UNIVERSITY OF PISA

Department of Computer Science

ONJAG
network overlays supporting distributed graph processing

by

Andrea Esposito

Supervisors

Laura Ricci Patrizio Dazzi

A thesis submitted for the

Master Degree of Computer Science

July 2014

UNIVERSITY OF PISA

ABSTRACT

DEPARTMENT OF COMPUTER SCIENCE

Master of Computer Science

by Andrea Esposito

The "Big Data" term refers to the exponential growth that is affecting the production of structured and unstructured data. However, due to the size characterising this data, usually deep analyses are required in order to extract its intrinsic value. Several computational models and various techniques have been studied and employed in order to process this data in a distribute manner, i.e. the capabilities of a single machine can not carry out the computation of this data. Today, a significant part of such data is modelled as a graph. Recently, graph processing frameworks orchestrate the execution as a network simulation where vertices and edges correspond to nodes and links, respectively. In this context the thesis exploits the Peer-to-Peer approach. The overlay concept is introduced and ONJAG ("Overlays Not Just A Graph"), a distributed framework, is developed. ONJAG runs over Spark™, a distributed Bulk Synchronous Parallel-like data processing framework. Moreover, a well-known problem in graph theory has studied. It is the balanced minimum k-way partitioning problem, which is also called minimum k-way cut. Finally, a novel algorithm to solve the balanced minimum k-way cut is proposed. The proposal exploits the P2P approach and the overlays in order to improve a pre-existent solution.

Contents

List of Figures

Acknowledgements

First of all, I would like to thank **Laura Ricci** who accepted to be my supervisor and "jumped" in that was a dark, obfuscated but full of opportunities and charming thesis. She always encouraged and supported me as long as she provided her meek knowledge. Dare I say she has been like a lighthouse keeper on this thesis. Indeed, I thank **Patrizio Dazzi** who joined not so far later as supervisor and immediately he generously lavished its experiences and critical thinking. Thank to **Emanuele Carlini** that actually have joined this adventure since the beginning and supported the thesis along the time. Especially thank to **Amir H. Payberah** and all the JA-BE-JA's authors that shared the original source codes of JA-BE-JA which has been simply fundamental to the success of the thesis. In addition, thank to **Alessandro Lulli** who enthusiastically joined later.
The regular meetings scheduled weekly have been more and more crowded along the time until no space was left in the room. This has been a clear sign of interest and support from everyone.

I also would like to express my gratitude to **Marco Grandi** who explored besides me the *Scala* programming language and supported me fighting against the compiler. Thank to **Roberto Ladu**, together we shared our thesis bugbears during the launch breaks sustaining somehow each other. Thank to **Walter Tommasi**, also called *Javaman*, he always supported me and he is a real trustful friend. Thank to **Marco Dondio** who had to put up with my concerns mediating my well-know argumentative character. Thank to **Andrea Bernabei** who numerous time saved me from "the developer syndrome" or more practically from my laptop failure. Indeed, a big thank to the *Food Bytes* community whose I have been for a while an enthusiastic and schizophrenic organizer. I sincerely had fun and enjoyed your companies which actually improved my Pisa journey experience.

Now let me switch to Italian to... ringraziare tutta la mia **famiglia** che mi ha sempre sostenuto e creduto in me nonostante le tante difficoltà succedutesi col tempo, con particolare riferimento ai miei *genitori* per il loro sostegno sia affettivo che economico, sempre pronti ad aiutare se richiesto. Senza ombra di dubbio la famiglia sono le persone che più si sono preoccupate per me e mi sono più care. Grazie agli zii che hanno avuto un ruolo rilevantissimo nella mia esperienza universitaria e non solo. Ringrazio in particolar modo **Zio Marco** per avermi insegnato molte cose e per avermi sempre fermamente sostenuto ad andare in *Erasmus*. Non ci sono parole per esprimere l'amarezza, il dolore, i rimorsi, i sensi di colpa ed il vuoto che ci ha lasciato.

1

Introduction

Welcome to the real world.

– Morpheus, *The Matrix*

The "Big Data" term refers to the exponential growth that is affecting the production of structured and unstructured data. The perfusion of sensors and the ability to measure and monitor almost everything leads companies, governments and researchers to take advantage of such data making feasible the extraction of useful information from it. However, due to the size characterising this data, usually deep analyses are required in order to extract its intrinsic value. Indeed, the gathering, storage, retrieval and manipulation operations are challenging. Clearly, the capabilities of a single machine can not carry out the computation of those analyses. Several computational models and various techniques have been studied and employed in order to process this data in a distribute manner. Among the others the MapReduce model [12], whose one of the implementations is powered by Google and therefore has been quite famous, has been extensively employed. It is inspired by two functions picked up from the high-order functional paradigm [30].
Today, a significant part of such data is modelled as a graph. The social network graphs, such as Facebook and Twitter, road networks [42] and biological graphs, such as protein structures [27] and human brain connectome [5, 61], are examples of big datasets characterised by many vertices and edges. The MapReduce demonstrated to be suitable for several tasks [6, 25, 26] but anyway, alternative abstractions and solutions, sometimes still based on the MapReduce, have been proposed so far.

Recently, have been proposed some graph processing frameworks that orchestrate the execution as a network simulation where vertices and edges correspond to nodes and links respectively. Pregel [46], which has been developed at Google, is the pioneer of such abstraction. Unfortunately, it is a proprietary product but alternative solutions exploiting the same model of computation have been created by third-parties. The

Pregel model is based over the Bulk Synchronous Parallel abstraction [66]. The BSP is an iterative hardware-agnostic bridging model proposed by Leslie Valiant in the late 80s and it is structured on 2 phases. A first computation phase where all the working units perform a certain task in parallel only using locally retrievable data. After that, a communication phase is performed between the units and a synchronization barrier takes place. The execution and communication phases define a single iteration which is called "superstep". Over the BSP a relatively simple orchestration of messages is instrumented by Pregel in order to simulate the graph as a network. The actual frameworks that implement such model could support a variety of architectures, e.g. large commodity network machines, clusters or super-computers. The effective way in which the hardware is exploited, is up to the framework because the model does not focuses on such details. The differences between the available frameworks are the set of graph operation exposed and how fault-tolerance is faced. An open source implementation, which is developed at Google, is Hama [59] and it is based on the MapReduce [12] Hadoop[1] framework. Hama does not put into account any mechanism to perform fault-tolerance (relying on the Hadoop replication mechanism). Also Yahoo! developed its own solution called Giraph[2]. Giraph does fault-tolerance through check-pointing. A checkpoint forces the data to be saved safely and reliably over stable storage. Re-computation consists in the first attempt to recover from faults. In case of a machine fault, the re-computation of the required chunks of data is performed starting from the most recent checkpoint. Other alternatives are GraphLab [24], GPS [56] and Spark [72]. The latter one relies over the concept of *Resilient Distributed Datasets* [71]. Spark does re-computation as primary fault-tolerance mechanism through check-pointing of the RDDs and is highly suitable for iterative algorithms.
In order to achieve an efficient execution over those frameworks, consists in a proper decomposition of the graph for achieving vertex-cut or less commonly the edge-cut. The aim is to divide the graph in order to efficiently assign each fragment to the machines and achieve the best performance by minimizing the inter-machines communications. In Chapter 2 is given an overview of the distributed graph processing computation and the abstract model the solutions rely on.

The distributed data processing could be suitable as a multi-agent system-friendly environment [17, 69, 39]. Virtually, at a finer grain level and in case of graph structures the vertices could be agents. Peer-to-Peer computations and Multi-Agent Systems share somehow a common background about the distributed environments and their assumptions [38]. Therefore, the P2P approaches are eligible as viable solutions. In addition, the graphs are usually so big that the resources required for managing such structures are extensively distributed over numerous commodity machines instead of relying over a High Performance Computing architecture, which usually can not store those vastness of data without incurring in expensive hardware, i.e. the trade-off between resources

[1] The Apache Software Foundation, Apache™ Hadoop®! homepage: http://hadoop.apache.org
[2] The Apache Software Foundation, Apache™ Giraph! homepage: http://giraph.apache.org

and money has to be taken into account. Thus, the currently architectures employed further motivate the exploration of the P2P networks.
In this context the thesis introduces the Peer-to-Peer approach in order to exploit the distributed graph processing as a network simulation execution. A P2P network is a network where each node is actively involved in the orchestration. Those networks are self-organising, completely distributed without any point of centralization. The latter property ensures a balanced computation scheduling in a distributed scenario. P2P networks usually deal, but they are not limited to, with resource discovering, querying and measure calculation. The P2P algorithms usually have to face with the churn phenomenon, which is the shrinking and expanding of the network due to the leaving and joining of nodes, and approximate data. Those abilities are very interesting in the distributed graph processing subject because, translating the concepts into this area, they aim to put into account streaming executions, i.e. change the data at run-time, and achieve good results also using stale data perhaps retrieved by opportunist policies, e.g. usage of a resources without performing synchronization dealing with race conditions. Usually, a P2P network protocol builds an overlay over the actual network's topology. An overlay is a logic topology on the "physical" network that is formed, maintained and used by peers. Each overlay allows each peer to take advantage of its structure in order to perform specific operations (e.g. routing) or achieve topologies aimed at specific aims (e.g. the number of hops to reach destination).
In particular we focus on Gossip P2P protocols for building overlays. The Gossip communication paradigm perfectly fits to both P2P networks and large scale systems. Basically, it is an epidemic process [13] and consists of an information spreading over the network as a probabilistic process. Sometimes this process is employed aiming to calculate some properties which are approximated along the computation. Indeed, Gossip does not rely over structured overlays so no assumptions over the topologies are required. In Chapter 2 both the P2P approach and the Gossip paradigm are explained.

At best of our knowledge, no distributed graph processing frameworks orchestrate explicitly the computation as a layered (P2P) network. This thesis presents ONJAG ("Overlays Not Just A Graph"), a distributed framework built on this idea. ONJAG runs over Spark™ a distributed Bulk Synchronous Parallel-like data processing framework. ONJAG orchestrates the computation as a network simulation by defining a stack of protocols, following an ISO-OSI-like network stack [40]. Such protocols define their own logics and perform independent communications. The framework enables inter-protocols communications enhancing the computation with complex instrumentation, i.e. protocols could communicate one each other and behave accordingly. These behaviours let ONJAG to give a contribution to an area that has not been well and deeply studied yet. State-of-the-art distributed graph processing algorithms optimise the execution steps maintaining, by definition, the graph "as-is", i.e. the graph's topology is a constant variable of the problem which cannot be exploited by transformations. Conversely, the idea at the basis of ONJAG is the exploitation of this aspect assuming that

promising and novel solutions could be conceived and exploited. The overlay concept is the main mechanism of data exploitation. In addition, P2P researchers have claimed the suitability of their solutions also for the distributed graph processing environments but, due to the lack of tools, they limited to run simulations over P2P simulators such as PeerSim [51] or PeerfactSim.KOM [62], which are well-known Peer-to-Peer simulators. Instead, ONJAG consists in a real enabling platform.
Our framework enables the definition of general-networked protocol. However, our focus is on the creation of an enabling platform exploiting P2P-approaches. In Chapter 3 is presented the ONJAG framework in details.

The framework is bundled with a protocols toolbox. The toolbox contains a few representative protocols whose scenarios are typical P2P-like orchestrations. The k-core decomposition, the (random) peer sampling service and a topology overlay manager named T-MAN have been developed. The Gossip paradigm is employed by the last two approaches. Chapter 4 extensively describes those protocols.

In addition, a well-known problem in graph theory has studied [11]. Namely, the balanced minimum k-way partitioning problem, which is also called balanced minimum k-way cut. The aim is to divide an unweighed graph into k partition minimizing the number of the edges that cross the boundaries and, in addition, each partition has to hold approximately the same amount of vertices. The solution of this problem has some relevant applications, including biological networks, parallel programming, and on-line social network analyses and can be used to minimise communication cost, and to balance workload. Indeed, the vertex cut partitioning previously described in the distributed graph processing frameworks is exactly the balanced minimum k-way cut.
As this considerable amount of potential applications suggests, the problem is not new, as also are the proposed solutions. In the last years, a large amount of algorithms have been conceived, implemented and optimised for achieving good graph partitioning [28, 15, 34, 49, 57]. Some of these solutions are parallel but most of them assume cheap random access to the entire graph. Clearly, in a distributed graph processing environment it is not feasible to guarantee this property. Despite the interest on the problem, only recently the problem of computing an efficient min-cut in a distributed fashion have been proposed. Among the others JA-BE-JA [53] consists in a decentralised local search algorithm. It has been rewarded as "Best Paper Award" by IEEE SASO on 2013. JA-BE-JA aims at dealing with extremely large distributed graphs. To this end, the algorithm exploits its locality, simplicity and lack of synchronisation requirements. However, the assumptions made by original authors about porting JA-BE-JA over a real framework, which is based on the Pregel abstraction, has stood out not to be suitable as they claimed. To give an evidence of this fact we provide counter-examples, workarounds and alternative solutions in order to face such issues. Our main contribution is the analysis of the algorithm, its arrangement and porting to the ONJAG platform and also

some attempts of improvements. The ONJAG's Protocols toolbox includes an implementation of the JA-BE-JA algorithm bundled with some arrangements in order to adapt it to the Spark's BSP/Pregel-like environment. In Chapter 5 are described those points.

After that, a novel algorithm to solve the balanced minimum k-way cut is proposed. The proposal extends the JA-BE-JA algorithm orchestrating more than one overlay. Indeed, it exploits the P2P-like approach and the consequent overlays in order to improve the pre-existent JA-BE-JA's logic. The proposal pretends to be a proof of concept of applicability of merging P2P, Gossip and distributed graph processing subjects mainly exploiting overlays. More precisely, the T-MAN topology manager has been used exploiting a competitive orchestration between the protocols. The JA-BE-JA overlay modifies the T-MAN one and through that benefits of better achieved results, on the other hand the T-MAN overlay has to be repaired. In this scenario the orchestration takes advantage of the Gossip paradigm where each node is able to contribute to the result without requiring any synchronization or global knowledge as long as the P2P ability to face the churn and so its algorithms are self-healing allowing a certain degree of damaging, such as for the T-MAN topology. In Chapter 6 the proposal is described besides some experiments proving its goodness.

Finally, Chapter 7 summarizes the contributions of this thesis presenting final remarks and concluding with future works.

Part I

2

Background

What is the Matrix?

– Neo, *The Matrix*

Nowadays "Big Data" has became a buzzword though, we are really overwhelmed by many sensors and tools which enable the measurement and monitor of many instruments and behaviours of our lives. These tools are developed in many areas [47] e.g. finance, engineering, medicine and computational science. Indeed the recognition of the importance of this data is more and more accepted. Lots of efforts have been spent in order to crunch information from data [52]. Among the others the Bulk Synchronous Parallel model is employed thus, different orchestrations are performed [31].
Our focus is limited to the processing of large, vast data described by graph structures without taking into account the many issues those data require such as, for example, their collection, the heterogeneous formats, aggregations, missing values and so on. In addition many architectures could be employed for this task but, despite that, we do not focus over a specific one. Indeed the High-Performance Computing [14], High-Throughput Computing [64, 44] and Many-Task Computing [54, 54] are all eligible as viable computational approaches.

In this chapter we briefly report the evolution of the computational resources and architectures, the Map-Reduce programming model [12], the Bulk Synchronous Parallel model [66] and its derived implementations. Finally an overview about the P2P networks is given in order to enable the readers to transpose those concepts into the distributed graph processing subject.

2.1 Evolution of computing

Over the last 50 years the computational architectures evolved from a single processor computer to a set of many heterogeneous machines interconnected each other according to different kind of computational models. At the beginning, the improvement of CPU's speed was incredibly fast. In 1965 Gordon Moore, one of the founders of Intel, conjectured that the number of transistors per square inch on integrated circuits would roughly double every year. It turns out that the frequency of doubling is not 12 months, but roughly 18 months. Despite this increment the computation was not enough for the research activities therefore new architectures and as well as the integration of many processors and machines have been exploited to achieve better performance. Looking to the Top500 supercomputer list along the years, but not limited only on that, revealed that the following architectures, ordered by ascending adoption, have been used mainly:

- vector processors, machines designed to efficiently handle arithmetic operations on elements of arrays called vector. These machines exploit data parallelism. Their CPU's architecture, according to the Flynn's taxonomy [18], is usually SIMD (Single Instruction, Multiple Data) or MIMD (Multiple Instruction, Multiple Data). Nowadays most CPUs provide instructions to operate on multiple data; actually the CPUs are scalar (operate on single data) but there are vector units that run beside them. In the last decade also GPUs were used for general-purpose computation [20, 19] as they also are SIMD;
- symmetric multiprocessors (SMPs), SMP is a computer system that has two or more processors connected in the same cabinet, managed by one operating system, sharing the same memory and an Uniform Memory Access (UMA), and having equal access to input/output devices. Application programs may run on any or all processors in the system; the assignment of tasks is decided by the operating system. One advantage of SMP systems is scalability; additional processors can be added as needed up to some limiting factor, such as the data rate to and from memory;
- massively parallel processors (MPPs), MPP is a machine with many interconnected processors. Each processor has usually a different access to the memory, i.e. Non-Uniform Memory Access (NUMA);
- clusters, a group of computers (each of them with one or more processors) that are interconnected by a high speed network and are jointly put to work on a single computational task;
- distributed systems, a network of interconnected computers (or clusters), usually in different physical locations, employed for a common computational task.
  Noticeable is Grid computing, a computing model that is essentially a distributed system but usually used for a variety of purposes instead of a specific one.

Our work focuses mainly on cluster or distributed system. The key point is that we are in presence of some machines, each one with different level of parallelism, and a Non-Uniform Memory Access (NUMA).

2.2 Bulk-Synchronous Parallel model

The Bulk-Synchronous Parallel Model [66] is a bridging model between software and hardware for parallel computation.
The model defines a computation scheme abstracting from the concrete underline architectures. The main aim of the model is the efficiency of computation and to hide to the programmers the details of memory management, performing low level synchronization and similar operations.

The model is composed by three attributes:

1. components (or processors) performing computations or memory functions;
2. routers that deliver messages between pair of components;
3. facilities for synchronizing all or a subset of the components at regular intervals.

A computation consists of a sequence of supersteps. Each superstep consists of some computation tasks followed by a communication, message-passing based, between components. Realizing this in hardware provides an efficient way of implementing tightly synchronized parallel algorithms. The way to manage the start and end of each superstep is the barrier-style synchronization mechanism, which involves no assumptions about the relative delivery times of the messages within a superstep. The barrier could be potentially costly but it avoids deadlock and simplifies fault tolerance. In Figure 2.1 is shown a superstep and the 2 phases: computation and communication.

The cost in time of a superstep is defined by the longest computation, the global communication and the barrier synchronization.
Defining h as the max number of messages that each component exchange each other and g the ratio of the total operations performed by all components to the throughput of the routers, the performance of the BSP model is: $gh + s$ where s is a constant cost to perform initialization.
Keeping g fixed as the number of processors increase, and so the number of messages exchanged, means that the throughput of the routers has to be faster to enable the same performance. In other words, the performance of a BSP execution is directly proportional to the amount of data that has to be exchanged between the components and the throughput of the routers. As a conclusion if number of machines scale up, communication improvements have to be taken into account.

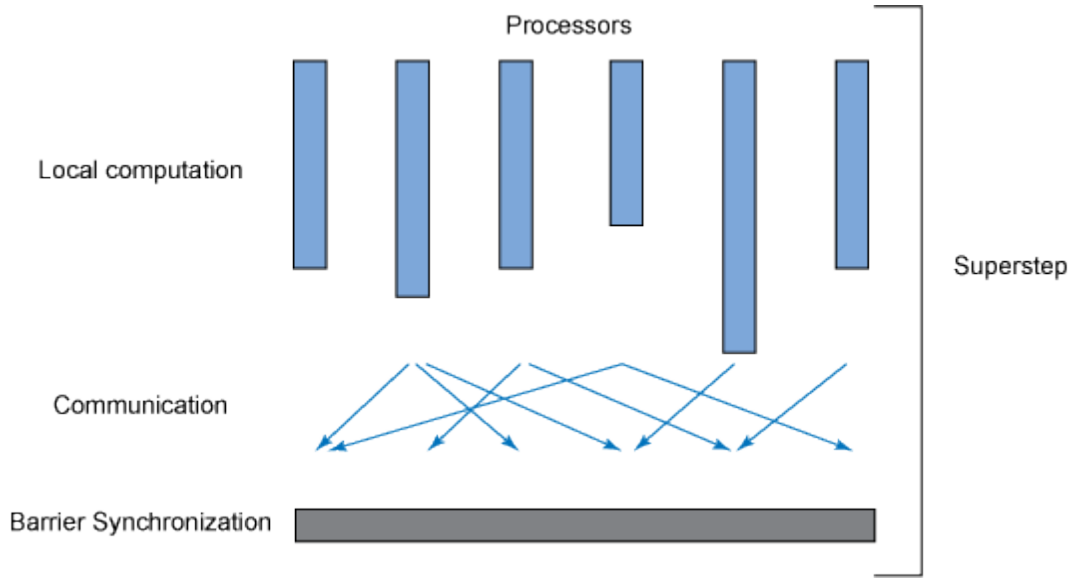


FIGURE 2.1: BSP's superstep

Examples of actual implementations of this model are MapReduce and Pregel, which are described in the following sections.

2.3 The MapReduce model

The MapReduce [12] is a programming model. Whose aim is to process large quantity of data using massive parallel computation. It has been shown to be suitable for many real tasks e.g. PageRank [6], K-Means Clustering [25], Multimedia Data Mining [26], Genetic Algorithms [1] and sometimes for Graph Analysis [9].
The programs written according to the model are automatically parallelizable and executable on distributed systems, clusters and even on commodity machines. The runtime system will take care about all the issues concerning the partition of the input data, load balancing and fault tolerance capabilities. Also, it will manage the inter-communication between machines and all the details that are architecturally dependent.
All these features provided by the runtime enable the programmer to write code without knowing actually the architecture where his code will be executed on, and also without high knowledge of parallel programming.

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs.

The computation is expressed by 2 functions: map and reduce. The abstraction is inspired by the map and reduce primitives present in Lisp [48] and many other functional languages [30].

1. The Map takes an input pair and produces a set of intermediate key/value pairs;

map	(k1, v1)	$\longrightarrow$ list(k2, v2)
reduce	(k2, list(v2))	$\longrightarrow$ list(v2)

TABLE 2.1: Signature of the map and reduce functions

2. The Reduce takes an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set. Typically just zero or one output value is produced per Reduce invocation.

The runtime groups together the pairs with the same key that are out-coming from the map invocation, and gives them to the reduce function. The final set of key/values pairs out-coming from the reduce execution is the result of the computation.

Note that, as shown in Table 2.1, the input pairs type has not to be the same in the final result.

At the beginning of a MapReduce computation, the input data is partitioned into small blocks. Each idle worker (machine) takes a block and performs on it the map function and finally writes on disk the intermediate pairs. After that, all the intermediate pairs are stored on disks and all the workers are idle. At this point each idle worker takes a block of intermediate data and executes the reduce function on it and again the result is stored on disk.
The consequence of this writing/reading mechanism of data is to enable the program to process large quantity of data that does not fit into the physical memory.
Figure 2.2, which is taken from [12], shows the overall architecture in a possible master-workers fashion.

In case of failures of one or more machines the runtime system reschedules the map/reduce tasks relying that are re-computable, i.e. the model relies on re-execution of tasks as the primarily mechanism of fault tolerance. This usually force the tasks to be independent.

Example

We briefly describe a simple and well-known example of MapReduce usage. The example aims to count the instances of different words of a given text document. The word-counter example takes as input the document and return a set consisting of pairs like *(word, #instances)*.
Taking as reference the document:

"Nunnery Scene", **Prince Amlet**. *William Shakespeare*

1: To be, or not to be, aye there's the point,
2: To Die, to sleep, is that all? Aye all

The word-counter program should return the following set of pairs: <*that,* 1>, <*is,* 1>,

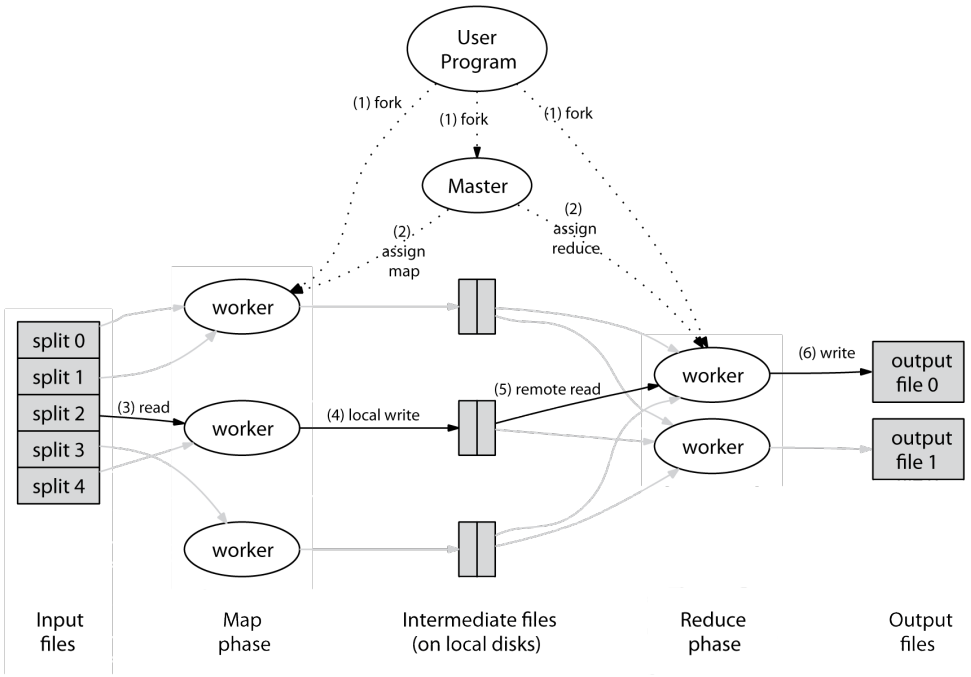


FIGURE 2.2: MapReduce possible architecture. Taken from [12]

<*the,* 1>, <*to,* 4>, <*sleep,* 1>, <*all,* 1>, <*there's,* 1>, <*die,* 1>, <*not,* 1>, <*aye,* 2>, <*point,* 1>, <*all?,* 1>, <*be,* 2> and <*or,* 1>.

The word-counter program has to be defined through the functions *map* and *reduce*. The *map* function takes a string line as input and returns a set consisting of the pairs like (word, 1). As a consequence, recalling the function signatures in Table 2.1, the *reduce* function takes as input the pair (word, list(number)) and returns the final pair (word, list(#instances)) where list(#instances) is a list consisting of only one entry which is the instances counter number.

The document's splitting into lines procedure is manage by the MapReduce environment which does it in parallel through distributed filesystems or manually managing the partitioned document onto the machines. At the beginning, the lines are processed by the *map* function that will split those lines into words and will return a simple set of (word, 1) pairs. The MapReduce environment will shuffle the pairs and the *reduce* function is called several times according to the key value of the pairs, i.e. the word, until no more merge operations are possible. The *reduce* function consists of summing the partial (already summed) values together until a single pair for each word is created. A pseudo-code listing of the 2 functions is:

```
1: function map(line: String):
2:     line.replaceAll(",", "") // Preprocess the line
3:     line.toLowerCase
```

```
4:    words ← line.split(" ") // split into words
5:    pairs ← EmptyCollection
6:    foreach word in words: // Creating the pairs
7:      pairs.add((word, 1))
8:    return pairs
9:
10: function reduce(key: String, values: List<Int>):
11:   sum ← 0
12:   foreach num in values:
13:     sum = sum + num // Sum the partial instances values
14:   return (key, sum)
```

The computation is embarrassingly parallel and the partial results are serialized on disks enabling the computation to manage documents that could not fit into memory.

Taking the reference document listed above, the *map* phase consists of 2 lines which are split into words, namely:

First line <*to,* 1>, <*be,* 1>, <*or,* 1>, <*not,* 1>, <*to,* 1>, <*be,* 1>, <*aye,* 1>, <*there's,* 1>, <*the,* 1> and <*point,* 1>

Second line <*to,* 1>, <*die,* 1>, <*to,* 1>, <*sleep,* 1>, <*is,* 1>, <*that,* 1>, <*all?,* 1>, <*aye,* 1> and <*all,* 1 >

Then the *reduce* phase sums the pairs with the same word as key value until the minimal set of pairs is created, e.g. *reduce*(to, list(1, 1, 1, 1)) ⇒ list(4) or, requiring more steps, *reduce*(to, list(1, 1)) ⇒ list(2), *reduce*(to, list(1, 1)) ⇒ list(2) and finally *reduce*(to, list(2, 2)) ⇒ list(4). The actual *reduce* operations order depends on the data partition onto the machines, the architecture where the program is running on and the implicit non-determinism of the parallel execution.

2.4 Pregel

Pregel [46] is a framework, created at Google, to compute algorithms on large scale graphs. It takes inspiration from the BSP model.

As said before, the MapReduce is suitable for some graph analysis algorithms but not for all of them. Some algorithms use global knowledge of the graph to compute their result or assume that the access to the data is uniform (e.g. the adjacent matrix stored in memory), in that cases MapReduce could suit well. Instead, other algorithms exploit a "network" vision of the graph, e.g. Centrality of vertices [55], and use mainly local information of a vertex and traverse the graph with a certain path along the execution.

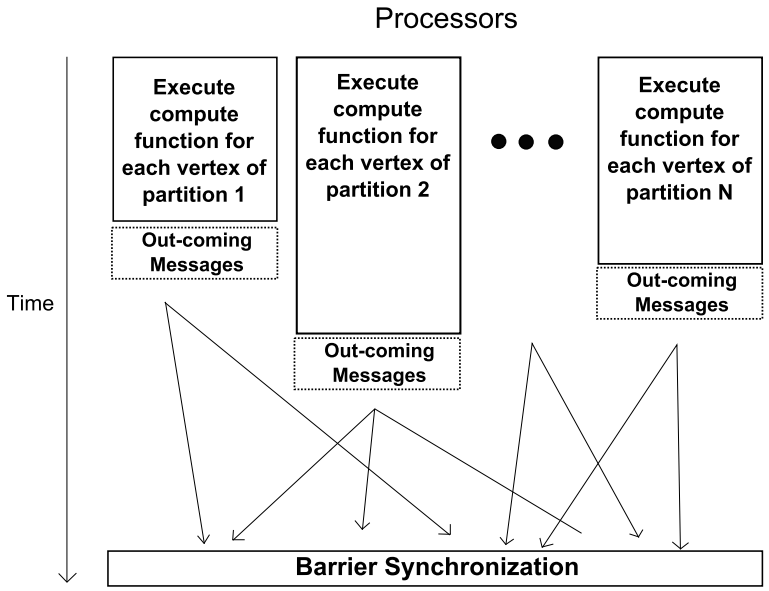


FIGURE 2.3: Pregel's superstep as BSP implementation

In these cases Pregel could suit better than MapReduce because it supplies the "lack of expressiveness" of MapReduce by changing the abstraction of the data.

Pregel's programs are vertex-centric, which means the operations are defined as "looking from the point of view of a vertex". As an inspired BSP model, a Pregel's computation is divided into supersteps and during each superstep executes the function associated to each vertex of the graph. The function is the same for all the vertices. Vertices communicate in a message-passing model, by sending and receiving messages. After the computation phase follows a set of out-coming messages from each vertex. The BSP's communication phase consists in dispatching these messages to the correct vertex. Each vertex during a superstep can vote to halt the computation. The execution ends when all the vertices vote to halt and no messages are out-coming at the end of a computation phase. Vertices are partitioned over the available machines.
Figure 2.3 shows a superstep from the point of view of the actual computation as BSP model, instead Figure 2.4 shows a partial execution from a higher point of view as the evolution of the graph. In Figure 2.4 the computation is executed by means of "graph as a network". The last point of view is the one that we are actually interested in.

The result of the total execution is changing the state of the vertices, for example inner attributes (e.g. K-Coreness [58]) but also the incident edges, i.e. the topology could be different at the end of the computation. In addition, along the supersteps, vertices could sum up some values into accumulators. Accumulators are global structures managed by the framework safely along the computations. An accumulator is parametrized by an accumulation function. The function, which has to be associative and commutative, sums up the current accumulator's value and the one given as parameter.

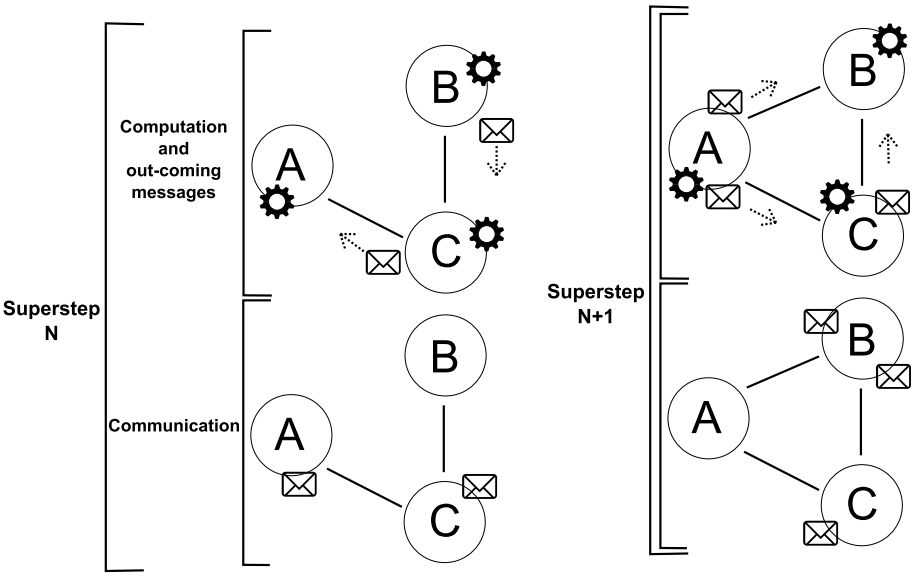


FIGURE 2.4: Pregel's superstep as graph evolution

Fault tolerance is achieved through checkpoints and re-computation. A checkpoint saves the data on disks in such a way that, in case of failure, it limits the number of the vertices that have to be recomputed and also they have not to be re-computed from the beginning, i.e. superstep 0, but from the last check-pointed superstep.

Each processor exchanges a lot of messages and it may happen that a subset of that is redundant or repetitive; an optimization is the use of combiners. Combiners are executed after the computation and the generation of the out-coming messages but before that the communication is actually performed. Combiners could merge the out-coming messages into a smaller set in such a way the data communication is lower (e.g. algorithms that perform a summation of values from incoming messages could create a combiner that merge out-coming messages that are going to the same vertex into one message with the value as the sum of the others).

2.4.1 Other implementations

Pregel is a closed and proprietary solution developed and used at Google. A lot of alternative implementations have been developed and each one differs for its actual implementation, the set of native operations provided to work on graph, and the fault tolerance mechanism that is employed.

First, it is important to notice that also other solutions exist which provide similar API to work on large scale graph but are not inspired directly on the BSP model. The most noticeable is GraphLab [45]. It works on shared memory and it is pretty fast on single machine and clusters. Actually the latest release utilizes also a BSP approach [24]. Another noticeable alternative is Combinatorial BLAS [7].

The main open-source Pregel implementation is Hadoop Hama [59]. Hadoop[1] is the open-source implementation of MapReduce and Hama is the implementation of Pregel that runs over Hadoop but does not have any fault tolerance mechanisms.
Yahoo! has also developed its own open-source framework called Apache Giraph![2]. It runs over Hadoop and performs fault tolerance through check-pointing.
Another framework is "GPS: A Graph Processing System" [56]. It is also BSP inspired and provides a specific domain language Green-Marl [29] that enables intuitive and simple expression of complicated algorithms.

2.4.1.1 Apache Spark framework

Our choice is the Apache Spark framework[3] [72]. It is also BSP inspired and, as the website reports, *"Spark is a general engine for large-scale data processing"*. It works over clusters and distributed systems.
It exploits data parallelism through *Resilient Distributed Datasets* (RDDs) [71]. RDD is a distributed memory abstraction consisting of immutable collection of objects spread across a cluster. A RDD is a distributed collection of objects that are read-only and can be generated only through deterministic and finite operations from either a dataset in a stable storage or other existing RDDs. The parallel distributed computation is expressed by means of operations over RDDs. A RDD is divided into a number of partitions, which are atomic pieces of information. Partitions can be stored on different nodes of a cluster. The RDD's operations, called transformations, are lazy computed. RDDs do not need to be materialized at all times instead, each RDD knows the chain of derivation that it is derived from, so it is capable to obtain the data by computing it from stable storage when needed. The derivations are computed in a lazy way through a logging operation, called *lineage*. The lineage is also the basic mechanism for fault tolerance, when fault occurs it is possible to recompute the RDD directly from the stable storage using the chain of derivation. The usage of checkpoints is also possible but it is expensive because of materializing of the intermediate results. On the other hand, it is useful in presence of long chains of derivations to prevent very long re-computation in case of faults.
The incremental transformation from a RDD to a new one is a constraint that gives some spots of exploitation and optimizations that, compared to the DSM (Distributed Shared Memory) that permits reading and writing operations in any memory location at any time, allows great improvements on fault tolerance and performances. However, the expressiveness is lower and RDDs are essentially suitable for algorithms that work with some data locality constraints over the data structures.
The Spark framework completely relies on the concept of RDDs. Spark is able to provide the MapReduce APIs and other operation (e.g. grouping by key and join) such a way

[1] The Apache Software Foundation, Apache™ Hadoop®! homepage: http://hadoop.apache.org
[2] The Apache Software Foundation, Apache™ Giraph! homepage: http://giraph.apache.org
[3] UC Berkeley AMPLab, Apache Spark™ homepage: http://spark.apache.org

it is possible to implement over it different kind of libraries (as opposite to Google and others that implement ad hoc solutions).
Some modules that cover different area of interest such as "MLlib" for machine learning and "GraphX" for graph analyses (it is an alpha experimentation in the moment of writing) are developed. Moreover GraphX [70] provides an implementations of Pregel. Actually in the previous versions of Spark *Bagel*, which is exactly an implementation of Pregel over Spark, was used instead of GraphX.

2.5 P2P networks

A network, e.g. such as Internet, is composed by a set of nodes partially connected each other through communication links. Each node seems like a computational unit with its own resources. It is usually identified over the network with an address or an identifier (id). The operations over the network are essentially message passing between nodes where they collaborate somehow each other to achieve some goals. Despite this quite wide generalization, the actual concretizations are mainly two scenarios: client-server and Peer-to-Peer (i.e. P2P).

In the client-server model there is a central node, called server, which provides some resources and the other nodes, called clients, exchange messages with it in order to gain access to some resources. This scenario is the main used on Internet (e.g. the World-Wide Web). The advantages of this model are the simplicity of the architecture and so the absence of communication overheads, and the great suitability to deal with proprietary resources so in cases where the resources are owned by someone and the clients "consume" them. The main disadvantage is the presence of a bottleneck over the network where clients communicate with the server and so scalability and availability issues eventually show up. Also the network's robustness is poor because in case the server fails the resources become immediately unreachable i.e. presence of a single point of failure. The workaround to deal with these issues is the replication of the server resources over more than one node and orchestrate them correctly (e.g. perform load-balancing technique [8]).

A P2P network is a network where each node is acting at the same time like server and client. Each node, called peer, provide some resources and it is interested in some others. As a consequence, the advantages are the network's robustness and scalability and the disadvantages are the overhead of the communications which is generated by the orchestrations of peers (i.e. the model is complex) and the lack of suitability in some scenarios (e.g. business private unshareable data).
Let's call *Protocol* the algorithms and application business logic that define the messages exchanged between peers.

A P2P network defines an overlay over the actual network's topology. An overlay is a logic topology of the "physical" network formed, maintained and used by peers. The overlays allow each peer to take advantage of their structure in order to perform various operations (e.g. routing) usually ensuring some properties (e.g. the number of hops to reach destination). A P2P network has to deal with the churn phenomenon. Churn is the presence of joining and leaving of peers along the time into the network. As a result, typically the networks have the property to be able to auto organize themselves. The overlays often ensure some properties therefore they are crucial for the protocols, which take advantage and rely on them. As a consequence, a lot of work and experiments have been done since the beginning about discovering different overlays that provide some properties and now they are categorized mainly in:

- unstructured, these overlays consist of a random topology where each peer has a certain amount of random neighbours. The properties provided by these overlays usually are not so strict and for that reason the scalability could be a problem. On the other side the overheads of managing and maintaining the topology is very low (e.g. fix the network against churn usually is not a big deal);
- structured, the topology is organized by following some rules and each peer has to take care of them and perform all the actions that are needed to maintain them. Usually the message routing complexity on such that overlays is logarithmic over the number of peers or its upper bound is anyway less than linear. This leads to better performances but also introduces overheads in order to maintain and manage the right topology. Churn is usually faced by maintaining backup links and over sized neighbours views.

2.5.1 Gossip

Gossip is a computational paradigm that perfectly fits to P2P networks and large scale systems. Basically, it consists of an information spreading over the network as a probabilistic process. Eventually each node reaches some local knowledge, which is actually a set of statistic variables of some measures. The information spreading is called *"epidemic"* (as the first time the paradigm was used [13]) because as a virus it contaminates the "community" or in other words the peers. Indeed, it is also called *anti-entropy* because tends to unify the diversification over the network or *rumour mongering* as an analogy.
More concretely, each peer does gossip with other *random* peers periodically and through this communication it updates its internal state and the approximation of the local knowledge. Comparing it to the P2P overlays, Gossip is an unstructured one that it is used in a different way to achieve different goals.
The main features of gossip protocols are:

- simple, a gossip protocol does not orchestrate a sequence of interactions and do not care about the network itself but it just defines the operations onto a single communication;
- scalable, as in an unstructured overlay there is no overhead maintaining a given net topology and indeed because each node does not need proper routing, the disadvantages of this kind of overlays do not occur;
- robust, intrinsically a gossip protocol manage the churn phenomenon and because of its random nature each node has eventually a fresh and coherent internal state;
- approximate, as a probabilistic process the knowledge obtained through this paradigm is usually an approximation.

A common gossip protocol relies over a *random peer sampling* service that provides a random subset of peers. This service is fundamental in order to execute the protocols. An implementation, which is also gossip-based, is presented in [33].

Analysing a gossip protocol as an epidemic process we can define the state of a peer as infected or susceptible. Moreover, the infection process along a single communication between two peers A and B, where A contacts B, could be categorized as:

- push, if A is infected then also B will be infected;
- pull, if B is infected then also A will be infected;
- push-pull, if any of them are infected then both of them will be infected.

It is useful to analyse the virus's speed spreading into the community for each category above. For clarity and simplicity we can suppose that each communication between pair of nodes occurs simultaneously over the network and we define each "communication phase" as a cycle. Taking an hypothetical initial network composed by n nodes whom only one of them is infected and all the other $n-1$ are susceptible, we can analyse each policy as:

- push, a susceptible node A at cycle i will not change state at cycle $i+1$ if none of the infected nodes contact it at cycle i or equivalently if all the infected nodes have chosen another node different from A at cycle i to contact. So the probability to be still susceptible at cycle $i+1$ is the probability to be susceptible at cycle i and no infection communications occur: $p_{i+1} = p_i \cdot (1 - \frac{1}{n})^{n \cdot (1-p_i)}$. And for instance with small p_i and large n it is approximately: $p_{i+1} = p_i e^{-1}$;
- pull, a susceptible node A at cycle i will not change state at cycle $i+1$ if it is contacted by another susceptible node at cycle i.
  The probability of that event is: $p_{i+1} = p_i^2$.

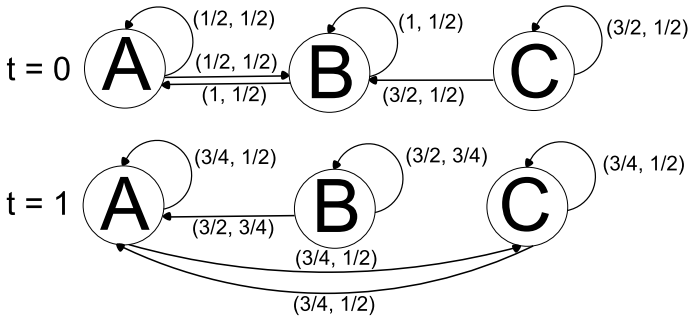


FIGURE 2.5: Example of the Push-Sum protocol calculating the average.

The pull probability decrease more faster than the push one and so it is more useful for the gossip purpose (i.e. to have a full infected community). Practically, the push-pull policy is usually used in order to take advantage of both the possibilities to infect nodes. More in details, it is likely the push policy mainly spreads the virus at the beginning because an infected node has great chance to contact a susceptible one instead for the pull policy the probability of a susceptible node to contact an infected one is low. Anyway, after a while it is the pull policy to contribute more to the epidemic phenomenon because the scenario will be reversed.
Demers at al. say [13] that, at least for the push policy, the number of cycles required to achieve a complete infected community are logarithmic to the number of nodes. As a consequence, gossip protocols are quite fast to converge to some valuable measures.

As an example we illustrate the *Push-Sum* gossip protocol [36]. The protocol computes measures like average or counting over all the nodes of the network. On each node i at cycle t are defined 2 variables: $s_{t,i}$ and $w_{t,i}$. By tuning the initial values of these two parameters is possible to calculate different measures.
The protocol's steps for each node i at cycle t are:

1: Let $\{(s_{t-1,r}, w_{t-1,r})\}$ be all pairs sent to i in round $t-1$ (generic node named as r)
2: Update the variables as $s_{t,i} \leftarrow \sum_{\forall r} s_{t-1,r}$ and $w_{t,i} \leftarrow \sum_{\forall r} w_{t-1,r}$
3: Choose a target node $j = rnd_i(t)$ uniformly at random
4: Send the pair $(\frac{1}{2}s_{t,i}, \frac{1}{2}w_{t,i})$ to j and i (yourself)
5: Update the measure's estimate s_i on node i as $s_i \leftarrow \frac{s_{t,i}}{w_{t,i}}$

As an example, putting the initial values $s_{0,i} = v_i$, $w_{0,i} = 1$ the protocol computes the average i.e. $\sum_{\forall i} \frac{v_i}{\#nodes}$ or $s_{0,i} = 1$ and $w_{0,i} = 0$ for all the nodes except one that has $w_{0,i} = 1$ computes the counting of the nodes in the network.

In Figure 2.5 are showed 2 cycles of a push-sum protocol example computing the

Cycle	**A**	**B**	**C**
0	$(1, 1)$	$(2, 1)$	$(3, 1)$
1	$(1, \frac{1}{2})$	$(\frac{3}{2}, \frac{1}{2})$	$(\frac{3}{2}, \frac{1}{2})$
2	$(3, \frac{7}{4})$	$(\frac{3}{2}, \frac{3}{4})$	$(\frac{3}{2}, 1)$

TABLE 2.2: The $(s_{t,i}, w_{t,i})$ pair values of each node for the first 2 cycles of the Push-Sum protocol example.

average over a network composed by 3 nodes; in Table 2.2 are showed the $s_{t,i}$ and $w_{t,i}$ values of each node at each cycle.

3

ONJAG: Overlays Not Just A Graph

Never send a human to do a machine's job.

– Agent Smith, *The Matrix*

In this chapter we introduce *ONJAG: Overlays Not Just A Graph*, a framework running over Spark and strongly inspired to Bagel which provides a network stack graph based view. It is written in *Scala*: a Java Virtual Machine compatible, functional and object-oriented programming language, which is highly suitable for parallel computations. The framework provides a structured view of the graph as a network that allows the definition of a stack of protocols (like the ISO-OSI stack). Each protocol could potentially create a completely independent overlay from the initial/real graph. This modularization into protocols enables the users to develop their solutions re-using third-party protocols. Thus, more interactions could occur between people and in this sense the framework could facilitate the design of new algorithms in a vertex-centred point of view.

Our main usage of *ONJAG* is the implementation of a stack of P2P protocols which is usually developed and designed for real networks. At best of our knowledge, this approach gives a contribution to a new area of distributed graph processing that has not been deeply explored yet.

Firstly, it is described the abstraction that overlays provide over the graph and the differences this approach implies if compared against the existing graph analyses on distributed systems. After that, we describe the architecture of the framework. Also an overview of the internals and Spark mechanisms are described. Finally, we present Misfire-Spark, a facility tool we created and used along our work.

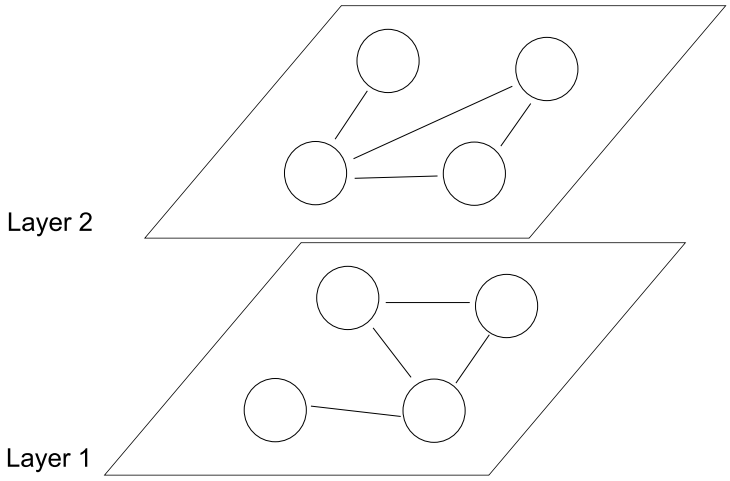


FIGURE 3.1: Example of two overlays seen as layers into an hypothetical stack.

3.1 Abstraction

The idea at the basis of our approach is to allow the users to create different topologies over the same vertex set. Ideally, the topologies are created starting from the same initial configuration (i.e. the initial edges) even if this is not mandatory. In case at each overlay is associated a protocol, the execution corresponds to run the distributed algorithms over the virtual networks. More precisely, the idea is to build a stack of protocols, where each protocol defines how to build the topology and provides the distributed algorithm logic. In Figure 3.1 are represented two topologies that share the same vertex set and are hierarchically organized as "layers" into a hypothetical stack of protocols.

The main difference between this approach and the existing graph analysing tools is to consider the data and the relation between data not as a fixed variable constrained into the problem which has to be solved. Lots of effort have been put onto the research of better methods on how solve problems over a graph but, by definition, the graph remains the same all over the computation. We think that some attempts trying to exploit also different abstractions are valuable and may they could lead to new results.
The main disadvantage of this approach is the problem of finding a good graph's partition onto the cluster machines in order to reduce communications between them, when the topology changes. The worst case clearly is if it does not exist a good partition and significant latencies are introduced into the computation. This aspect is accepted as a trade-off when the abstractions lead to faster (as supersteps needed) or better solutions.

Our approach is useful, mainly, when the problem to solve needs global knowledge but the solution differs for each vertex, i.e. global knowledge and local solutions. Sore of the current graph analyses algorithms exploit a similar approach but are not bound to this kind of problems, therefore, they do not build any facilities and they are not optimised for that.

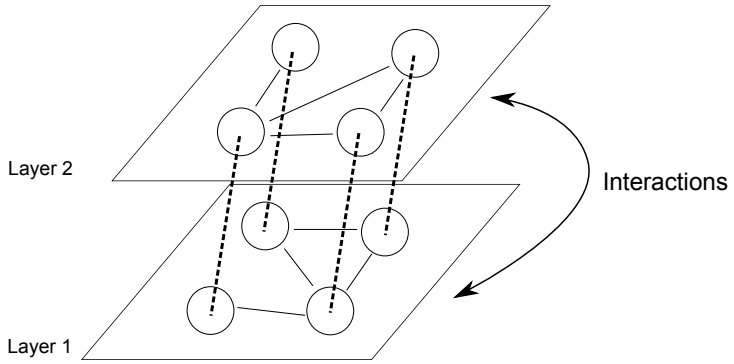


FIGURE 3.2: Intra-communication feature between two layers.

The iterative computations of *Bagel* could be actually be a computation of a single layer in the perspective of our approach. Somehow, from that point of view, our abstraction is a kind of "layered Bagel". Just a kind of because at each superstep, as showed in Figure 3.2, we allow each protocol to communicate with the others (respectively, inside each vertex) in order to exchange useful information and orchestrate themselves, i.e. intra-protocol communications onto the same vertex are performed. As a result, this feature allows to define algorithms that are built over more than one layer in order to reach the solution. Indeed, each protocol is able to cooperate or not with the others and so it can give a noticeable contribution.

3.2 Architecture

The framework's architecture is really simple to foster the user in focusing on the development of his/her own protocols. The architecture is inspired by the Bagel one (Pregel-like). Figure 3.3 shows the UML class diagram of the overall architecture.

The core, where all computations are performed, is the *OnJag* class. It is composed of a stack of user-defined Protocols. It provides the functions to create the initial set of vertices and messages. It manages all the stuff concerning the protocols and the communications between them. It is also possible to specify minor aspects like the maximum number of supersteps, the Spark's storage level and the persistence policy RDDs associated to. More in details, these policies concern about when the intermediate results, which are still RDDs, have to be dropped out from memory and disk.

As a Pregel-like framework, *Combiner* and *Aggregator* facilities are provided. For each protocol the user can define a combiner and an aggregator. Moreover, in order to allow fine grain tuning, it is possible to define conditions to drive protocol's check-pointing i.e. each protocol is treated separately by the framework.

The *Protocol* abstract class represents the abstraction of a single algorithm that runs on each vertex. Actually, the Strategy Design Pattern [22] takes place. The design pattern

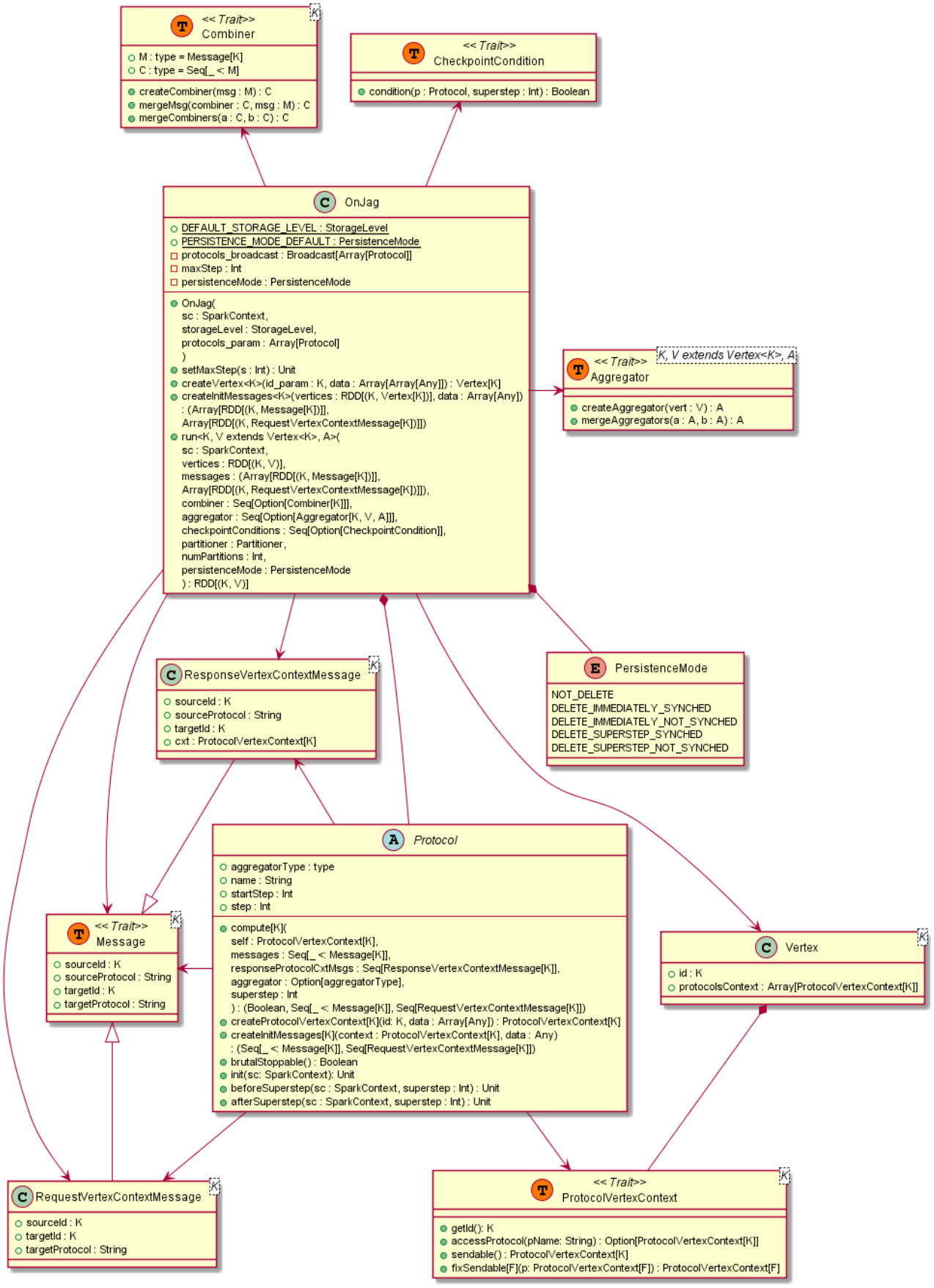


FIGURE 3.3: UML Class Diagram of the overall framework architecture

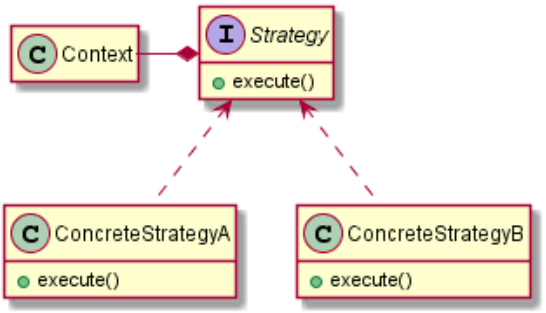


FIGURE 3.4: Strategy Behavioural Design Pattern

is illustrated in Figure 3.4. As a behavioural design pattern it defines the runtime behaviour and allows at runtime the usage of different algorithms into the same context by the client. The *Context* is the *OnJag* class and the abstract class *Protocol* is the *Strategy* interface. Each protocol orchestrates the protocol contexts which define the state of the vertices and the messages that are exchanged. A protocol also defines both the way to create the protocol context for each vertex and the initial set of messages. The user has to define his/her own protocols by extending this abstract class.
In order to decide when the computation is terminated and also to permit the definition of protocols that run all over the execution but without having a stop condition, each protocol has to declare if it can be stopped "brutally". The computation will be terminated according to the *vote to halt* policy. At each superstep, considering only the not brutal stoppable protocols, if all the vertices have voted to halt and the out-coming set of messages is empty then, the computation would be terminated.
Each protocol defines a custom starting superstep and the frequency that it will be executed with, i.e. how many supersteps will be skipped before it has to be computed again.
Moreover, in order to allow a finer usage of Pregel-like accumulators, there is the possibility for each protocol to define its own *"beforeSuperstep"* and *"afterSuperstep"* methods. Each method takes the Spark context as parameter allowing the creation and managing of RDDs by the protocol.

The class *ProtocolVertexContext* defines the vertex's state relative to a specific protocol. Its main feature is the possibility to communicate with other contexts of the same vertex. The access to the other protocols on the same vertex is guaranteed by the framework and it is completely transparent to the users. In addition, each context must be able to provide as needed a read-only message-passing-ready version of itself, called sendable version, in order to be retrieved by other vertices through message-passing. The sendable contexts are employed in order to reduce the required execution memory. Thus, a sendable context must be an optimised version of the original one but it has to be read-only, low memory footprint and no linking to other contexts are allowed, due to the potential loops of references. Therefore, those kind of contexts are not able to communicate with the other contexts on the same vertex. Nevertheless, that functionality

is pointless because such contexts are supposed to be used only during exchange data between vertices and just read-only operations with local scope are required. Notice that in the Pregel-like model there is not direct access to the neighbourhood neither other vertices although a certain level of exploitations could be adopted, due to the shared-memory support available into the partition over the machines.
As a result, a context that needs to access another one but, on another vertex (its sendable version i.e. like a read-only mode) it has to issue a RequestVertexContextMessage message and the framework will provide the information the freshest version packed into the *ResponseVertexContextMessage* message.

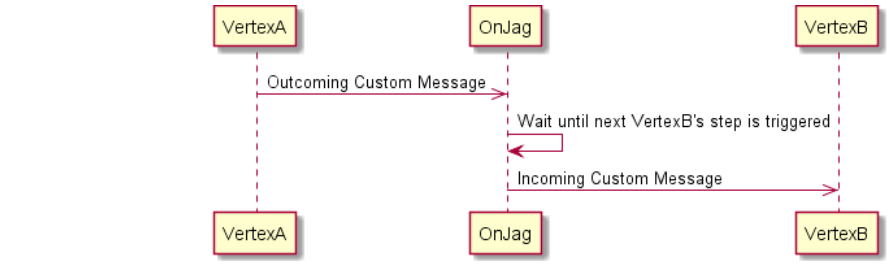
The framework manages 2 kind of messages: custom-made by the users and framework ones. The messages defined by the users are dispatched according to the sender and receiver "activation frequency" hence, at superstep t of the sender protocol, its outcoming messages are put since the $t+1$ superstep into the incoming set of the receiver. According to the receiver frequency eventually the protocol will be processed and at that time the messages are actually received. On the other side, the framework messages are the *RequestVertexContextMessage* and *ResponseVertexContextMessage* classes. They are used to communicate with the framework by each vertex's context in order to retrieve another context of a different vertex. The dispatch of these kind of messages is different from the custom-made of the users. A message sent at superstep t is immediately carried out by the framework and it will be processed according to the sender protocol's frequency. Calling that superstep j ($j > t$), at that time the message will be processed and the response will be delivered instantaneously to the protocol's context, i.e. the messages are processed, delivered and received during the superstep j.
In Figure 3.5 are shown both kind of messages dispatches as UML Sequence Diagrams.

As a conclusion, in order to develop a new protocol a user has to define a concrete *Protocol* class where the *compute* method describes the logic for all the contexts, one or more concrete *ProtocolVertexContext* classes and, optionally, custom messages by extending the default *Message* class. Inside the new defined protocol it is possible to access the other contexts of the stack through the *"accessProtocol"* method of *ProtocolVertexContext* and so use existing previous developed or third-party protocols.

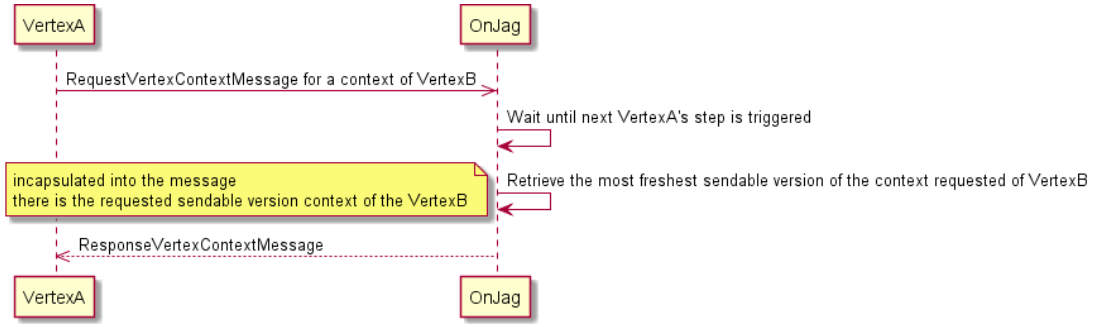
3.2.1 Internals

In this section we describe the framework's RDDs internals and how they have been structured in order to achieve a BSP computation stack and multi-protocol based.

Figure 3.6 shows the execution of an iteration for a protocol from the point of view of the RDDs.
First of all, the vertices are stored on a separate RDD. At each iteration, any protocol is described by the couple (*RequestMessage*s, *Message*s). An iteration starts creating the



(a) Custom Messages Sequence Diagram



(b) Framework Messages Sequence Diagram

FIGURE 3.5: UML Sequence Diagram of the 2 type of Messages

*ResponseMessage*s and combining the messages with the given combiner if present. After that, a join phase is involved, the operation is one of the most expensive ones because it involves the usage of all the RDDs. The result is an auxiliary RDD that is concretely used for the actual custom computation of the current protocol. The out-coming values of the protocol's execution are the updated state of vertices, the next *RequestMessage*s RDD and the out-coming *Message*s. ONJAG supports the communication between any protocol and, in order to realize it, a dispatch phase is required. This phase filters the out-coming *Message*s and appends them to the correct incoming RDD. This operation is clearly related to the number of intra-communications that occur along the computation and according to those could be an expensive time-consuming phase. Although, our experiments always showed that it was not a bottleneck at all. For convenient reasons the current messages and the newly generated ones are both maintained during an iteration and, at the end, only the new ones are saved. The customizable protocol's speed requires to maintain the "old" messages during the creation of the new one because some protocols could skip the current iteration and so the new generated messages are the previous one plus the out-coming from other protocols through the dispatch phase. At the end of each superstep the stop condition has to be verified. Recalling that the stop condition is verified when no vertices (of not brutal stoppable protocols) are active and no messages are dispatched, an intermediate computation has to be performed. The protocols could be defined as "brutally stoppable" and in that case the stop condition has to filter appropriately the messages in order to calculate proper value. More precisely, when no vertices are active, all the messages are filtered by dropping the ones sent by a brutal stoppable protocol. If no message survives the stop condition is verified.

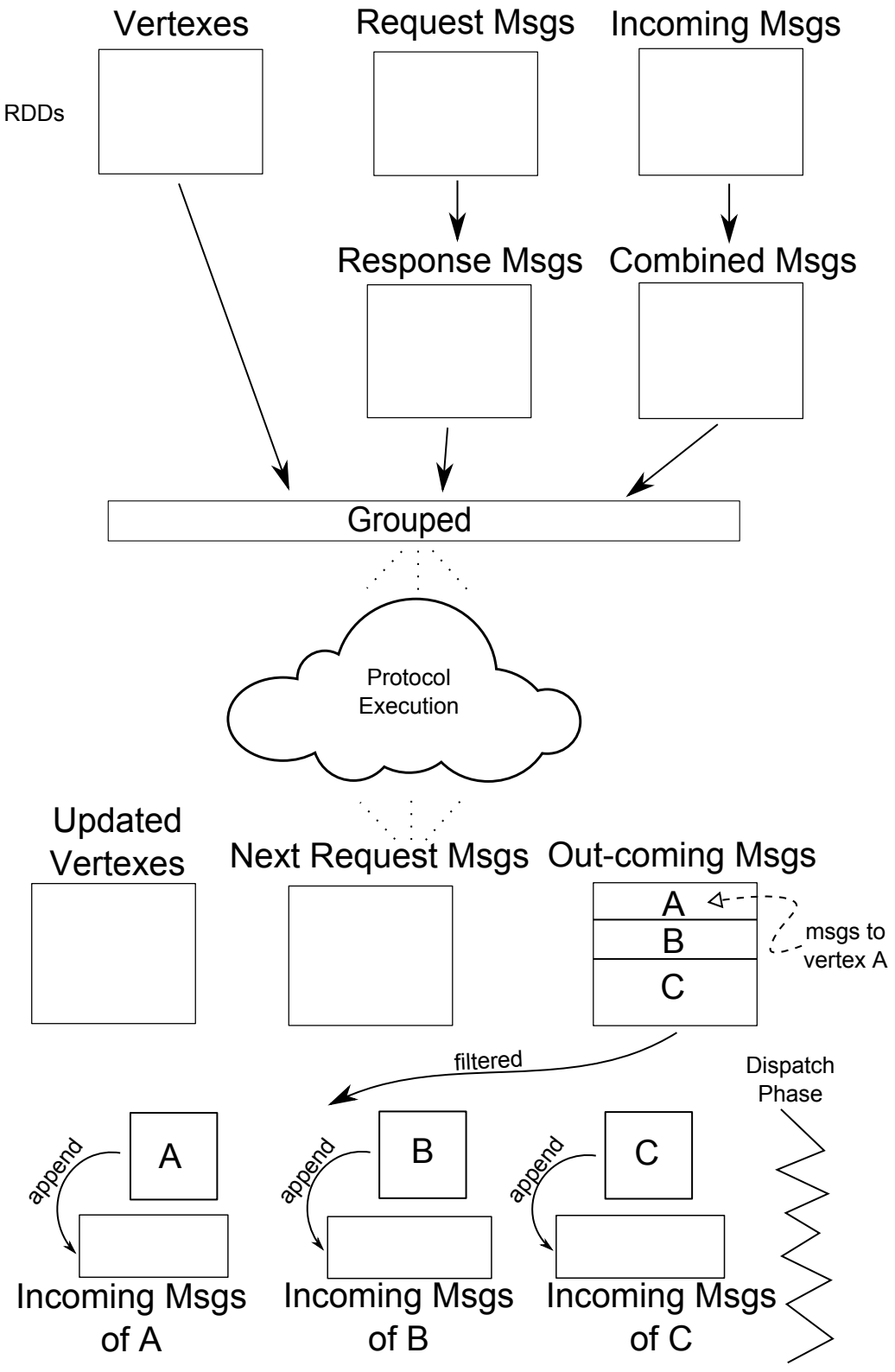


Figure 3.6: ONJAG RDDs

3.3 Spark Mechanisms

The development of the ONJAG framework stood out some aspects that have to be taken into account in order to execute a computation without errors. Spark is a quite new framework which is already being used by lots of major industries and out-performs the standard Hadoop Map-Reduce. However, we found that its maturity regarding (massive) iterative algorithms computation, at least related to the API's usability, has not been always clear and stable since the beginning. Nevertheless, lots of improvements have been realised, since the first version that we used (0.7.3), by the developer community. This also implied that we had to change our work following the continuously out-coming fixes, new functionalities as well as updates to APIs.

In this section we report our experiences describing the Spark basics and pointing out which issues have stood out giving the applied solutions. In addition, the ONJAG's details are taken into account regarding the Spark mechanisms.

First of all, the RDDs are not automatically managed by Spark but the developer is in charge of doing it. The RDDs are computed in a lazy way and there are some APIs, called "actions", that trigger the materialization of the RDDs (e.g. *count* and *foreach* APIs are actions). The materialization is like a temporary RDD until it will be persisted becoming a concrete RDD. The materialization is used just to perform the action and no data would be saved if the RDD's storage level has not been set appropriately. As a result, calling some actions iteratively without cache or persist the new temporary RDD leads to recompute it every time doing the *lineage* of operations that are needed to construct it. There are few policies regarding how the RDD could be persisted: on the main memory or on secondary memory storage or both of them, the serialization format (binary, using Java serializer or Kryo serializer). In order to develop an application that will not recompute all the data, we had to persist for each iteration the RDDs. Beside the explicit materialization and storage on memory of the RDDs, also their memory dropping has to be managed (i.e. no garbage collector exists). For that reasons, in order to avoid "Out of Memory" exceptions we had to take care also of the "unpersist" phase of the RDDs. This phase has been applied mainly on the intermediate RDDs that have been used just to calculate temporary data, and saving only the ones that are used in the next iteration. As a result, we defined and used the following usage pattern along the iterations:

```
newRdd = oldRdd.map(myFun) // Some operations over the oldRdd
newRdd.persist(myStorageLevel) // Flagged to be persisted
newRdd.foreach(x ⇒ {}) // Force evaluation ⇒ materialization and so
    persistence on the storage
oldRdd.unpersist(true) // True means that the RDD will be dropped
    synchronously
```

Another source of "Out of Memory" exceptions could be the presence of references in the current RDDs to former RDDs or partial data of them. The only fix is to ensure that no back-references occur. In addition, also Spark's internals require memory and until the 0.9.1 version it does not manage it appropriately. This is clearly an important behaviour and we struggled a lot facing workarounds but fortunately in the last few months an automatic manager has been developed (actually it is still under development, not completely finished yet) and it is available since the 1.0 version of the framework. Actually, many issues showed up that slow down our work considerably but along the time Spark is becoming more and more stable and it is being enriched with new functionalities.

Spark maintains for each RDD its lineage so the sequence of operations to derive it. With iterative algorithms the lineage will be eventually too long and consequently "Stack Overflow" exceptions are thrown. This is a systematic issue that the developer has to face with. The fix could be to increase the stack size, if the iterations are not too many, or check-pointing frequently. A RDD flagged to be check-pointed, when it will be actually materialized, will be saved on stable storage (e.g. HDFS) and its lineage will be truncated. The checkpoint operation is quite costly but it is the provided trade-off facility by the framework in order to compute iterative algorithms. In the time of writing there are discussions in the Spark's community about creating a "special" desirable RDD for the cases where the lineage is truncated without the needs of materializing on stable storage but relying on the Spark internals replication memory system. Attention must be put on the order of flagging the RDD as checkpoint-able and actually materialize it. The checkpoint would be performed only if it has been set BEFORE the materialization will take place.
Caching and check-pointing could seem similar but they actually do different things mainly regarding the lineage. The cache operation puts the RDD on the storage memory according to the storage level value but it does not truncate the lineage. In fact, in case the RDD would be persisted on the memory and a machine fault occurs the RDD is lost and through its lineage it is recoverable by the other workers. On the other hand, the checkpoint operation persists the RDD over a stable storage (i.e. no matters the storage level value) because it is fault-tolerant and so relying on that the lineage is truncated.
According with the above usage pattern and introducing check-pointing the actual usage pattern is:

```
newRdd = oldRdd.map(myFun) // Some operations over the oldRdd
newRdd.persist(myStorageLevel) // Flagged to be persisted
newRdd.checkpoint() // ⇐ checkpoint flag here
newRdd.foreach(x ⇒ {}) // Force evaluation ⇒ materialization,
    persistence and check-pointing
oldRdd.unpersist(true) // True means that the RDD will be dropped
    synchronously
```

RDDs are immutable so no modification should be done inside them instead new copies have to be created. Actually, it is possible to modify the objects that are contained into a RDD but carefulness must be put in that case because fault-tolerance and algorithm logic could fail if they are not taken into account appropriately. More precisely, in case of a partial fault, the missing part will be recomputed but, because the underline algorithm relied on side-effects, unexpected behaviour could show up. Nevertheless, if it has been managed adequately it would take advantage of that and the total computation requires a reduced memory footprint. ONJAG violates deliberately the "immutability contract" in order to take advantage of the memory exploitation. At each iteration the vertices are computed but, instead of saving the new states into a fresh RDD through a *map* operation, they are maintained onto the same RDD performing a *foreach* operation. ONJAG delegates the correctness of the underline protocols to themselves, i.e. the *compute* function of each Protocol must be stateless and does not rely on data that are not managed by Spark. We believe this is not a limitation due to the intrinsically nature of the computation that require the stateless constraints.

Custom activation frequency could be set for each Protocol along a computation, for that reason the internal ONJAG's implementation manages separately each protocol's data over some RDDs. As a consequence, a dispatch phase has to be performed in order to send the out-coming messages of a protocol to the incoming set of the receiver one.

Last noticeable detail is that the Protocol instances are broadcast over all the workers in order to permit the access of their attributes from the functions that are executed in parallel on each worker. Indeed, the attributes are really needed to be broadcast to ensure that the methods will use the same data. The broadcast of that data ensures that the workers will read coherently but no write operations are permitted. Therefore, in case an attribute is also written and not just read, it has to be defined as an *Accumulator* in order that Spark takes care also of it, i.e. attributes of a broadcast Protocol are already broadcast to the workers but could be needed to declare also some of them as accumulator values. In Figure 3.7 is showed the broadcast performed by a protocol regarding its attributes. The developer must be take care of this aspect in order to access the Protocol's attributes correctly. The protocols are re-broadcast after the invocation of their methods *init*, *beforeSuperstep* and *afterSuperstep*, and in that methods exceptionally is allowed also to update or replace the protocol's attributes values (either "normal" and accumulator ones).

3.4 Misfire-Spark: a development and testing facility

Even if Spark is becoming more and more mature at each new version, we had to struggle with many aspects: the not-well documented APIs, some unexpected behaviours, no oblivious optimization facilities for iterative graph algorithms, workarounds that are

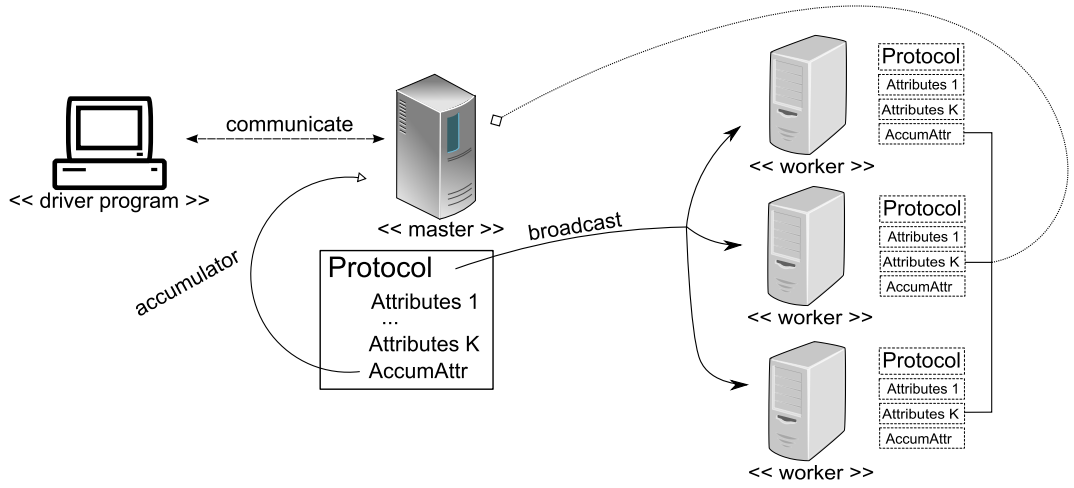


FIGURE 3.7: Protocol broadcast

known just inside the community and, last but not least, the reproducibility of a computation given the same initial seed.

The actual materialization of RDDs, the novel but in progress work for meta-data framework's internals garbage collection, no fault-tolerant accumulators (i.e. unexpected over-accumulation of data in case of a fault), shuffles always performed on disks, lack of debug breakpoints on closures which run over RDDs (i.e. no possibility to adequately debug) and randomness of data execution have been the main issues we had to face with.

For that reasons, we develop Misfire-Spark, a Spark wrapping tool in order to facilitate the developing and the testing phases. Misfire-Spark does only in-memory computations, exploiting the *Scala* native parallelism management and adopting ad-hoc solutions in order to enable fast execution in case the datasets fits in memory. It is valuable mainly during the developing phase when its often useful to verify the correctness of an algorithm. On the other hand, Misfire-Spark does not provide any distributed facilities and it is not fault-tolerant. These simplifications allow low memory meta-data footprints. All the data and RDDs are under the JVM garbage collector ensuring that a well-known memory management process is performed.

The Misfire-Spark's APIs are a wrapping of the Spark's ones and because of that they are tightly-coupled to the Spark's interfaces. The tool provides: the custom context, the operations managing RDDs, the operations managing PairRDDs (a special API set in case the RDD's type is fo the form $(K,\ V)$), accumulators and broadcasts. Actually it provides a large subset of all the APIs included the operations that we needed during our work. An important difference compared to the original APIs is that they are executed immediately i.e. no lazy computations, indeed the materialization is done automatically updating the current RDD instantaneously. The no materialization and caching managements have simplified the overall computations. In fact operations are exploited by "just in place" computation fashion.

In order to enable the possibility to reproduce tests and debug the algorithms there are provided two modes: sequential and parallel. In case the sequential mode is enabled all

the computations are performed sequentially, that is no parallelisms are exploited. As a consequence, the test's reproducibility is achieved and also debugging is simplified. On the other hand, in case of debug of a parallel unexpected behaviour the parallel mode enables parallel executions but maintains the capability to trigger breakpoints over any RDDs closures. Moreover, the parallel mode is parametrized over a N grade of parallelism parameter in order to monitor the computation over different range of parallelization.

Regarding the performance Misfire-Spark achieves great results because it exploits in-memory data, *Scala* native parallelism management and does ad-hoc solutions for some critical operations. More precisely, the *cogroup*/*groupWith*/*join* operations and in general all the grouping ones could be very time-consuming. For that operations a Divide&Conquer, which is a well-known algorithmic design methodology [37, 65], skeleton [10, 43] (also called ForkJoin) has been placed. The problem is: given two or more RDDs with the pairs $(K,\ A_1),\ \cdots,\ (K, A_n)$ grouping that RDDs into an unique RDD where each record is of the type $(K,\ Seq[A_1],\ \cdots,\ Seq[A_n])$ (i.e. each entry with the same K value are merged together). The Divide&Conquer skeleton is composed by a split phase where the data is divided into chunks and computations are done over them, and a merge phase where the results of the previous computations are collected and merged into the final RDD. The 2 phases could exploits parallel exploitation. In our case the split phase divides the RDDs into chunks of approximately equal size and the grouping is performed over each group set in parallel. The result is an half set of sub-grouped results. Thus, recursively the merge operation is performed over the groups until the final completely grouped RDD is created. Each merge operation is done sequentially due to the memory contention and race conditions. The conquer phase exploits parallelism over the form of a reversed binary tree where each sub-merge actions of the same level of deepness are done in parallel. The tree's deepness is determined by the partitions number, which is a parameter of ONJAG, and the RDD sizes.

4

The bundled Protocols Toolbox

Do not try and bend the spoon. That's impossible. Instead... only try to realize the truth.

– Spoon boy, *The Matrix*

In this chapter we present a set of protocols which are developed beside the framework. We create them as validations of our framework's implementation as long as to check the effectiveness of porting the P2P approach into the distributed graph analysing subject.

Each algorithm has been ported as a protocol by picking it considering the scenarios it deals with and so which challenges the framework has to face with. The implemented algorithms are: Distributed k-Core Decomposition, Random Peer Sampling and T-MAN. Actually another algorithm, called JA-BE-JA, has been implemented but it stands out many downsides. Therefore, our focus has been re-directed on a deep analysis of that algorithm and subsequently its porting. We present JA-BE-JA in the next chapter as a standalone case.

4.1 Distributed k-Core Decomposition

The k-Coreness protocol implements the Distributed k-Core Decomposition [50]. It is an algorithm that computes the k-coreness measure of each node in the network. The k-coreness metric is a well-established method for identifying a special family of maximal induced subgraphs of a graph called k-cores, or k-shells [3]. Informally, a node has coreness k if all its neighbours have coreness $\geq k+1$ and at least one has coreness equal to k. The process of computing the k-coreness is also called k-core decomposition: nodes are said to have coreness k if they belong to the k-core but not to the $(k+1)$-core. Figure 4.1 shows an example of k-core decomposition on a small graph.

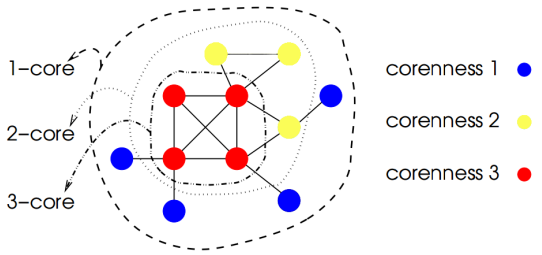


FIGURE 4.1: Example of k-core decomposition. Taken from [2]

As stated in the original paper the algorithm is directly translatable to the Pregel model and so also as an ONJAG protocol. We implement the algorithm in order to prove its actual suitability. In addition, it does not change the topology of the graph (i.e. only a static "overlay") and exploits only communications with the neighbourhood. Definitely, it is a perfect minimal, but still noticeable, usage case scenario.

The description follows with the formal problem definition, solution and implementation. The formal description we give has been deduced by the one presented by the authors of the original paper [50].

4.1.1 Problem Statement

Let G be an undirected graph $G = (V, E)$ with $N = |V|$ nodes and $M = |E|$ edges. We denote $d_G(u)$ the degree of u in G, whereas $G(C) = (C, E|C)$ is the subgraph of G induced by subset of nodes C, where $E|C = \{(u, v) \in E : u, v \in C\}$.
The concept of k-core decomposition is substantially given by the following two definitions:

Definition 4.1. A subgraph $G(C)$ induced by the set $C \subseteq V$ is a k-core if and only if $\forall u \in C : d_{G(C)}(u) \geq k$, and $G(C)$ is maximal, i.e., for $\bar{C} \supset C$, there exists $v \in \bar{C}$ such that $d_{G(C)}(v) < k$.

Maximality of k-cores guarantees uniqueness, i.e., there exists at most one k-core in G for every $k = 1, 2, \ldots$.

Definition 4.2. A node in G is said to have coreness k if and only if it belongs to the k-core but not the $(k + 1)$-core.

4.1.2 The Solution

The distributed algorithm is based on the property of locality of the k-core decomposition: due to the maximality of cores, the coreness of node u is the largest value k such that u has at least k neighbours that belong to a k-core or a larger core.

Notation. Let $k_G(u)$ denote the coreness of u in G and $neighbour(u) = \{v : (u, v) \in E\}$.

Theorem 4.3 (Locality)**.** $\forall u \in V : k_G(u) = k$ *if and only if*

(i) there exist a subset $V_k \subseteq neighbour(u)$ *such that* $|V_k| = k$ *and* $\forall v \in V_k : k_G(v) \geq k$ *and*

(ii) there is no subset $V_{k+1} \subseteq neighbour(u)$ *such that* $|V_{k+1}| = k + 1$ *and* $\forall v \in V_{k+1} : k_G(v) \geq k + 1$.

The locality property tells that the information about the coreness of the neighbours of a node is sufficient to compute its own coreness. Indeed, the algorithm's steps are:

1. each node computes the current estimation of its own coreness
2. each node communicates the estimation to the neighbourhood
3. each node receives the neighbour's estimates and update its neighbour's descriptors view accordingly

Each node executes the steps until the coreness estimation value does not change at the first step. The process goes on until convergence i.e. no change occurred. The algorithm is listed in the pseudo-code 1 and the core estimation function in the pseudo-code 2. The *core* variable contains the current estimation of the coreness and the array *est* represents the neighbour's descriptors view, i.e., each position of index n contains the last core received of the n-*th* neighbour.

4.1.3 Implementation

We implemented a straightforward protocol derived directly from the solution and the pseudo-code presented in the original paper. More precisely, we took into account the "one-to-one" model. The "one-to-one" model corresponds to the fully distributed peer-to-peer system. We ran the protocol over the same original paper's datasets. Actually, we were unable to retrieve the same version of each dataset, therefore we used the most keep updated ones directly from the SNAP library [41] at the time of writing. The results are listed in Table 4.1. In conclusion, the suitability of the approach, which is our most aim, is verified successfully because ONJAG executes the single-layer protocol without any issues and performed really fast computation due to the strict neighbourhood communications, i.e. we are interested on the viability of the computation.

Algorithm 1 Distributed algorithm to compute the k-core decomposition; routine executed by node u.

```
on initialization do
    changed ← false
    core ← d(u)
    foreach v ∈ neighbour(u) do est[v] ← +∞
    send <u, core> to neighbour(u)

on receive ⟨v, k⟩ do
    if k < est[v] then
      est[v] ← k
      t ← computeCoreness(est, u, core)
      if t < core then
        core ← t
        changed ← false

repeat at each superstep
    if changed then
      send ⟨u, core⟩ to neighbour(u)
      changed ← false
```

Algorithm 2 **int** computeCoreness(**int**[] est, **node** u, k)

```
for i = 1 to k do count[i] = 0
foreach v ∈ neighbour(u) do
    j ← min(k, est[v])
    count[j] = count[j] + 1
for i = k downto 2 do
    count[i − 1] ← count[i − 1] + count[i]
i ← k
while i > 1 and count[i] < i do
    i ← i − 1
return i
```

Graph	$\|V\|$	$\|E\|$	k_{avg}
CA-AstroPh	18772	19808	32.47
CA-CondMat	23133	93468	12.55
p2p-Gnutella31	62586	73946	2.52
soc-sign-Slashdot090221	82144	274601	7.33
soc-Slashdot0902	82168	474232	16.98
Amazon0601	403394	1693694	11.44
web-BerkStan	685230	38002975	13.57
roadNet-TX	1379917	192166	4.51
wiki-Talk	2394385	2510705	2.25

TABLE 4.1: Results obtained for "one-to-one" distributed k-core decomposition. Name of the data set, number of nodes, number of edges and average coreness

4.2 Random Peer Sampling

Generally, gossip algorithms are based over a hypothetical service where they can pick at any time a random peer of the network. Indeed, the Random Peer Sampling service [33] is crucial for such kind of gossiping. The service retrieves the peer's ids demanding to the users the conceivable collecting of other data.
In addition, the Random Peer Sampling service would be the only viable solution in case the user wants to pick a random vertex and no hypothesis exist over the key set of the vertices. A scenario would be the presence of a huge graph, which is a result of intermediate transformations, that does not preserve the initial key set or, another scenario would be in case stream parallelism is taken into account and so at real-time vertices join and leave the network i.e. presence of churn.

This service has been implemented as a protocol because of its importance and also because its overlay is the extreme case where at each superstep the topology changes drastically. Whenever such overlays will occur, challenges to find a good partition over the distributed system stand out. As a random process it is not possible to find an actual good partition. Anyway, as described in the original paper, the service is based on local knowledge and random-walk path discovering. In fact looking at the local peer picking process it is random but, however, global randomness is not really achieved due to the topology's changes that are somehow correlated between each pair of supersteps. Therefore, this peculiarity could be exploited along the execution as changes are likely friendly among the network-routing facilities and data locality memory systems. The Random Peer Sampling service collects a set of ids. ONJAG massively relies over the *RequestVertexContextMessage*s and *ResponseVertexContextMessage*s in order to retrieve the desired actual peer's data. The execution runs successfully enabling the intra-protocol communication feature to be tested in order to retrieve the peers discovered by the service.

4.2.1 The Algorithm

The problem statement is to provide a local service for each vertex defining an API *getPeer()* returning a random id of another vertex of the network. The service has to be local and so it has to use only local information. It has also to face the churn phenomenon.

The idea of the service is really simple. Starting from the actual topology it exchanges with a vertex of the current view a percentage of the view itself and merges the current view with the received one. The variables have been put into account are the policy by which vertex has to be elected and exchange the view with, how percentage of the view to share and consequently how merge the received one, the view size and also, as all the gossip algorithms, if the communications have to be push, pull or push-pull.

In order to face churn, each entry of the view contains an "age" field in order to be able to recognize and perhaps drop the most older ids.
The policies that are implemented and tested are:

- regarding how select the peer to exchange the view with: RAND, uniformly picked at random, or TAIL, which means the most older one i.e. the one with the greater "age" field value. The TAIL policy is more valuable compared to the HEAD one (select the most newest) because it implies to contacts a vertex that has changed hopefully a lot its view since last time instead the HEAD policy means to contact the freshest vertex that, by definition, has not any new ids to offer;
- the percentage of the view exchanged is fixed at $50\% - 1$ (half view minus one because it is sent also the vertex itself);
- how merge the current view with the one received by another vertex is tuned by these parameters:
  - H, healer parameter indicates how many vertices have to be dropped because they are the elder ones;
  - S, swapper parameter indicates how many vertices of the current views have to be dropped in order to maintain in the merged one the most newest;
  - R, random parameter indicates how many vertices of any views (current or received one) have to be dropped selected uniformly random.

  More precisely, the current view has size c and the received one has size $\frac{c}{2}$ so, in order to preserve the dimension, the parameters have the constraint: $H+S+R = \frac{c}{2}$. The extreme profiles are: blind ($R = \frac{c}{2}$), healer ($H = \frac{c}{2}$) and swapper ($S = \frac{c}{2}$). The H parameter is the only self-healing parameter that the service has employed. The healer is dropping all the old vertices and faces quite well the churn but the exchanged ids set is the minimum valuable one. As opposite, the swapper completely trusts the newest vertices thus, it likely fails in case of churn but maximize the entropy of the network increasing the diversity between views. In our work static graphs are considered so the swapper profile is always used. Although, topology changes occur for real-time scenarios or dynamic graph along the execution and a trade-off between H and S has to be found;
- two of the three infection patterns are taken into account: push and push-pull. Pull pattern is not taken into account because it is trivial to verify that if at any time no incoming links have existed to a vertex, which can happen spontaneously because of the randomness, it would be disconnected without any chances to join again the network.

The *getPeer()* API is implemented such a way it returns the next entry of the view starting from the one which is at zero position i.e. uses the view as a queue doing a

pop() call each time. In case all the view's entries are already returned the API returns an entry picked at random. Each time the view is refreshed by the background gossip service the API restarts by giving the first entry. As the original paper states until the queue (the view which is used as that) is not empty the local randomness is preserved. On the contrary, the case of returning an entry picked at random it does not preserve the property. Although, it is quite trivial not to fall in the worst case by tuning the frequencies of the API calls, the background gossip service speed cycle and the view size c.

The results of using the above policies are that the TAIL policy, compared to the RAND one, presents a randomness behaviour tendency along the supersteps to be more auto-correlated (the process is "more deterministic") but outperforms the RAND policy against the churn phenomenon. The last behaviour occurs because each time the elder vertices are contacted, indeed as quickly as it possible, the dead vertices are recognized and removed from the view. As a consequence, the in-degree distribution of the TAIL policy is more unbalanced compared to the RAND one because, for the same reason of the churn, choosing always the ones with the greater "age" value implies along the time to increase the neighbourhood of the elder vertices, i.e., their in-degrees. As a result for static network that does not present churn the RAND policy is preferable otherwise furthermore tuning is required.
As expectable the push-pull communication pattern converges more faster than the push one.

4.2.2 Implementation

The implementation of the Random Peer Sampling service as a protocol adheres exactly the original paper so providing the tuning of the parameters such as: c, H, S, TAIL and RAND policy. The Random Peer Sampling protocol is listed in pseudo-code 3. The original push and pull threads of the original implementation are merged together in order to fit with our platform. Every time a gossip algorithm has been implemented always we had to face with this merge operation due to the fact that platform does not explicitly manage the 2 kind of agent (i.e. push and pull) because it has been developed for general purposes stack-based computations.

4.3 T-MAN

Overlays are crucial in P2P networks and their characteristics are usually mainly described by the topology. The T-MAN algorithm [32] is a gossip-based overlays topology management. It allows to create and manage from scratch, i.e. from a random gossip network, a custom overlay topology. Managing the overlay implies the capability to face

Algorithm 3 Random Peer Sampling Protocol

```
   procedure randomSamplingService(Message[] incomingMessages):
 2:    p ← view.selectPeer()
       if push then
 4:      buffer ← createBuffer()
         send buffer as PushMessage to p
 6:    else
         send null as PushMessage to p
 8:    foreach m in incomingMessages
         if m is PushMessage then
10:        buffer ← createBuffer()
             if pull then
12:            send buffer as PullMessage to p
           merge(c, H, S, m.buffer)
14:      else if m is PullMessage then
           if pull then
16:          buffer ← createBuffer()
             merge(c, H, S, m.buffer)
18:    view.increaseAge()
       getPeerIndex ← 0 // Reset the index in order to restart from the head
20:
       function createBuffer():
22:      buffer ← ((myId, 0)) // Initialize the buffer, 0 is the initial "age"
         view.permute()
24:      view.moveOldest(H) // Moves the oldest entries to the end
         buffer.add(view.head(c/2 − 1))
26:      return buffer

28:    procedure merge(int c, int H, int S, buffer):
         view.append(buffer)
30:      view.removeDuplicates()
         view.removeOldEntries(min(H, view.size − c))
32:      view.removeHead(min(S, view.size − c))
         view.removeAtRandom(view.size − c)
34:
   function getPeer():
36:    if getPeerIndex < view.size then
         ret ← view.get(getPeerIndex)
38:      getPeerIndex = getPeerIndex + 1
       else
40:      ret ← view.pickAtRandom()
       return ret
```

the issues raised up by the churn phenomenon.
Recalling that exists mainly 2 types of P2P overlay: unstructured and structured ones; the topology that it constructs are mainly the structured ones.
Indeed, T-MAN is very interesting for our purposes because it enables to define, over the constructed overlays, all the algorithms that are using structured topologies and also it performs a kind of clustering over the graph that could be useful per se or to be exploited by other protocols. The macro behaviour of the peers' aggregation into groups or a well-defined structured onto the network is definitely a clustering feature.

4.3.1 The Algorithm

The problem statement is to organize a random network to a well-defined structured topology through an ad hoc overlay and also to maintain and manage it along the time.

The idea is that each peer would know or would be able to recognize which is its best neighbourhood if it has been allowed to decide. In another point of view it is possible to define a similarity function that captures this ability of "local ordering" for each peer. The question that has to be answered by each peer is: "How far is this peer relatives to myself?". Indeed, the overlay topology has to be defined by a *ranking function*. The function takes 2 peers and return a real number. More 2 peers are similar smaller it is the returned value.
Clearly, as the original paper reports, the idea puts a hypothesis about the presence of the "local ordering" thus, it has to be investigated how many overlay topologies are actually able to be defined in such way. Despite that, lots and useful topologies have been described so far.

The T-MAN algorithm is gossip based. It relies on a random peer sampling service. It is described by its push and pull phases. The implementation consists of a sequence of views that are exchanged at each cycle, each peer merges the incoming views with its one in order to maintain only the most similar peers according to the ranking function. At the beginning the views are exchanged according to the initial topology and some random peers are picked up from the random peer sampling service. The usage of the random sampling service is mainly to speed-up the convergence in fact the service provides peers as such long range links that could be useful in order to short-cut the network and do not walk through all of it, e.g. at the beginning a very dissimilar neighbour set could occur. The random peer sampling service is also the primary mechanism to recover and maintain the topology against the churn phenomenon. More precisely, the view's entries have an "age" field and according to that old peers are removed along the cycles in order to maintain only the live nodes.

4.3.2 Implementation

The protocol's implementation of the T-MAN algorithm, which is listed in pseudo-code 4, has to merge the push and pull phases according to the platform. In addition, due to the message-passing system of the BSP model, no real-time neighbour's states were available. Thus, it was not possible to add a random peer to the view as-is like the original paper [32]. We implement a pipelined process in which, between the push and the pull phases, there is a superstep of delay in order to retrieve the random peer contexts that are needed. The protocol's attributes are c, H and R and they describe respectively the view size, the healing parameter (i.e. how aggressive the protocol has to be in order to face churn) and how many random peers are picked up from the random peer sampling service for each message exchange.

We validate our implementation creating 3 standard, well-known topologies: line, ring and torus. Line and ring are 2D simple topologies and torus is a 3D one. We generate the coordinates for each peer and we used the euclidean distance metric as ranking function, which is defined as:

$$eucledian_distance(a,b) \;=\; \sqrt{\sum_{i=1}^{N}(a_i+b_i)^2}$$

where a, b are two points and N is the dimension of the space.
The torus azimuthally symmetric about the z-axis is described by the following parametric equations:

$$x = (c + a\ cos\nu)\ cos\mu$$

$$y = (c + a\ cos\nu)\ sin\mu$$

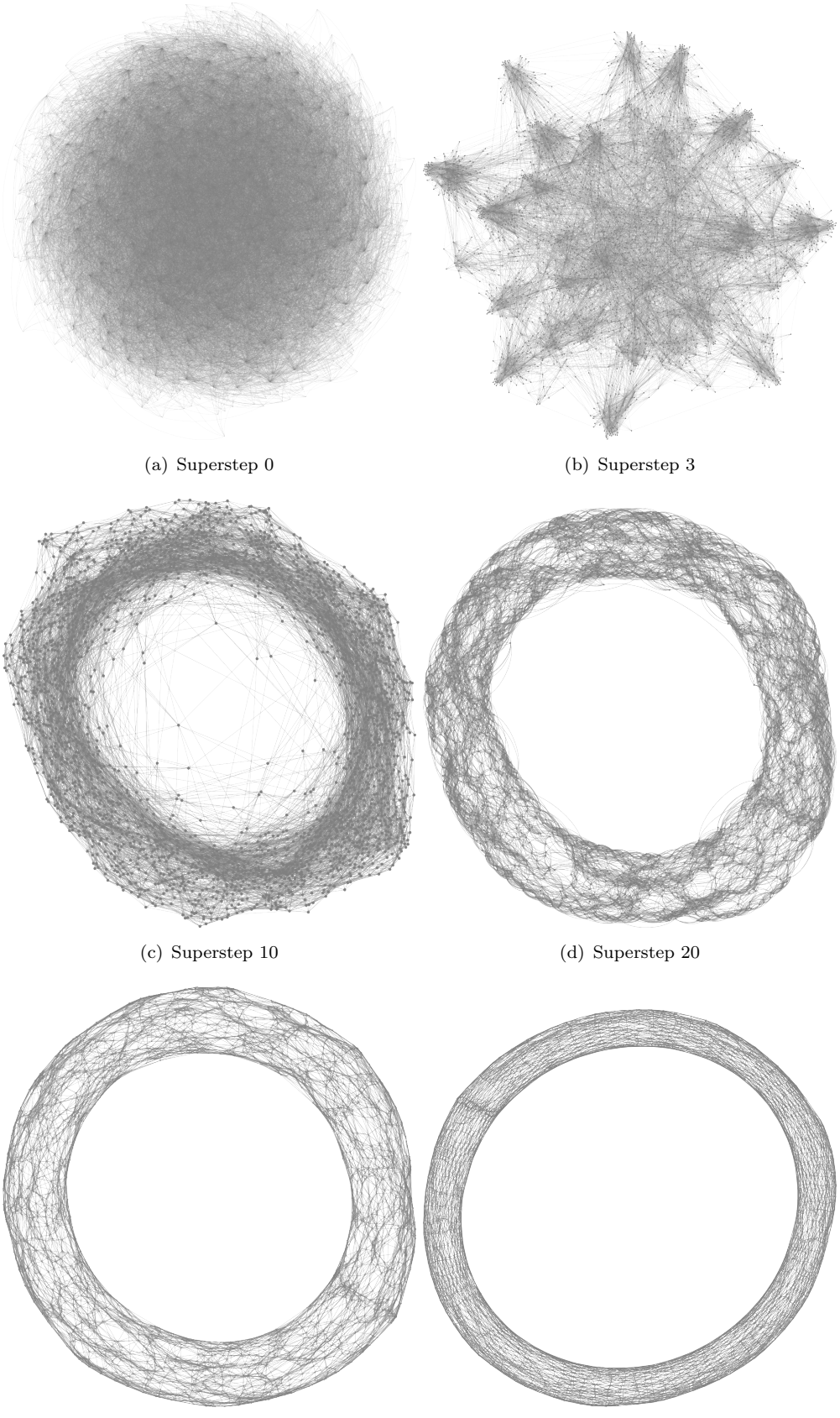
$$z = a\ sin\mu$$

where c is the radius from the center of the hole to the center of the torus tube and a is the radius of the tube.

In Figure 4.2 is shown the evolution of a synthetic network composed by 2700 nodes and 181951 edges to a torus running the T-MAN protocol. The random network has been created through the Erdos-Reyni [16] method using a wiring probability of 5%. The torus's parameters are $c = 15$, $a = 5$ and the angle sampling are 90 steps for the torus ($\nu \in [0; 2\pi]$ with a step of $\frac{2\pi}{90}$) and 30 steps for the tube ($\mu \in [0; 2\pi]$ with a step of $\frac{2\pi}{30}$). The T-MAN protocol's attributes are: $c = 8$, $H = 0$ and $R = 1$. The supersteps required to create the topology are very few and indeed a good approximation is reached after 20 supersteps.

Algorithm 4 TMAN Protocol

```
   procedure TMAN(Message[] incomingMessages):
 2:    p ← view.selectPeer()
       randomChunk ← randomChunks.next()
 4:    // Ask immediately for a new chunk because it is sure that will be a push phase
   also at the next superstep
       ask next random chunk of R peers
 6:    // Prepare push view buffer sorting it according to myself
       pushView ← prepareView(this, randomChunk)
 8:    send pushView as PushMessage to p
       foreach pullPeer in pendingPullList
10:      randomChunk ← randomChunks.next()
         // Prepare pull view buffer sorting it according to the other peer
12:      pullView ← prepareView(pullPeer, randomChunk)
         send pullView as PullMessage to pullPeer
14:   pendingPullList.clear()
      randomChunks.clear()
16:   globalView ← view.clone()
      foreach m in incomingMessages
18:      if m is PushMessage then
            globalView.merge(m.view)
20:         pendingPullList.append(m.sender)
            ask next random chunk of R peers
22:      else if m is PullMessage then
            globalView.merge(m.view)
24:   view.selectView(c, globalView)
      view.increaseAge()
26:
      function prepareView(Peer peer, Peer[] randomPeers):
28:      buffer ← EmptyView
         buffer.merge(view)
30:      buffer.merge(peer)
         buffer.merge(randomPeers)
32:      buffer.removeOldest(H)
         // Select view sorting the entries according to the peer parameter
34:      buffer.selectView(c, peer)
         return buffer
36:
   class View:
38:   procedure selectView(int size, View buffer):
         selectView(size, buffer, ownerPeer)
40:   procedure selectView(int size, Peer pivotPeer):
         selectView(size, EmptyView, pivotPeer)
42:   procedure selectView(int size, View buffer, Peer pivotPeer):
         view.merge(buffer)
44:      view.removeDuplicates()
         view.remove(pivotPeer)
46:      sort view with
            (a,b) ⇒ rankingFunction(pivotPeer, a) < rankingFunction(pivotPeer, b)
48:      shrink(0, size)
```



(a) Superstep 0

(b) Superstep 3

(c) Superstep 10

(d) Superstep 20

(e) Superstep 50

(f) Superstep 100

FIGURE 4.2: Evolution of a random network to a torus using TMAN

Part II

5

JA-BE-JA: a distributed balanced minimum k-way graph partitioning algorithm

What is "real"? How do you define "real"?

– Morpheus, *The Matrix*

In this chapter we present JA-BE-JA [53], a distributed algorithm to calculate the balanced minimum k-way graph partition problem. The problem is also known as the balanced minimum k-way cut. It is a well-known problem in graph theory [11] but there is recent interest [4, 35]. The aim is to divide an unweighed graph into k partitions minimizing the number of the edges that cross the boundaries and, in addition, each partition has to be approximatively of the same size. The solution of this problem has some relevant applications, including biological networks, parallel programming, and on-line social network analysis and can be used to minimise communication cost, and to balance workload. Some of these solutions are parallel and, most of them, implicitly assume to have a not expensive random access to the entire graph. JA-BE-JA performs the computation in a distributable manner. It exploits a local search optimization relying only on the local knowledge for each node, i.e. the state of the node and its neighbourhood, and on an asynchronous communication system which is executed periodically. Nodes communicate only through messages over edges of the graph. The original authors stated the algorithm's suitability to the Pregel and GraphLab frameworks and generally to the BSP/Pregel-like models. Indeed, they claim that simple adaptations will be needed to port the algorithm to the actual frameworks. However, we found that the JA-BE-JA's assumptions are not verified into such models and it is not adaptable "as-is" to Pregel or GraphLab frameworks. Further, we give counter-examples, workarounds, and alternative solutions that describe our experience on porting JA-BE-JA over Pregel.

The JA-BE-JA algorithm is studied among two models in the original paper: one host-one node ("one-to-one") and one-host-multiple nodes ("one-to-many"). The "one-to-one" model describes a fully distributed scenario such as a P2P network. The "one-to-many" model corresponds to a hypothetical distributed framework environment which exploits shared-memory among each machine short-cutting the communications performed between the vertices located on the same machine. This latter solution is not Pregel-like because introduces the concept of subsupersteps (involving intra-superstep communications) and different user-defined functions are executed for each vertex, depending on the exploitation of the shared-memory. If on the one hand this approach speeds-up the computation, on the other hand it is not adequate to be exploited by most of the Pregel-like distributed parallel data processor. Therefore, our focus is on the "one-to-one" model.

The ONJAG's Protocols toolbox includes an implementation of the JA-BE-JA algorithm comprehensive of some arrangements in order to adapt it to the Spark's BSP/Pregel-like environment. The chapter follows giving the formal problem statement of the balanced minimum k-way cut, which is derived by the one presented by the authors of the JA-BE-JA paper, the proposed solution by JA-BE-JA and an analysis of the issues that stand out. At the end, our results and the achieved improvements are presented.

5.1 Problem Statement

Consider an undirected graph $G = (V, E)$, where V is the set of nodes (vertices) and E is the set of edges. A k-way partitioning divides V into k subsets. Intuitively, in a good partitioning the number of edges that cross the boundaries of components is minimized. This is referred to as the min-cut problem in graph theory. Balanced (uniform) partitioning refers to the problem of partitioning the graph into equal-sized components. The equal size constraint can be softened by requiring that the partition sizes differ only by a factor of a small ϵ.

A k-way partitioning can be given with the help of a partition function $\pi : V \rightarrow \{1, \cdots, k\}$ that assigns a colour to each node. Hence, $\pi(p)$, or π_p for short, refers to the colour of node p. Nodes with the same colour form a partition. We denote the set of neighbours of node p by N_p, and define $N_p(c)$ as the set of neighbours of p that have colour c:

$$N_p(c) = \{q \in N_p : \pi_q = c\}$$

The number of neighbours of node p is denoted by d_p, and $d_p(c) = |N_p(c)|$ is the number of neighbours of p with colour c. We define the *energy* of the system as the number of edges between nodes with different colours (equivalent to edge-cut). Accordingly, the energy of a node is the number of its neighbours with a different colour and the energy

of the graph is the sum of the energy of the nodes:

$$E(G, \pi) = \frac{1}{2} \sum_{p \in V} (d_p - d_p(\pi_p))$$

where we divide the sum by two since the sum counts each edge twice. Now, we can formulate the balanced optimization problem: find the optimal partitioning π^* such that

$$\pi^* = \underset{\pi}{\operatorname{argmin}}\, E(G, \pi)$$

$$s.t.\ |V(c_1)| = |V(c_2)|,\ \forall c_1, c_2 \in \{1, ..., k\}$$

where $V(c)$ is the set of nodes with colour c.

5.2 The Solution

The basic idea is to initialize colours on each node uniformly at random, and then to apply heuristic local search to push the configuration towards lower energy states (min-cut). We recall that the energy of the system is defined as the number of edges between nodes with different colours. The energy of a node is the number of its neighbours with a different colour. When applying local search, the key problem is to ensure that the algorithm does not get stuck in a local optimum. For this purpose, the algorithm employs the simulated annealing technique [63].

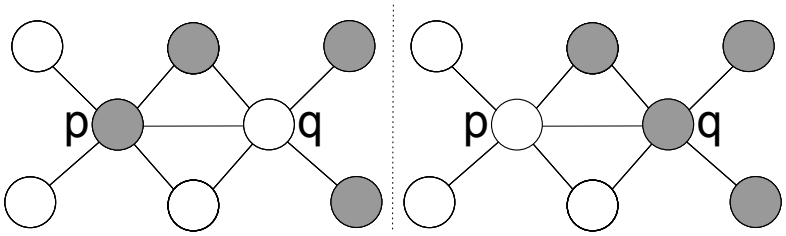
The heuristic local search consists of running in parallel all the nodes in which each node tries to swap its colour with a neighbour or random one in order to decrease the energy. More precisely, they swap their colours according to the following equation:

$$d_p(\pi_q)^\alpha + d_q(\pi_p)^\alpha > d_p(\pi_p)^\alpha + d_q(\pi_q)^\alpha \tag{5.1}$$

where α is a parameter of the energy function. In case of $\alpha = 1$ the swap occurs exactly when the operation will bring the new configuration to a lower energy state. In case of $\alpha > 1$ the swap occurs even when the new configuration will not be directly better but it leads to a higher probability of colour swapping in the future. Figure 5.1 shows an example in which a swap takes place between the nodes p and q. More precisely the equation, assigning $\alpha = 1$, is evaluated as:

$$3 + 3 > 1 + 1$$

To avoid becoming stuck in a local optimum, it is used the well-known Simulated Annealing (SA) technique. It is introduced a temperature (T) and it is decreased over time,



(a) Before swap (c) After swap

FIGURE 5.1: Example of a swap according to the JA-BE-JA's algorithm.

similarly to the cooling process in [63]. The updated equation becomes:

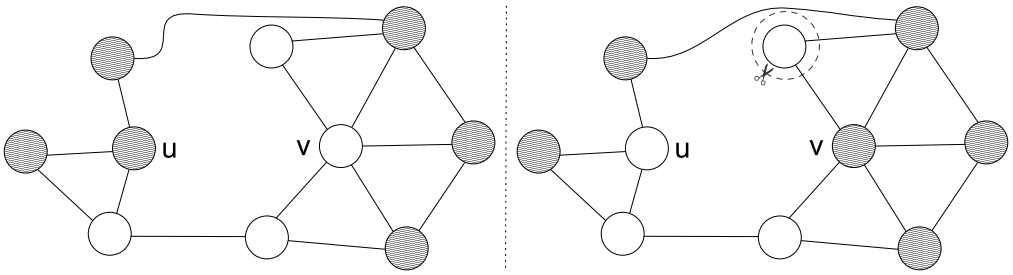
$$(d_p(\pi_q)^\alpha + d_q(\pi_p)^\alpha) \times T > d_p(\pi_p)^\alpha + d_q(\pi_q)^\alpha \tag{5.2}$$

As a result, at the beginning the configuration might move to a direction that degrades the energy function, i.e. nodes exchange their colours even if the energy is increased. Over time, however, the swaps will become more conservative and will not allow those exchanges that result in a higher energy. The two parameters of the SA process are (i) T_0, the initial temperature, which is greater than or equal to one, and (ii) δ, that determines the speed of the cooling process. The temperature at cycle r is calculated as $T_r = T_{r-1} - \delta$. When the temperature reaches the lower bound 1, it is not decreased anymore. From now on, the decision procedure falls back on using equation 5.1. It is also performed a multi-start search, by running the algorithm many times, starting from different initial states.

5.2.1 Objections

Analysing the JA-BE-JA algorithm has revealed some objections regarding the correctness about the final partition result. More precisely, the returned partitions could not be valid according to the k-way cut definition. The k-way cut problem defines the final k partitions $P_i | \forall i \in [1, k]$ to be connected components, a property which is well-known in the graph theory. In other words, for each partition, which is a set of vertices, must exists a path between any pair of the contained vertices, formally: $\forall n \in [1, k]$ $\exists p$: path s.t. $p = v_i \rightarrow \cdots \rightarrow v_j$ $\forall v_i, v_j \in P_n$. The description follows by pointing out how such property is violated.

First of all, the JA-BE-JA's bootstrap phase consists of colouring all the vertices in order to create the initial partition sets. The balance between each partition's size is guaranteed by an user pre-defined partitions configuration or by a picking up process of a colour for each vertex according to a certain probability distribution. In case of doing it through the picking process (but also a wrong user pre-defined configuration), the initial partition configuration will likely be not valid because there is the possibility



(a) Before swap (c) After swap

FIGURE 5.2: Scenario where the usage of $\alpha = 2$ leads to loose the connected components property

that $\exists P_i \in P_{1...k},\ v_i, v_j \in P_i$ s.t. $\nexists p$: path where $p = v_1 \to \cdots \to v_j$. JA-BE-JA has no mechanisms to verify the correctness of the initial configuration, neither the ability to fix it along the computation. As a result this scenario could bring invalid final partitions according to the connected components property requirement of the k-way cut.

Unfortunately, the new requirement of a valid coloured partitions configuration at the beginning is not enough to guarantee the correctness of the final one. The space-solution problem presents lots of spots for local minimum therefore workarounds have to be placed in order to avoid them. JA-BE-JA employed the Simulated Annealing technique defining the energy as the edge-cut. However, recalling equation 5.2, the SA technique deliberately does wrong swaps at the beginning until the temperature T reaches 1. These swaps do not take into account the preservation of the connected components property. Hence, partitions fragmentation phenomenon could occur, e.g. 2 nodes swap their colours each other and at the resulting configuration they are isolated, i.e., no neighbour has the same colour of them. At the end not valid partitions configuration could show up. As a consequence this issue does not permit the usage of the simulated annealing without breaking the property resulting in a not proper min-cut.

The initial coloured configuration of the partitions is fundamental to lead to a valid result, alternatives that could be investigated are the capabilities of merging the sparse "sub-partitions" of the same colour to an only one. Clearly, this approach is hard to be resolved and no advantages stand out w.r.t. colouring properly the initial configuration. If the SA technique has avoided, then the decisional equation for each swap would be 5.1.

The usage of α is motivated in [53] because there are some cases where doing the exchange also when it is not properly convenient for both the nodes leads to better solutions, however this behaviour does not always preserve the connected components as showed in Figure 5.2. The vertices pair (u, v) will swap their colours because:

$$d_{uu} = d_{vv} = 2,\ d_{uv} = 1,\ d_{vu} = 3$$

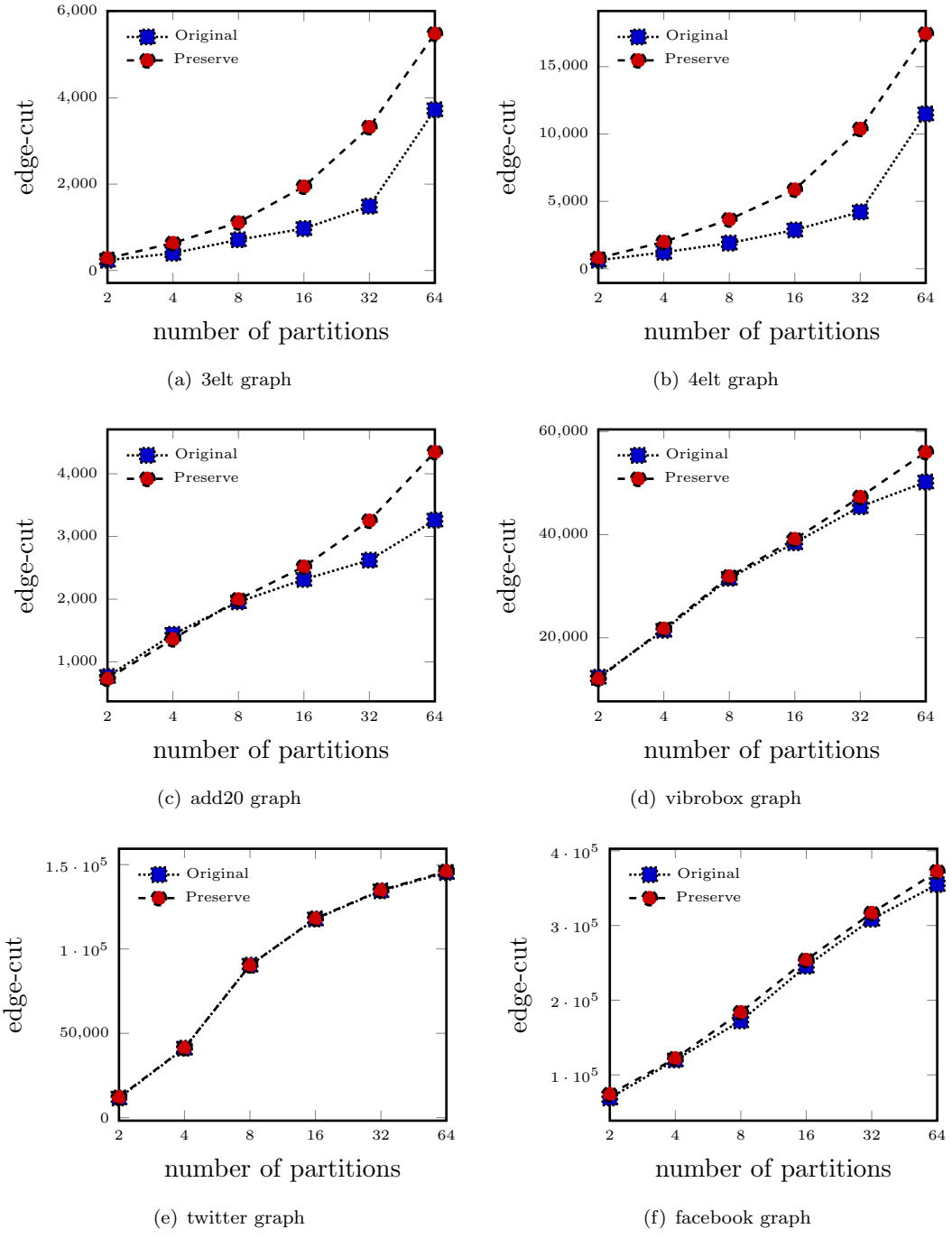
Graph	$\|V\|$	$\|E\|$
add20	2395	7462
3elt	4720	13722
4elt	15606	45878
vibrobox	12328	165250
Twitter	2731	164629
Facebook	63731	817090

TABLE 5.1: Graphs that are used for the tests

calculating the decisional equation 5.1: $1^2 + 3^2 > 2^2 + 2^2$ so $10 > 8$. As a result the swaps will take place and the white connected components is lost leading to a not valid partitioning result.
Hence, the α parameter has to be set to 1 in order to do not loose the connected components property.

All these objections restrict noticeably the algorithm's effectiveness. Given an initial valid partition configuration, a fix to JA-BE-JA could be to verify if swaps would preserve the connected components property and only in that case perform them. Let's call this variant "preserve-version". The problem of this solution is the overhead that is introduced in order to check the connected components property *at each potential swap*. The performances could noticeably fall down. In addition the solution space is less exploitable because such constraints will limit its exploration. Alternately the SA technique must not be used and also the decisional equation must have $\alpha = 1$. Whenever a communication occurs between 2 peers, either as neighbours or picked at random, these restrictions ensure that the connected components property of the graph will be preserved.

Since the JA-BE-JA's authors that have shared the original source code, we implemented the connected components check routine that has to be run at each possible swap over the original "one-to-one" JA-BE-JA source code. We recall the one host-one node model corresponds to the fully distributed system (i.e. P2P). The JA-BE-JA's source code runs over PeerSim [51], a P2P simulator. The PeerSim environment allows to check cheaply the connected components property through the shared-memory environment and the sequential execution of the simulator. We run tests in order to verify the actual effectiveness of the algorithm. We considered a large subset of graphs that also the original paper uses that are enumerated in Table 5.1. The datasets are taken from *The Walshaw Archive* [68], which is an archive of well-known graph where many balanced minimum k-way cut algorithms are running over them, and 2 social network graphs: a sample of the Twitter graph [21] and a sample of the Facebook graph [67].
The JA-BE-JA's parameters are the same as the original tests: temperature $T = 2$, $\delta = 0.003$ and $\alpha = 2$. The tests have been run for 10 times for each graph. The balanced minimum k-way cut has been calculated for $k = 2, 4, 8, 16, 32, 64$. Figure 5.3 shows the



(a) 3elt graph

(b) 4elt graph

(c) add20 graph

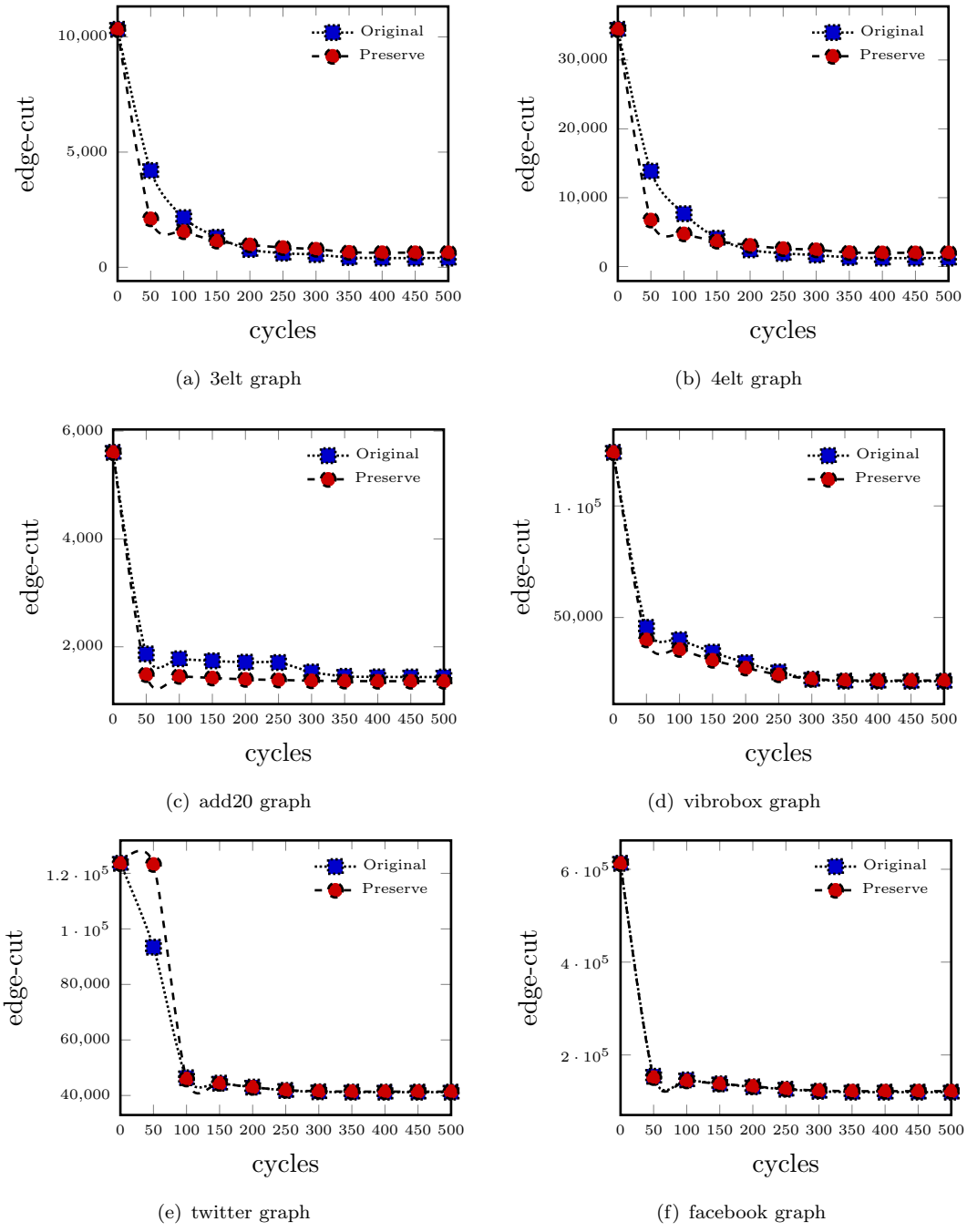
(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.3: Original JA-BE-JA Vs. JA-BE-JA Preserve on the Edge-cut average results relative to the partitions number

JA-BE-JA edge-cuts and the preserve-version ones relatives to the k partition. In Figure 5.4 are plotted the edge-cut average trends along the cycles for the partitions $k = 4$. The preserve version converges as fast as the original one but the edge-cut values achieved are slightly higher. As expected, the actual preservation of the connected components property influences the final edge-cut value and so the final coloured partition configuration that will be found.



(a) 3elt graph

(b) 4elt graph

(c) add20 graph

(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.4: Edge-cut mean trends of Original JA-BE-JA and the Preserve version along the cycles

5.2.2 A Pregel-compliant version

The JA-BE-JA algorithm has been designed for distributed system models and some versions of it have been investigated like the one host-one node (also called "one-to-one") model that corresponds to the P2P environment so each vertex of the graph is a peer; the one host-multiple nodes (also called "one-to-many") model that maps a subset of vertices of the graph to each host machine, takes advantage of the shared-memory that

it is available along the computation. The BSP model uses message-passing as the main mechanism to exchange the information between each machine and so in case of a graph-computation that it is equivalent to exchange messages between the vertices. Indeed, the BSP model is suitable as the one host-multiple nodes model of JA-BE-JA. However, as the original paper claims, the Pregel-like frameworks are actually the one host-one node model case. More precisely Spark, the framework that we are using, does not allow to distinguish between the operations that are performed on the same machine and the ones done through message passing, so it does not allow to take advantage of the shared memory for each subset of peer. The Spark's APIs define a set of operations that perform the computation over each partition (so over each vertex subset) separately. This is not sufficient because the developer does not have the access to the actual memory structure and so it is difficult to distinguish which data is on the same machine and the others that are instead some kind of place-holder/pointers, which are to be retrieved through message passing. This abstraction does not allow to implement easily the one host-multiple nodes model over Spark but despite that, we strongly believe that a custom implementation of a specialized RDD could solve the issue.

We implemented the ONJAG framework using the current API without developing a custom RDD leaving this to future works. As a consequence ONJAG (over Spark) does not take advantage of the shared memory. The "one-to-one" model has to be taken as reference with the difference of the communication timing pattern. In case of a P2P network the JA-BE-JA algorithm runs asynchronously over all the peers and no barriers are performed, instead using the BSP model the message exchange is performed synchronously by a barrier-style synchronization and so a well-defined repetitive pattern occurs. The implication of a repetitive and well-defined communication pattern is the emerging of a set of scenarios that have to being take into account in order to maintain correct and efficient the protocol.
The scenarios that mainly change the JA-BE-JA's behaviour are:

- each peer sends or receives a *swap request* message at each superstep. In order not to be paired with more than one peer at a time, it has to refuse the incoming request and, for the same logic, also the out-coming requests will be refused resulting in a stuck case where no swaps occur. The reasons is because all the peers are using the same logic. Somewhat differentiation between each peer's logic must be done. The usage of a random flag as guard to the sending of the out-coming swap requests has been placed, i.e. each peer flips a coin and accordingly send or not the requests;

- the synchronous communication pattern biases how the peers swaps their colour each other. More in detail, the swaps occur all together and so the exploration of the solution space is massively different compared to the completely asynchronous one. Let us make an analogy starting by simplifying the completely asynchronous message exchanges to a sequence of communication as a waterfall so to be possible

to be labelled and ordered. Then, the "one-to-one" model is like the *online* learning algorithm in the "machine learning" where the space is explored trying to optimise the current solution moving on each axis sequentially. Instead the BSP/Pregel-like model is like the *batch* learning algorithm and so the exploration of the space is done trying to move inside it considering all the axes together. More concretely in the BSP model the neighbourhood of a peer will give less spot to exchange the colour with them compared to the "one-to-one" model. The execution is discretized. This aspect has to be verified in order to ensure that the same optimal solution set is found.

We modified the original source code in order to simulate the BSP environment. More precisely the PeerSim "one-to-one" implementation has been modified in order to act artificially as a BSP computation. The shared-memory environment and the sequential simulator behaviour have not been changed. Also the preserve version is ported to the BSP simulation. Experiments have been executed.

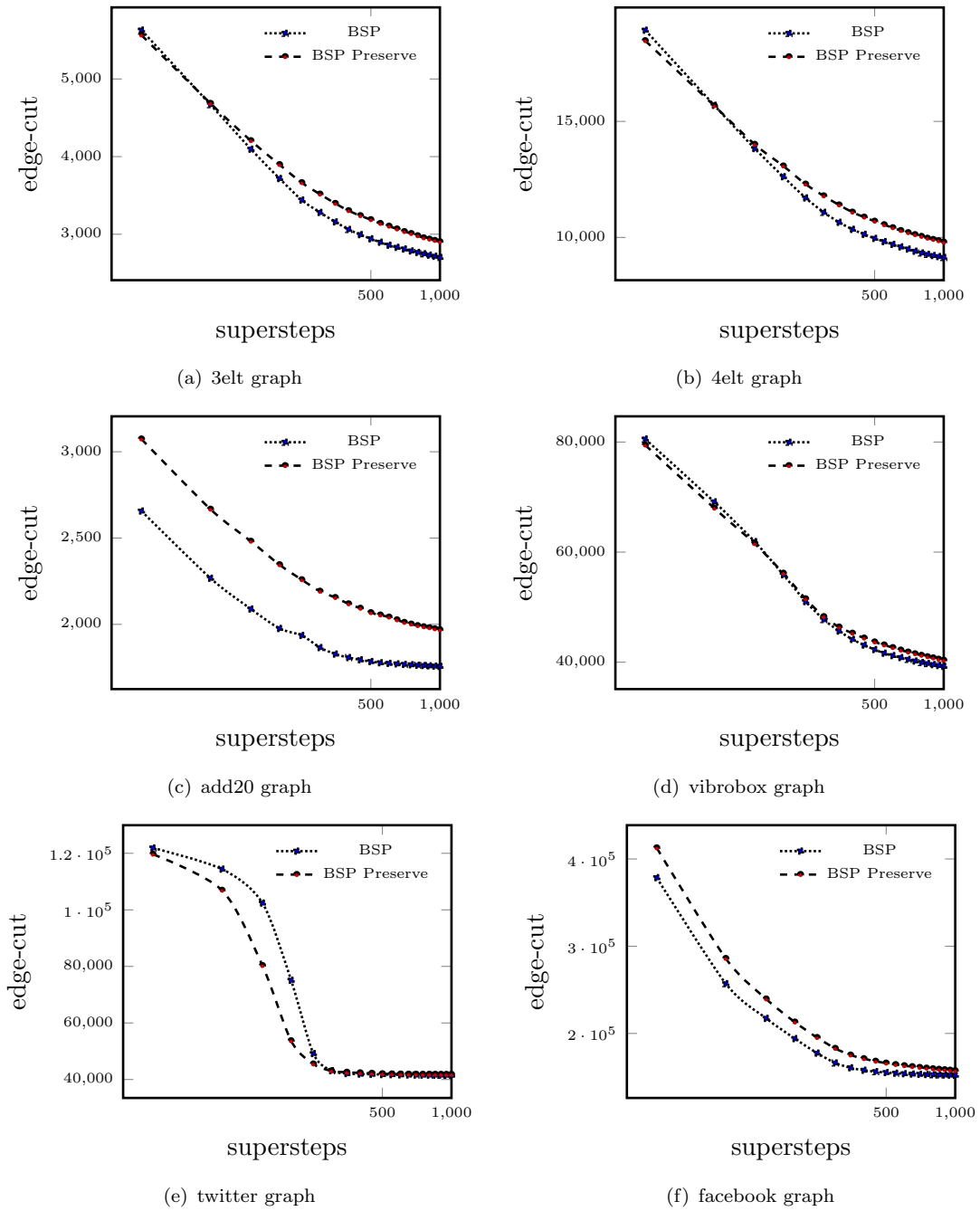
The JA-BE-JA's tests over the BSP simulation on PeerSim were re-run for 10 times. The algorithm over the partitions obtained for $k = 4$ was executed for 1000 supersteps and, as showed in Figure 5.5, it is noticed that the BSP computation converges likely as the original one. Thus, the tests for partitions $k = 2, 8, 16, 32, 64$ were also computed for 1000 supersteps. Figure 5.6 shows the final edge-cut values relative to the k partitions. The BSP final edge-cuts always have been considerably higher than the original ones except for the twitter graph which it seems to be really JA-BE-JA-friendly for all the executions. The scale up is in the average of the 30%. At the end, the BSP/Pregel-like environment is less exploitable for the algorithm but for bigger graphs it seems that the trends would be better. Intuitively, the spots of finding an exchange partner are more and swaps would likely occur leading towards better partitioning.

5.3 Implementation

We implemented an "as-is" implementation of JA-BE-JA derived by the paper [53] as an ONJAG protocol. For performance reasons, the calculus of the Min-Cut is done by a global accumulator variable natively provided by Spark instead of running a distributed algorithm as suggested by the JA-BE-JA's authors. The multi-start search is not provided explicitly because we preferred a "keep it simple" approach. Nevertheless it is still possible to do it through re-running the execution.

The experiments were executed over a cluster of machines named "barbera" at *National Research Council*[1]. The cluster is composed by 4 machines connected to each other by a Gigabit Ethernet. Each machine is equipped with 2 Opteron 6276, which has 16 3.2

[1]Consiglio Nazionale delle Ricerche (CNR), http://www.cnr.it/



(a) 3elt graph

(b) 4elt graph

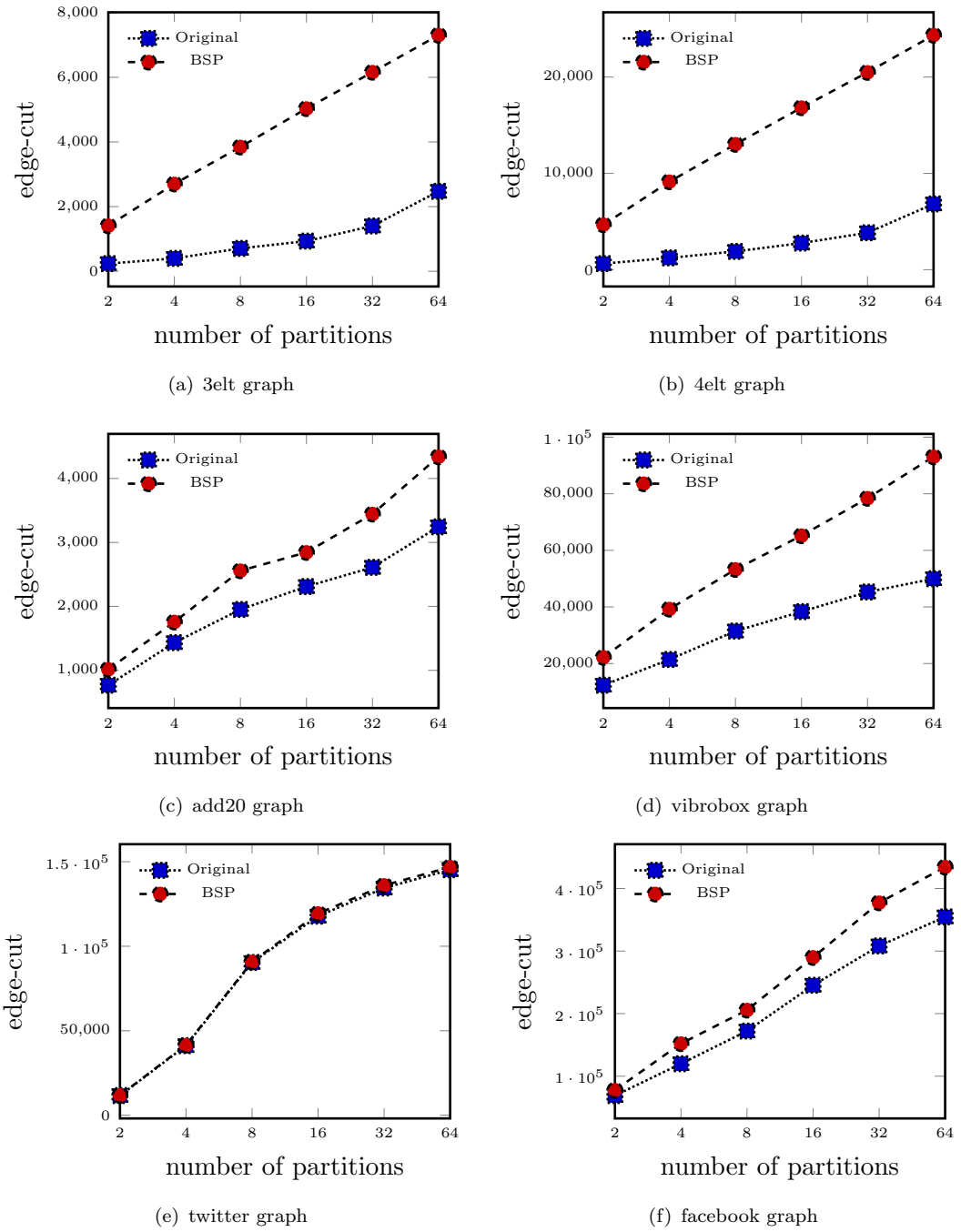
(c) add20 graph

(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.5: Edge-cut mean trends of BSP JA-BE-JA and its Preserve version along 1000 supersteps



(a) 3elt graph

(b) 4elt graph

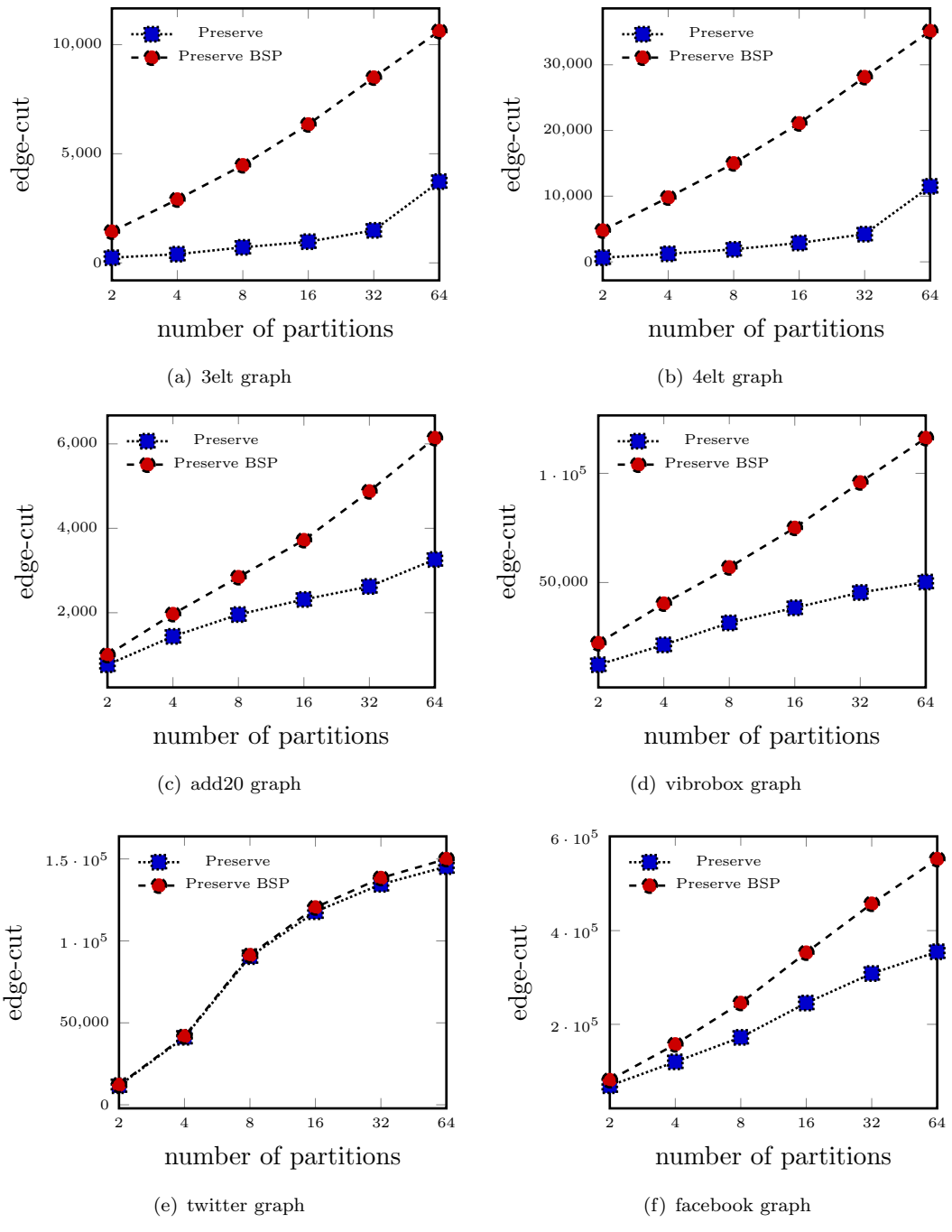
(c) add20 graph

(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.6: Original JA-BE-JA Vs. BSP JA-BE-JA about Edge-cut average results relative to the partitions number (1000 supersteps)



(a) 3elt graph

(b) 4elt graph

(c) add20 graph

(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.7: Preserve JA-BE-JA Vs. BSP Preserve JA-BE-JA about Edge-cut average results relative to the partitions number (1000 supersteps)

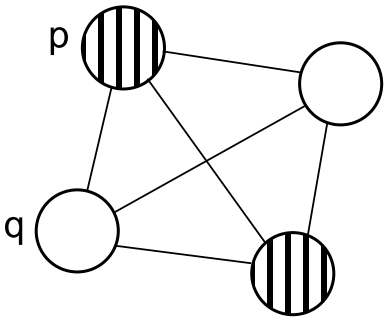


FIGURE 5.8: Example of a configuration that brings to an useless colours swapping between p and q.

GHz cores, for a total of 32 cores. Each Opteron has a NUMA architecture as memory model. Each machine has 128 Gigabyte of RAM and 3 Terabytes of storage.

As a result, the implementation revealed few issues and also some arrangements has taken place:

- the original paper defines for each node a function, called *getDegree*, calculating $d_p(\pi_x)$, which is required for the decisional equation to find a partner to exchange the colour with. In case a node p invokes the *getDegree* of a neighbour q to calculate $d_q(\pi_p)$ an exceptional case could happen in which the exchange takes place but actually should not. More precisely, the calculus of $d_q(\pi_p)$ does not put into account that the current neighbour p, which is involved in such calculus, should not be considered coloured with the current colour because in case of the swap would take place it would be false. Thus, an unwanted decision will occur. Actually the swap brings to an equivalent configuration and so the operation is still correct but it should not have been performed. Moreover it could bring the execution to loops, i.e. the occurrences of such cases could lead the computation to re-execute indefinitely those swaps. A configuration example is shown in Figure 5.8, recalling the equation 5.1 the values are: $d_p(\pi_p) = d_q(\pi_q) = d_p(\pi_q) = d_q(\pi_p) = 1$. For simplicity and clarity we shall put $\alpha = 1$. Considering the original *getDegree* function: $\bar{d}_p(\pi_q) = \bar{d}_q(\pi_p) = 2$. Therefore the equation holds and the swap takes place. Although after the swap have taken place, $d_p(\pi_p)$ and $d_q(\pi_q)$ are not equal to 2 but they are still equal to 1, i.e. the *getDegree* function misled the values.

- the original paper actually does not explicitly define how orchestrate the nodes communications, nevertheless the awareness must be taken into account. Indeed the partition sizes could change if they are not well managed. Moreover when a node tries to swap its colour with another one, it must not accept any other incoming *swap request* in order not to show up "ghost" node. Therefore, it could be paired into more than one conversation and despicable it could swap its colour with more than one other node, i.e. the partition sizes are not preserved. Thus, each node must be paired into a swap operation once a time.

The main and most important difference between the paper hypotheses and the Spark environment is the availability of the neighbourhood information and more in general of the other nodes. The original algorithm assumes to directly access the neighbour information at any time instead into the Spark environment, as a message-passing model, a sequence of messages as to be orchestrated in order to retrieve such data. As a result the performance and also the correctness had to be verified.

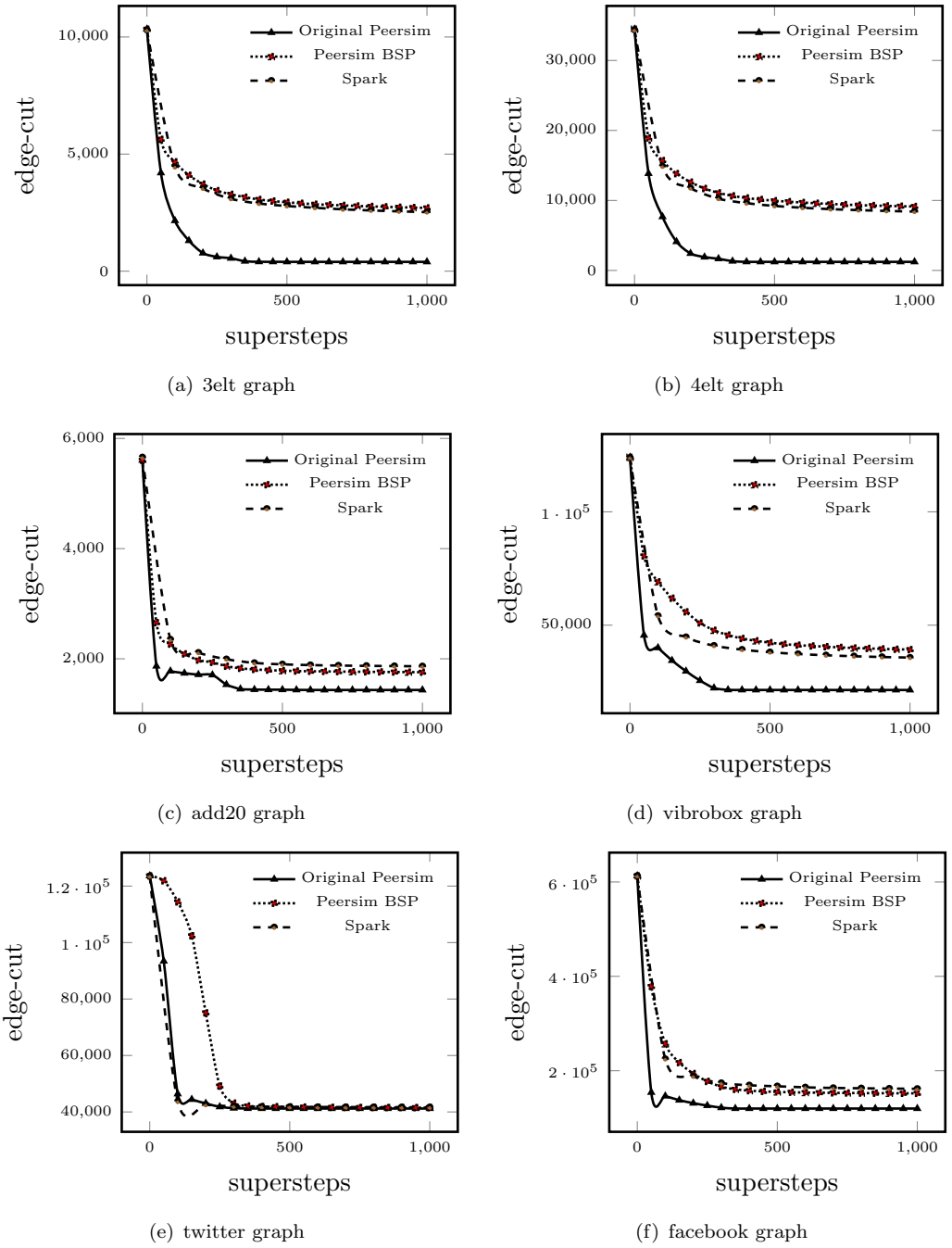
Nonetheless, the Min-Cut calculation differs from the paper one because the value is calculated through an *Accumulator* provided by Spark. Accumulators are global variables accessible by every machine of the cluster where it is possible perform on them an associative and commutative function. In our case the Min-Cut computation consists of $\sum_{\forall p \in V} |N_p| - d_p(\pi_p)$ because the graph is undirected and each edge weight is set to 1. So we sum the local min-cut value of each node p into an accumulator.
Beside that, in order to get correct Min-Cut value we must ensure that the configuration is stable during the computation and so there are not messages that could change the colour of some node in the meanwhile. JA-BE-JA continuously modifies the partition configuration and it does not provide any entry-points for such calculus therefore ad-hoc orchestration has to be performed. More precisely, these operations are:

1. suspend the JA-BE-JA protocol committing each node to its actual colour;
2. retrieve all the neighbourhood information;
3. finally calculate the local min-cut value and sum it into the accumulator.

As said above the actual unavailability of the peers in the BSP/Pregel-like environment needs to figure out workarounds and alternative solutions. The neighbour's context could be refreshed exchanging them between each peer at each superstep, it is expensive but ensures the coherence of each view. More precisely $2 \cdot |E|$ messages are required at each superstep. Although the random peer's contexts are not obliviously retrievable, two possibilities have been explored: blind *swap request* and caching. The blind strategy is quite simple: a peer, which has not found a neighbour peer to exchange its colour with, sends the request to a random peer that in turn evaluates the decisional equation 5.2 and in case of the equation is verified the swap takes place. On the other hand, the context's collecting could be done by caching a random view at each superstep and pipe-lining the actual retrieval and the subsequent usage. Caching enables a "wise" decision for the nodes but introduces a factor of incoherence because the cache could be invalidated and so misleading partner are chosen. The correctness is still maintained because the misled partner will refuse such swaps. As a consequence, both the sender and the receiver peers have to evaluate the decisional equation. If the cache is refreshed at each superstep no incoherence would show up. Anyway, both the neighbours and the random peers strategies introduce a significant overhead number of messages into the network at each superstep.

Both the strategies have been implemented and tested. The caching strategy showed up a very early stop of the algorithm resulting to very poor (i.e. high edge-cut) coloured partition configurations. Analysing the reasons of this behaviour showed that at least in the BSP environment each peer does not find very quickly any partner to swap its colour with. Thus a stuck network occurs and the JA-BE-JA algorithm stops. Indeed caching introduces a latency of at least one superstep and it is despicable because that time is crucial in order to avoid finding false-positive partners. The blind strategy does not expose this behaviour because a brute-force swap request yields to a less likely stuck network since the algorithm lasts for more supersteps. As a result, for our tests we choose the blind strategy. The results (in Figure 5.9) show that our Spark implementation performs grossly as the simplified PeerSim one with the synchronization barrier. In Figure 5.10 are reported the average edge-cuts as function of the partition sizes for 1000 supersteps.

Analysing the Spark executions, a significant low percentage of success swaps has been monitored, i.e., few peers agreed on swaps. Figure 5.11 shows the actual percentages. JA-BE-JA does a greedy local search optimization. Relaxing the search to a set of best nodes it would differentiate noticeably each vertex's logic leading to heterogeneous interactions over the network. Hopefully, this will lead to a higher percentage of successful swaps and perhaps better final edge-cut values, e.g. the second best candidate peer would be chosen for swapping. A pick probability function has been placed such a way that the partner is picked up onto the best candidate peers, i.e. peers are ordered according to the decisional equation and, according to a probability distribution, the partner is picked. The pick probability considers just the neighbours and not the random sample because of the blind strategy that does not practically retrieve a set of peers. Considering this fact and the reported data in Figure 5.12, the final edge-cut results that are achieved along the partitions are close to the ones without the pick probability procedure. In Figure 5.13 are shown the edge-cuts as function of the partitions with and without the usage of the pick probability procedure. If on one hand this behaviour could be exploited, on the other hand the overhead of introducing more messages in the network at each superstep does not actually lead to a advantageous trade-off. From now on, all the experiments will not consider the pick probability procedure.



(a) 3elt graph

(b) 4elt graph

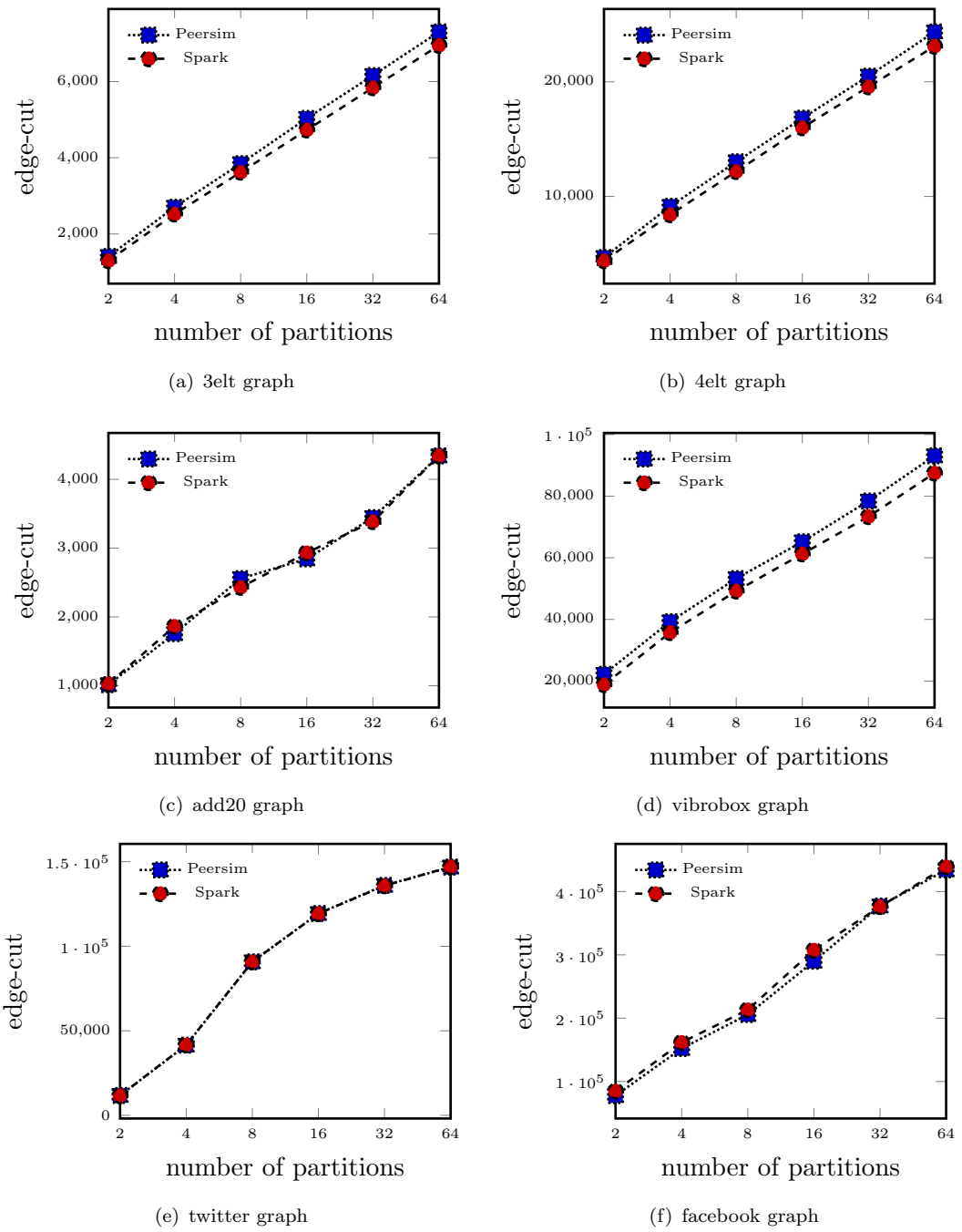
(c) add20 graph

(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.9: Edge-cut mean trends of Original Peersim JA-BE-JA, Peersim BSP and Spark JA-BE-JA along 1000 supersteps



(a) 3elt graph

(b) 4elt graph

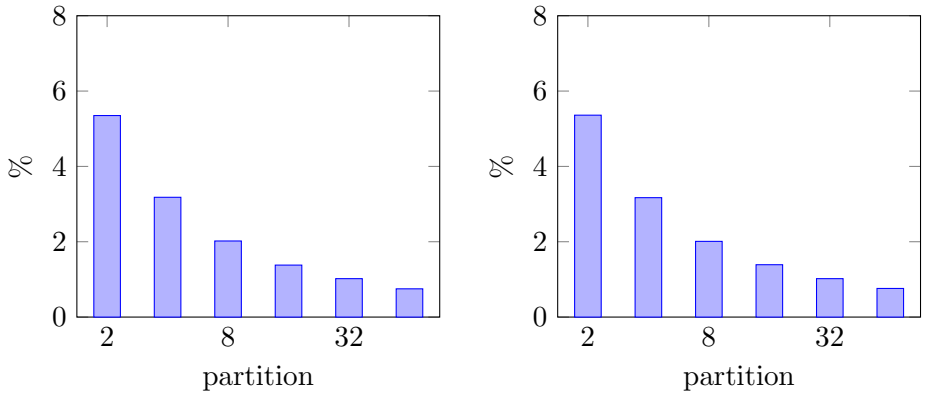
(c) add20 graph

(d) vibrobox graph

(e) twitter graph

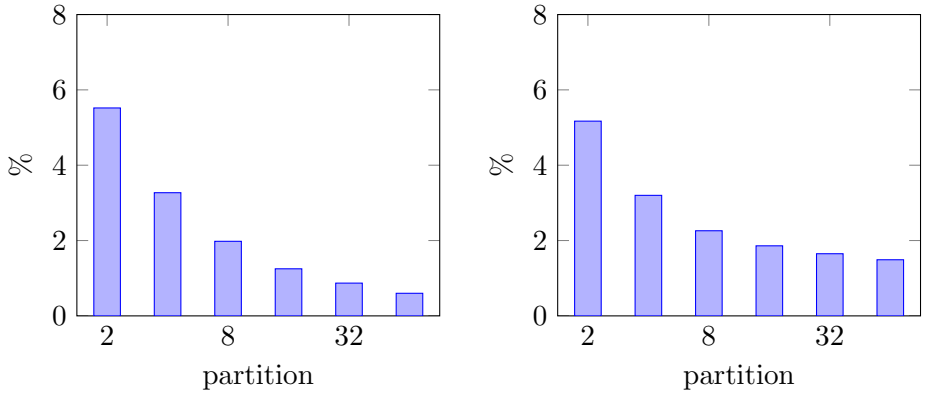
(f) facebook graph

FIGURE 5.10: Peersim BSP JA-BE-JA Vs. Spark JA-BE-JA about Edge-cut average results relative to the partitions number (1000 supersteps)



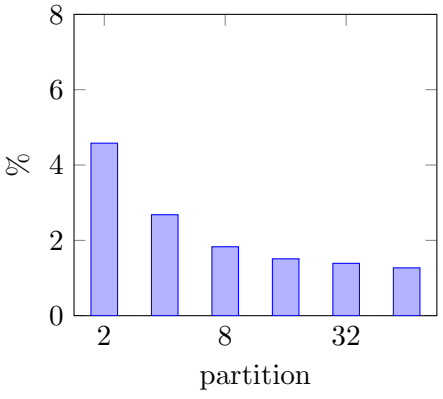
(a) 3elt graph

(b) 4elt graph

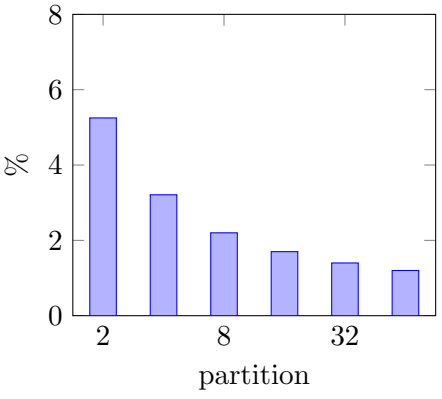


(c) add20 graph

(d) vibrobox graph

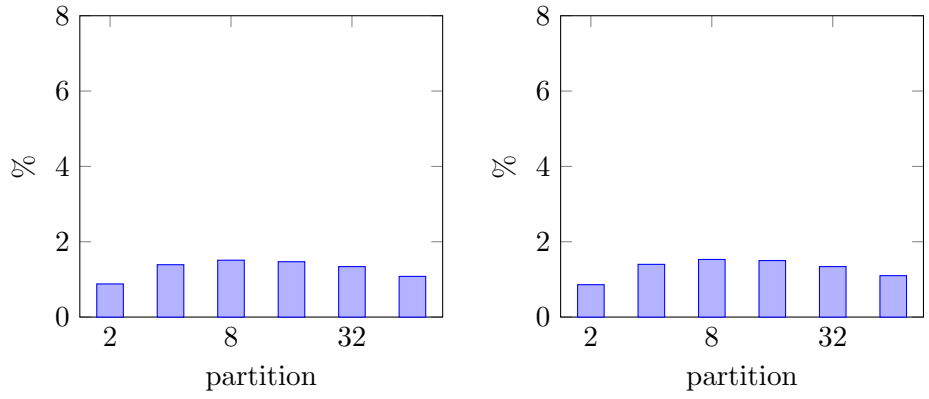


(e) twitter graph



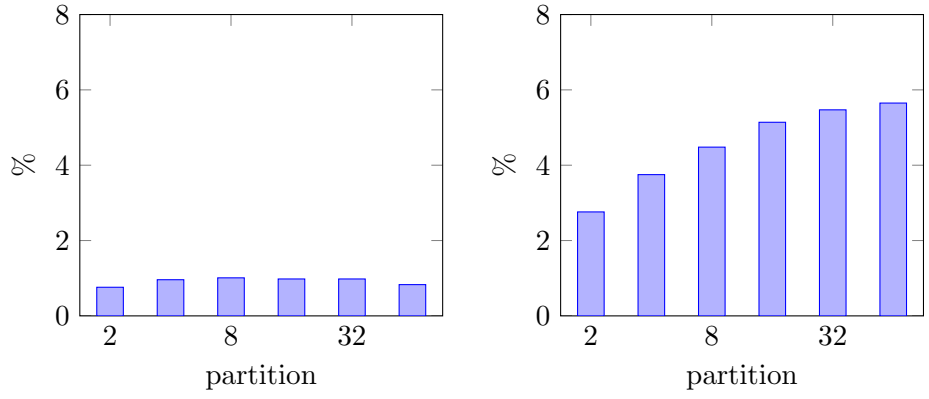
(f) facebook graph

FIGURE 5.11: % of *swap request*s that actually succeed for the Spark execution along the partitions number



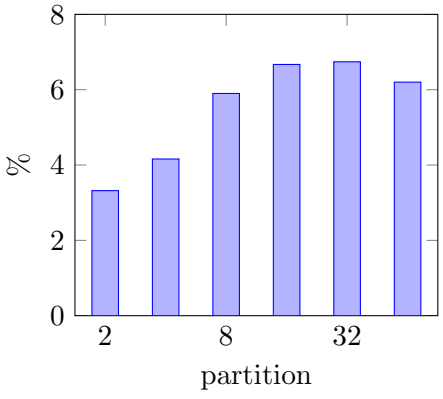
(a) 3elt graph

(b) 4elt graph

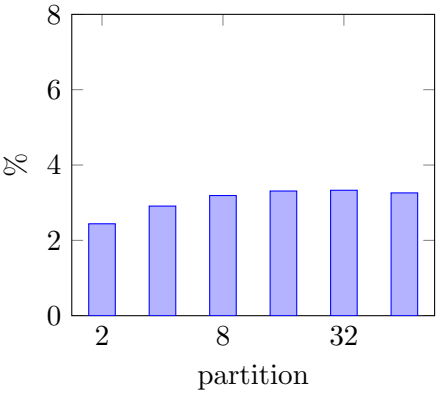


(c) add20 graph

(d) vibrobox graph

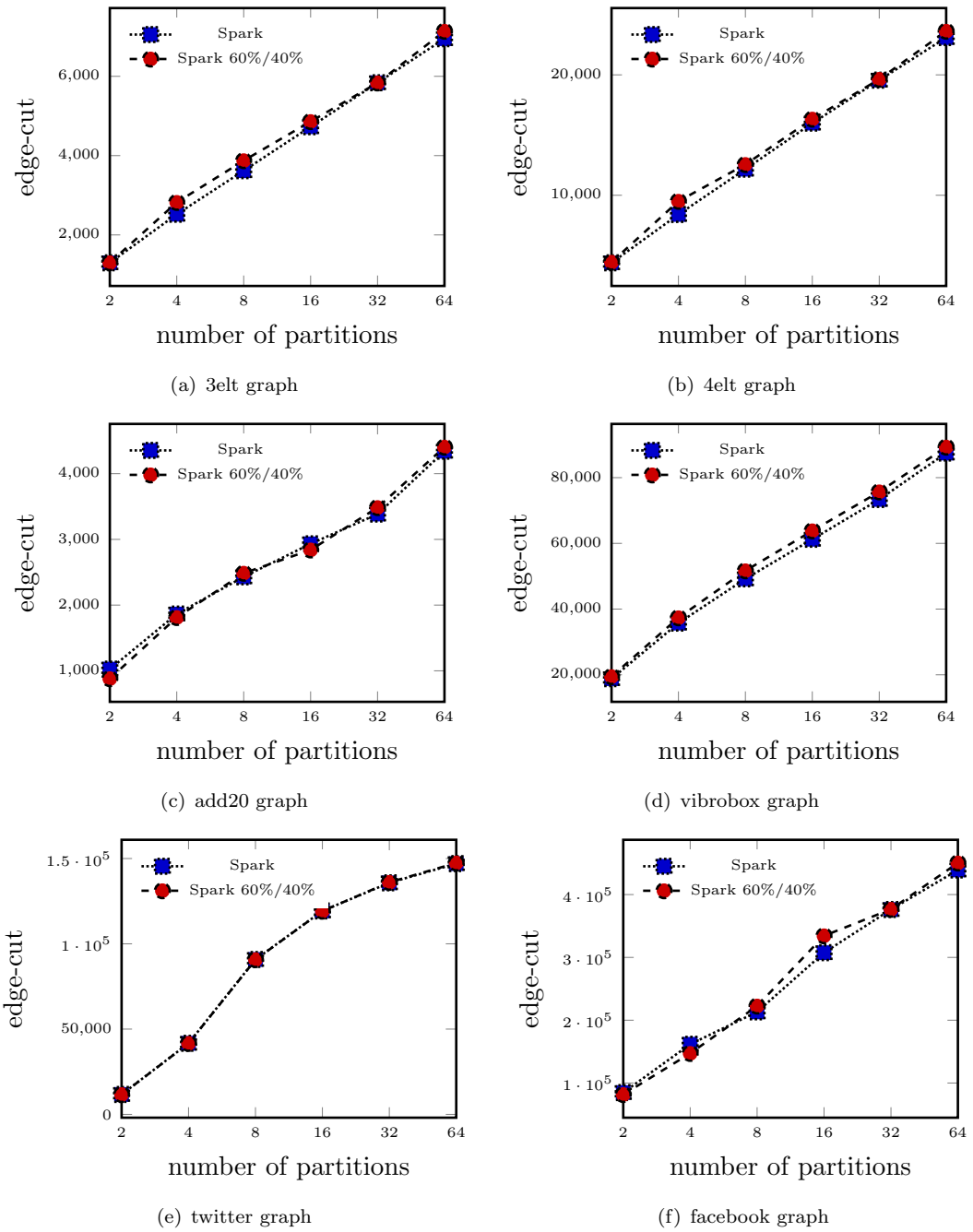


(e) twitter graph



(f) facebook graph

FIGURE 5.12: % of *swap requests* sent to neighbours for the Spark execution along the partitions number



(a) 3elt graph

(b) 4elt graph

(c) add20 graph

(d) vibrobox graph

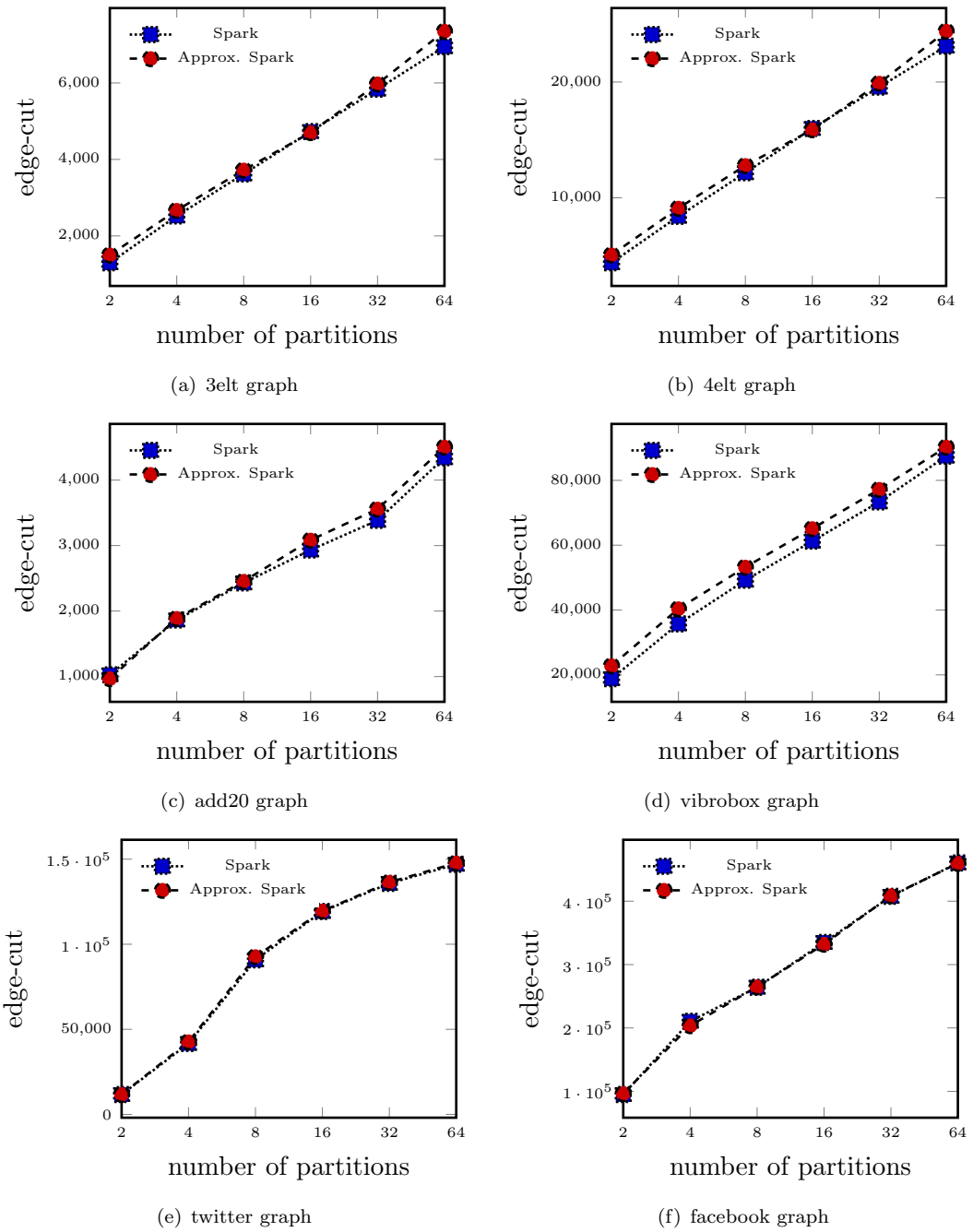
(e) twitter graph

(f) facebook graph

FIGURE 5.13: Spark JA-BE-JA Vs. Spark JA-BE-JA with pick probability of 60% – 40% about Edge-cut average results relative to the partitions number

5.3.1 An approximate implementation

The main difference between the BSP/Pregel-like computational model such as Spark and the JA-BE-JA's assumptions is the direct access to the neighbourhood and a small random sample of the graph. In addition, the decisional equation 5.2 requires also the knowledge of the neighbourhood of each neighbour. Therefore, we developed a message sequence procedure in order to retrieve such contexts at each superstep for all the peers. The trade-off of this workaround is between the correctness of the JA-BE-JA algorithm and the performances in terms of messages and number of supersteps. More precisely, $2 \cdot |E|$ messages are required in order to implement such procedure at each superstep. After figuring out which performances and final edge-cuts are achieved by strictly adhering to a JA-BE-JA-compliant implementation over a Pregel-like support, we relaxed the assumption allowing incoherent and approximate contexts along the computation. In other words, the contexts are not anymore refreshed at each superstep, therefore the decisional equation could be evaluated over stale data. The contexts are retrieved opportunistically and lazily piggybacking into the already existing orchestration message performed by JA-BE-JA. In other words, $SWAP\ REQUEST/ACK/NACK$ are exploited for context updating. In spite of this modification the obtained results are very close to the strict-complaint version. In Figure 5.14 are showed the final edge-cut values for each partition. Indeed the same behaviour, as shown in Figure 5.15, is reproduced in case the "pick probability" procedure is used. This modification greatly improves the performances. In Figure 5.16 is possible to see the speed-up compared to the strict compliant JA-BE-JA version and the approximated one. Hence, the modification also requires less memory allocation. As a consequence, from a Big-Data point of view, a larger set of viable datasets becomes computable in a reasonable time. The speed up does not seem *completely* related to the partition size. However, the best speed up is achieved for the partition $k = 2$. Intuitively, the probability to mislead the colour is 50% (as flipping a coin) and so the global process acts mostly like as the strict-complaint one. In conclusion the barrier synchronization is the main difference between the "one-to-one" model and the BSP/Pregel-like environment. The non-shared-memory environment and the approximate JA-BE-JA version alternatives have not affected significantly the final results.



(a) 3elt graph

(b) 4elt graph

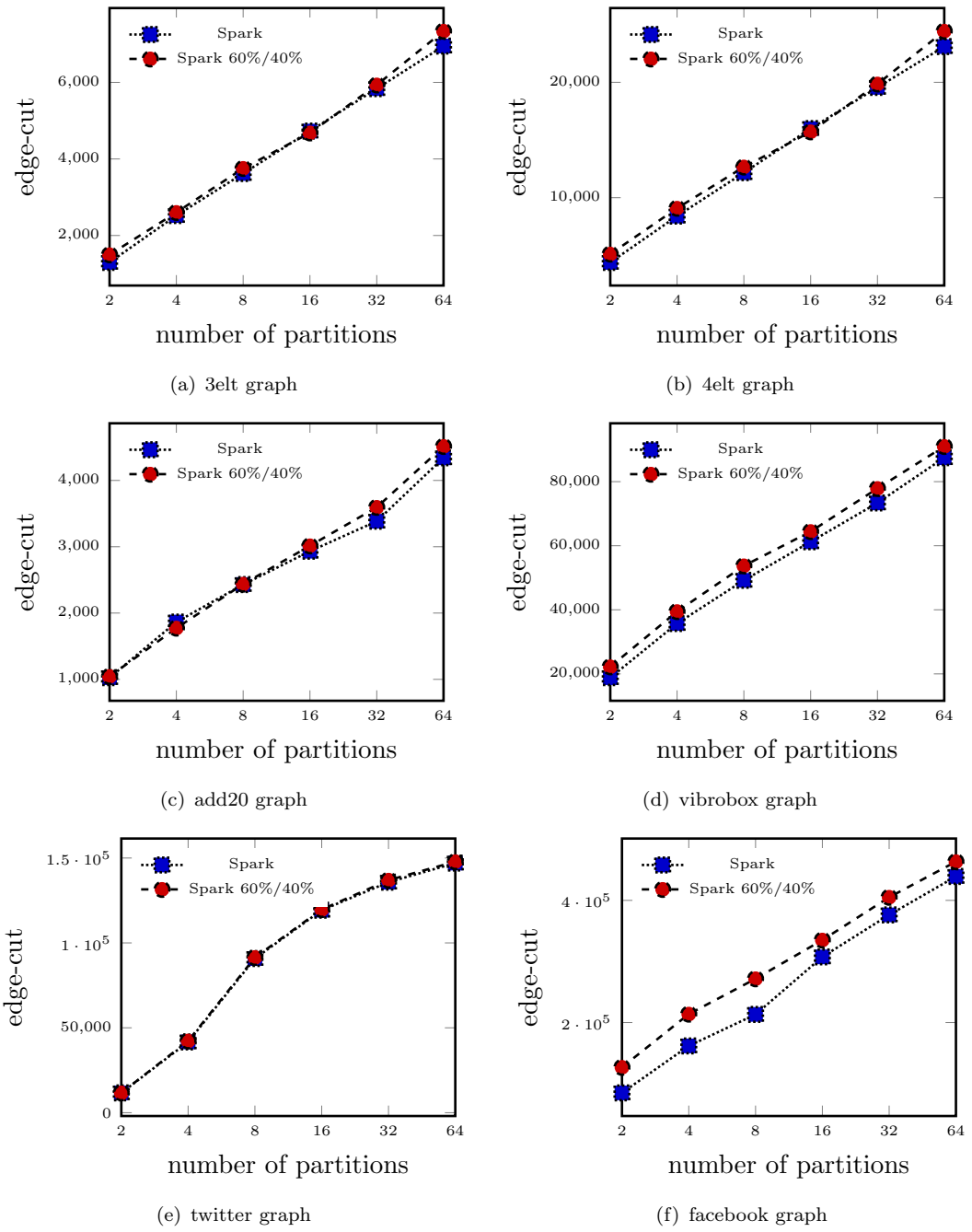
(c) add20 graph

(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.14: Spark JA-BE-JA Vs. Approximate Spark JA-BE-JA about Edge-cut average results relative to the partitions number



(a) 3elt graph

(b) 4elt graph

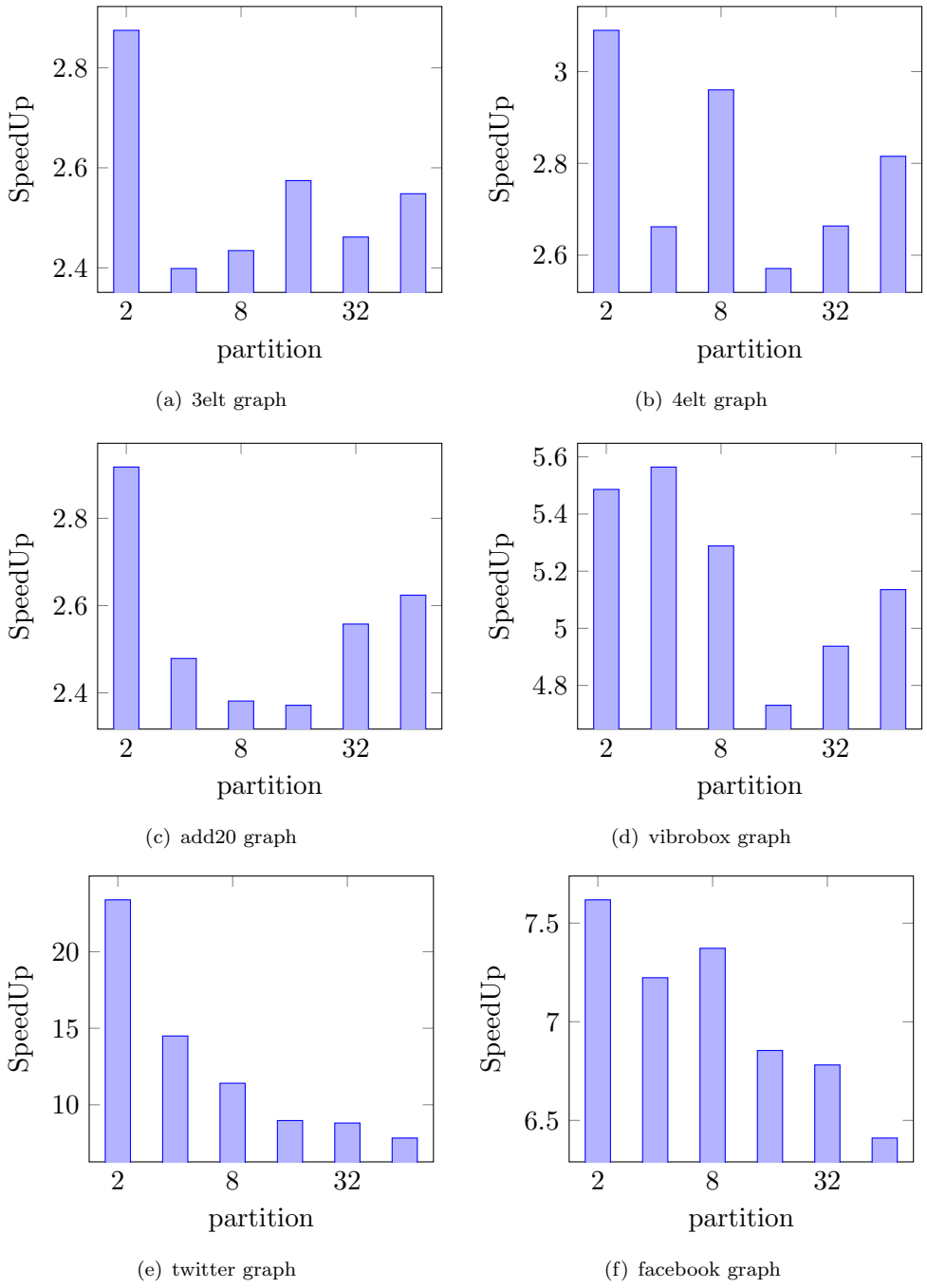
(c) add20 graph

(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.15: Spark JA-BE-JA Vs. Approximate Spark JA-BE-JA with pick probability of $60\% - 40\%$ about Edge-cut average results relative to the partitions number



(a) 3elt graph

(b) 4elt graph

(c) add20 graph

(d) vibrobox graph

(e) twitter graph

(f) facebook graph

FIGURE 5.16: SpeedUp of Approximate Spark compared to Original Spark along the partitions number

6

JA-BE-JA meets Overlays

> *I'm trying to free your mind, Neo. But I can only show you the door. You're the one that has to walk through it.*
>
> – Morpheus, *The Matrix*

In this chapter we describe a proof-of-concept application of the "overlays" approach introduced in Chapter 5. We recall that the idea consists of introducing a pile of layers over the graph, whose layers are able to communicate each other in order to orchestrate complex behaviours. Our main contribution is the creation of a stack of P2P-like layers. Some of those layers (called Protocols in the ONJAG nomenclature) define an own overlay. The approach is the exploitation of such overlays massively relying on the intra-communications between layers. In this chapter an actual application is presented that, to a certain extent, endorses the ONJAG conception and creation.

As described in the Chapter 5, JA-BE-JA encountered some difficulties during the porting on a Pregel-like framework, contrasting the original authors believing. Thus, we deeply analysed such algorithm and as a result we have been able to contribute somewhat to its improvement. In addition, the min-cut problem is well-known in graph theory and indeed it is challenging and actually employed in numerous areas. In conclusion, our effort have been placed over JA-BE-JA. Nevertheless the approach could be extended to other algorithms or new ones could be created ad-hoc.

We focus on Gossip, a computation model, which is friendly to P2P networks. Note that we do not exhaustively verified our proposal in a completely distributed environment and on the distributed graph processing one but our contribution is exclusively on the latter case. Our results are verified into the ONJAG environment.

The chapter continues as follow. Firstly, an alternative and deep analysis of the JA-BE-JA algorithm is given. After that, our contribution is presented contextualising JA-BE-JA into the P2P environment. Then, a quasi min-cut definition is given in

order to properly evaluate the JA-BE-JA algorithm. Finally, the ONJAG framework is taken into account, concluding with findings over experiments and practical remarks.

6.1 An alternative JA-BE-JA review

In Chapter 5 is presented JA-BE-JA, a distribute balanced minimum k-way graph partitioning algorithm. It is extensively studied and tested in order to bring it to a Pregel-like platform. We recall that the algorithm mainly relies over a coloured metaphor where each node could be coloured and partition are defined as the nodes with the same colour. JA-BE-JA orchestrates the execution swapping the colour between pair of nodes according to the equation (5.2), which includes also the employment of the Simulated Annealing technique. The algorithm just employs local search optimization to achieve global improvements. Each performed swap, excluding when the simulated annealing temperature is greater than 1, moves the graph towards a lower *energy*, state better edge-cuts.

Recalling equation (5.1), which does not employ the SA technique, and putting the $\alpha = 1$:

$$d_p(\pi_q) + d_q(\pi_p) > d_p(\pi_p) + d_q(\pi_q) \tag{6.1}$$

The meaning of such equation is to verify if the possible swap would bring the local edge-cut towards a lower value. More precisely such aspect is evaluated for the p node either for the q one, indeed the equation is verified only if both respective edge-cuts are decreased. We define

$$H_p = \frac{d_p(\pi_p)}{|N_p|} \tag{6.2}$$

as the percentage of the neighbours with the same colour of the node p. Contextualizing the equation (6.1) regarding the H value it is clear that the equation is verified only when $H_p < 50\% \wedge H_q < 50\%$, i.e. the neighbourhood colours of both nodes must be mostly different to the p and q ones respectively. Moreover in case all the nodes present a H value greater than 50% no swaps are performed anymore and the algorithm stops. As a consequence this kind of logic tries to exploit the nodes that have a H value < 0.5 in order to achieve better min-cut configurations and, it does not consider nodes with $H \geq 0.5$. Let's name this behaviour "loneliness exploitation".

Regarding the α value, it is introduced in order to bring the configuration to equivalent states but more promising in the future. Recalling equation (5.1) and contextualising it relatives to the H value it is interesting to state that $\alpha > 1$ allows swaps even if the equation does not hold per se and, more interestingly to figure out that such policy sometimes advantages nodes with greater H values, i.e. in case $|N_p| \approx |N_q|$ the swap occurs in favour of the greater H value holder increasing the unbalance between the nodes. The effectiveness of such behaviour is directly proportional to the α value. This

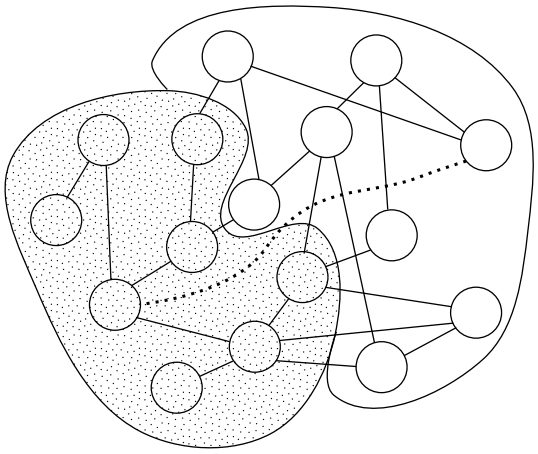


FIGURE 6.1: Stain Metaphor of the JA-BE-JA's topology

aspect apart from increasing the H values of some nodes, decreases the H value of others thus, the "loneliness exploitation" is endorsed allowing exploitation of new configuration.

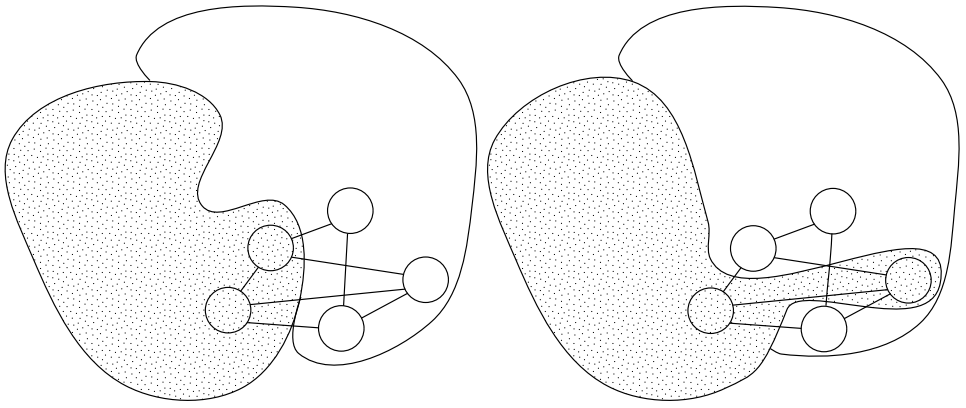
6.2 The TMAN add-on

The "overlays" approach exploits orchestration between several protocols relying over the intra-communication facility. Our proposal adds a new overlay that interacts with the JA-BE-JA one. First of all, the JA-BE-JA's overlay is the actual graph topology plus some long range links that are created through the retrievement of a small sample of random nodes for each vertex.

As shown in Figure 6.1, such overlay could be thought as a set of stains which interact each other "moving" like fluids. In this metaphor, the long range links fold the stains in a N dimensional space, where N is greater than the actual dimension, allowing the interaction between areas of the stains that are not physically near each other. The JA-BE-JA algorithm orchestrates those stains in order to minimize the edge-cut. Whenever a swap succeeds, analogously in the stain metaphor, is like the stains flow rolling in some directions, as shown in Figure 6.2.

6.2.1 Our Proposal

T-MAN, described in section 4.3, is a gossip algorithm whose aim is to create and manage topology overlays. The overlays are created and maintained according to a *ranking function* which measures the similarities between two peers. Our proposal is to introduce an ad-hoc overlay and exploit over it better choices. More precisely, JA-BE-JA interacts with such overlay in a sense of the stain metaphor. The interaction's type is competitive, which means JA-BE-JA takes advantage of such overlay but as side-effects it damages such topology, i.e. the 2 overlays are somehow related and a modification



(a) Topology before the swap (b) Topology after the swap

FIGURE 6.2: Swap in the stain metaphor of the JA-BE-JA's topology

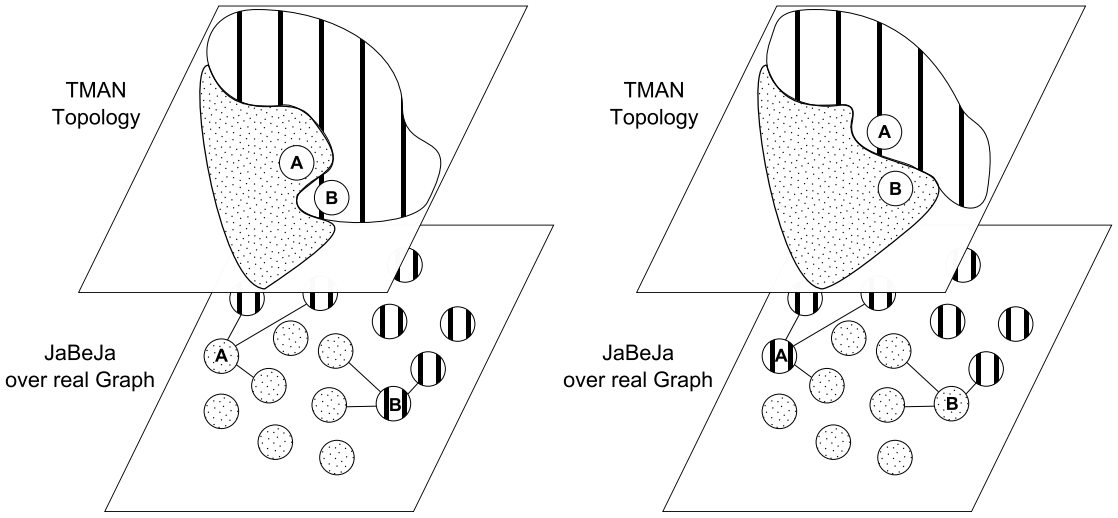
on one of them implies the transformation of the other. Thus, the T-MAN's ability to maintain the topology against dynamic changes is crucial. Indeed, the balance between the JA-BE-JA ruination and T-MAN repairing has to be tuned.

The T-MAN algorithm relies over the definition of a *ranking function*. Hence, the ad-hoc topology has to be defined through that function. The idea is to abstract the property of "borderness" of each peer from the actual topology, i.e. the idea is to point out the peer's property to be on a border of some stains. Whenever 2 peers are on the borders the swap could take place and so figuratively the peers cross the boundaries. Moreover, the boundaries where the swaps take place do not have to be the same where the swap equation is evaluated on. At the end, we exploit the T-MAN overlay for the equation evaluation and perform the swap over the actual JA-BE-JA one. More precisely, the equation evaluation is executed firstly over the T-MAN overlay and in case no partner is found the algorithm falls back to the original evaluation, i.e. the neighbourhood and the random sample are taken into account. The side-effect of such orchestration is the damages that the T-MAN overlays is subjected to, i.e. the T-MAN overlay is defined starting from the JA-BE-JA one.
The *ranking function* using the JA-BE-JA notation is defined as:

$$rankFunction(peerA, peerB) = \begin{cases} +\infty, & \text{if peerA.colour == peerB.colour} \\ |H_A - H_B|, & \text{otherwise} \end{cases}$$

$$\text{where } H_P = \frac{d_p(\pi_p)}{|N_p|}$$

We recall that two peers A and B are more similar as the $rankFunction(A, B) \to 0$. The function ranks similar two peers that have the same percentages of neighbours of the same colour. Moreover they have to be coloured differently to each other, i.e. the peers are ranked among their "borderness" property and their relative colours. As a consequence, peers that are in the middle of a partition (a stain in the metaphor



(a) Before the colour exchange

(b) After the colour exchange

FIGURE 6.3: Example of an exchange colour using the TMAN layer over the real JA-BE-JA coloured graph.

nomenclature) are similar to the ones that are in the middle of another partition (100% case, $H_P = 1$) and, more useful for our purposes, peers which are on "the borders" of a partition result similar to others that are also on the same kind of border, i.e. peers that are at the same distance from the border are similar. As a result, in a figurative way as shown in Figure 6.3, the stains flow between each other like a liquid with the aim of finding a good solution mapping the actual topology to another more friendly for the purpose. Moreover, such abstraction allows to find hopefully better candidates to exchange the colour with filtering and organizing the peers among their "borderness" property.
The decisional equation is evaluated over the T-MAN overlay and in case the swap will succeed the JA-BE-JA underline topology is updated. Therefore, the T-MAN overlays will be out-of-date at least in such peers and it has to rearrange their neighbourhood in order to adhere to their new "borderness" value. Optimistically the "borderness" value is updated incrementally allowing smoothly changes to the T-MAN overlay, otherwise the protocol strongly relies over the random sample to reconstruct its topology.

The JA-BE-JA algorithm performs local search optimization and the achieved solution could be stuck in a poor local minimum edge-cut value. In order to avoid those local minima the Simulated Annealing technique is employed. A temperature T is employed in order to manage the *energy* of the system, which could be increased at the beginning until a conservative policy is adopted falling back to the original algorithm. More precisely, JA-BE-JA introduces the temperature into the decisional equation, as shown in equation (5.2). The result of contextualizing such parameter into the stain metaphor is shown in Figure 6.4. The temperature defines an area over the border where those peers are able to interact each other, i.e. the temperature defines a stripe and peers that are into it acts like to be on the same border. The cooling process, i.e. the temperature

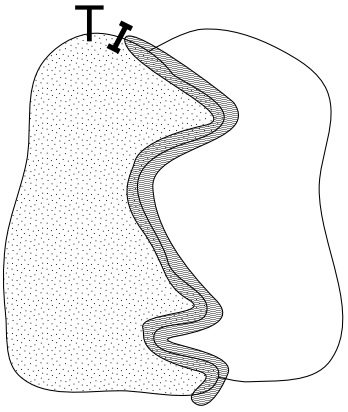


FIGURE 6.4: Contextualization of the Simulated Annealing into the Stain Metaphor

decreasing, corresponds to tightening the stripe. A temperature value equal to 1 corresponds to vanish the stripe. Intuitively at the beginning many peers are able to interact each other whenever if they are actually on the border or not. Along the execution the stripe will be tighten and tighten reducing the number of peers which are able to interact, i.e. the simulated annealing creates (biased) chaos converging eventually to the usual behaviour. This aspect advantages the T-MAN overlay because softer requirements are requested along the computation and a less strict-timing repair capability is needed allowing the usage of more cycles to the T-MAN protocol.

6.3 A quasi min-cut measure

The optimal bi-partitioning of a graph $G = (V, E)$ is the one that minimizes the *cut* value, which is defined as:

$$cut(A, B) = \sum_{v \in A,\ u \in B} w(u, v)$$

where A, B are the 2 partitions and w is a function that returns 1 in case an edge between u and v exists, 0 otherwise. However, the *cut* measure has been shown not to be always sufficient in order to achieve a good balance between the partitions. Thus, alternative definitions that take into account also other characteristics have been proposed. A noticeable one is proposed by Shi-Malik [60], the *normalized cut* (*Ncut*). The *Ncut* derives from the original *cut* taking into account the internal connectivity of each partition and the connectivity with the rest of the graph. The *Ncut* is also called the disassociation measure. It is defined as:

$$Ncut(A, B) = \frac{cut(A, B)}{assocc(A, V)} + \frac{cut(A, B)}{assocc(B, V)} \tag{6.3}$$

where $assoc(X, V) = \sum_{u \in X, t \in V} w(u, t)$. It counts the edges that link the nodes in X to the entire graph.

Similarly the association measure is defined as:

$$Nassoc(A,B) = \frac{assoc(A,A)}{assoc(A,V)} + \frac{assoc(B,B)}{assoc(B,V)} \tag{6.4}$$

where $assocc(A,A)$ are the number of edges within the same partition. This measure defines how tightly the nodes are connected.
Interestingly, the disassociation and association measures are related:

$$\begin{aligned} Ncut(A,B) &= \frac{cut(A,B)}{assocc(A,V)} + \frac{cut(A,B)}{assocc(B,V)} \\ &= \frac{assoc(A,V) - assoc(A,A)}{assoc(A,V)} + \frac{assoc(B,V) - assoc(B,B)}{assoc(B,V)} \\ &= 2 - (\frac{assoc(A,A)}{assoc(A,V)} + \frac{assoc(B,B)}{assoc(B,V)}) \\ &= 2 - Nassoc(A,B) \end{aligned}$$

Extension and derivation

We derived the Shi-Malik *normalized cut* extending it straightforward to k-partitioning the graph and allowing to evaluate a no proper min-cut, which breaks the connected components property, i.e., we allow fragmentation of each partition and the measure has to evaluate also that aspect.

The natural *Ncut* extension to k-partitioning is defined as:

$$Ncut(X_1, \cdots, X_k) = \sum_{i=1}^{k} \frac{cut(X_i)}{assocc(X_i,V)} \tag{6.5}$$

where $cut(X_i) = \sum_{P \in (V \setminus X_i)} \sum_{u \in X_i, v \in P} w(u,v)$.
Similarly the *Nassoc* measure is extended as:

$$Nassoc(X_1, \cdots, X_k) = \sum_{i=1}^{k} \frac{assocc(X_i,X_i)}{assocc(X_i,V)} \tag{6.6}$$

Finally, the relationship between the association and disassociation measures is:

$$Ncut = k - Nassoc(X_1, \cdots, X_k) \tag{6.7}$$

The fragmentation of each partition into small sub-partitions requires to evaluate the following details, namely, the number of fragments, their absolute and relative sizes and how they are connected absolutely and relatives to each other in the graph. Moreover a proper evaluation could be application dependent. More precisely the measure has many trade-offs, and it is not possible to define a priori the best, e.g. is it preferable to deal

with several fragments relatively of the same sizes or a bunch of that to be significantly bigger than the others?
Therefore, our derivation does not give a fixed biased evaluation function but defines a generic function that will be defined contextualising it according to the application domain. Nevertheless, we define our actual function used for the evaluation of the JA-BE-JA algorithm.

The fragments of a X partition are notated as:

$$\overline{x} \in P(X) | P(X) = \{fragment_{X_1}, \cdots, fragment_{X_m}\}$$

The *assocc* measure is modified as:

$$assocc(X, X) = \sum_{\overline{x} \in P(X)} assocc_{frag}(\overline{x}, \overline{x})$$
$$assocc_{frag}(\overline{x}, \overline{x}) = \sum_{u \in \overline{x}, t \in \overline{x}} w(u, t)$$

Indeed $assocc(X, V) = \sum_{\overline{x} \in P(X)} assocc_{frag}(\overline{x}, V)$ in which similarly to the previous definition $assocc_{frag}(\overline{x}, V) = \sum_{u \in \overline{x}, t \in V} w(u, t)$. For simplicity, we use $assocc$ any time is clear from the context overloading the above functions.
We derived the *normalized cut* from the *Nassocc* association measure. The disassociation measure, i.e. the *Ncut*, is derived subsequently. The *Nassoc* measure over k partition is defined as:

$$Nassoc(X_1, \cdots, X_k) = \sum_{i=1}^{k} f(X_i) \cdot \sum_{\overline{x} \in P(X_i)} \frac{assoc(\overline{x}, \overline{x})}{assocc(\overline{x}, V)} \tag{6.8}$$

where

$$f(X_i) : partition \rightarrow \{v \in \mathbb{R} | v \in [0, 1]\} \tag{6.9}$$

is the evaluation function that measures the quality of a partition as the fragments which is composed by. Higher is the value returned by $f(X_i)$ better is the quality of the fragments. The function is not defined at priori and it is likely domain dependent.
Our evaluation function evaluates how many fragments are present, the variance between the fragment sizes and the shortest path to another fragment. The external fixed values are a, b, c and α. The external variables weight each aspect according to the user interest, e.g. $\alpha \geq 1$ penalizes unbalanced fragments instead $\alpha < 1$ advantages such cases. We defined our evaluation function as:

$$f(X_i) = a \cdot ln(|P(X_i)| - 1 + e)^{-1} + \tag{6.10}$$
$$b \cdot ln(VAR(X_i)^{\alpha} + e)^{-1} + \tag{6.11}$$
$$c \cdot ln(Ndistances(X_i) + e)^{-1} \tag{6.12}$$
$$a + b + c = 1$$

where the

$$\mu(X_i) = \frac{\sum_{\overline{x} \in P(X_i)} |\overline{x}|}{|P(X_i)|}$$

and

$$VAR(X_i) = \sigma(X_i) = \sqrt{\frac{\sum_{\overline{x} \in P(X_i)} (|\overline{x}| - \mu(X_i))^2}{|P(X_i)|}}$$

$$Ndistances(X_i) : partition \rightarrow \{v \in \mathbb{R} | v \in [0, +\infty)\}$$

Finally the *normalized quasi cut* (*NQcut*) is derived as:

$$\begin{aligned} NQcut(X_1, \cdots, X_k) &= k - Nassocc(X_1, \cdots, X_k) \\ &= k - (\sum_{i=1}^{k} f(X_i) \cdot \sum_{\overline{x} \in P(X_i)} \frac{assoc(\overline{x}, \overline{x})}{assocc(\overline{x}, V)}) \\ &= k - (\sum_{i=1}^{k} f(X_i) \cdot \frac{assocc(X_i, X_i)}{assoc(X_i, V)}) \\ =& \frac{assocc(X_1, V) - (f(X_1) \cdot assocc(X_1, X_1))}{assocc(X_1, V)} \\ &+ \cdots + \\ &\frac{assocc(X_k, V) - (f(X_k) \cdot assocc(X_k, X_k))}{assocc(X_k, V)} \end{aligned}$$

The *NQcut* acts as the *Ncut* in case no fragmentation occurs. As a normalized measure the *Ncut* either the *NQcut* are in the interval $(0, k]$. The extreme case $NQcut = k$ occurs if the connectivity within each partition is poor even if the fragments are evaluated as poorly suitable.
Unfortunately, we could not be able to implement the $Ndistances$ function due to the time constraints. Nevertheless we presented our evaluation function including such detail despite in the next sections it is not performed. We set the external variable as: $a = 0.8$, $b = 0.2$, $c = 0.0$ and $\alpha = 1.0$.

6.4 Experiments

Tests have been executed in order to verify the actual feasibility of the T-MAN add-on in the distributed graph processing computations. The tests were run over ONJAG on the Spark platform so the completely distributed environment is not taken into account, i.e. no PeerSim simulations have run. Nonetheless, in Chapter 5 we compared the BSP/Pregel-like platforms and the P2P environment evaluating them over the JA-BE-JA executions. Thus, the tests and the findings are already collocated opportunely

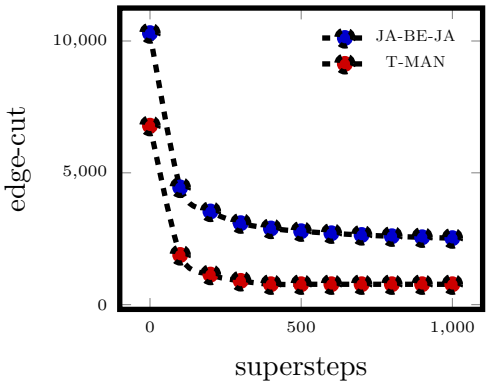
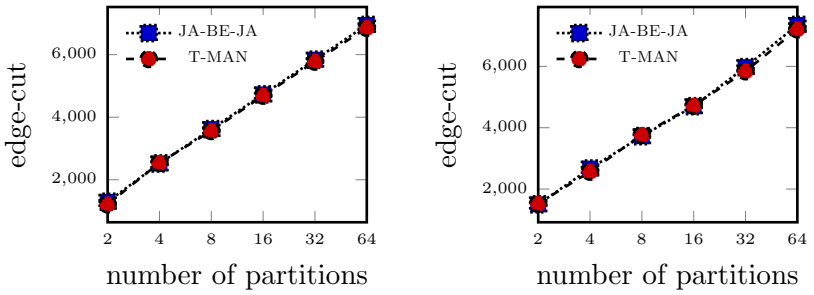


FIGURE 6.5: Edge-cut mean trends of Spark JA-BE-JA and the T-MAN add-on along 1000 supersteps for the *3elt* graph with $k = 2$

contextualizing the results relatively to the BSP/Pregel-like Spark framework. An exhaustive approach has not been feasible because of the time constraints. However, in order to validate properly also the add-on in a P2P environment, we are aware that more tests are needed perhaps with the PeerSim simulator.
We run the tests over the "barbera" cluster, which is described in 5.3. The graphs that are taken into account are *3elt*, *vibrobox* and *Facebook*. The dataset details are showed in Table 5.1.

First of all, we repeated the tests of Spark Original JA-BE-JA just adding the T-MAN add-on, i.e., the data coherence is maintained and no attempts of improvements have been taken into account, e.g. caching. The convergence resulted to be approximately the same. In Figure 6.5 is showed the plot for the graph *3elt* with $k = 2$. As showed in Figure 6.7 the edge-cut values achieved by the T-MAN add-on are, for the smallest graph, lower than the original JA-BE-JA algorithm, i.e. better results are found. However for the *Facebook* graph worst results are found. Looking to the *Normalized Quasi Mincut* (*NQcut*), showed in Figure 6.8, the same trends are found. The quality of the final configuration is improved by a 5/10% except for the *Facebook* graph where opposite findings are found. Intuitively, the smaller graphs required less effort and so better results are achieved. It is noticeable to observe the results over k, the partition number, the best improvements are mainly achieved by lower k values instead for partition $k = 16, 32, 64$ the edge-cuts are not comparable as the former ones. A higher k value means more stains in the stains metaphor so more complex overlays are built. Moreover the T-MAN's *ranking function* discerns essentially between 2 colours and intrinsically it is optimised for $k = 2$. Clearly, the function has to be generalized and improved in order to take into account more complex situations. The *Facebook* graph does not show such trend because if on one hand 2 partitions optimise the *ranking function* behaviour, on the other hand the self-healing system of T-MAN strongly relies over the random samples which are required in case the neighbourhood is not sufficient and for 2 partitions the latter case it is more likely to occur (the graph size contributes to the phenomenon amplification). The *NQcut* values achieved with the T-MAN add-on version follows



(a) not approximate (b) approximate

FIGURE 6.6: Spark JA-BE-JA No Cache Vs. Spark JA-BE-JA with T-MAN add-on and cache is employed about Edge-cut average results relative to the partitions number on the *3elt* graph

the same trend of the edge-cuts but there is less margin between them. This is not an oblivious result because the *NQcut* measures the *quality* of the partitions, not only their edge-cuts. We can state that the T-MAN add-on does not fragments the graph more than the original JA-BE-JA algorithm and maybe it could perform better increasing the fragments quality.
After that, we relax the assumption about the data coherence and the approximate versions have been tested. Figure 6.9 shows the edge-cuts average trends along the partitions. The trends are comparable to the not approximate ones re-enforcing that JA-BE-JA performs a good local optimization also in presence of stale data. However, as shown in Figure 6.10, the *NQcut* values are sensible higher compared to the not approximate ones.

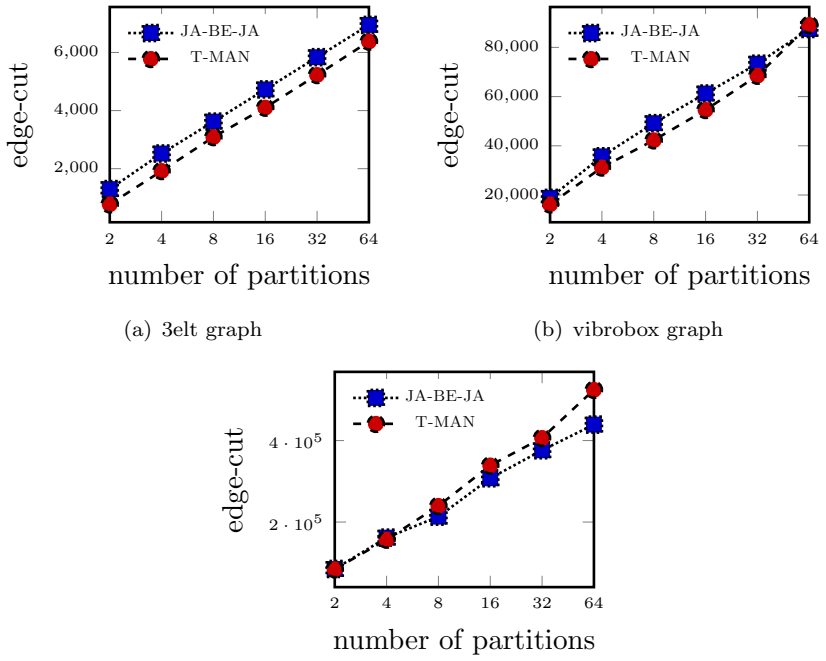
We also re-introduced the caching technique employed in 5.3 but for the T-MAN nodes. The T-MAN overlay provides to each JA-BE-JA peer a set of nodes. They could be cached or, employing the blind policy, one at random is chosen as partner. As for the JA-BE-JA caching also for the T-MAN layer such policy is somehow too slow due to the high variability of the network. More precisely compared to the previous tests, the equation evaluation is not performed only on the T-MAN overlay but it is also evaluated falling back to the original order, i.e. neighbourhood and random sample. Thus, the T-MAN view often does not provide a swap partner and the computation results as a usual JA-BE-JA execution. Figure 6.6 shows the edge-cut values for the *3elt* graph both in case of the data coherence model even the approximate version one. The final results are grossly as the JA-BE-JA ones. In conclusion, the caching policy does not give any improvements, and consequently not employed in the following tests.

In addition, we tested also the Simulated Annealing technique utility. We increased the initial temperature T and maintained the same δ. The temperature has been set to 3. Figure 6.11 and Figure 6.12 show the edge-cut average trends and *NQcut* values, respectively. The achieved results are better compared to the ones with $T = 2$ confirming the border stain metaphor showed in Figure 6.4. Therefore, we argue that, increasing

the temperature T until the stripe covers half size of the stains, it would achieve the best results. The *NQcut* values confirm that the *ranking function* is linearly dependent to the k partitions. Also the approximate version has been run and the results are showed in Figure 6.13 and Figure 6.14. In this case the stripe is built over an approximate stain metaphor and a bit worst results are achieved because mislead data occurs. The *NQcut* values are comparable to the approximate version with $T = 2$.

In order to verify the T-MAN's effectiveness we executed the T-MAN protocol at double frequency relatives to the JA-BE-JA one. In this way the built overlay likely adheres to the *ranking function* metric. In Figure 6.15 and Figure 6.16 are showed the results. In spite of building a more well-defined topology, the results are not better compared to the previous tests. Looking to the *NQcut* values is noticeable to stand out that the better values are still for lower ks. Thus, we argue that the *ranking function* is the main reason of this poor results re-enforcing the needs of further investigations regarding the similarity functions.
Another way to verify the T-MAN's effectiveness is to increase the random samples. The resulting topology is likely more coherent to the *ranking function*. Figure 6.17 and Figure 6.18 show the edge-cuts and the *NQcut* values, respectively. As expected the same trends are found, like the case of double frequency described previously. The results are actually a bit better and it is reasonable because the topology is construct by a completely random sample instead of traversing the graph. At the end, the T-MAN ability to construct the desired overlay is confirmed and efforts on better definition of the *ranking function* are required.

In conclusion, the tests successfully show the T-MAN add-on as a valid proof-of-concept of the "overlay approach" often achieving better results compared to the plain JA-BE-JA algorithm. Although, the *ranking function* clearly shows not to be perfectly suitable in general, therefore further investigations are required.

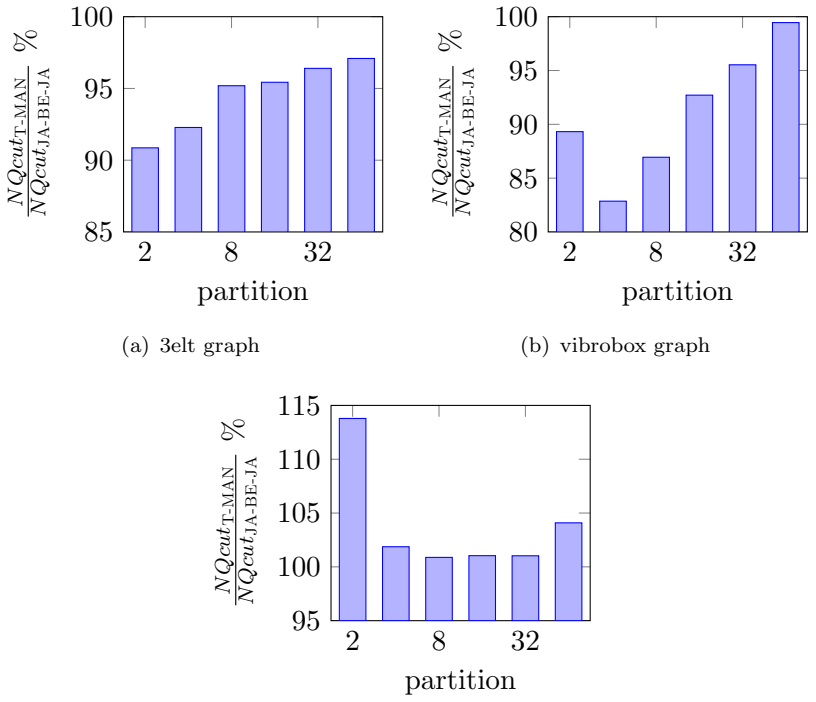


(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.7: Spark JA-BE-JA Vs. Spark JA-BE-JA with T-MAN add-on about Edge-cut average results relative to the partitions number

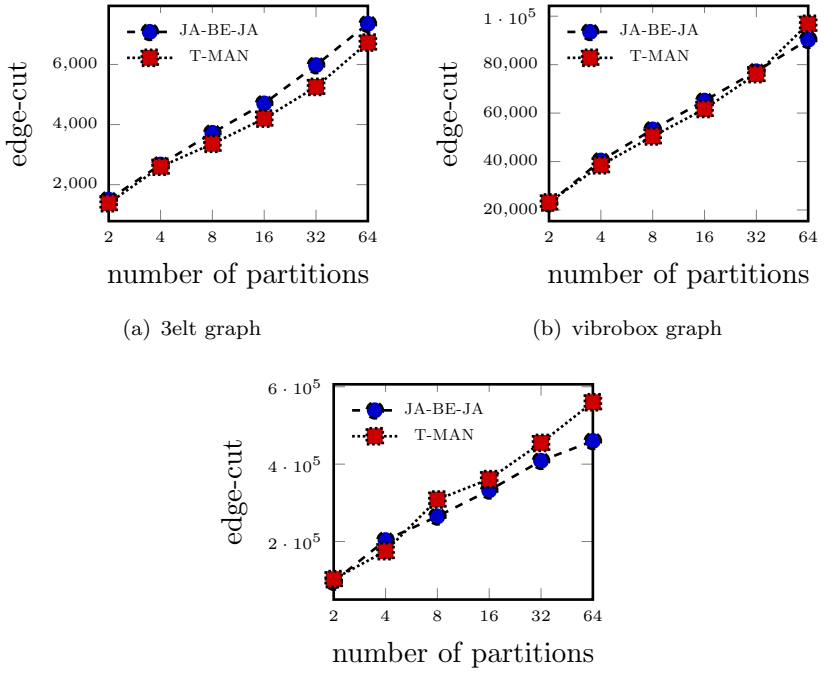


(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.8: Spark JA-BE-JA with T-MAN add-on relatives to Spark JA-BE-JA about NQcut values

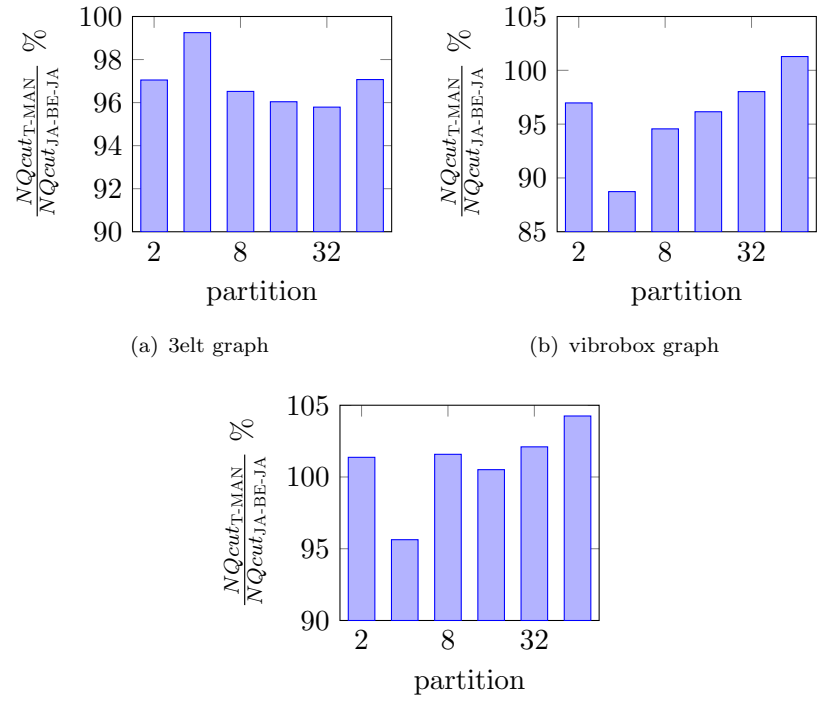


(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.9: Approximate Spark JA-BE-JA Vs. Approximate Spark JA-BE-JA with T-MAN add-on about Edge-cut average results relative to the partitions number

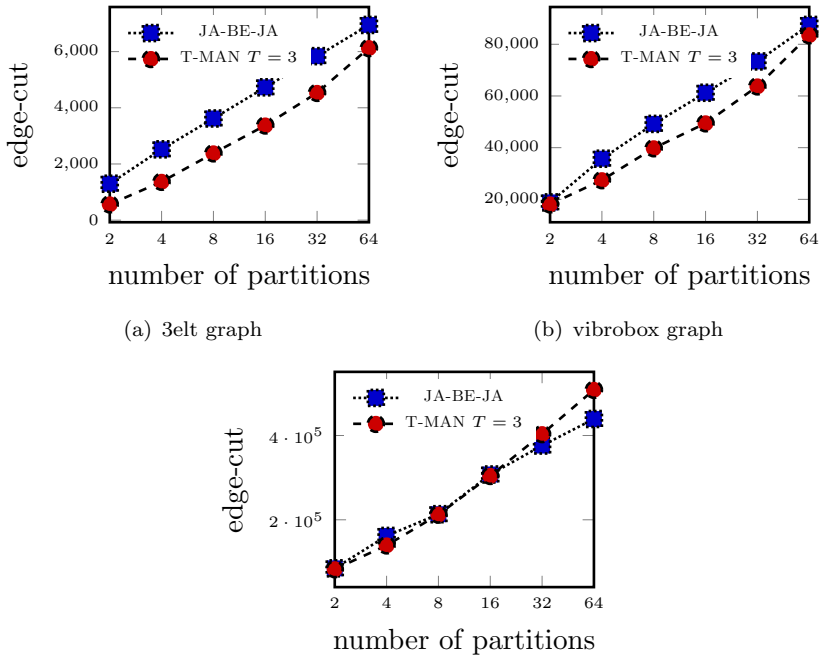


(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.10: Approximate Spark JA-BE-JA with T-MAN add-on relatives to Approximate Spark JA-BE-JA about NQcut values

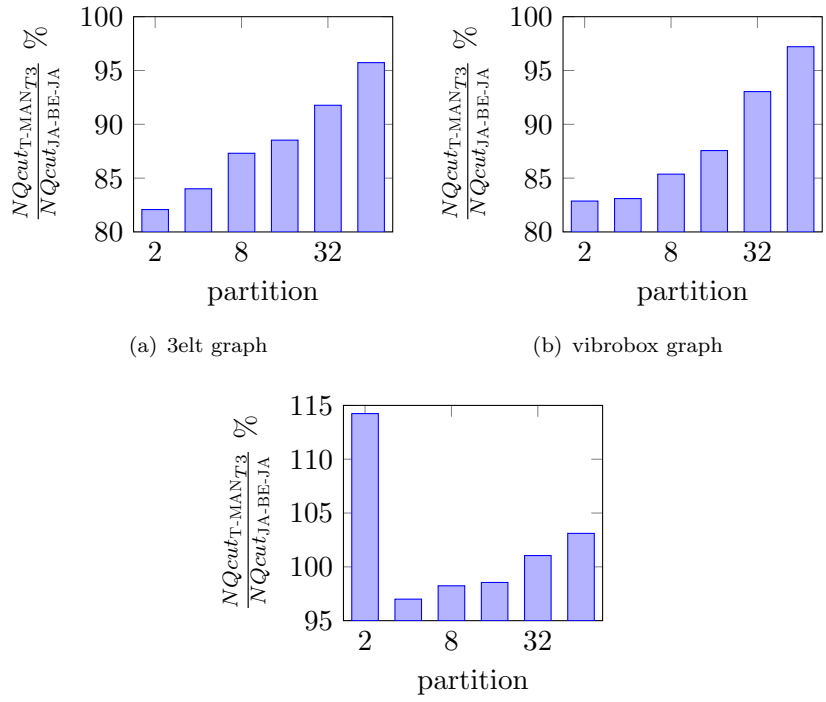


(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.11: Spark JA-BE-JA Vs. Spark JA-BE-JA with T-MAN add-on and Temperature = 3 about Edge-cut average results relative to the partitions number

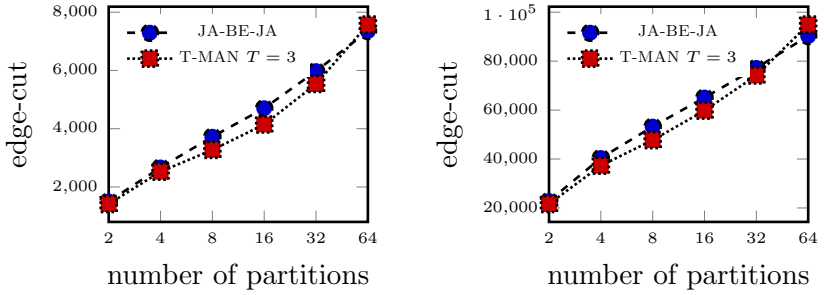


(a) 3elt graph

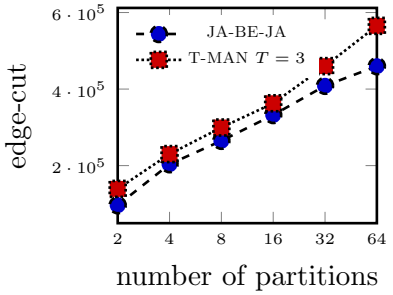
(b) vibrobox graph

(c) facebook graph

FIGURE 6.12: Spark JA-BE-JA with T-MAN add-on and Temperature = 3 relatives to Spark JA-BE-JA about NQcut values

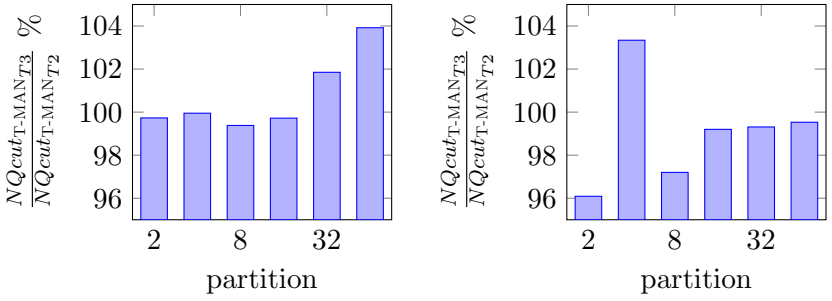


(a) 3elt graph (b) vibrobox graph

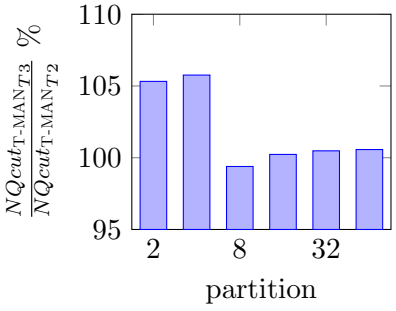


(c) facebook graph

FIGURE 6.13: Approximate Spark JA-BE-JA Vs. Approximate Spark JA-BE-JA with T-MAN add-on and Temperature = 3 about Edge-cut average results relative to the partitions number

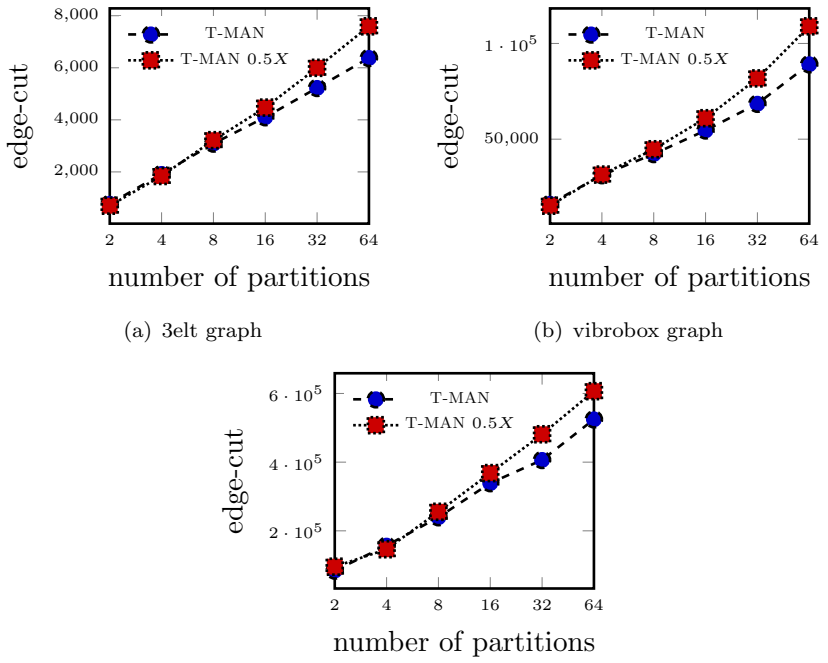


(a) 3elt graph (b) vibrobox graph



(c) facebook graph

FIGURE 6.14: Approximate Spark JA-BE-JA with T-MAN add-on relatives to Temperature = 2, 3 about NQcut values

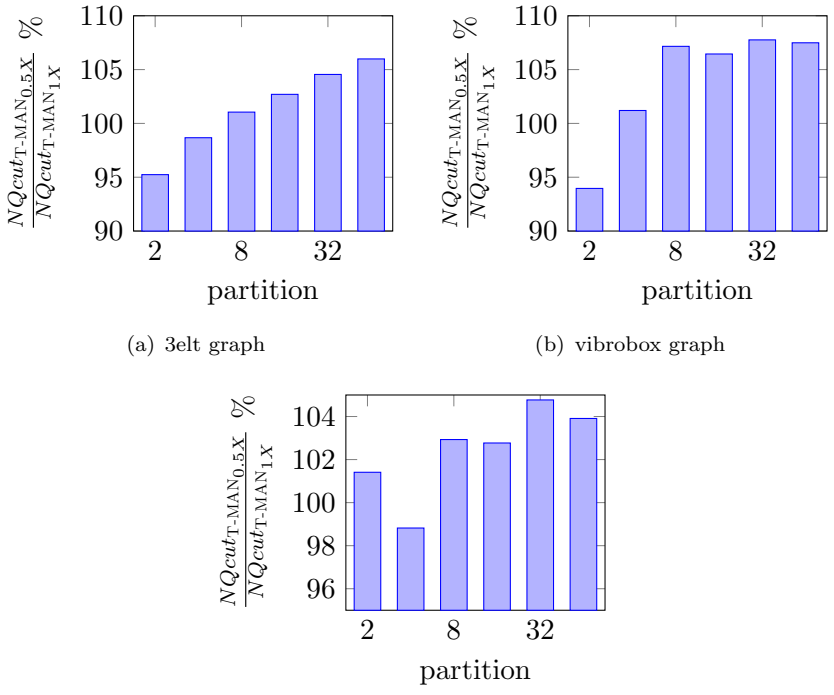


(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.15: Spark JA-BE-JA with T-MAN add-on original frequency Vs. Spark JA-BE-JA with T-MAN add-on double frequency about Edge-cut average results relative to the partitions number

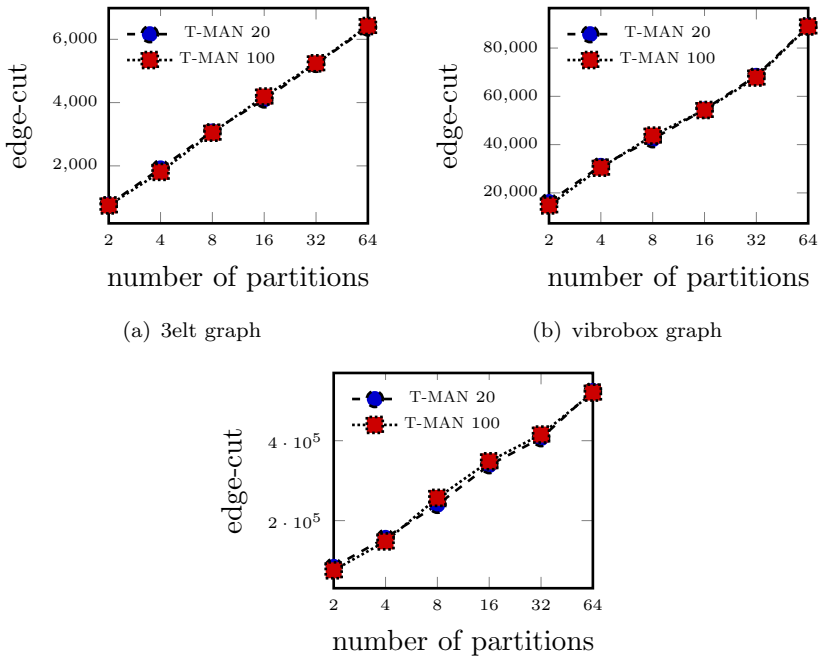


(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.16: Spark JA-BE-JA with T-MAN add-on relatives to the original frequency and double frequency running about NQcut values

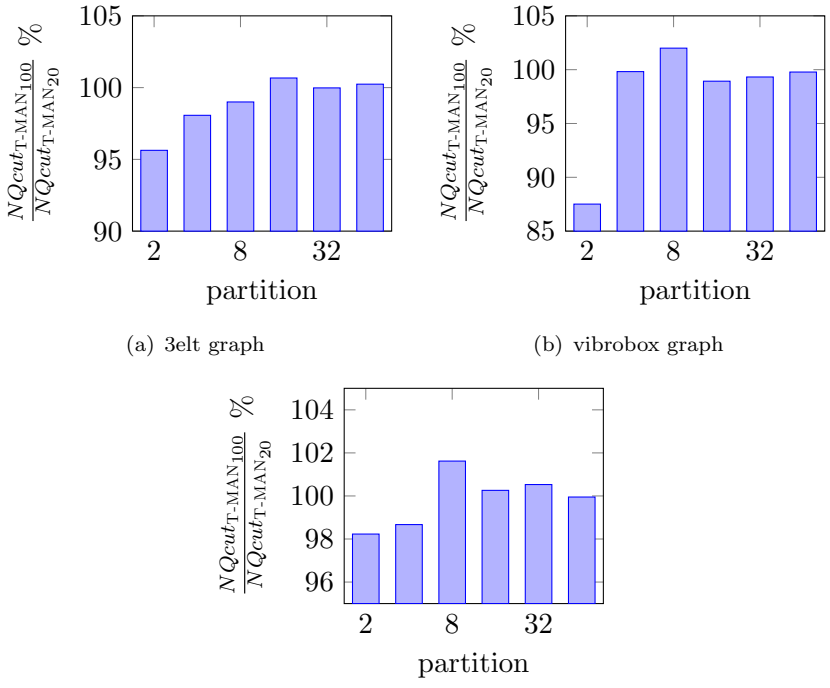


(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.17: Spark JA-BE-JA with T-MAN add-on and random sample size = 20 Vs. Spark JA-BE-JA with T-MAN add-on and random sample size = 100 about Edge-cut average results relative to the partitions number



(a) 3elt graph

(b) vibrobox graph

(c) facebook graph

FIGURE 6.18: Spark JA-BE-JA with T-MAN add-on relatives to the random sample size = 20, 100 NQcut values

7

Conclusions

Everything that has a beginning has an end, Neo.

– Agent Smith, *Matrix Revolutions*

Recently some contributions have been emerged on the distributed graph processing subject by the P2P community, who re-uses its experiences on orchestrating complex environments. However, at best of our knowledge, no tools providing graph computations as P2P simulations have been developed and published so far. Furthermore, Pregel model employs a network vision of the graph processing. Therefore, we proposed ONJAG, a specific distributed networked graph processing abstraction. It provides a layered network environment that recalls a structure similar to the ISO-OSI network stack. Its peculiarity is the inter-communications taking place between the layers composing the stack. Those interactions support the realisation of complex orchestrations. In addition, ONJAG enhances the execution in terms of data exploitation, i.e., computations are no longer expressed as a set of distinct operations on a graph, instead can be seen as an evolutionary process over the graph. From an architectural point of view the definition of independent protocols through modularization allows re-usability of existing approaches. In addition, this supports a gross-fertilization across different research communities by means of a tool behaving as glue between bricks developed independently. To this end we exploited both the P2P approach and the Gossip paradigm. Moreover, ONJAG has been implemented over a state-of-the-art solution: Spark$^{\text{TM}}$, a distributed data processing framework. We extensively tested the effectiveness of the abstraction by implementing a P2P protocols toolbox. The toolbox is composed by a set of protocols. Its aim is to cover a wide range of interaction patterns that can take advantage by the exploitation of P2P-like solutions. The feasibility of the ONJAG abstraction has been discussed and verified. Most of the solutions extracted from the P2P environment are gossip-based. Basically, an epidemic approach supporting highly dynamic changes affecting the data, i.e. the streaming ability has, to a certain extend, already being taken into account.

The balanced minimum k-way cut problem has been studied. It is a widespread recurrent problem in many areas. Indeed, a good vertex-cut partitioning is crucial to let distributed graph processing frameworks efficiently run. To this end, we adopted JA-BE-JA, a state-of-the-art decentralized solution, which aims to deal with extremely distributed large graphs. We have found some issues during our in depth investigations. Therefore, the JA-BE-JA's solution, under certain terms, does not adhere to the balanced minimum k-way cut properties. More precisely, the connected components property is sometimes not guaranteed. In order to verify our hypotheses we contacted the authors of the JA-BE-JA paper that shared its source code with us. We conducted some evaluations with the original JA-BE-JA algorithm in order to give an estimation on average of such solution. The results we obtained have been also used as comparative baseline data for the other investigations we conducted in our thesis.
Moreover, the original authors claim the algorithm's suitability for BSP/Pregel-like distributed frameworks. Unfortunately, the assumptions do not hold with the actual state-of-the-art tools. We re-arranged the orchestration to obtain a Pregel-compliant version by means of Spark operations then, we compared it with the original one using the PeerSim simulator. Indeed, we developed an artificial synchronization barrier into PeerSim in order to study its effects maintaining the other JA-BE-JA assumptions. The results show a significant degradation of the final edge-cut values by the Pregel-compliant implementation compared to the completely distributed one. To face with the performance degradation we studied a few strategies exploiting the Spark platform and the JA-BE-JA logic. We tuned such strategies and finally released completely the JA-BE-JA's assumptions about the neighbourhood coherence constraints. In spite of working with stale data the approximate solution achieves results that are as good as the original edge-cuts. A certain degree of incoherence along the computation allows to adopt a simpler orchestration dealing with speed-ups between $2X$ and $20X$, i.e. 200% and 20000%. The achievable speed-up depends on the topology, the number of partition k and the graph's size.

Finally, a proof-of-concept of the "overlay approach" has been conceived, developed and tested. The proof-of-concept regards the balanced minimum k-way cut. First of all a new measure, called *Normalized Quasi cut* (*NQcut*), has been defined. It is derived from the well-known *Normalized cut* (*Ncut*) of Shi-Malik. It measures the partition configuration taking into account also no-proper mincuts, relaxing the connected components property, i.e. allowing fragmentation. As Shi-Malik state in their paper, the edge-cut is not sufficient to properly evaluate the quality of the final partition configuration. to this end, the *NQcut* has been defined in order to properly evaluate our proposal. After that, we proposed a T-MAN add-on for JA-BE-JA exploiting a stain metaphor of its executions. It consists of evaluating the JA-BE-JA decisional equation by putting into account the "borderness" of the peers provided by the T-MAN protocol. Indeed, the equation evaluation involves the T-MAN layer but the swaps are performed over the JA-BE-JA's topology overlay.

According to the *NQcut* measure, the results are showed to be promising of 5/10% compared to the JA-BE-JA ones. However, for bigger graphs the *ranking function* shows to be quite unsatisfactory requiring further investigations for actual efficacy. Nonetheless, it goes besides the scope of this thesis. We recall that the aim is to provide a proof-of-concept. To this end it is successfully achieved. The T-MAN add-on is an attempt of data exploitation that the ONJAG abstraction enables to be exploited.

7.1 Future Works

Essentially, ONJAG is an abstraction which is agnostic with respect to the underlining distributed framework. Therefore, evaluation on top of other platforms and distributed computational orchestrations should be considered, e.g. the promising *Stratosphere project*[1] recently moved into the Apache incubator as *Apache Flink*[2].

Nonetheless a Spark™ implementation has been developed. The software is an application consisting of orchestrations of *RDD*s, i.e. it is a user-defined API. Along the development and usages it has strongly stood out the needs of a custom RDD in order to exploit properly the distributed environment. The one host-many nodes model is the actual aim that such RDD should implement, i.e. the BSP has to be taken into account instead of the Pregel model and, maintaining the network vision of the latter one, the exploitation of the shared-memory over each machine has to be taken into account. Thus, the computation for each node would be described by 2 functions: a local logic which depends only by current data and a global one that orchestrates the computation without relying over any kind of locality.
Some of the overlay scenarios that could be instantiated, are intrinsically costly in a distributed environment, the random peer sampling service is an example. Indeed, a completely random overlay requires a continuous shuffling of the graph topology, with a consequent overhead affecting computation and communication resources. Thus, the need for a native service directly supported by the framework layer, which carries out this service is strongly desirable. Pre-fetching, opportunistic retrieval and pipe-lining are some of the possible techniques that could be adopted for realising such native service.
This thesis mainly focuses on distributed graph processing targeting non-streaming computations. However, it also deals with streaming executions that require relatively small computational efforts. The streaming support is provided by means of P2P and Gossip algorithms, which supports it "by design". Indeed, the extension of the ONJAG framework, the design of a custom RDD and a native random peer sampling service have to be considered also for the streaming computations.
Finally, Misfire-Spark, a testing and developing facility, has been developed but only a

[1] Stratosphere project's homepage: http://stratosphere.eu/
[2] Apache Flink's homepage: http://flink.incubator.apache.org/

subset of the Spark's APIs are currently provided. The development of the remaining APIs would be a quite trivial operations which has to be done.

The toolbox of protocols developed bundled with the ONJAG framework is a representative set of scenarios that developers can exploit to realise their works. The toolbox includes only P2P and Gossip algorithms although the development of more general network protocols is desirable. Anyway, in order to provide an almost exhaustive complete set of P2P solutions, a few other approaches should be realised, including a grouping/-clustering algorithm and a *Distributed Hash Table* (DHT).
Among the others we developed the distributed k-core decomposition. Unfortunately, it has not been possible to retrieve the datasets used in the original paper and conduct a detailed comparison of our solution with the original one. As a consequence, the correctness of our solution has not been verified. Thus, the employment of external tools for computing the k-core decomposition of the datasets we used has still to be done.

The balanced minimum k-way cut problem has been taken into account. The JA-BE-JA solution, which is a distributed heuristic local search optimization algorithm, has been extensively analysed. Unfortunately, JA-BE-JA sometimes does not calculates a valid partitioning configuration leading to a not proper balanced minimum k-way cut. More precisely, the connected components property could be not satisfied. We proposed a simple workaround doing a local property check along the supersteps. However, alternative orchestration approaches could be designed. Moreover, the initial configuration is affected by the same issue, suggesting that specific solutions can be proposed either in this area. Instead, the initial configuration and the JA-BE-JA execution could be maintained "as-is", limiting our intervention to the design of a merging and fixing solution, i.e. the fragments are merged together re-assembling the partitions but the edge-cut values are somehow maintained.
The tests showed up a very low percentage of swaps occurring in each execution, at least in computations run on the distributed framework. This led to a slow convergence and worse results. Some improvements could be performed in this area, e.g. a transaction system could be designed where a fact-finding swap set is employed for each nodes in order to ensure, within a defined amount of cycles, agreements between pair of nodes.

As a proof-of-concept of the ONJAG abstraction we proposed a T-MAN add-on to the JA-BE-JA algorithm. Our findings look promising even if further experiments have to be run to confirm the results achieved. The tests showed up that the T-MAN protocol requires a certain effort for tuning its parameters in order to find a configuration that is effectively JA-BE-JA-friendly.
In addition, the *ranking function* stood out not to be satisfactory when $k > 2$ and it does not distinguish between partitions and fragments, hence new ranking functions should be conceived.
Drastically another possible choice is to replace the T-MAN algorithm with another overlay topology manager, e.g. SCAMP [23], or something similar, e.g. solutions for

distributed clustering.
Besides that, we defined an original mincut measure which takes into account the fragmentations in case the connected components property is not verified. It is derived starting from the *normalized mincut* by Shi-Malik. An *evaluation function*, which is defined in equation (6.9), is employed in order to measure the quality of fragmentation. However, the *normalized quasi mincut* (*NQcut*) does not define a fixed bias evaluation function because it is domain dependent. To overcome this limitations, in Chapter 6 we defined our $f(X_i)$ which measures the total fragmentation, the relative sizes of the fragments and how they are placed one to each other. This last metric, called *Ndistances*, has not been implemented yet. In the next future we plan to develop it and re-evaluate our tests accordingly.

Bibliography

[1] V. Abhishek, L. Xavier, D. E. Goldberg, and R. H. Campbell. Scaling genetic algorithms using mapreduce. *Intelligent Systems Design and Applications, International Conference on*, 0:13–18, 2009.

[2] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. k-core decomposition: a tool for the visualization of large scale networks. *CoRR*, abs/cs/0504107, 2005.

[3] V. Batagelj and M. Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.

[4] A. Berger, A. Grigoriev, and R. Van der Zwaan. Complexity and approximability of the k-way vertex cut. *Networks*, 63(2):170–178, 2014.

[5] S. L. Bressler and V. Menon. Large-scale brain networks in cognition: emerging methods and principles. *Trends in Cognitive Sciences*, 14(6):277 – 290, 2010.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Seventh International World-Wide Web Conference (WWW 1998)*, 1998.

[7] A. Buluç and J. R. Gilbert. The combinatorial blas: design, implementation, and applications. *International Journal of High Performance Computing Applications*, 2011.

[8] V. Cardellini, I. Roma, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3:28–39, 1999.

[9] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.

[10] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[11] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 241–251. ACM, 1992.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12. ACM, 1987.

[14] J. Dongarra. Trends in high performance computing: a historical overview and examination of future developments. *Circuits and Devices Magazine, IEEE*, 22(1):22–27, 2006.

[15] A. J. Enright, S. V. Dongen, and C. A. Ouzounis. An efficient algorithm for large-scale detection of protein families. *Nucleic.Acids.Res.*, 30:1575–84, 2002.

[16] P. Erdős and A. Rényi. On the evolution of random graphs. In *Publication of the mathematical institute of the hungarian academy of sciences*, pages 17–61, 1960.

[17] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1999.

[18] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, 1972.

[19] J. Fung and S. Mann. Using multiple graphics cards as a general purpose parallel computer: applications to computer vision. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 1, pages 805–808 Vol.1, 2004.

[20] J. Fung, F. Tang, and S. Mann. Mediated reality using computer graphics hardware for computer vision. In *ISWC*, pages 83–89. IEEE Computer Society, 2002.

[21] W. Galuba, K. Aberer, D. Chakraborty, Z. Despotovic, and W. Kellerer. Outtweeting the twitterers - predicting information cascades in microblogs. In *Proceedings of the 3rd Wonference on Online Social Networks*, WOSN'10, pages 3–3. USENIX Association, 2010.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[23] A. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *Computers, IEEE Transactions on*, 52(2):139–149, Feb. 2003.

[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th*

USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pages 17–30. USENIX Association, 2012.

[25] L. Hai-Guang, W. Gong-Qing, H. Xue-Gang, Z. Jing, L. Lian, and W. Xindong. K-means clustering with bagging and mapreduce. *2013 46th Hawaii International Conference on System Sciences*, 0:1–8, 2011.

[26] W. Hanli, S. Yun, W. Lei, Z. Kuangtian, W. Wei, and C. Cheng. Large-scale multimedia data mining using mapreduce framework. *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, 0:287–292, 2012.

[27] V. Helms. *Principles of Computational Cell Biology: from protein complexes to cellular networks*. Wiley-VCH, Weinheim, 2008.

[28] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95. ACM, 1995.

[29] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. *SIGPLAN Not.*, 47(4):349–362, March 2012.

[30] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, Sept. 1989.

[31] IBM, P. Zikopoulos, and C. Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011.

[32] M. Jelasity and O. Babaoglu. T-man: Gossip-based overlay topology management. In *Proceedings of the Third International Conference on Engineering Self-Organising Systems*, ESOA'05, pages 1–15. Springer-Verlag, 2006.

[33] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), Aug. 2007.

[34] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96. IEEE Computer Society, 1996.

[35] K.-i. Kawarabayashi and M. Thorup. Minimum k-way cut of bounded size is fixed-parameter tractable. *CoRR*, abs/1101.4689, 2011.

[36] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491, Oct. 2003.

[37] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[38] M. Koubarakis. Multi-agent systems and peer-to-peer computing: Methods, systems, and challenges. In M. Klusch, A. Omicini, S. Ossowski, and H. Laamanen, editors, *Cooperative Information Agents VII*, volume 2782 of *Lecture Notes in Computer Science*, pages 46–61. Springer Berlin Heidelberg, 2003.

[39] Y. Kubera, P. Mathieu, and S. Picault. Ioda: an interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23(3):303–343, 2011.

[40] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley, 6/e edition, 2013.

[41] J. Leskovec. Snap library, July 2009. http://snap.stanford.edu/.

[42] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[43] M. Leyton and J. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296, Feb. 2010.

[44] M. Livny and C. Team. Condor manual, 1990. http://www.mi.infn.it/condor/manual/Contents.html.

[45] Y. Low, J. Gonzalez, A. Kyrola, B. Danny, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[46] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146. ACM, 2010.

[47] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity, May 2011. http://www.mckinsey.com/Insights/MGI/Research/Technology_and_Innovation/Big_data_The_next_frontier_for_innovation.

[48] J. L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *j-CACM*, 3(4):184–195, April 1960.

[49] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–13, April 2008.

[50] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.

[51] A. Montresor and M. Jelasity. Peersim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Sept. 2009.

[52] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism, HotPar*, volume 11, pages 10–10, 2011.

[53] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *SASO*, pages 51–60. IEEE, 2013.

[54] I. Raicu, I. T. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11. IEEE, 2008.

[55] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[56] S. Salihoglu and J. Widom. Gps: a graph processing system. *SSDBM*, 2012.

[57] P. Sanders and C. Schulz. Engineering multilevel graph partitioning algorithms. In C. Demetrescu and M. Halldórsson, editors, *Algorithms – ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer Berlin Heidelberg, 2011.

[58] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269 – 287, 1983.

[59] S. Seo, E. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726, 2010.

[60] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, Aug. 2000.

[61] O. Sporns, D. R. Chialvo, M. Kaiser, and C. C. Hilgetag. Organization, development and function of complex brain networks. *Trends in cognitive sciences*, 8(9):418–425, 2004.

[62] D. Stingl, C. Gross, J. Ruckert, L. Nobach, A. Kovacevic, and R. Steinmetz. Peerfactsim.kom: A simulation framework for peer-to-peer systems. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 577–584, July 2011.

[63] E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[64] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[65] J. D. Ullman, A. V. Aho, and J. E. Hopcroft. The design and analysis of computer algorithms. *Addison-Wesley, Reading*, 4:1–2, 1974.

[66] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[67] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2Nd ACM Workshop on Online Social Networks*, WOSN '09, pages 37–42. ACM, 2009.

[68] C. Walshaw. The graph partitioning archive, Aug. 2012. http://staffweb.cms.gre.ac.uk/~wc06/partition.

[69] G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[70] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6. ACM, 2013.

[71] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2. USENIX Association, 2012.

[72] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10. USENIX Association, 2010.