, MapReduce, Local execution could be used too [50].

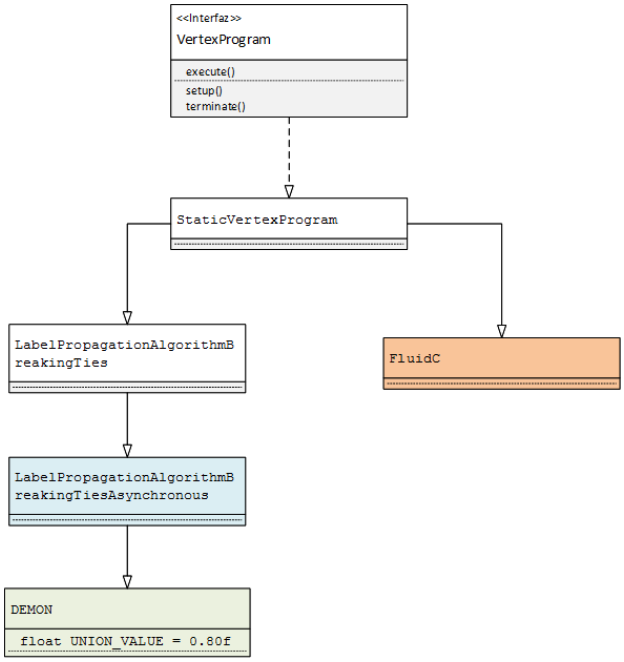


Figure 30 - UML classes Diagrama

In the state of the art in community detection algorithms we find many of algorithms with different complexity and quality in community detection.

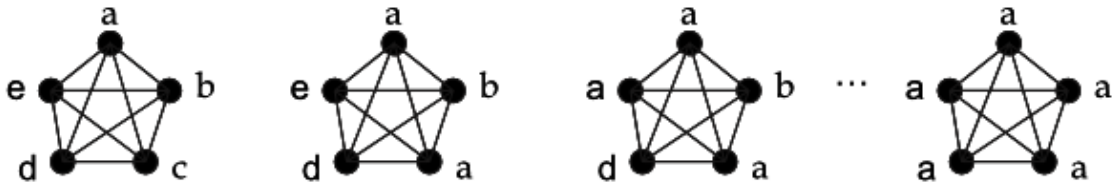
4.5.2 Label Propagation algorithm

LPA is the fastest algorithm for community detection algorithm where each vertex is initialized with an id label the for each iteration this label is propagated between the neighbors of the entity and each entity adopt the label with the major frequency [45], LPA does not receive parameters.

Each vertex has id label, then this label is propagated through vertex´s neighbors. Each vertex adopts the label with more frequency between vertex neighbors.

The next image is an example of the algorithm:

Each vertex is initialized with an id label, in each iteration the vertex to compute take the label with more frequency between their neighbors and then propagate their new id label through their neighbors.



The next function is used to update the label of a vertex x

Equation 4 - Label Propagation , update label function

$$C(x) = f(C_{xi1}(t), \dots, C_{xim}(t), C_{xi(m+1)}(t-1), \dots, C_{xik}(t-1))$$

$xi1$,..., xim are neighbors of x that have already been updated in the current iteration while $xi(m+1)$,..., xik are neighbors that are not yet updated in the current iteration

The stop condition in the LPA is:

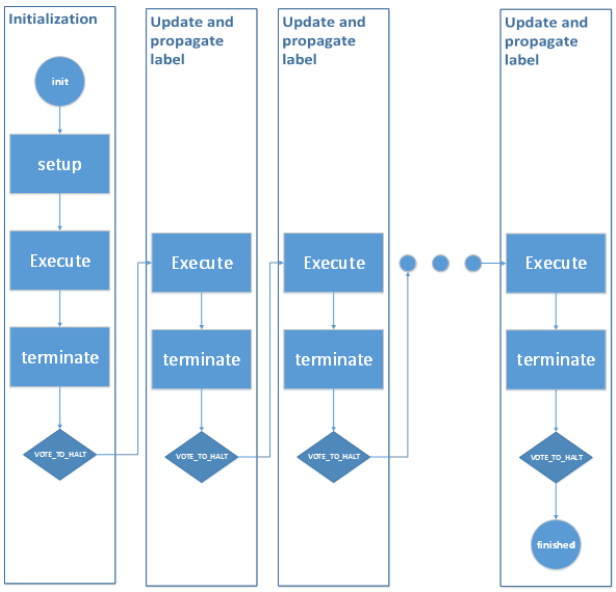
If $C_1, \dots, C_p$ are the labels that are currently active in the network and $d_i^{C_j}$ is the number of neighbors node i has with nodes of label C_j , then the algorithm is stopped when for every node i, If i has label C_m then $d_i^{C_m} > d_i^{C_j}$ ∀ j.

4.5.3 TinkerPop Label propagation implementation

The TinkerPop label propagation implementation is into the class "LabelPropagationAlgorithmBreakingTiesAsynchronous.java", this algorithm has the next flow:

- The first iteration is for the algorithm initialization.
  - Execute the "setup" global method to initialize the global memory
  - Execute the "execute" method for each vertex to initialize the vertex properties

    - Execute the “terminate” global method to validate if all vertices converge
- The other iterations, each vertex updates their label and propagates their label to their neighbors, until all vertices do not change their label converge.
    - Execute the “execute” method for each vertex to update and propagate their label
    - Execute the “terminate” global method to validate if all vertices converge



The next image represents the communities detected in karate graph for the execution of the class “LabelPropagationAlgorithmBreakingTiesAsynchronous.java”, where each color represents a community

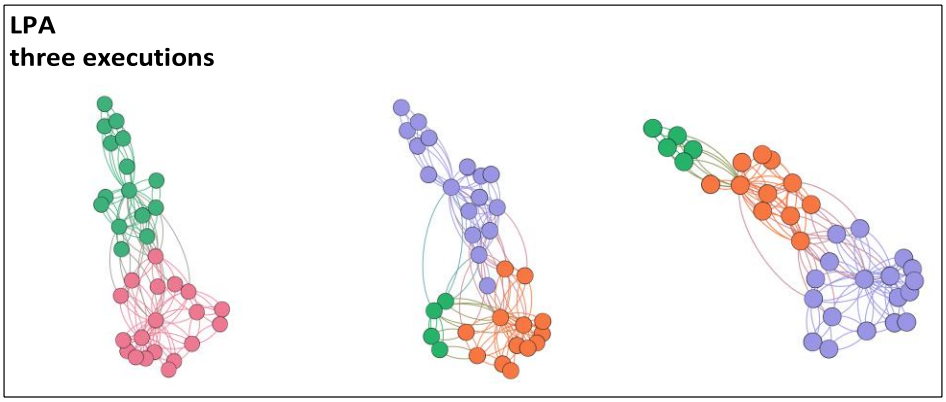


Figure 31 - Label Propagation example

I get different result from the three executions of the same algorithm, this is for the random selection of label when the vertices have two or more neighbor labels that have the same frequency.

Label Propagation Pseudo-code

```
####################################
# SETUP METHOD                     #
####################################
Require:GLOBAL_MEMORY

GLOBAL_MEMORY.vote_to_halt <= false



####################################
# EXECUTE METHOD                   #
####################################
Require:GLOBAL_MEMORY, VERTEX, MESSAGES

If(GLOBAL_MEMORY.number_of_iteration==0)
     VERTEX.neighbours <= collectionOf[idVertex,community_label]
     VERTEX.vote_to_halt<=false
     GLOBAL_MEMORY.vote_to_halt.And(VERTEX.vote_to_halt)
Else if((VERTEX.vote_to_halt == false) or INPUT_MESSAGES.hasMessages())
     for each(neigbour_message in MESSAGES.INPUT_MESSAGES)
          VERTEX.neighbours.updateCommunity(neigbour_message.idVertex , neigbour_message.community_label)
     end for each
     local_community = getCommunityWithMoreFrecuency(VERTEX.neighbours)
     finish <= (local_community == VERTEX.community)
     if(!finish)
          MESSAGES.SendMessageToNeighbors([VERTEX.id,local_community])
     end if
     VERTEX.community <= local_community
     VERTEX.vote_to_halt <= finish
     GLOBAL_MEMORY.vote_to_halt.And(finish)
End if

####################################
# TERMINATE METHOD                 #
####################################
Require:GLOBAL_MEMORY

If(GLOBAL_MEMORY.vote_to_halt)
     finishProgram()
End if
nextBSP_Iteration(GLOBAL_MEMORY)
```

Pseudo-code 1- Label Propagation

4.5.4 DEMON algorithm

Democratic Estimate of the Modular Organization of a Network is an algorithm for overlapping community detection where we have one parameter ϵ that indicate if two overlapping communities are merged when they have ϵ percent of entities not in common [43]. This algorithm is for Large graphs due to it take in mine the local communities for each entity in the network.

DEMON algorithm defines two basic graph operations.

The Ego Network extraction EN. Given a graph G and a node $v \in V$, $EN(v, G)$ is the subgraph $G'(V', E')$ where $V'$is the set containing v and all its neighbors in E, and E' is the subset of E containing all edges (u, v) where $u \in V' \wedge v \in V'$

The Graph vertex difference $-g$: $-g(v, G)$ will result in a copy of G without the vertex v and all edges attached to v.

The combination of these two functions yield the EgoMinusEgo function:

Equation 5 - EgoMinusEgo Function

$$EgoMinusEgo(v, G) = -g(v, EN(v, G))$$

Given a graph G and a node $v \in V$, the set of local communities $C(v)$ of node v is a set of (possibly overlapping) sets of nodes in $EgoMinusEgo$, where each set $C \in C(v)$ is a community according to node similarity: each node in C is more similar to any other node in C than to any other node in $C' \in C(v)$, with $C \neq C'$.

The set of global communities, or simply communities, of the graph G as:

Equation 6 - DEMON merge function

$$C = Max(\bigcup_{v \in V} C(v))$$

From local to global communities by selecting the maximal local communities that cover the entire collection of local communities. Each found in the $EgoMinusEgo$ network of each individual node.

Two communities C and I are merged if and only if at most the $\epsilon\%$ of the smaller one is not included in the bigger one; in this case C and I are removed from the set of Communities and their union is added to the result set.

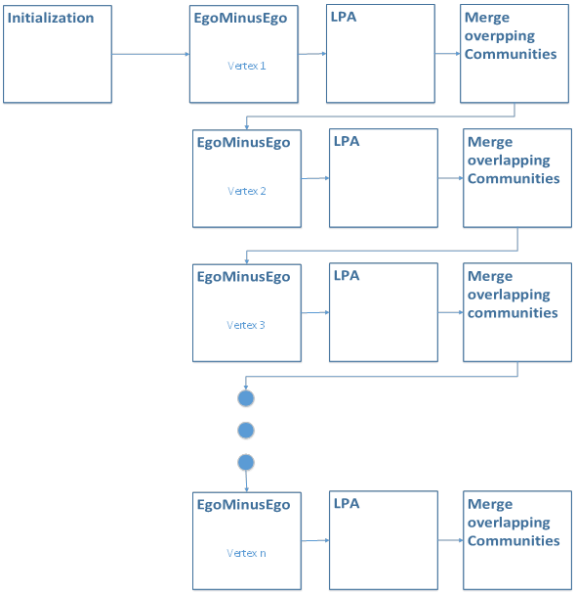
4.5.5 TinkerPop DEMON implementation

The TinkerPop DEMON implementation is into the class “DEMON.java”

The flow of the algorithm is the next: I create functional modules from a set the iterations

- The modulo Initialization: this module is for the initialization of the parameters and initializes the vertex properties.
- The EgoMinusEgo module: this module is composed by a set of iterations and applies the EgoMinusEgo function to the graph.

- LPA module: this module executes the Label propagation algorithm implemented into the class “LabelPropagationAlgorithmBreakingTiesAsynchronous.java” to find the local communities of the current vertex.
- The Marge overlapping Communities module: this module merges the new local communities with the global overlapping communities, only if at most the $\epsilon\%$ of the smaller community is not included in the bigger community.
- This algorithm executes the EgoMinusEgo, LPA, Merge overlapping Communities modules for all vertices into the graph



The next image represents the communities detected in karate graph for the class DEMON.java execution, using the parameter to merge communities equals to 0.8

Below In the left image, the function EgoMinusEgo is applied to the blue vertex and the local communities are discover using the label propagation algorithm.

The right imagen is the result. The greens vertices are the communities for the previous blue vertex, and the pink vertices do not have a cluster at this moment.

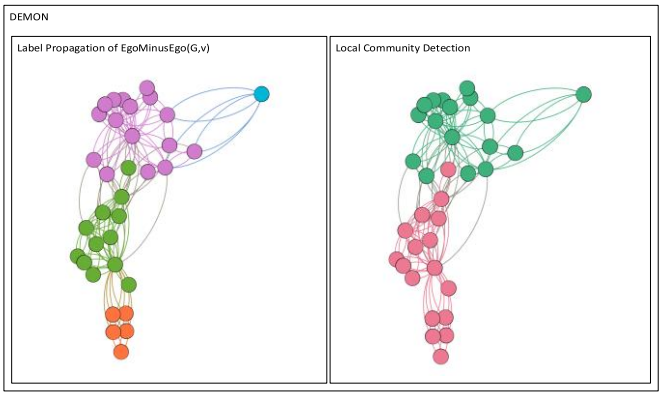


Figure 32 -DEMON example

The next image is the sequence of 3 nodes that applied the EgoMinusEgo function to blue vertices then discovering the local communities with the label propagation algorithm, at last show the overlapping communities: the strong purple share two vertices with the orange community

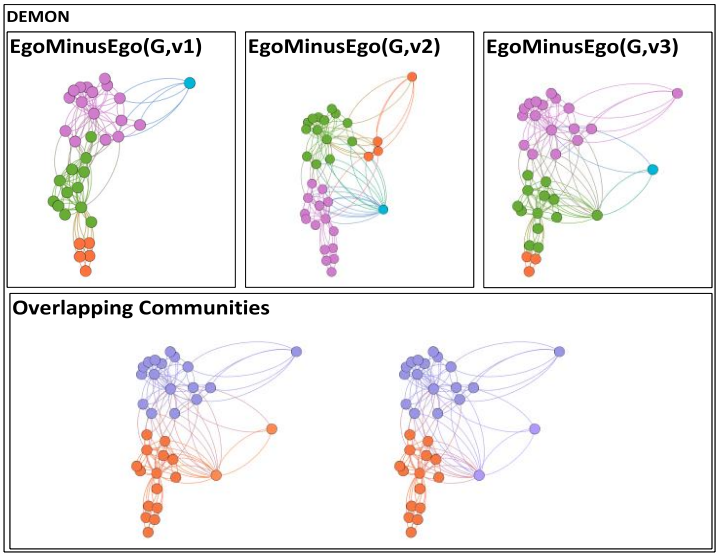


Figure 33- DEMON with 3 vertices example

4.5.6 FluidC algorithm

FluidC is an algorithm that imitate the behavior of liquids and depending of the structure of the network and their density of the liquid, it expands and pushes each other on the network. This algorithm receives the maximum number of communities to detect in the graph [44].

Consider a graph $G = (V, E)$ formed by a set of vertices V and a set of edges E. FluidC initialize k fluid communities on k different random vertices of V , communities that will begin expanding throughout the graph. At all times, each fluid community has a total

summed density of 1. When a fluid community is compacted into a single vertex, such vertex holds the full community, which is also the maximum density a single vertex may hold. As a community spans through multiple vertices, its density becomes evenly distributed among the vertices that compose it.

The updating rule is defined as follow:

Equation 7 - FluidC function

$$C_v^{'} = argmax_{c \in C} \sum_{w \in \{v, N_v\}} D_w \, \delta(C_w, c)$$

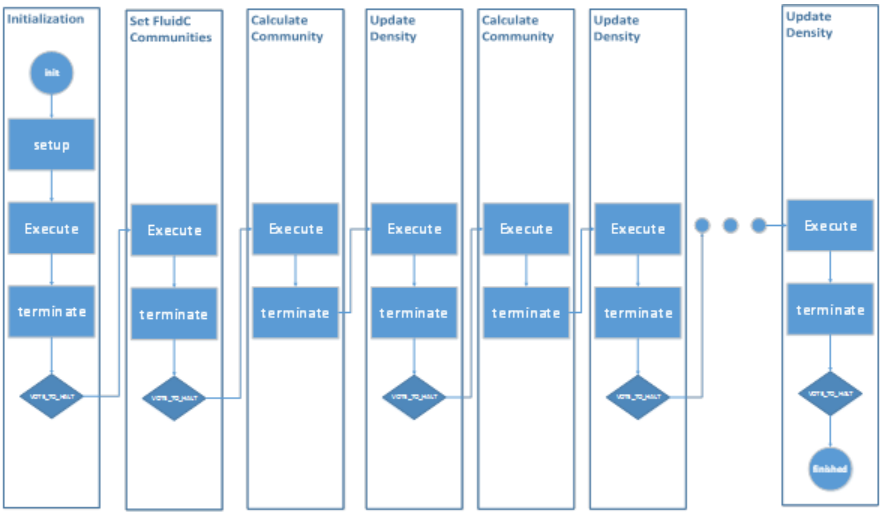
$$\delta(C_w, c) = \begin{cases} 1, if \; C_w = c \\ 0, if \; C_w \neq c \end{cases}$$

Where v is the vertex being updated, $C_v^{'}$ is the updated community of v , N_v are the neighbors of v , c refers to a community from the set of all communities C, D_w is the density assigned to vertex w and $C_w$is the community vertex w belong to. $\delta(C_w, c)$ is the Kronecker delta between C_w and c community.

4.5.7 TinkerPop FluidC implementation

The TinkerPop FluidC implementation is into the class “FluidC.java”, the flow of the algorithm is the next.

- The first iteration: it is for the variables initialization.
- The second iteration: Initialize the n number of FluidC Communities over random vertices
- For Each pair of iterations ” Process Community” in time t and “Update Density” in time (t+1) until the algorithm converge:
    - Iteration ”Process Community”: it calculates and update the new vertex Community for each vertex in the graph
    - Iteration “Update Density”: it calculates the new value of the Fluid Community Density for each Fluid Community, until the algorithm converges.
    - The algorithm converge until all vertices does not change their community.



The next image represents the communities detected in karate graph for the class FluidC.java execution changing the number of number of fluid community parameter, Each color represents a community

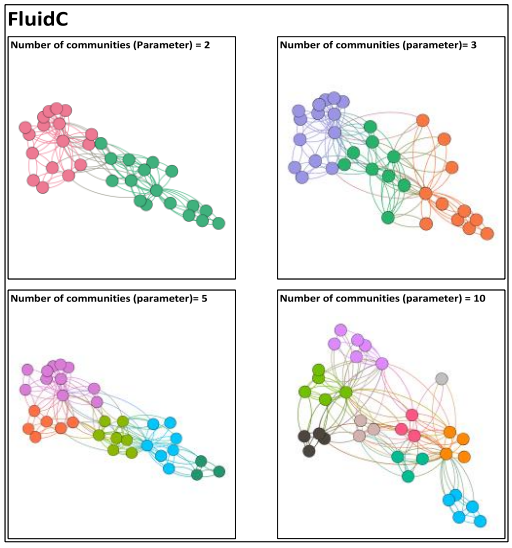


Figure 34- FluidC example

FluidC Pseudo-code

```
####################################
# SETUP METHOD                     #
####################################
Require:GLOBAL_MEMORY

GLOBAL_MEMORY.vote_to_halt <= false
GLOBAL_MEMORY.step <= "initialization"
GLOBAL_MEMORY.numberFluidCommunities = x_number


####################################
# EXECUTE METHOD                   #
####################################
Require:GLOBAL_MEMORY, VERTEX, MESSAGES

If(GLOBAL_MEMORY.step == "initialization")

     VERTEX.vote_to_halt <= false
     VERTEX.density <= 0
     GLOBAL_MEMORY.get_X_number_RandomVertices(GLOBAL_MEMORY.numberFluidCommunities , VERTEX.id)
     GLOBAL_MEMORY.vote_to_halt.And(VERTEX.vote_to_halt)

Else if(GLOBAL_MEMORY.step == "set communities" )

     If(GLOBAL_MEMORY.listOfVerticesToinitFluidC.contains(VERTEX.id))
          VERTEX.community <= VERTEX.id
          VERTEX.density <= 1
          MESSAGES.sendMessagestoNeighbors([VERTEX.id,1])
     Else
          VERTEX.vote_to_halt <= true
     End If

Else if((GLOBAL_MEMORY.step == "Calculate Community") and VERTEX.vote_to_halt == false)
     finish <= (VERTEX.community_Tminus1 == VERTEX.community)
     If(GLOBAL_MEMORY.DensityCommunities.contains(VERTEX.community_Tminus1))
          VERTEX.community <= VERTEX.community_Tminus1
          VERTEX.density <= GLOBAL_MEMORY.FLUID_DENSITY.get(VERTEX.community_Tminus1)
     End If
     MESSAGES.sendMessageToNeighbors([VERTEX.community,VERTEX.density])
     GLOBAL_MEMORY.vote_to_halt.And(VERTEX.vote_to_halt)

Else if ((GLOBAL_MEMORY.step == "Update Density") and (VERTEX.vote_to_halt == false or MESSAGE.hasMessages()))
     community <= CalculateFluidCommunity(VERTEX,MESSAGE.INPUT_MESSAGES)
     VERTEX.community_Tminus1 <= community
     GLOBAL_MEMORY.FLUID_COMMUNITIES.sum(community,1)
     VERTEX.vote_to_halt <= false
End if


####################################
# TERMINATE METHOD                 #
####################################
Require:GLOBAL_MEMORY

If(GLOBAL_MEMORY.vote_to_halt)
     finishProgram()
Else If(GLOBAL_MEMORY.step == "initialization" )
     GLOBAL_MEMORY.listOfVerticesToinitFluidC == GLOBAL_MEMORY.get_X_number_RandomVertices_List()
Else If(GLOBAL_MEMORY.step == "Calculate Community")
     GLOBAL_MEMORY.step <= "Update Density"
Else if(GLOBAL_MEMORY.step =="set communities")
     GLOBAL_MEMORY.step <= "Calculate Community"
Else if(GLOBAL_MEMORY.step == "Update Density")
     For each(fluidC in GLOBAL_MEMORY.FLUID_COMMUNITIES)
          GLOBAL_MEMORY.FLUID_DENSITY.update(fluidC,1/fluidC.size)
     End For each
     GLOBAL_MEMORY.step <= "Calculate Community"
End if
nextBSP_Iteration(MEMORY)
```

Pseudo-code 2- FLuidC

5 Results

This section I present the results for the graph processing previously mentioned

5.1 Marenostrum III configuration

The configuration of Hadoop on Marenostrum III has the next sequence, this sequence was implemented into LSF script.

1. Execute MPI program to retrieve the host names of the resources reserved for the job
2. Configure Dynamically the Hadoop file using the host names
    a. The $TMPDIR needs to be configured in core-site.xml, hdfs-site.xml, yarn-site.xml Hadoop files
3. Configure TinkerPop environment
4. Deploy HDFS, YARN cluster
5. Execute groovy script to launch the Tinkerpop console.
    a. Upload the graph to HDFS
    b. Execute the community detection algorithm
    c. Retrieve the results

Marenostrum III present many restrictions to configure Hadoop, from install the plugins of TinkerPop to execution of the Hadoop jobs

- Marenostrum III does not have a manual to use Hadoop
- Marenostrum III require special configuration to quit some constraints, specially a clean process
    - Each 5 minutes, the clean process kills Hadoop process services.
- Marenostrum III does not have an exterior connection to install TinkerPop plugins from their repository.
- It is difficult track the Hadoop job from LSF.

Quit the Marenostrum III clean process to execute the Hadoop jobs requires a special configuration, for this reason and the short time to finish the Thesis I don't create tests of the Communty Detection algorithms on Marenostrum III.

5.2 AMAZON EMR configuration

The configuration of Amazon EMR requires little configuration to work with Giraph and SPARK using TinkerPop 3

The next JSON is for works with JAVA 8 and add the number of counters on Hadoop. The JSON is used when the cluster is configured before the AMAZON EMR cluster is running.

Amazon EMR - JSON Hadoop Configuration

```
[
    {
        "Classification": "hadoop-env",
        "Configurations": [
            {
                "Classification": "export",
                "Configurations": [],
                "Properties": {
                    "JAVA_HOME": "/usr/lib/jvm/java-1.8.0"
                }
            }
        ],
        "Properties": {}
    },
    {
        "Classification": "spark-env",
        "Configurations": [
            {
                "Classification": "export",
                "Configurations": [],
                "Properties": {
                    "JAVA_HOME": "/usr/lib/jvm/java-1.8.0"
                }
            }
        ],
        "Properties": {}
    },
  {
    "Classification": "mapred-site",
    "Properties": {
      "mapreduce.job.counters.max": "1024",
      "mapreduce.job.counters.limit": "1024"
    }
  }
]
```

The m4.large instances were used in this thesis and m4.large has the next Default Hadoop configuration :

m4.large

Configuration Option	Default Value
mapreduce.map.java.opts	-Xmx2458m
mapreduce.reduce.java.opts	-Xmx4916m
mapreduce.map.memory.mb	3072
mapreduce.reduce.memory.mb	6144
yarn.app.mapreduce.am.resource.mb	6144
yarn.scheduler.minimum-allocation-mb	32
yarn.scheduler.maximum-allocation-mb	6144
yarn.nodemanager.resource.memory-mb	6144

TinkerPop 3 requires one property file to configure the Hadoop Graph and the Computer Graph(SparkGraphComputer , GiraphGraphComputer), in this thesis I use the next configuration.

TinkerPoP 3 configuration for SPARK and GIRAPH

```
###################################
# TinkerPop 3 Configuration       #
###################################
gremlin.graph=org.apache.tinkerpop.gremlin.hadoop.structure.HadoopGraph
gremlin.hadoop.graphReader=org.apache.tinkerpop.gremlin.hadoop.structure.io.gryo.GryoInputFormat
gremlin.hadoop.graphWriter=org.apache.tinkerpop.gremlin.hadoop.structure.io.gryo.GryoOutputFormat
gremlin.hadoop.jarsInDistributedCache=true
gremlin.hadoop.defaultGraphComputer=org.apache.tinkerpop.gremlin.spark.process.computer.SparkGraphComputer

gremlin.hadoop.inputLocation=#GRAPH_NAME
gremlin.hadoop.outputLocation=#RESULT_GRAPH_NAME_

###################################
# SparkGraphComputer Configuration #
###################################
spark.master=yarn-client
spark.executor.memory=SIZE_OF_EXECUTOR
spark.serializer=org.apache.tinkerpop.gremlin.spark.structure.io.gryo.GryoSerializer
spark.executor.extraClassPath=/home/hadoop/lib/*


###################################
# GiraphGraphComputer Configuration #
###################################
giraph.maxMasterSuperstepWaitMsecs=150000
mapred.job.tracker=yarn
mapred.map.max.attempts=10
giraph.minWorkers=#NUMBER_OF_WORKERS
giraph.maxWorkers=#NUMBER_OF_WORKERS
giraph.zkServerCount=1
giraph.useOutOfCoreGraph=true
giraph.useOutOfCoreMessages=true
giraph.numInputThreads=NUMBER_OF_THREADS
giraph.numComputeThreads=NUMBER_OF_THREADS
giraph.maxMessagesInMemory=1000000
giraph.pure.yarn.job=false
giraph.SplitMasterWorker=true
```

5.3 Performance

This Section presents the performance of the Community Detection algorithms previously mentioned using the next graphs from SNAP repository

Where DBLP and Amazon graphs have the same scale but their properties are different, the next tables show some properties and scale of the graphs

Name	Type	Nodes	Edges	Communities	Description
YouTube	Undirected, Communities	**1,134,890**	**2,987,624**	**8,385**	Youtube online social network
DBLP	Undirected, Communities	**317,080**	**1,049,866**	**13,477**	DBLP collaboration network
Amazon	Undirected, Communities	**334,863**	**925,872**	**75,149**	Amazon product network

Name	LDBC Graphalytics Scale	Graph500 Scale	Triangles	Diameter	Average Clustering coefficient	Kryo file Size (MB)
YouTube	6.61	**20**	3056386	**20**	0.0808	215.37
DBLP	6.13	**18**	2224385	**21**	0.6324	66.4
Amazon	6.1	**18**	667129	**44**	0.3967	63.8

5.3.1 Experiments

The fist experiment consists in the execution of the Label Propagation, DEMON and FluidC Algorithms implemented in TinkerPop and measure the time execution from the algorithm begins the execution (stup method) to the algorithm converges or it completes a given number of iterations (terminate method).

The maximum number of iterations is set to 20.

The Graphs were migrated to Kryo format and storage in the HDFS.

Name	Kryo file Size (MB)
YouTube	215.37
DBLP	66.4
Amazon	63.8

5.3.2 Label Propagation

The next images represent the Label Propagation execution time using 2,5,10 Workers for SPARK and Giraph on AMAZON EMR, the missing bars represent memory exception

Giraph:

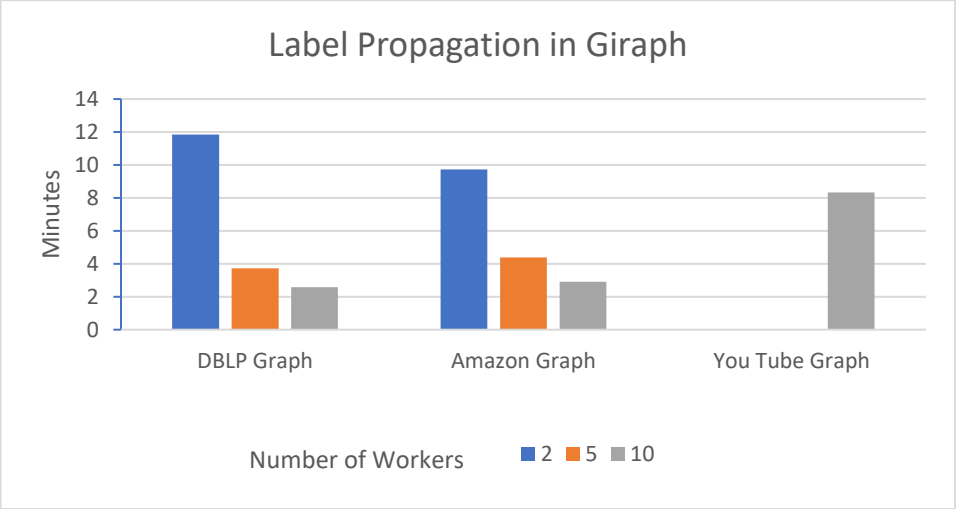


Figure 35 - label propagation execution time with Giraph

Giraph has a peak of 8 MB/s in use of network, 50 GB of memory and 50% of use of CPU to process the Amazon graph with 10 workers

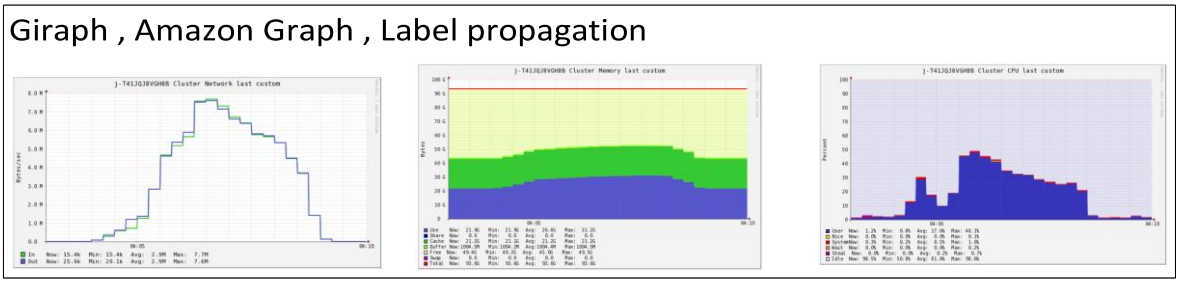


Figure 36 - LPA resource consumption in Giraph

Giraph errors:

Giraph does not complete the Execution of the YouTube graph with 5 and 2 workers due to The errors were produced by the Garbage collector due to it exceeded the limit of memory configured in the JAVA program.

SPARK:

SPARK complete the execution of the program faster than Giraph, also finished the process for The Youtube Graph with 5 workers

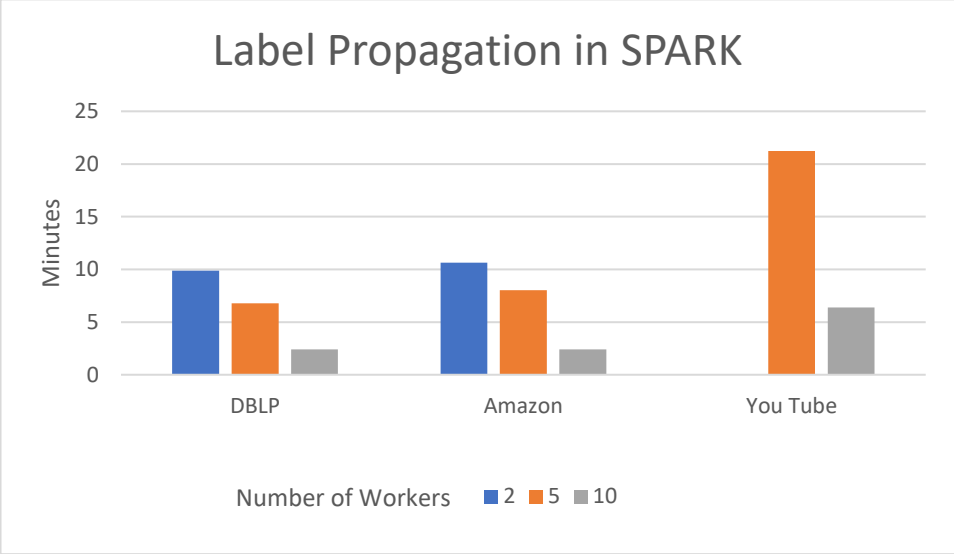


Figure 37 - Label Propagation execution time SPARK

SPARK has a peak of 45 MB/s in use of network, 64 GB of memory and 50% of use of CPU to process the Amazon Graph with 10 Workers

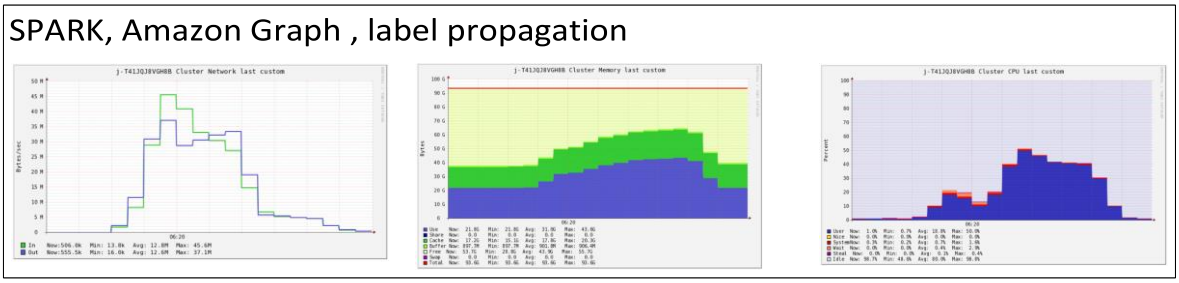


Figure 38 – LPA resource consumption in SPARK

5.3.3 FluidC

The FluidC algorithm receive as parameter the number of fluid communities, the algorithm groups the vertices into the fluid communities. For the executions of this test, the Fluid Communities is set to the number of communities that exist in each graph (number of communities in the ground-truth, YouTube Graph = 8,385, Amazon Graph = 75,149, DBLP Graph = 13,477).

Giraph:

The next image shows the execution time for 20 iterations

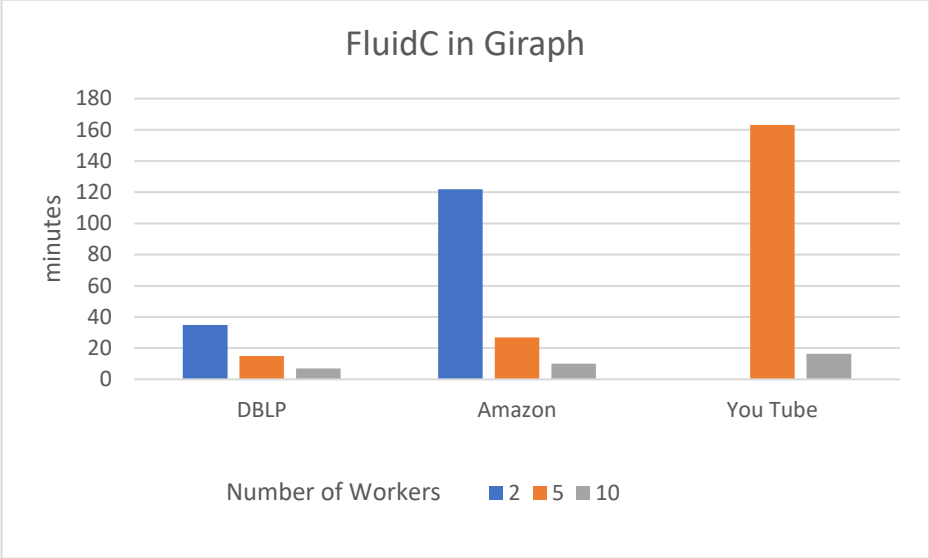


Figure 39 - FluidC execution Time in Giraph

The next image represents the general behavior of the cluster when FluidC algorithm is executed with 8,385 fluid communities, with 10 Workers, the result is a peak of 9 MB the use of Network, near to 50 GB in Memory and a peak of 60% of use the CPU in whole the cluster.

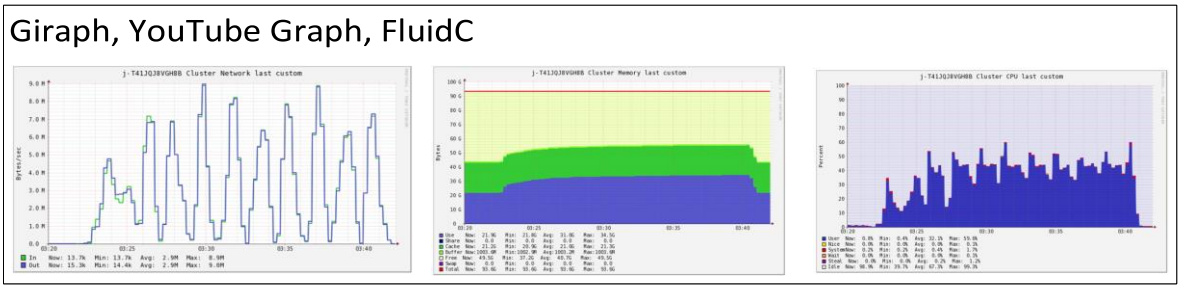


Figure 40 - resource consumption in Giraph

SPARK:

The next image shows the FluidC execution time for 20 iterations on SPARK

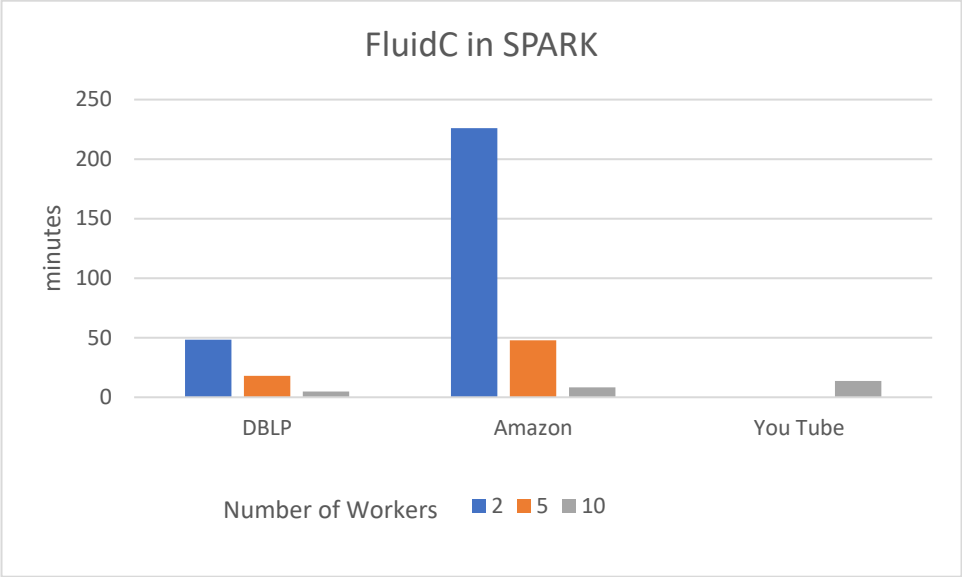


Figure 41 - FluidC execution time in SPARK

The next image represents the general behavior of the cluster when FluidC algorithm is executed with 8,385 fluid communities, with 10 Workers

The result is a peak of 55 MB the use of Network, near to 80 GB in Memory and a peak of 50% of use the CPU in whole the cluster.

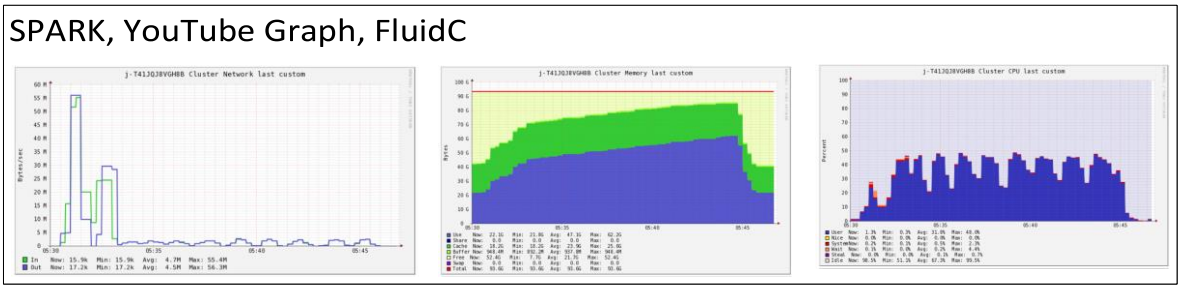


Figure 42 - resource consumption in SPARK

5.4 Quality in the Community Detection for FluidC algorithm

The next images provide information of the quality of the detection algorithm, I use Recall and the precision methods to measure the quality of the detection with respect to the ground-truth communities of the graphs.

The Rank is the sorter communities from lower to higher Recall or Precision score versus the Recall or Precision score, for big graphs decreases the quality of the detection.

Figure 43- FluidC quality in community Detection

5.5 FluidC vs Label Propagation and SPARK vs Giraph

The time execution of the label propagation algorithm is lower that the FluidC, the label propagation is known because it detects monster communities, The next image shows the execution time for the label propagation and FluidC implementations , the graphs are DBLP, YouTube and Amazon with SPARK and Giraph for 2, 4 and 10 workers and maximum number of iterations equals to 20.

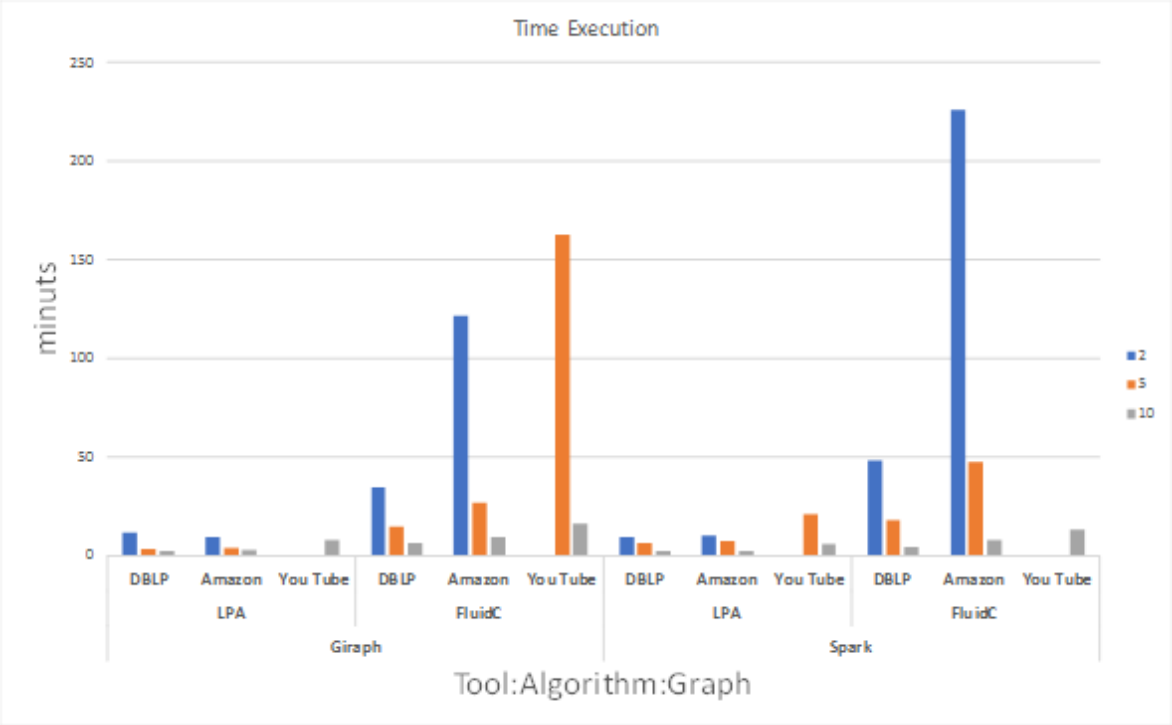


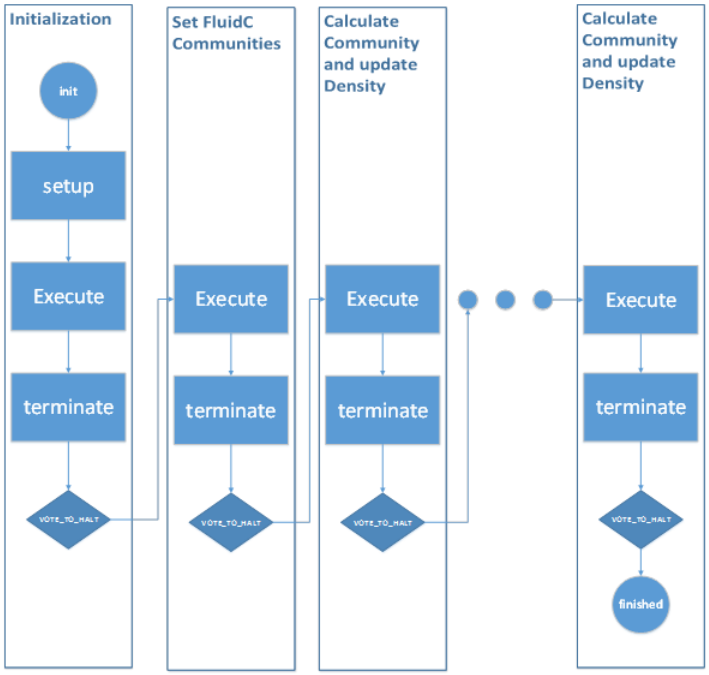
Figure 44- time Execution

Comparing the execution time of the Label propagation versus the FluidC algorithms, I modified the FluidC algorithm with the objective of reduce the time execution and increase the use of CPU.

The second implementation of fluidC (FluidC_2) has the next flow

- The first iteration: this iteration is for the variables initialization.
- The second iteration: Initialize the Fluid Communities on random vertices
- For Each iteration until the algorithm converges:

- Each iteration Calculates and updates the new vertex Community for each vertex in the graph, also in the iteration calculates the new value of the Fluid Community Density for each Fluid Community, until the algorithm converges, this implementation each vertex saves the communities of their neighbors like the label propagation implementation to reduce the traffic in the network.



The advantage of this flow is: for each iteration, the algorithms converge faster than the previous version, also reduce the number of synchronizations. But increase the memory consume to reduce the traffic on the Network

The image Below show the Pseudo-code of the FLuidC_2

The FLUID_2 Pseudo-code

```
###################################
# SETUP METHOD                    #
###################################
Require:GLOBAL_MEMORY

GLOBAL_MEMORY.vote_to_halt <= false
GLOBAL_MEMORY.step <= "initialization"
GLOBAL_MEMORY.numberFluidCommunities = x_number


###################################
# EXECUTE METHOD                  #
###################################
Require:GLOBAL_MEMORY, VERTEX, MESSAGES

If(GLOBAL_MEMORY.step == "initialization")
    VERTEX.neighbours <= collectionOf[idVertex,community_label]
    VERTEX.vote_to_halt <= false
    GLOBAL_MEMORY.get_X_number_RandomVertices(GLOBAL_MEMORY.numberFluidCommunities , VERTEX.id)
    GLOBAL_MEMORY.vote_to_halt.And(VERTEX.vote_to_halt)

Else if(GLOBAL_MEMORY.step == "set communities" )

    If(GLOBAL_MEMORY.listOfVerticesToinitFluidC.contains(VERTEX.id))
        VERTEX.community <= VERTEX.id
        VERTEX.density <= 1
        MESSAGES.sendMessagestoNeighbors([VERTEX.id,1])
    End If

Else if((GLOBAL_MEMORY.step == "Calculate Community and Update Density") and (MESSAGE.hasMessages() or VERTEX.vote_to_halt ==
false))
    For each(neigbour_message in MESSAGES.INPUT_MESSAGES)
        VERTEX.neighbours.updateCommunity(neigbour_message.idVertex , neigbour_message.community_label)
    End for each
    community <= CalculateFluidCommunity(VERTEX,MESSAGE.INPUT_MESSAGES, GLOBAL_MEMORY.FLUID_DENSITY)
    finish <= (community == VERTEX.community)
    GLOBAL_MEMORY.vote_to_halt.And(finish)
    VERTEX.vote_to_halt <= finish
    If(!finish)
        VERTEX.community <= community
        MESSAGES.sendMessagestoNeighbors([VERTEX.id, community])
    End If
    MESSAGES.sendMessageToNeighbors([VERTEX.community,VERTEX.density])
    GLOBAL_MEMORY.FLUID_COMMUNITIES.sum(community,1)
Else if(GLOBAL_MEMORY.step == "Calculate Community and Update Density")
    GLOBAL_MEMORY.vote_to_halt.And(VERTEX.vote_to_halt)
    if(VERTEX.community != null)
        GLOBAL_MEMORY.FLUID_COMMUNITIES.sum(VERTEX.community,1)
    End If
End if


###################################
# TERMINATE METHOD                #
###################################
Require:GLOBAL_MEMORY

If(GLOBAL_MEMORY.vote_to_halt)
    finishProgram()
Else If(GLOBAL_MEMORY.step == "initialization" )
    GLOBAL_MEMORY.listOfVerticesToinitFluidC == GLOBAL_MEMORY.get_X_number_RandomVertices_List()
    GLOBAL_MEMORY.FLUID_DENSITY.clear()
    For each(node in listOfVerticesToinitFluidC)
        GLOBAL_MEMORY.FLUID_DENSITY.put([node,1])
    End for each
    GLOBAL_MEMORY.step <= "set communities"

Else If(GLOBAL_MEMORY.step == "Calculate Community and Update Density" )
    For each(fluidC in GLOBAL_MEMORY.FLUID_COMMUNITIES)
        GLOBAL_MEMORY.FLUID_DENSITY.update(fluidC.id,1/fluidC.size)
        GLOBAL_MEMORY.FLUID_COMMUNITY(fluidC.id,0)
    End For each
End if
nextBSP_Iteration(MEMORY)
```

Pseudo-code 3- FluidC_2

5.5.1 FluidC vs FluidC second implementation (Fluid_2)

The time is similar but the computation increases due to FluidC updates and computes the fluid communities 9 times with 20 iterations. The FluidC_2 updates and computes the fluid communities 18 times with 20 iterations.

Giraph: all processes were completed by FluidC_2

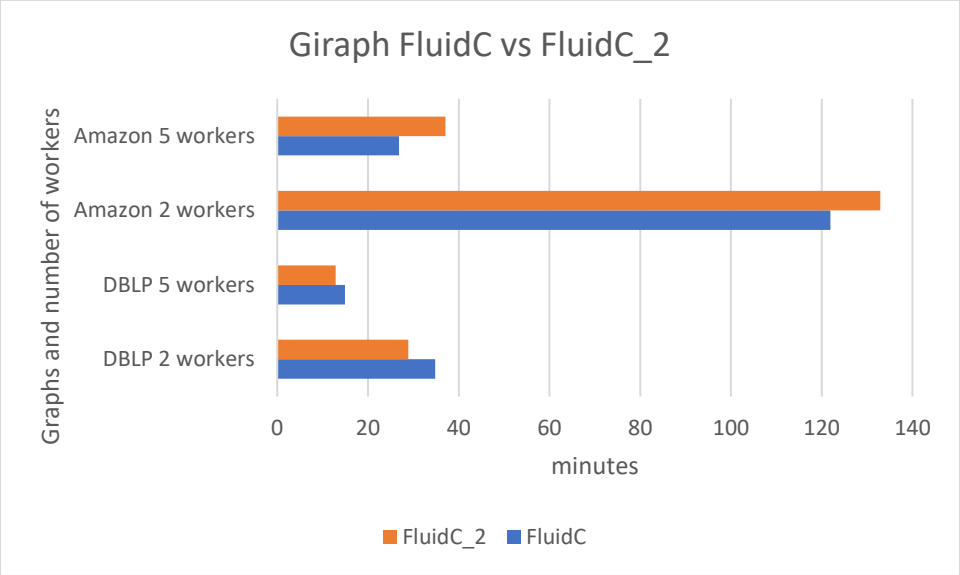


Figure 45 - FluidC vs FluidC_2 execution time Giraph

SPARK: FluidC_2 does not complete the execution for Amazon and DBLP graphs using 5 workers but first FluidC implementation completes the process for the same graphs.

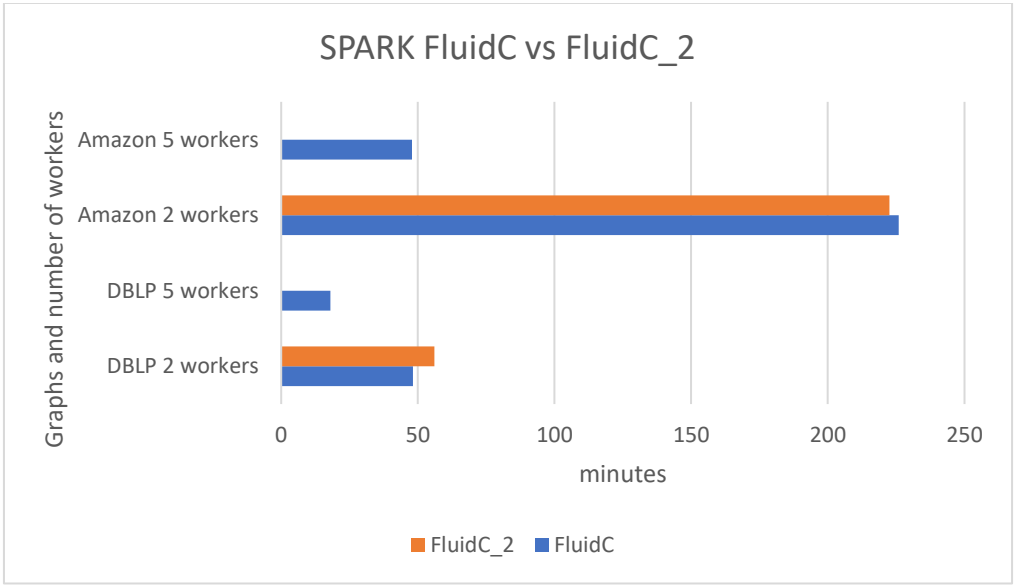


Figure 46- Figure 45 - FluidC vs FluidC_2 execution time SPARK

The follow test uses the Amazon graph and executes the FLuidC_2 implementation, increasing the fluid community parameter, the maximum number of iterations is 5

If we increase the number of communities to detect in the fluidC_2 implementation also increase the time execution.

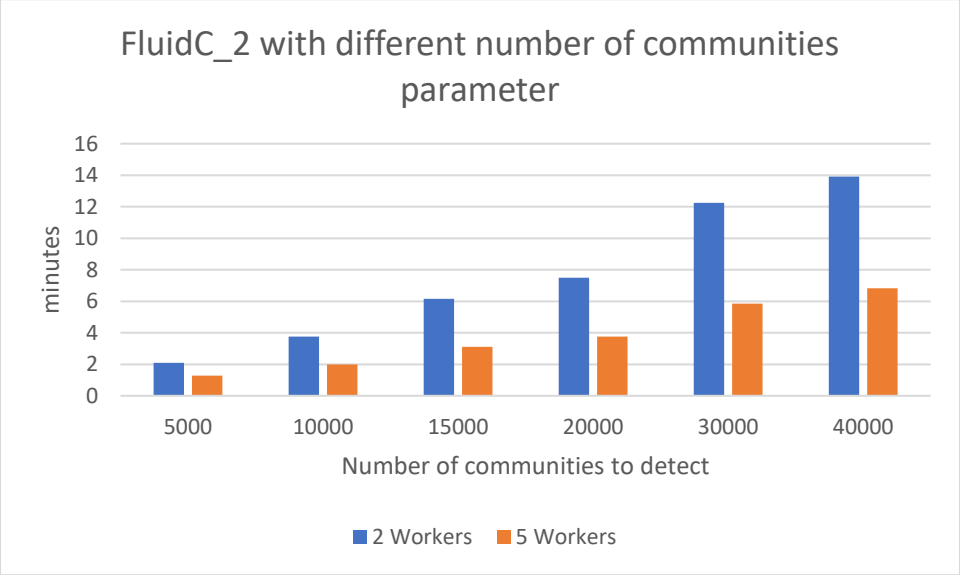


Figure 47- FluidC Execution time with different size of fluid communities

5.5.2 Fluid_2 Performance

FluidC_2 works better in Giraph than SPARK due to SPARK does not distribute the work in an efficient form. This unbalances the workflow and produce memory exceptions in run time

The next images show the use of resources for each worker in SPARK and Giraph.

Giraph uses 4 workers for compute the graph and 1 server of Zookeeper

SPARK uses 5 workers for compute the graph.

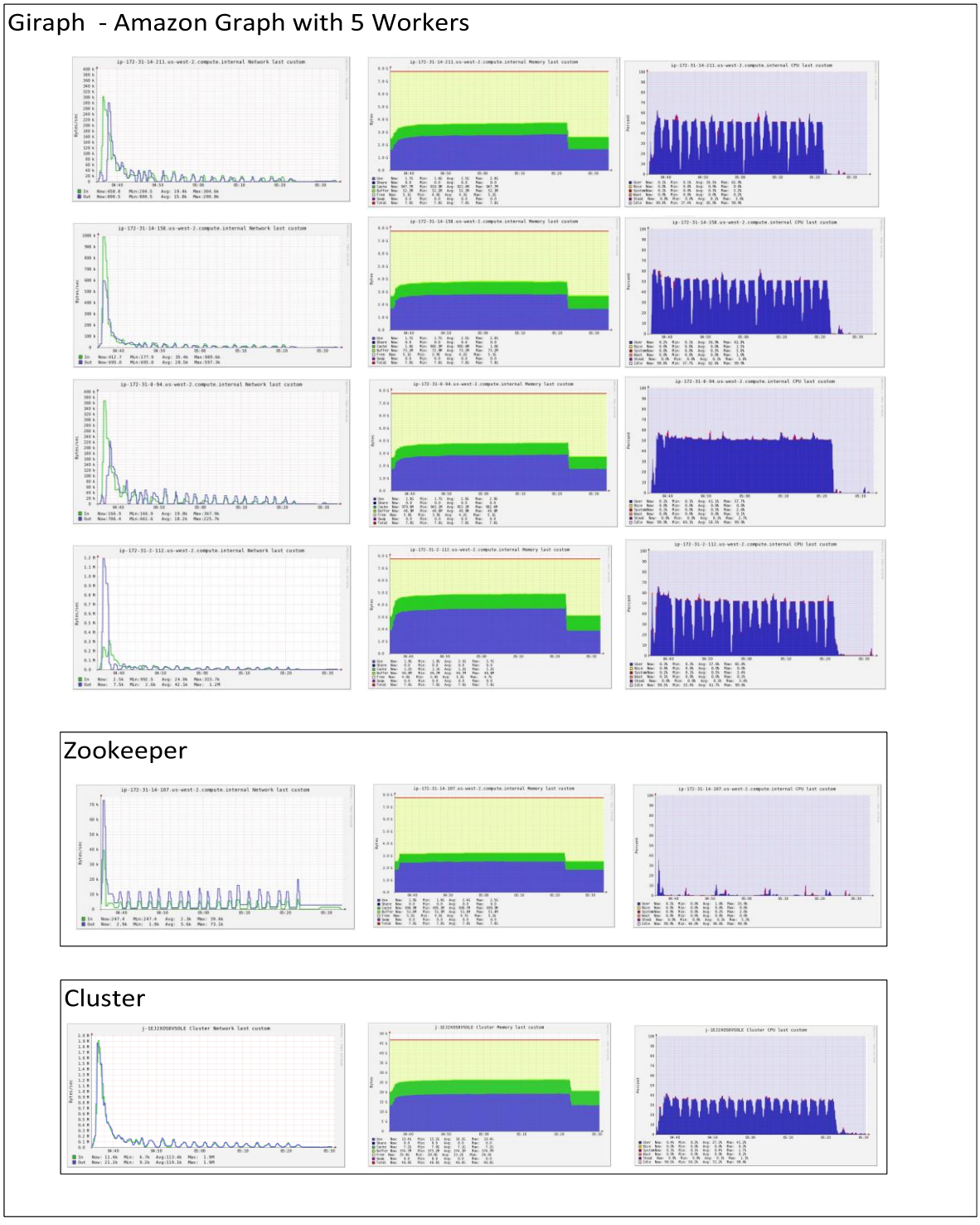


Figure 48- Giraph resource consumption for each worker

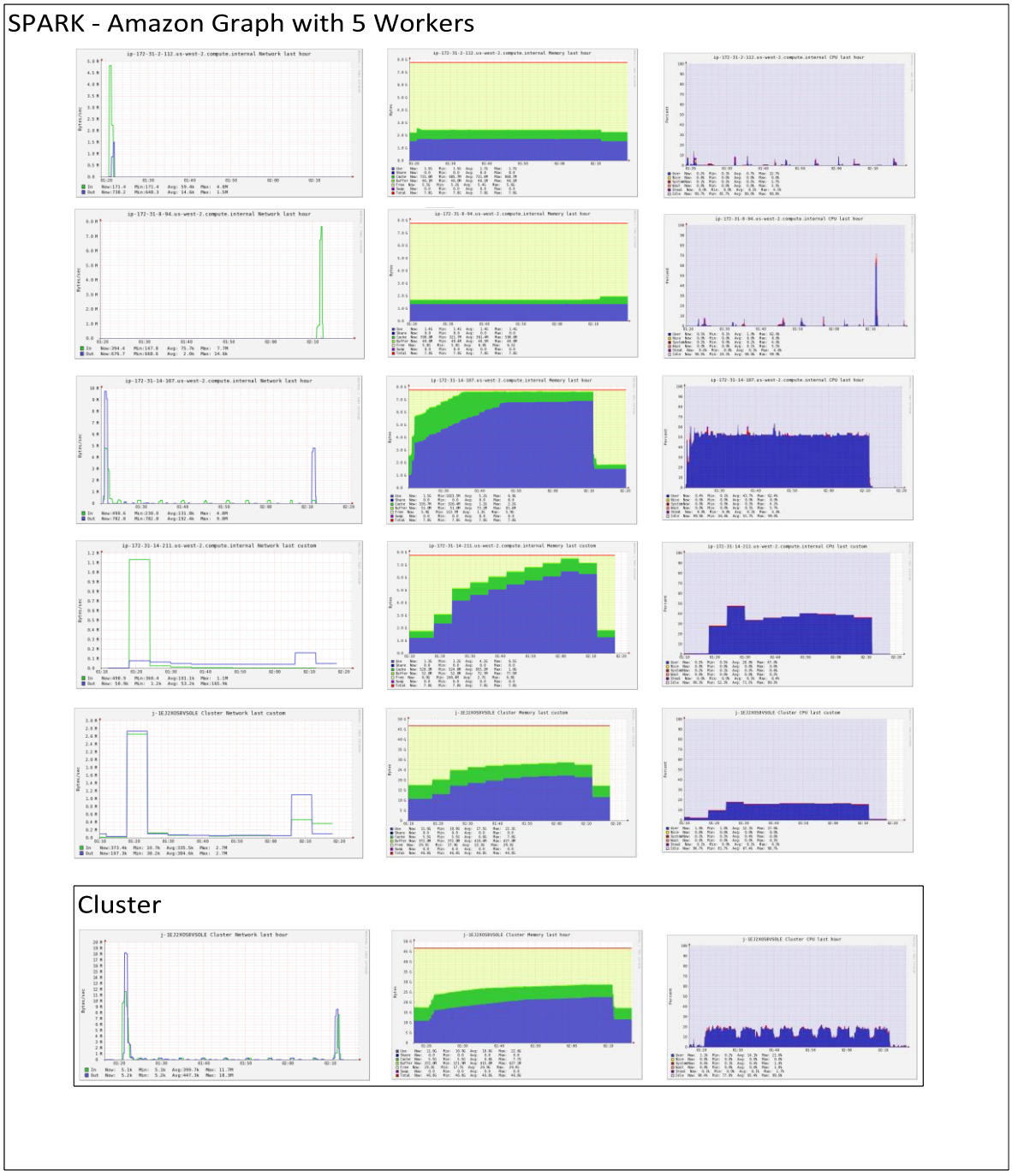


Figure 49- SPARK resource consumption for each worker

6 Conclusions

I can conclude the next:

- SPARK and Giraph has low use of CPU in the execution, this is due to the execution time takes more time in the memory access than the operation to detect communities as the challenge mentioned in the section 3.4
- Giraph distribute better the workflow (figure-48)between their workers than SPARK (figure-49)
    - SPARK suffers when the graph process demands more memory and could produce the memory exception in run time this is produced by the work unbalance in the figure 49 1 worker has the whole memory (8GB) in use but others have between 1 to 2 GB of Memory in use.
    - Its necessary Analyze the Partition strategy.
- The TinkerPop tool scales well in horizontal form due to the properties of YARN and the programming model, but the graph structure and graph properties could produce the work unbalance in run time, it is necessary for the graph processing sets good configurations in the YARN containers and the JAVA options
  We previously need to analyze the properties of the graph.
- The graphs in Kryo format and TinkerPop consume more memory due to the cost of Java and serialization of the objects.
- The precision of the community detection in large graphs is low in general also for fluidC - figure 43.
- The supercomputer Marenostrum III is not a platform that interacts in a natural form with Apache BigData Tools.
- The Cloud Computing on Amazon EMR and the use of Apache BigData tools on it, creates a good platform to process Graphs. Take in mine that we can scale horizontal and vertical form depending of the workload.
- To have better precision and recall scores we need create new overlapping community detection algorithms due to the ground-truth communities for the SNAP graphs are overlapping communities.
- DEMON algorithm has expensive cost in time because we need to execute the label propagation n time the number of vertices of the graph to complete the algorithm.
- I need create more tests to discover the real advantage of DEMON, LPA, FluidC algorithms on large graphs.

7 Future work

- Implement more parallel community detection algorithms to compare the performance and precision of them

- Modify Java parameters to optimize the JVM
- Create test with vertical scale
- Implement this work in Marenostrum 4.
- create the algorithms in a HPC version to reduce the overhead with a tool in the State of the Art.

8 References

1. What is network science ?. ULRIK BRANDES,GARRY ROBINS, ANN McCRANIE, STANLEY WASSERMAN. @Cambridge University Press 2013, 1-15.2013
2. Community Detection in networks: A user guide. Santo Fortunato, Darko Hric. 3-Nov 2016. Physics.soc-ph.
3. Defining and Evaluating Networks Communities based on Ground-truth. Jaewon Yang, Jure Leskovec, 6 Nov 2012
4. One Trillon Edges: Graph Processing at Facebook-Sacale. Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, Sambavi Muthukrishnan, Proceedings of the VLDB Endowment, Vol. 8 No. 12.2015
5. Apache Giraph[online][April 29, 2017] http://giraph.apache.org/
6. Fraud Detection Discovering Connections Using Graphs Databases. Gorka Sadowksi & Philip Rathle.
7. Pregel: A system for Large-Scale Graph Processing. Grzegorz Malewicz, Matthew H. Austern, AartnJ. C. Bik, James C. Dehnert, LLan Horn, Naty Leiser, and Grzegorz Czajkowski. Google, Inc. SIGMOD´ 10. June 6-11,2010
8. UNDERSTANDING SUPERCOMPUTING WHIT MARENOSTRUM SUPERCOMPUTER IN BARCELONA. Jordi Torres. September 2016
9. Amazon EMR [online][April 29, 2017] https://aws.amazon.com/es/emr/
10. MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. Google, Inc. OSDI 2004
11. Resilient Distributed Dataset: A fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, Mosharaf Chowdhuty,Tathagata Das,Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Iron Stoica. University of California, Berkeley.
12. GraphX: A Resilient Distribute Graph System on Spark: Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin. Ion Stoica. AMPLab, EECS, UC Berkeley
13. Message Passing Interface(MPI)[online][cited: April 27, 2017] (MarcadorDePosición1)https://computing.llnl.gov/tutorials/mpi/#What
14. Mixed-Precision programming with CUDA 8. [online][29 April 2017] https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/

15. CUDA 8 Features Revealed. [online][29 April 2017] https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/
16. Graph DataBases – NEW OPPORTUNITIES FOR CONNECTED DATA. Ian Robinson, Jim Webber & Emil Eifrem. Second Edition. O´REILLY,2015 Neo Technology,Inc.
17. Kryo[online][29 April 2017]. https://github.com/EsotericSoftware/kryo
18. GraphML[online][29 April 2017]. http://graphml.graphdrawing.org/
19. GraphSON[online][29 April2017]. https://github.com/thinkaurelius/faunus/wiki/GraphSON-Format
20. CHALLENGE IN PARALLEL GRAPH PROCESSING. ANDREW LUMSDAINE and DOUGLAS GREGOR, BRUCE HENDRICKSON and JONATHAN BERRY. December 2006
21. LDBC Graphalytics: A Benchmark for large-Scale Graph Analysis on Parallel and distributed Platforms. Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, MihaiCapota, Narayanan Sundaram, Michael Anderson, LLie Gabriel Tanase, Yinglong Xia, Lifeng Nai, Peter Boncz.2016 VLDB Endowment
22. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. Mihi Capota, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, Peter Boncz.31 May 2015.
23. Graph500 challenge[online][cited: April 28, 2017]. http://www.graph500.org/
24. How well do Graph-Processing Platforms Perform? An empirical Performance Evaluation and Analysis. Yong Guo. Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, Theodore L. Willke. In IEEE IPDPS, pages 395-404 IEEE, May 2014.
25. LDBC[online][April 29, 2017]. http://ldbcouncil.org/
26. Apache TinkerPop[online][April 29, 2017] http://tinkerpop.apache.org/
27. ElasticSearch[online][April 29, 2017] https://www.elastic.co/
28. Apache Titan[online][April 29, 2017] http://titan.thinkaurelius.com/
29. Neo4j[online][April 29, 2017] https://neo4j.com/product/
30. Apache Cassandra[online][April 29, 2017] https://cassandra.apache.org/
31. Apache Hadoop[online][April 29, 2017] http://hadoop.apache.org/
32. Scale graph[online][April 29, 2017] http://scalegraph.org/web/
33. IBM SystemG[online][April 29, 2017] http://systemg.research.ibm.com/
34. Apache Spark[online][April 29, 2017] http://spark.apache.org/
35. Apache Zookeper[online][April 29, 2017] https://zookeeper.apache.org/
36. NVIDIA nvGRAPH[online][April 29, 2017] https://developer.nvidia.com/nvgraph
37. Extreme Scale Breadth-First Search on Supercomputers. Koji Ueno, Toyotaru Suzumura, Naoya Maruyama, Katsuki Fujisawa, Satoshi Matsuoka. 2016 IEEE
38. Extreme Big Data Processing in Large-Scale Graph Analytics and Billion-Scale Socual Simulation. Toyotaro Suzumura. In ICPE´14 , March 22-26, 2014

39. Towards Next-Generation Graph Processing and Management Platform. Toyotaro Suzumura in HPGP´16 May 31-30 2016
40. Explore Efficient Data Organization for Large Scale Graph Analytics and Storage. Yinglong Xia, Ilie Gabriel Tanase, Lifeng NAi, Wei Tan, Yanbin Liu, Jason Crawford and Ching-Yung Lin. 2014 IEEE.
41. Scale Graph: A High-Performance Library for Billion-Scale Graph Analytics. Toyotaro Suzumura, Koji Ueno. 2015 IEEE
42. Community detection in networks: Structural communities versus ground truth. Darko Hric, Richard K. Darst, and Santo Fortunato, 11 Dec 2014
43. DEMON: a Local-First Discovery Method for Overlapping Communities, Michael Coscia, Giulio Rossetti, Fosca Giannotti, Dino Pedreschi. KDD´12,August 12-16,2012
44. Fluid Communities: A Community Detection Algorithm, Parés, F, Garcia-Gasulla D., Vilalta A. , Moreno, J. Ayguadé E. , Labarta, J. Cortes U., & Suzumura, T. March 29, 2017.
45. Near linear time algorithm to detect community detection structures in large-scale networks. Usha Nandini Raghavan, Réka Albert , and Soundar Kumura. 11 September 2007.
46. Introduction to High Performance Scientific Computing, Victor Eijkhout with Edmond Chow, Robert van de Geijn. 2nd Edition 2014.
47. Hadoop The Definitive Guide STORAGE AND ANALYSIS AT INTERNET SCAL. Tom White, April 2015, Fourth Edition. O´REILLY
48. My Hadoop – Hadoop-on-Demand on traditional HPC Resources, Sriram Krishan, Mahidhar Tatineni, Chaitanya Baru.
49. Hadoop on HPC: Integrating Hadoop and Pilot-based Dynamic Resource Management, Andre Luckow, Ioannis Paraskevakos, George Chantzialexiou, Shantenu Jha. 31 Jan 2016.
50. Apache TinkerPop 3.2.4 [online][April 29, 2017 http://tinkerpop.apache.org/javadocs/current/full/