Large Scale Social Network Analysis

By

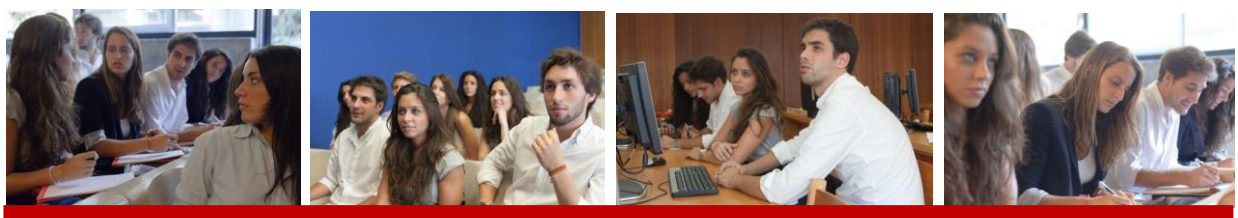
Rui Portocarrero Sarmento

2013

Master Thesis – Data Analysis and Decision Support Systems

Supervisors:

João Gama
Albert Bifet



Keywords: network analysis, large graph networks, parallel computing, network community structure

Biography

Rui Sarmento has a degree in Electrical Engineering by University of Porto, Faculty of Engineering. He has worked in several areas from 3G mobile networks with functions in an international technical support center to software development companies focusing on Communications and Intranet solutions with Linux based Enterprise Operating Systems. Finally he has also worked for the main public transportation company in his hometown, Porto, as a project management engineer in the informatics and communications area.

He is currently also collaborating with LIAAD (Laboratory of Artificial Intelligence and Data Analysis) in INESCP.

Acknowledgments

First, I would like to thank my advisors João Gama and Albert Bifet for their advices and support. This thesis would not have been possible without them and without the encouragement they have given me over the last two years. I'll thank specially to João Gama for giving me the opportunity for the time I spent at LIAAD (Artificial Intelligence and Data Analysis Lab) surrounded by great Doctorate Students. They were always supporting and very helpful. I would also like to thank Albert Bifet for being a never ending resource of information for new and arising tools used throughout the thesis.

I thank a lot to Marcia Oliveira. The time spent studying with her at LIAAD was great. I thank her for the help she gave me understanding important theoretical subjects and for making the thesis development task a lot easier.

I also would like to thank to FEUP (Faculty of Engineering - University of Porto) for lending the "grid cluster". That made the task of dealing with large datasets feasible, for that I would like to thank Jonathan Barber for the immense enthusiasm and support when I was dealing with supercomputers and clusters without previous knowledge on the subject.

I want to thank Américo Amaral for helping me with the installation of Hadoop on VM's (Virtual Machines). His help was much appreciated and his insight on Hadoop install was always of great value on the beginning of thesis research.

The thesis final phase consisted on the development of an algorithm previously not developed with the Green-Marl language and in spite being a relatively fresh language I had the pleasure to be guided by Martin Sevenich, Hong Sungpack and Austin Gibbons, some of them contributing for the development of the language or that had contributed to it in the past. They were all great help in understanding the language syntax and behavior in parallel environments.

Finally I want to thank Tiago Cunha for the great companionship while we were studying for the master courses and particularly because he helped me when gathering some considerable quantity of data in study on this document.

Abstract

Throughout this document we present an in depth study and analysis of very large scale social networks. Besides the explanation of how to use and install several tools available, we explain in detail the basis of these tools. Several algorithms will be used to give the reader knowledge of different tools and technique results. Using tools like Graphlab or using Hadoop and Hadoop Map Reduce based tools like Pegasus or Giraph we will compute some important metrics. We will also use an optimized tool for graph analysis, it is called Snap (Stanford Network Analysis Platform). Although this tool is not inherently a parallel computing one, it can serve as a reference for non parallel graph analysis software.

Several metrics will be computed for several different size networks including a case study using data previously achieved from the CrunchBase databases. One of this particular Crunchbase network has relationships between technological companies and financial organizations. Another network is also derived from Crunchbase databases with relationships between persons and technological companies.

Finally, using parallel computing paradigm, two distinct algorithms will be implemented, a community detection algorithm and also a similarity ranking algorithm. Both algorithms behavior will also be subject of studies with test networks.

PORTUGUESE VERSION

O objetivo deste documento é explorar em profundidade o estudo das redes sociais de grande escala. Além da exposição ao leitor do método de utilização e instalação de diversas ferramentas disponíveis também será explicada a arquitetura funcional dessas ferramentas. Serão utilizados vários algoritmos para dar ao leitor uma noção das técnicas de funcionamento e correspondentes resultados para cada uma das ferramentas. Serão calculadas algumas métricas importantes, usando ferramentas como o Graphlab ou usando o Hadoop e ferramentas baseadas no Hadoop Map Reduce como o Pegasus ou o Giraph. Adicionalmente utilizaremos ferramentas otimizadas para a análise de redes sociais como o Snap (Stanford Network Analysis Platform) que embora não sendo uma ferramenta de computação paralela serve como referência neste campo.

Vários algoritmos serão computados para redes de diferentes tamanhos incluindo um caso de estudo com redes obtidas da base de dados Crunchbase. Esta rede Crunchbase é composta pelas relações entre empresas tecnológicas e organizações financeiras. Também derivada da base de dados Crunchbase está outra rede com as ligações entre personalidades e as empresas tecnológicas.

Finalmente, utilizando as bases da computação paralela, foram desenvolvidos dois algoritmos distintos. Um algoritmo de deteção de comunidades e um algoritmo de cálculo do ranking de similaridades (simrank) entre nós de uma rede. Ambos os algoritmos serão também sujeitos a estudos de comportamento com redes de teste.

Table of Contents

Figures & Algorithms

Tables

1. Introduction

1.1. Motivation

Graphs are the main used representation for the social networks structure. In graph theory, a graph is a representation of a network of entities, objects or beings where some of them are connected by links. They are abstracted by being represented by nodes, vertices or vertexes and the links between them are called edges. They are visually presented typically by a diagram with a set of dots for the vertexes and joined by curves or lines for the edges. The edges may be directed or undirected. For example, in a scientific conference the public will know the orator but the orator might not know all the elements in the audience so the connections between the audience and the orator will be represented by directed connections. If the orator gets to know some particular person in the audience them the connection will be therefore undirected since the orator knows the audience member and the audience member knows the orator.

Graph computations are often completely data-driven, dictated by the vertex and edge (node and link) structure of the graph on which it is operating rather than being directly expressed in code. In particular, the above properties of graph problems present significant challenges for efficient parallelism. As a result, parallelism based on partitioning of computation can be difficult to express because the structure of computations in the algorithm is not known a priori.

The data in graph problems are typically unstructured and highly irregular. Graph data makes it difficult to extract parallelism by partitioning the problem data. Scalability can be quite limited by unbalanced computational loads resulting from poorly partitioned data.

Performance in contemporary processors is predicated upon exploiting locality. Thus, high performance can be hard to obtain for graph algorithms, even on serial machines.

In graph algorithms computation there is typically a higher ratio of data access than for other scientific applications computation. Since these accesses tend to have a low amount of exploitable locality, runtime can be dominated by the wait for memory fetches. All this problems are discussed extensively by Lumsdaine *et al.* (2007) and will be exposed in this document. The majority of tools used for this thesis development

address the problems searching for solutions and specifically addressing large graph analysis issues.

1.2. Thesis Overview

This document tries to gather on one single document as much information possible about the parallel computing tools available nowadays for the purpose of social networks analysis, more concretely for those of large scale. Several tools and different algorithms were used to gather information on several different networks of large scale, impossible/very difficult to study on a normal commodity machine and with sequential software due to time consuming processing.

On Chapter 2 we describe the state of the art of parallel computing architectures, hardware and software approaches to the subject of study in this document, i.e. large scale graph analysis.

Chapter 3 introduces the reader to the tools available for graph analysis, describe their functional characteristics and prepare the user to use introduced tools on practical use cases. On Section 3.4, previous to explore practical use cases with tools previously introduced also on Chapter 3, the characteristics of the data used for this task are also explained.

On Chapter 4 we describe the development process of two parallel algorithms. There is an introduction to these metrics and then the developed code results on some test case data specifically used to focus on the algorithms characteristics.

Finally on Chapter 5 we take conclusions on the overall work developed for this document and explain also the possible further developments of this work and what we think could be a good update for it in the future.

1.3. Contributions

With this document we tried to compile as much information to compare the tools available for graph analysis nowadays. There will be a comparison of these tools regarding several important subjects like advantages and disadvantages, offered algorithms from installation and also methods for installing and running these tools.

Other main contributions of this work are novel implementations of an algorithm for community detection and also of a similarity ranking calculation algorithm, with a recently developed specific domain language. They were developed and experimented with a language for graph analysis domain called Green-Marl. This specific language tool is also exposed and explained on this document's Section 4.4.

Resuming this document has these main contributions:

1. Aggregation of information:
    a. What tools to use for analyzing large social networks
    b. How to install the tools
    c. What algorithms are already implemented with these tools
    d. How to run the offered algorithms

2. Implementation of algorithms for large scale Social Network analysis:
    a. Community Detection algorithm implementation with Green-Marl language
    b. Similarity Ranking algorithm implementation also with Green-Marl language

2. State of the art

This Chapter introduces the state of the art architectures and software strategies available recently, that are scalable to large networks since they use parallel processing. Therefore this Chapter is dedicated only to parallel processing. We will write about major technologies used by data scientists to approach the problem of big data particularly on the large/very large graphs subject.

Sections 2.1 to 2.3 are an overview based on the important paper by Lumsdaine *et al.* (2007) addressing on parallel computing for graph analysis. This way we intend to expose recent research made on this subject and subsequently its evolution as time evolved until today.

2.1. Parallel Architectures & Programming Models

Nowadays most machines are built based on standard microprocessors, with the use of hierarchical memory. The processing is usually optimized reducing latency with fast memory to store values from memory addresses that are likely to be accessed soon. Although for the majority of modern applications, this is a way to improve performance, it is not particularly effective for unstructured graphs calculations as we will see.

Distributed Memory Machines

This type of machines is usually programmed by explicit message passing by the user. He is responsible for the division of data among the memories and also responsible for the assigning of different tasks to the processors.

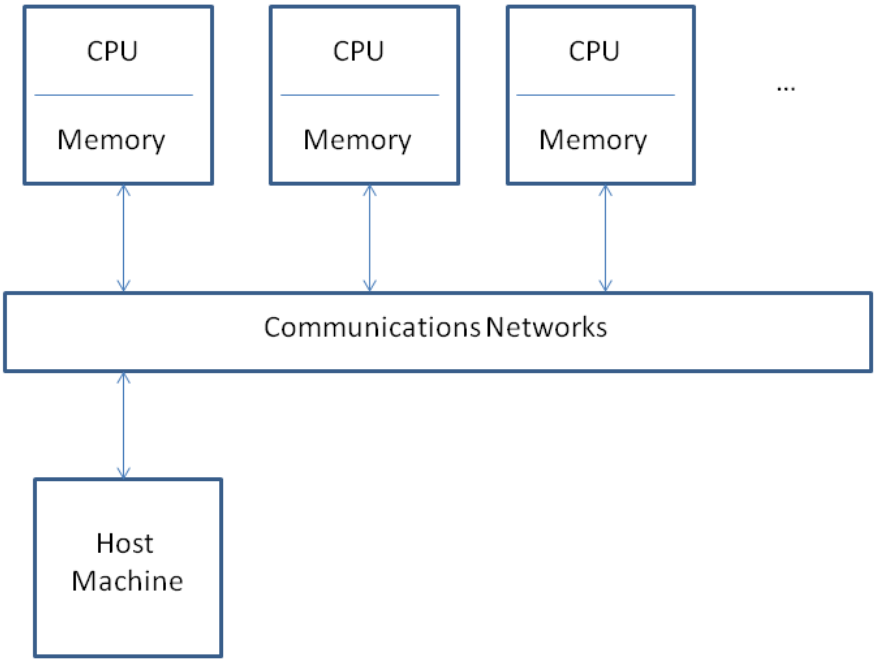


Figure 1: Distributed Memory Machines

The exchanging of data between processors is governed by user controlled messages, generally with the help of MPI communication library from Borchers and Crawford (1993). This way and for many users applications high performance is achieved but the high detail in messages control can be fastidious and errors might be usual.

Normally programs are written in a way that processors might work independently on local data and might also work collective in a group of processors with operations based on communications between them as specified from Valiant (1990). However data cannot be exchanged instantly and processing demands that it can only be done on breaks between computation tasks. This characteristic makes it tedious to explore fine-grained parallelism making distributed memory machines not suited to this kind of parallelism.

Partitioned global address space computing

Partitioned global address such as UPC from El-Ghazawi *et al.* (2003) is more adequate for fine-grained parallelism. The feature of a global address space makes easier the writing of applications with data access patterns of higher complexity.

As can be seen on Fig. 2 UPC is based on a single communication layer therefore parallel programs of fine-grained type achieve better performance than using MPI library for communication between CPU's, memory and host machines.

One constraint of UPC programs, as for MPI is that the number of threads is limited and constant, usually equal to the number of processors. As will be pointed on the sections below the lack of dynamic threads makes it generically difficult to build up superior performance software for graph analysis.

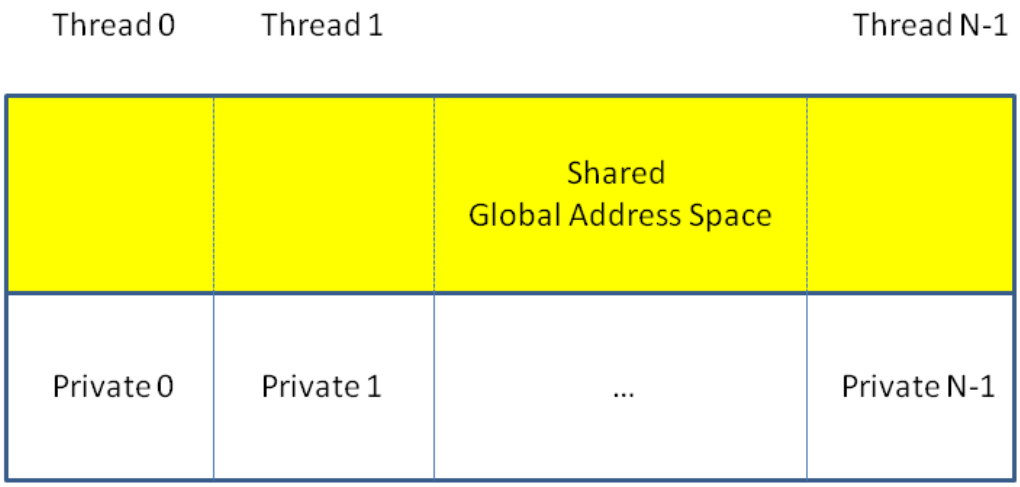


Figure 2: Global Address Space Computing

2.1.1. Shared-memory computers

UPC features a globally addressable memory by software on distributed memory hardware but it can also be provided in plain hardware. Shared memory computers can be categorized in several groups. Here it is only considered cache-coherent computers and also massively multithreaded computers.

Cache-coherent parallel computers

With SMPs (symmetric multiprocessors) global memory is globally reachable by each processor. UPC might be used to program these machines although the most usually used is OpenMP from Dagum and Menon (1998) or even a POSIX threading approach from Society (1990). In this Thesis we will use exclusively SMPs machines for computing metrics for graph analysis in Section 3.4 and will also use OpenMP in chapter 4 for the development of graph analysis algorithms.

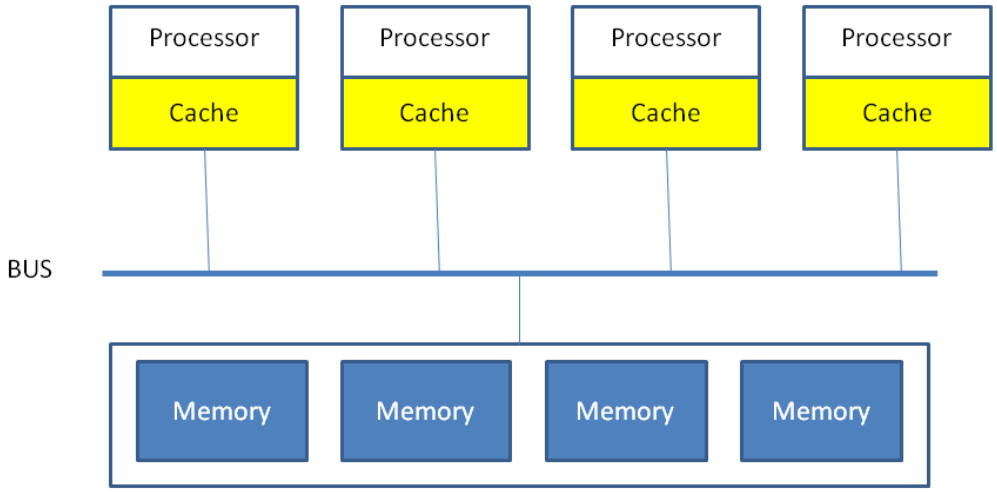


Figure 3: SMP global memory

SMP characteristics make it possible for a program to access any addresses in global memory directly and sensibly fast because of its hardware support. Therefore unstructured problems can achieve better performance than is possible on distributed memory machines. SMPs are therefore dependant of faster hardware for accessing memory and subsequently with lower latency.

As seen above in Fig. 3 processors possess a memory hierarchy in which a small amount of data is kept in cache, a faster memory for quick access and to ensure read operations get the most recent values for variables.

In a multiprocessor computer with multiple caches, cache-coherence is a great and challenging task adding overhead which can degrade performance. For problems in which reads are much more prevalent than writes, cache coherence protocols impact on performance and scalability.

Another challenge with SMPs is thread synchronization and scheduling being possible that some threads be blocked for a period of time. Recent versions of OpenMP required that the number of threads be equal to the number of processors and therefore a blocked thread corresponds to an idle processor and that may impact on performance as will see in section 4.6.2 with some practical use cases and for developed algorithms that use OpenMP.

Massively multithreaded architectures

Massive multithreaded machines are built upon custom processors which are more expensive and have much slower clock frequency than mainstream microprocessors. MTA-2 from Anderson *et al.* (2003) is an example of this type of machine and it has also a non-standard programming model although it might be considered simple.

2.2. Mapping Parallel Graph Algorithms to Hardware

Parallel graph algorithms have been classified to be difficult to develop. The challenging characteristics of software and hardware to take care with in the development process are the following:

Task Granularity: With centrality measures computations it is common to use many shortest path calculations and therefore there is a significant quantity of coarse-grained parallelism. Resuming, each shortest path could be a separate task, but for the majority of graph calculations parallelism is exclusively found on fine-grained parallelism. Hardware architecture that makes it easy to use fine-grained parallelism would be more suited to run such type of algorithms.

Memory contention: In global address space systems, multiple threads try to simultaneously access the same memory. This reduces performance on the majority of situations. This problem grows in the same measure the degree of parallelism increases and is maximized with multithread machines. A graph algorithm will usually not write within the graph input but it has to create and write its own data structures and therefore memory addressing must be handled with care.

Load Balancing: For some cases of graph algorithms, for example breadth-first search, load balancing might change over time (few vertices to visit in the beginning and more in the end). This problem is less worrying with shared-memory machines because work tasks can be migrated between processors without having to move data from and to memory.

Simultaneous Queries: A large graph may be queried by a group of analysts simultaneously, and for that, architecture should focus on throughput.

Distributed-memory architectures

Distributed memory and message passing machines have the least propensity to fine-grained parallelism and are hard to make them perform dynamic load balancing. On the other side and with a more generic behavior MPI programs will run on almost all parallel platforms.

With edges and vertices of a graph partitioned among processors in a distributed memory system, if a processor owns a vertex, it needs to have a mechanism to find his vertex's neighbors. This issue is solved widely in many applications by keeping a local sub-data structure with the information of all the neighbors/adjacent vertex's (also called ghost cells) to the vertex's owned by the processor or local to a process. This kind of solutions is well applied to graph structures where a low amount of edges are spread across different processors, and these kind of graphs are usual in scientific graph problems. In addition, high-degree vertices cause problems in distributed memory, as they may overload the memory available on a single processor.

An alternative to ghost cells is to use a hashing scheme to assign vertices to processors. Although hashing can result in memory savings compared to ghost cells, it can incur significant computational overhead.

Partitioned global address space computing

In this case global address space makes it obvious for the need for ghost cells, facilitating finer-grained parallelism and dynamic load balancing. Data layout may be important for performance though, since the graph is partitioned and non-local accesses induce overhead. UPC language implementations might be difficult because of its limited support and portability.

Cache-coherent, shared-memory computers

SMPs have all the advantages of partitioned global address space computing. They have lower latencies because they provide hardware support for global address access though they have a limitation of one thread per processor. They also have complicated memory access patterns making processor idles usual while waiting for memory.

Massive multithreaded machines

Massive multithreaded machines support both coarse and fine-grained parallelism and are amenable to load-balancing and simultaneous queries. Adding to these good features they do not have the complexity and performance costs of implementations of cache-coherence of SMPs.

The main problem with massively multithreaded algorithms is the amount of threads in itself because if it is in numbers much greater than the number of processors memory contention issues are more common. This technology is also said to have an uncertain future so the commitment to development based on this architecture may not be advised or is considered risky.

2.3. Software Approaches

Parallel Boost Graph Library

By abstracting away the reliance on a particular communication medium, the same algorithm in the Parallel BGL (Boost Graph Library) from Gregor *et al.* (2005) can execute on distributed-memory clusters using MPI (relying on message passing for communication) or SMPs using Pthreads (relying on shared memory and locking processors for communication).

With parallel BGL, multiple algorithm implementations may be required to account for radical differences in architecture, such as the distinction between course-grained parallelism that performs well on clusters and some SMPs and fine-grained parallelism that performs well on massively multi-threaded architectures like the MTA-2.

Multi-Threaded Graph Library

The MTA-2 and XMT simple programming model assure its high level propensity for the generic programming but it had constraints because of its novelty and immature status regarding its software library.

Another solution is the MultiThreaded Graph Library from Berry *et al.* (2006), inspired by the serial Boost Graph Library, developed at Sandia National Laboratories to provide a near-term generic programming capability for implementing graph algorithms on massively multithreaded machines. Like the Parallel BGL, underlying data structures are leveraged to abstract parallelism away from the programmer. The

key to performance on MTA/XMT machines is keeping processors busy, and in practice this often reduces to performing many communicating, asynchronous, fine-grained tasks concurrently. The MTGL provides a flexible engine to control this style of parallelism. The MTGL was developed to facilitate data mining on semantic graphs, i.e., graphs with vertex and edge types. Furthermore, the XMT usage model allows many users to run algorithms concurrently on the same graph. The MTGL is designed to support this usage model.

SNAP, small-world network analysis and partitioning framework

SNAP (Small-world Network Analysis and Partitioning) is a modular graph infrastructure for analyzing and partitioning interaction graphs, targeting multicore and many core platforms. SNAP is implemented in C language and uses POSIX threads and OpenMP primitives for parallelization. The source code is freely available online from [1]. In addition to partitioning and analysis support for interaction graphs, SNAP provides an optimized collection of algorithmic "building blocks" (efficient implementations of key graph-theoretic kernels) to end-users. Novel parallel algorithms for several graph problems were designed and run efficiently on shared memory systems. SNAP framework team does implementations of breadth-first graph traversal, shortest paths, spanning tree, MST, connected components, and other problems achieve impressive parallel (multicore) speedup for arbitrary, sparse graph instances. SNAP provides a simple and intuitive interface for network analysis application design, whose objective is hiding the parallel programming complexity involved in the low-level kernel design from the user, as mentioned by Bader and Madduri (2008).

Recent Approaches

To deal with big data applications, more recently, a new software paradigm has appeared. These programming systems are designed to get their parallelism not only from a "supercomputer," but from "computing clusters" – large groups of hardware, including conventional processors or "nodes" connected by some particular mean (Ethernet cables or switches) on a computer network. The software stack works with a new form of file system, called a "distributed file system," which features an extension of any disk array in a conventional operating system. Distributed file systems ("DFS")

[1] http://snap-graph.sourceforge.net/

are also prepared to protect against the frequent failures that occurs when data is distributed over hundreds or thousands of compute nodes, and DFS does it by providing replication of data or redundancy. Keeping in mind these kinds of file systems many different programming systems have been developed. Map-Reduce was one of them and has been used extensively for the most common calculations on large-scale data performed on computing clusters. Map-Reduce is used in lots of ways because is efficient for most calculation cases and is tolerant of hardware failures during the computation. We will deal with this new approach with more detail on the next sections of this thesis.

2.4. Recent Approach: Distributed File System

Normally most computing is done on a single node processor, with its main memory, cache, and local disk. Not long ago, applications that called for parallel processing, such as large scientific calculations, were done on special-purpose parallel computers with many processors and specialized hardware. However, the new computing facilities existing today have given rise to a new generation of programming systems. These systems take advantage of the power of parallelism and at the same time avoid the reliability problems that arise when the computing hardware consists of thousands of independent components. This section discusses the characteristics of this type of specialized file systems that have been developed to take advantage of large sets of nodes. Later in this document's chapter 3, several tools for graph metrics computations will be introduced. The vast majority of these introduced tools are also DFS based, typical in a distributed computation environment.

2.4.1. Architecture of compute nodes

Normally compute nodes might be stored on racks of computers. On each rack computers might be connected with gigabit Ethernet switch or even fiber optics network cards and switches, if exists more racks these are connected by another network. It is expected greater bandwidth capacity for the hardware connecting the racks because it is essential for efficient communication between large racks in need for much more bandwidth than the communication process between nodes in each individual rack.

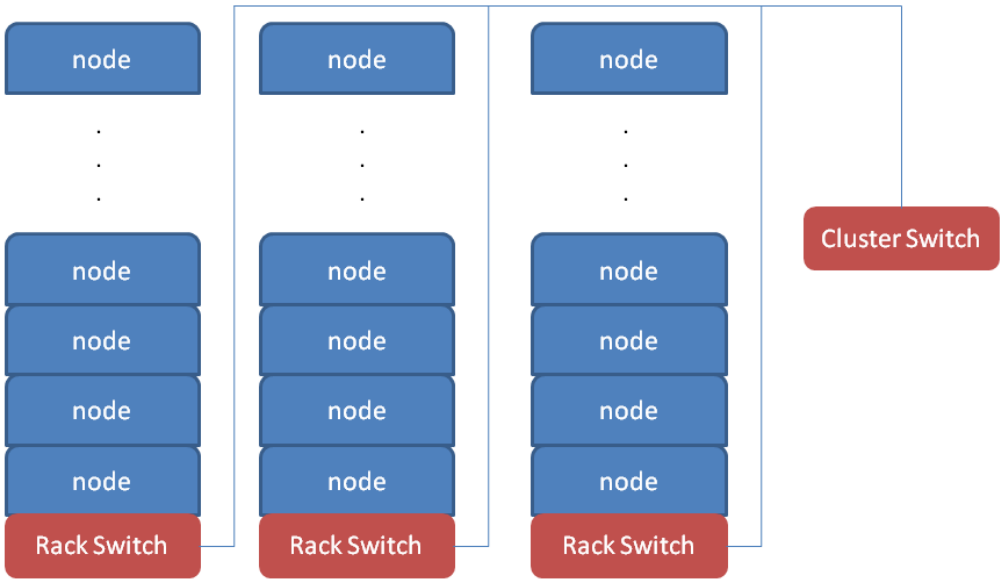


Figure 4: Compute nodes in racks, connected by rack switches interconnected by a cluster switch

For systems such as Fig. 4, the principal points of failure modes are the loss of a single node when for example the disk crashes or because of the network card malfunctions or the loss of an entire rack when for e.g. the rack switch fails to communicate with the cluster switch.

There are solutions to this problem that can take two different shapes:

1. Files are stored redundantly. The files are duplicated at several compute nodes. This new file system, often called a distributed file system or DFS:

    a. Examples of DFS systems:
        i. Google File System (GFS)
        ii. Hadoop Distributed File System (HDFS)
        iii. CloudStore
    b. DFS systems are often used in these situations:
        i. used with big files, possibly files with terabytes of size.
        ii. Files are rarely updated.
    c. How does “DFS” work?
        i. Normally, both the chunk size and the degree of replication can be decided by the user, an example feature of a “DFS” could be:

1. Chunks that are replicated, perhaps four times, at four different compute nodes.
2. The nodes containing copies of data are located at different racks of computers therefore avoiding loss of data if rack fails.
3. There is a *master node* or *name node* controlling the location of file chunks and therefore every node using DFS knows where the files are located.

2. Division of computations into tasks, such that if any one task fails to execute to completion, it can be restarted without affecting other tasks. This strategy is followed by the map-reduce programming system.

2.5. Introduction to Hadoop

Hadoop is a framework developed for running applications on large clusters. Apache Hadoop is the open source implementation of Google's Map/Reduce methodology, where the application is divided into several small fragments of work and each may be executed or re-executed on any node in the cluster. For that purpose Hadoop provides a distributed file System (HDFS) that stores data on the several nodes. Hadoop framework also automatically handles node failures regarding Map/Reduce tasks and also the HDFS system as cited by Mazza (2012).

Map/Reduce is a set of code and infrastructure for parsing and building large data sets. A *map* function generates a key/value pair from the input data and this data is then reduced by a *reduce* function that merges all values associated with equivalent keys. Programs are automatically parallelized and executed on a run-time system which manages partitioning the input data, scheduling execution and managing communication including recovery from machine failures.

Regarding its architecture, Hadoop Cluster code is written in JAVA and consists of compute nodes, also called "TaskTrackers" managed by "JobTrackers". It is also composed by a distributed file system (HDFS) i.e. a "namenode" with "datanodes".

The “JobTracker” coordinates activities across the slave “TaskTracker” processes. It accepts Map-Reduce job requests from clients and schedules *map* and *reduce* tasks on “TaskTrackers” to perform the work.

The “TaskTracker” is a daemon process that spawns *map* and *reduce* child processes to perform the actual *map* or *reduce* work. Map tasks typically read their input from HDFS, and write their output to the local disk. Reduce tasks typically read the map outputs over the network and write their outputs back to HDFS.

Please see Figure 5 explaining interactions between “JobTrackers” and “TaskTrackers”:

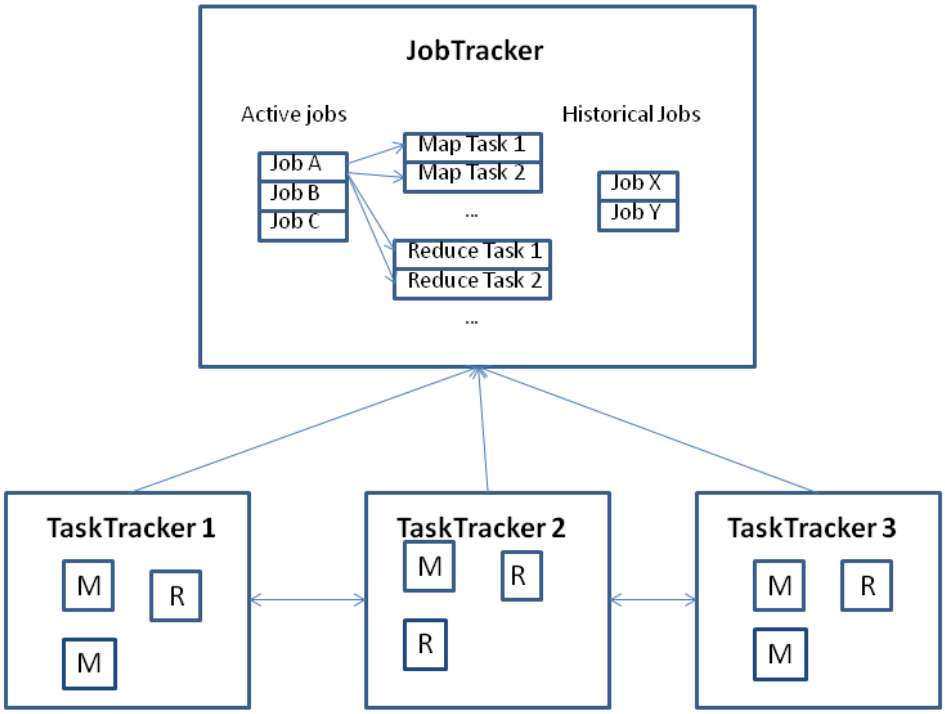


Figure 5: Map-Reduce Job and Task Tracking

The “TaskTrackers” send heart beats signaling to the “JobTracker” at regular intervals, with the heart beat they also indicate when they can take new *map* and *reduce* tasks for execution. Then the “JobTracker” consults the Scheduler to assign tasks to the “TaskTrackers” and sends the list of tasks as part of the heart beat response to the “TaskTrackers”.

2.5.1. Physical Architecture

Hadoop's component ZooKeeper requires an odd-numbered of machines so the recommended practice is to have at least three of them in any reasonably sized cluster.

It's true that Hadoop can run on any kind of servers, even the old ones, but for better results mid-level rack servers with dual sockets, as much RAM as is affordable, and SATA drives optimized for RAID storage. Using RAID, however, is strongly discouraged on the "DataNodes", because of HDFS being already implementing the replication and error-checking by nature; but on the "NameNode" it's strongly recommended for additional reliability.

From a network topology perspective with regards to switches and firewalls, all of the master and slave nodes must be able to open connections to each other. For small clusters, all the hosts would run 1 GB network cards connected to a single, good-quality switch. For larger clusters look at 10 GB top-of-rack switches that have at least multiple 1 GB uplinks to dual-central switches. Client nodes also need to be able to talk to all of the master and slave nodes, but if necessary that access can be from behind a firewall that permits connection establishment only from the client side as mentioned by Holmes (2012).

2.5.2. Hadoop Users

Hadoop has a high level of penetration in high-tech companies and is spreading across other sectors. As a small example the following web companies use Hadoop:

1. Facebook uses Hadoop to store copies of internal log and dimension data sources and use it as a source for reporting/analytics and machine learning. Currently Facebook has two major clusters, one with 1100-machine with 8800 cores and about 12 PB raw storage. They have yet another 300-machine cluster with 2400 cores and about 3 PB raw storage. For both this clusters each commodity node has 8 cores and 12 TB of storage.

2. Yahoo! uses Hadoop in more than 100,000 CPUs on a 40,000 computers cluster. Their biggest cluster has 4500 computers. Hadoop is used to support

research for Ad Systems and Web Search. It is also used to do scaling tests to support development of Hadoop on larger clusters.

3. Twitter uses Hadoop to store and process tweets, log files, and many other types of data generated across Twitter. They use Cloudera's CDH2 distribution of Hadoop, and store all data as compressed LZO files.

This information and additional information for many other web companies is available on the Hadoop Wiki page from Leo (2012).

2.5.3. Hadoop Available Algorithms

For further research about Hadoop algorithms there is a good compilation on publications that explain how to implement algorithms with this tool on [2]. There is also a compilation of map-reduce patterns on [3] and finally if the reader is more interested on machine learning algorithms with Hadoop it might be useful to check the Mahout page on [4].

Hadoop Mahout's algorithms are implemented on top of Apache Hadoop using the map/reduce paradigm. Mahout's core libraries are optimized to allow also for good performance even for non-distributed algorithms i.e. pseudo-distributed installations of Hadoop.

Hadoop Mahout is appropriate for several use cases including recommendation mining for example in commercial applications, clustering tasks for example with sets of text documents and therefore grouping them into groups of topically related documents. For example Mahout can also be applied to classification by learning from existing categorized documents. Mahout then tries to find what documents of a specific category look like and assigns unlabelled documents to the predicted category. Mahout can also be applied to Frequent item set mining taking a set of item groups and identifying which individual items usually appear together. This has applications for example on commercial environments with product transactions lists.

[2] http://atbrox.com/2010/05/08/mapreduce-hadoop-algorithms-in-academic-papers-may-2010-update/
[3] http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/
[4] http://mahout.apache.org/

2.5.4. Hadoop Advantages and Disadvantages

This section presents Hadoop Map Reduce advantages and disadvantages. This is important because Map Reduce serves as basis for several used tools available to do data analysis nowadays. Some of these tools are introduced in Chapter 3 and used for metrics computations on section 3.4. Table 1 gives a summary:

Table 1: Advantages and Disadvantages – Hadoop Map Reduce

Tool	Hadoop MR
Advantages	• Ability to write MapReduce programs in Java, a language which even many non computer scientists can learn with sufficient capability to meet powerful data-processing needs • Ability to rapidly process large amounts of data in parallel • Can be deployed on large clusters of cheap commodity hardware as opposed to expensive, specialized parallel-processing hardware • Can be offered as an on-demand service, for example as part of Amazon's EC2 cluster computing service Washington (2011)
Disadvantages	• One-input two-phase data flow rigid, hard to adapt - Does not allow for stateful multiple-step processing of records • Procedural programming model requires (often repetitive) code for even the simplest operations (e.g., projection, filtering) • Map Reduce nature is not specially directed to implement code that presents iterations or iterative behavior • Opaque nature of the map and reduce functions impedes optimization from Zinn (2010)

2.5.5. Hadoop installation - Physical Architecture

For this thesis we use an HP machine with 12 cores (Intel(R) Core(TM)2 Duo CPU T7700 @ 2.40GHz) and 55GB RAM lend by FEUP (Faculty of Engineering University of Porto). The OS installed on FEUP machine is a CentOS 6 Linux distribution. This machine has a pseudo-distributed installation of Hadoop based on the web page by Noll (August 5, 2007).

The Hadoop installation procedures for a small test cluster setup are available in this document in APPENDIX A.

2.6. Map-Reduce

In brief, a map-reduce computation executes as follows and is essentially defined by the developed *map* and *reduce* functions as mentioned by Rajaraman *et al.* (2012):

1. Within the Map tasks scheduler each mapper is given one or more pieces of the data in the distributed file system. These Map tasks turn the chunk of data into a sequence of key-value pairs. The way key-value pairs are produced from the input data is determined by the code written by the user for the **Map function**.
2. The key-value pairs from each Map task are collected by a master controller and sorted by key. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same **Reduce task**.
3. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way. The manner of combination of values is determined by the code written by the user for the **Reduce function**.

The following figure translates the Map-Reduce process:

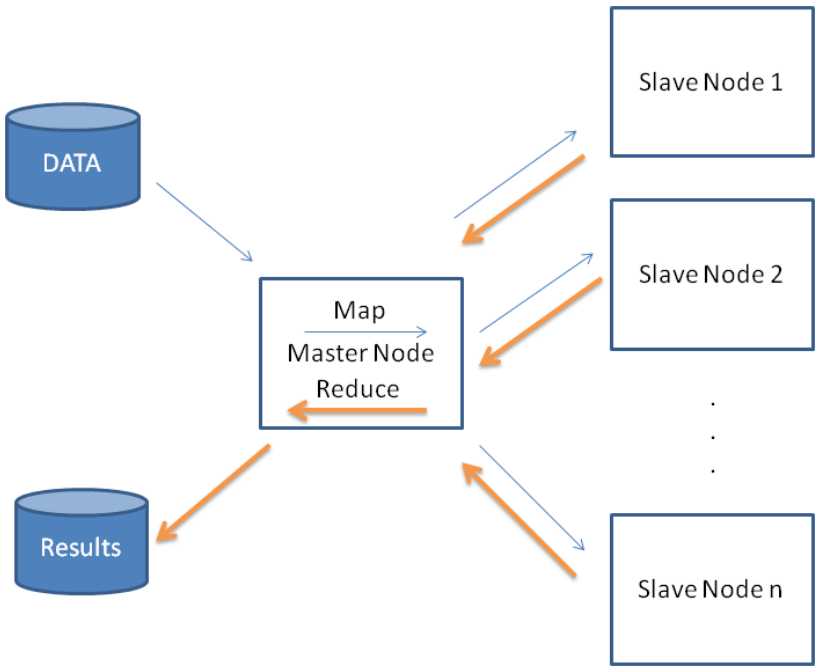


Figure 6: Schematic of Map-Reduce Computation

2.6.1. The Map processing

The role of the map-reduce user is to program/define *map* and *reduce* functions, where the *map* function outputs key/value tuples, which are processed by *reduce* functions to produce the final output. Map function is defined with a Key/value pair as input and that

represents some excerpt of the original files/file, for example a single document or document line. The *map* function produces zero or more Key/value pairs for that input but it can have also a filtering purpose when it outputs only if a certain condition is met.

2.6.2. The Reduce processing

The *reduce* function is called once per each Key outputted by *map* function and also as an input to reduce are all the values outputted by *map* function for some specific key.

Like the *map* function the *reduce* function can output from zero to many key/value pairs, in the end of the process the output can be written to DFS or a database for example.

2.6.3. The Shuffle and Sort Process

The shuffle and sort phases are responsible for determining the reducer that should receive the map output key/value pair (called partitioning); and ensuring that, for a given reducer, all its input keys are sorted.

Map outputs for the same key (such as “Yahoo“ in figure 7) go to the same reducer, and are then combined together to form a single input record for the reducer. Each reducer has all of its input keys sorted.

Figure 7 gives an example of the Shuffle & Sorting process used with Map-Reduce applications. This example is related to text mining documents for company’s news. The mapper splits each document line into distinct words, and outputs each word (the key) along with the word's originating filename (the value). MapReduce partitions the mapper output keys and ensures that the same reducer receives all output records containing the same key. MapReduce sorts all the map output keys for a single reducer, and calls a reducer once for each unique output key, along with a list of all the output values across all the reducers for each unique output key. The reducer collects all the filenames for each key, and outputs a single record, with the key and a comma-separated list of filenames.

The high-level algorithm for such a task would be like this:

```
mapper (filename, file-contents):
  for each keyword in file-contents:
    emit (keyword,filename)

reducer (keyword, values):
  for each keyword:
     for each values:
        add values to list-of-filenames
  emit (keyword, list-of-filenames)
```

Algorithm 1: High-level example of Text mining with Map/Reduce

In this example case we list Google, Yahoo, Microsoft and Apple and following previous algorithm the Shuffle & Sort would be like in Figure 7:

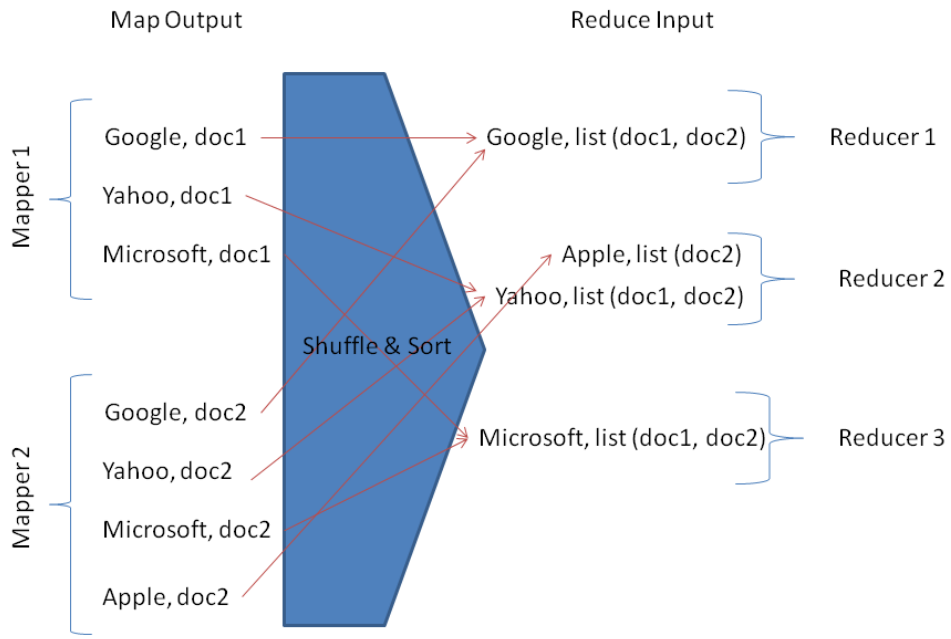


Figure 7: Schematic of Map-Reduce Shuffle & Sorting

2.7. Resumed evolution over recent times

This section ends this chapter resuming the recent evolution describing the milestones of large scale graph mining and analysis. The following figure illustrates this evolution and gives some insight on future developments of this subject. It is a recent subject of study and the development or use of parallel computing tools to approach big data

problems and specifically graph analysis fundamentally starts in the beginning of the 21st Century with the creation of Boost Graph library written in C++. Four years latter an evolution of the library appear in the form of Parallel Boost Graph Library also written in C++ and simultaneously the appearance of Hadoop written in JAVA which would became preponderant some years later. In the years between we also have seen the development of SNAP framework written in C and OpenMP (for multicore architectures) but finally Hadoop and the HDFS assumed to be the most used tool among the vast majority of graph analysis frameworks.

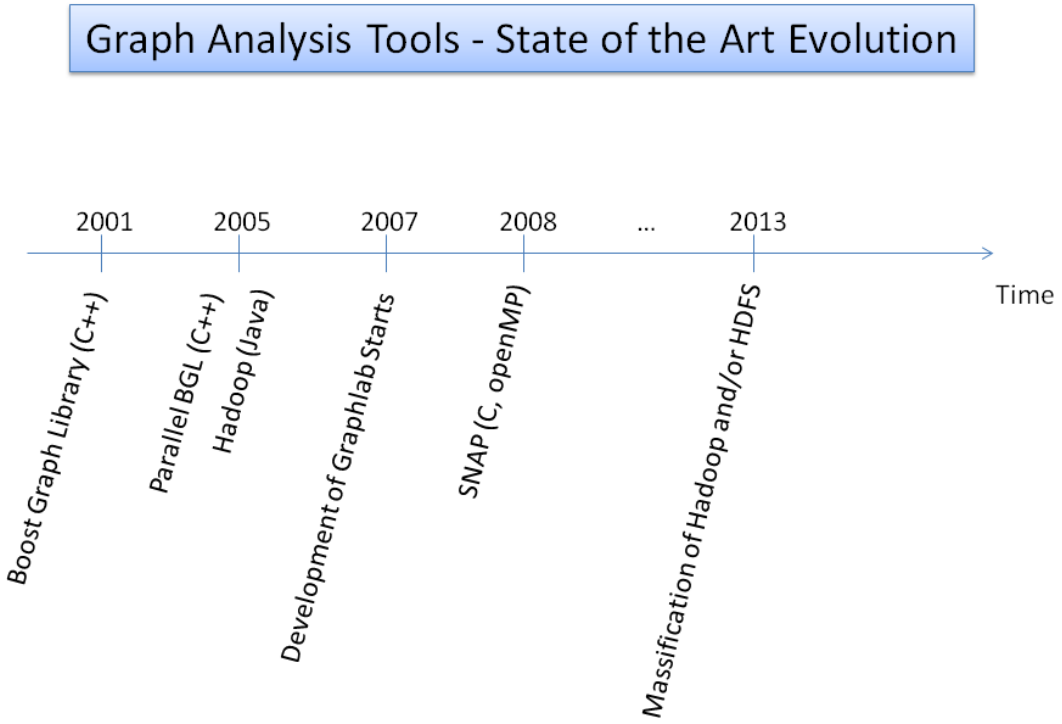


Figure 8: State of the Art – recent evolution

3. Graph Analysis Tools

Many of the tools used on this thesis development are parallel/distributed computing tools not necessarily developed to be used for graph analysis but generically developed to fulfill the need for tools to analyze big data on machines with many cores/clusters of computers.

3.1. Tools Introduction

Most graph tools use Hadoop or HDFS as its basis to work with clusters of computers and distributed data files. Tools like Apache Giraph, Pegasus, Map Reduce, Graphlab and others use it and depend on it for proper communication between nodes on a cluster. Another used tool, in fact not dependent of Hadoop is Snap C++ packages published by Stanford. The introduction to this tool and the experimental results obtained with it for graph analysis metrics are presented respectively in sections 3.1.4 and 3.4.8.

3.1.1. Pegasus

The first used tool that is based in Hadoop was Pegasus. Pegasus is an open-source, graph-mining system with massive scalability. It is fully written in JAVA language and it runs in parallel, distributed manner as mentioned in Kang (2012).

Pegasus provides several algorithms already implemented so the user can apply them directly to social networks and graphs (section 3.2). The details about Pegasus can be found on a paper by Kang and Tsourakakis (2009). The instructions for Pegasus installation procedures can be found also on this document (APPENDIX A).

3.1.2. Graphlab

Graphlab (2012), is a high-level graph-parallel abstraction that efficiently and intuitively expresses computational dependencies. Unlike Map-Reduce where computation is applied to independent records, computation in GraphLab is applied to dependent records which are stored as vertices in a large distributed data-graph. Computation in GraphLab is expressed as vertex-programs which are executed in

parallel on each vertex and can interact with neighboring vertices. In contrast to the more general message passing and actor models, GraphLab constrains the interaction of vertex-programs to the graph structure enabling a wide range of system optimizations. GraphLab programs interact by directly reading the state of neighboring vertices and by modifying the state of adjacent edges. In addition, vertex-programs can signal neighboring vertex-programs causing them to be rerun at some point in the future. The instructions for Graphlab installation procedures can be found on this document, APPENDIX A and the algorithms available for graph analysis are mentioned on section 3.2.

3.1.3. Giraph

Giraph implements a graph-processing framework that is launched as a typical Hadoop job to use existing Hadoop infrastructure. Giraph builds upon the graph-oriented nature of Pregel developed by Google from Malewicz *et al.* (2010) but additionally adds fault-tolerance to the coordinator process with the use of ZooKeeper as its centralized coordination service.

Giraph follows the bulk-synchronous parallel model relative to graphs where vertices can send messages to other vertices during a given super-step. Checkpoints are initiated by the Giraph infrastructure at user-defined intervals and are used for automatic application restarts when any worker in the application fails. Any worker in the application can act as the application coordinator and one will automatically take over if the current application coordinator fails as mentioned from Apache (2012). The instructions for Giraph installation procedures can be found on this document (APPENDIX A) and the algorithms available for graph analysis are mentioned on section 3.2.

3.1.4. Snap (Stanford Network Analysis Platform)

As cited on the project's webpage [5] Snap from Leskovec (2012) is a general purpose, high performance system for analysis and manipulation of large networks. The core SNAP library is written in C++ and optimized for maximum performance and compact graph representation. It easily scales to massive networks with hundreds of millions of

[5] http://snap.stanford.edu/snap/

nodes, and billions of edges. It efficiently manipulates large graphs, calculates structural properties, generates regular and random graphs, and supports attributes on nodes and edges. Besides scalability to large graphs, an additional strength of Snap is that nodes, edges and attributes in a graph or a network can be changed dynamically during the computation.

Snap was originally developed by Jure Leskovec in the course of his PhD studies. The first release was made available in Nov, 2009. Snap uses a general purpose STL (Standard Template Library) like library GLib developed at Jozef Stefan Institute. Snap and GLib are being actively developed and used in numerous academic and industrial projects. The instructions for Snap installation procedures can be found on this document, APPENDIX A and the algorithms available for graph analysis with this tool are mentioned on section 3.2.

3.2. Comparison of basic features of graph analysis tools

Almost all of the tools proposed in this document are introduced on previous chapters and include toolkits ready to be used immediately after install.

Pegasus, Graphlab, Snap and Giraph have several algorithms dedicated to networks analysis. Pegasus is exclusively dedicated to network analysis, Graphlab has several toolkits available but the *graph analytics toolkit* is the more appropriate for the subject of this thesis. Snap is, like other tools, dedicated to graphs analysis and presents a myriad of algorithms ready to use. Giraph is a tool still under heavy development and has an algorithm library with some few simple example algorithms as we will see on Table 2.

Table 2: Comparison of tools – Algorithms

Software	Pegasus	Graphlab	Giraph	Snap
Algorithms available from software install	• Degree • PageRank • Random Walk with Restart (RWR) • Radius • Connected Components	• approximate diameter • kcore • pagerank • connected component • simple coloring • directed triangle count • simple undirected triangle count • format convert • sssp • undirected triangle count	• Simple Shortest Path (available from [6]) • Simple In Degree Count • Simple Out Degree Count • Simple Page Rank • Connected Components	• cascades • centrality • cliques • community • concomp • forestfire • graphgen • graphhash • kcores • kronem • krongen • kronfit • maggen • magfit • motifs • ncpplot • netevol • netinf • netstat • mkdatasets • infopath
Parallel computing	YES	YES	YES	NO
Can user configure number of cores or machines?	YES	YES	YES	NO

On Table 2, among the toolkits/example algorithms available for each tool it is also exposed the capacity of these several tools to work on a parallel computing environment and also if the selected number of processor cores or processing machines is available to be specified from configuration. This information is important for further use of these tools scalability and if for example the numbers of computing nodes available on the user cluster vary.

The algorithms names in Table 2 are self explanatory in considerable amount but for Snap and Graphlab there are situations where the purpose of the algorithm might not be clear to the reader. This is a brief explanation on the acronyms in Table 2 and what they mean:

- For **Graphlab**:

SSSP: single source shortest path vertex program.

[6] https://cwiki.apache.org/confluence/display/GIRAPH/Shortest+Paths+Example

- For **SNAP,** from readme file on [7]**:**

cascades: Simulate a SI (susceptible-infected) model on a network and compute structural properties of cascades.

centrality: Node centrality measures (closeness, eigen, degree, betweenness, page rank, hubs and authorities).

cliques: Overlapping network community detection (Clique Percolation Method).

community: Network Community detection (Girvan-Newman and Clauset-Newman-Moore).

concomp: Manipulates connected components of a graph.

dynetinf: Implements stochastic algorithm for dynamic network inference from cascade data (more at http://snap.stanford.edu/proj/dynamic/).

forestfire: Forest Fire graph generator.

graphhen: Common graph generators (Small-world, Preferential Attachment, etc.).

graphhash: Graph hash table for counting frequencies of small graphs.

kcores: Computes the k-core decomposition of the network.

kronem: Estimates Kronecker graph parameter matrix using EM algorithm.

krongen: Kronecker graph generator.

kronfit: Estimates Kronecker graph parameter matrix.

maggen: Multiplicative Attribute Graph (MAG) generator.

magfit: Estimates MAG model parameter.

motifs: Counts the number of occurrence of every possible subgraph on K nodes in the network.

ncpplot: Computes Network Community Profile (NCP) plot.

netevol: Computes properties of an evolving network, like evolution of diameter, densification power law, degree distribution, etc.

netinf: Implements netinf algorithm for network inference from cascade data (more at http://snap.stanford.edu/netinf).

netstat: Computes statistical properties of a static network, like degree distribution, hop plot, clustering coefficient, distribution of sizes of connected components, spectral properties of graph adjacency matrix, etc.

MakeDatasets: creates datasets for the SNAP website. The code demonstrates how to load different kinds of networks in various network formats and how to compute

[7] https://github.com/snap-stanford/snap/blob/master/README.txt

various statistics of the network, like diameter, clustering coefficient, size of largest connected component, and similar.

3.3. Advantages and Disadvantages

This section resumes the advantages and disadvantages of the tools used for graph analysis in this thesis. The following table resumes the general opinion about the tools:

Table 3: Advantages and Disadvantages - Comparison of tools

Tool	Pegasus	Graphlab	Giraph	Snap
Advantages	• Similar positive points to Hadoop MR (please see section 2.5.4)	• Algorithms can be described in a node-centric way; same computation is repeatedly performed on every node. • Significant amounts of computations are performed on each node. • Can be used for any Graph as long as their sparse.	• Several advantages over Map Reduce: - it's a stateful computation - Disk is hit if/only for checkpoints - No sorting is necessary - Only messages hit the network as mentioned from Martella (2012)	• Optimized for Graph processing. • Written with C++ which is intrinsically considered a fast language
Disadvantages	• Similar negative points to Hadoop MR (please see section 2.5.4)	• Programmability: user must restructure his algorithm in a node centric way. • There is an overhead of runtime system when the amount of computation performed at each node is small. • Small world graphs: Graphlab lock scheme may suffer from frequent conflicts for such graphs.	• Still in a very immature phase of development • Lack of a complete offered algorithm library	• Not developed to take advantage of parallel or distributed processing of tasks • Some algorithms can be time consuming even for relatively small graphs due to the number of graph characteristics covered (eg. "centrality" algorithm)

3.4. Computing Metrics for Graph Analysis

This section presents an overview of the tools used for computing graph metrics. These tools are Graphlab, Pegasus, Hadoop Map/Reduce and Snap. There will be also a brief exposure of the results obtained with each tool. First we will start by explaining the origin and details of the data networks used for tools tests.

When it was necessary to use Hadoop and HDFS based tools there is a need for inputting the data files, edge or adjacency lists to HDFS and that task was done with the following *"put"* command:

```
$hadoop fs -put  <localsrc>  <dst>
```

where `<localsrc>` is the path of the local file that we want to send to HDFS and `<dst>` is the destination file name we want the source file to have in HDFS.

3.4.1. Case Studies

The experimental evaluation described in Chapter 3 uses several different datasets. One of the dataset represents the relationships between technological companies spread around the world and financial organizations. Another dataset is related to relationships between persons and companies also on the technological universe. Next section, we explain the datasets characteristics.

3.4.1.1. Characteristics of the original data

We used networks downloaded from [8] for this chapter containing computation of networks metrics. Therefore we are using Amazon's products network where network edges represent links of commonly co-purchased products (from now on designated by Network C) from Leskovec *et al.* (2005). We use also Youtube's online social network (from now on designated by Network D) and LiveJournal online social network (from now on designated by Network E) from Backstrom *et al.* (2006). These networks are available among others from Leskovec (2009).

As mentioned before the Networks A an B represent data that was downloaded from the CrunchBase website, a directory of technology companies. The Network A

[8] http://snap.stanford.edu/data/index.html

represents the connections between technological companies and financial organizations and Network B represents the connections between personalities and technological companies. For achieving both this networks we used the CrunchBase API from Thanedar (2012) that provides JSON representations of the data found on CrunchBase. The output of the items is JavaScript Object Notation, a lightweight format for data exchange. JSON is pure JavaScript, an alternative to XML. To handle this format is not necessary to use DOM or any specific framework.

The API currently supports three actions: "show", "seeks" and "list".

Example of original data Entity

For information about a specific entity in CrunchBase, we use a URL as follows:

```
http://api.crunchbase.com/v/1/<namespace>/<permalink>.js
```

The namespaces available are:

- company
- person
- financial-organization
- product
- service-provider

Example Company: Google

```
http://api.crunchbase.com/v/1/company/google.js?api_key=...
```

Example Investment Fund: Accel Partners

```
http://api.crunchbase.com/v/1/financial-organization/accel-partners.js?api_key=...
```

Example Person: Brad Fitzpatrick

```
http://api.crunchbase.com/v/1/person/brad-fitzpatrick.js?api_key=...
```

Please note that for using CrunchBaseAPI commands, we use an API Key previously obtained after registration on the Crunchbase website. If, for example, your API key is 1234 the previous command would be:

```
http://api.crunchbase.com/v/1/person/brad-fitzpatrick.js?api_key=1234
```

Entities List

For a list of all entities of a particular namespace on CrunchBase, we use a URL as follows:

```
http://api.crunchbase.com/v/1/<plural-namespace>
```

The plural namespace available are:

- companies
- people
- financial-organizations
- products
- service-providers

In this work we used these following namespaces: **companies, people,** and **financial organizations**.

Number of firms: 88.269

`http://api.crunchbase.com/v/1/companies.js?api_key=...`

Number of investment funds: 7.697

`http://api.crunchbase.com/v/1/financial-organizations.js?api_key=...`

Number of persons: 118.394

Therefore and for all the networks used in this chapter we have the following number of nodes and edges:

- Network A with 16.339 vertexes and 30.313 edges.
- Network B with 107.033 vertexes and 128.746 edges.
- Network C with 334.863 vertexes and 925.872 edges.
- Network D with 1.134.890 vertexes and 2.987.624 edges.
- Network E with 3.997.962 vertexes and 34.681.189 edges.

3.4.1.2. Data Preprocessing

To deal with extraction of the data for networks A and B, a Windows Application was used, it communicates with the site API. The final output was a directory with JSON files with all the items available for the selected entities.

After having extracted all items it was necessary to generate statements in order to export the items to a database and make the relationship between entities, for this task a Windows Application was used. We were using MySql DBMS initially but after several performance problems we chose SQL Server. Depending on the tool used for data

analysis it might be necessary to translate an edge list originally retrieved from the database to an adjacency list. This conversion from edge list to adjacency list was done with programming code made with R language. This code is available on APPENDIX B, page 84 among other code developed also for preprocessing of data.

3.4.2. Degree Measure with Pegasus

The following command was then used from Pegasus console to run algorithm:

```
PEGASUS> compute deg comp-finorg
Enter parameters: [in or out or inout] [#_of_reducers]: inout 2
```

where *comp-finorg* is the graph name already uploaded on HDFS (*"add"* command explained on users guide from Kang *et al.* (2010)). Pegasus asks if we want to retrieve in-degree or out-degree or if we want generic degree information. It also asks how many reducers we want to use and this number is dependent of number of node machines in the cluster and is calculated with the next assumption:

number of reducers = 2* number of machines

The results are then available on the HDFS directory pegasus/graphs/[GRAPH_NAME]/results/[ALGORITHM_NAME]. So, to obtain these results on pegasus/graphs/comp-finorg/results/deg we have to get them from HDFS, the following command was written on OS console:

```
$hadoop fs -get /user/110414015/pegasus/graphs/comp-finorg/results/deg /results
```

the results are then divided in two folders, one with the node degree count where we can see for each value of degree count the quantity of these occurrences in the graph. Here goes an example of output:

2	**3186**
4	1369
6	566
8	258
10	141
12	72
14	45

this results expose the existence in this network of 3186 nodes with node degree value of 2 i.e. two neighbors for each node in this group of nodes and this undirected graph.

The output for the node degree count expectedly outputs the node degree for each node in the graph, for example, the node with Id 2 has 30 neighbors:

2	**30**
4	224
6	59
8	13
10	48
12	113
14	12

3.4.3. Triangles with *Graph Analytics* Graphlab Toolkit

For the next experiences with the data and Graphlab's *Graph Analytics Toolkit* from Graphlab (2012) we followed the website relative to the algorithm available on [9].

The following command was used on graph analytics toolkit directory:

```
$ ./undirected_triangle_count --graph=/home/110414015/Relationships-Companies-
FinancialOrg.tsv --format=tsv
```

The raw output of this command was:

```
  This program counts the exact number of triangles in the provided graph.

  INFO:     mpi_tools.hpp(init:63): MPI Support was not compiled.
  TCP Communication layer constructed.
  INFO:      metrics_server(launch_metric_server:219): Metrics server now listening on
http://hpcgrid-centos6:8090
  INFO:       distributed_graph.hpp(load_from_posixfs:1823): Loading graph from file:
/home/110414015/Relationships-Companies-FinancialOrg.tsv
  INFO:     distributed_ingress_base.hpp(finalize:166): Finalizing Graph...
  INFO:     distributed_ingress_base.hpp(exchange_global_info:493): Graph info:
           nverts: 16339
           nedges: 30313
           nreplicas: 16339
           replication factor: 1
  Number of vertices: 16339
  Number of edges:    30313

  Counting Triangles...
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 16339
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
  INFO:     synchronous_engine.hpp(start:1373): 1 iterations completed.
  Updates: 16339
```

[9] http://docs.graphlab.org/graph_analytics.html#graph_analytics_triangle_undirected

```
Counted in 0.047622 seconds
70 Triangles
Metrics server stopping.
```

The following command was used on graph analytics toolkit directory and for the Network B studied:

```
$./undirected_triangle_count --graph=/home/110414015/Relationships-Persons-Companies.tsv --format=tsv
```

The raw output of this command was:

```
This program counts the exact number of triangles in the provided graph.

INFO:     mpi_tools.hpp(init:63): MPI Support was not compiled.
TCP Communication layer constructed.
INFO:      metrics_server(launch_metric_server:219): Metrics server now listening on http://hpcgrid-centos6:8090
INFO:        distributed_graph.hpp(load_from_posixfs:1823): Loading graph from file: /home/110414015/Relationships-Persons-Companies.tsv
INFO:     distributed_ingress_base.hpp(finalize:166): Finalizing Graph...
INFO:     distributed_ingress_base.hpp(exchange_global_info:493): Graph info:
          nverts: 107033
          nedges: 128746
          nreplicas: 107033
          replication factor: 1
Number of vertices: 107033
Number of edges:    128746
Counting Triangles...
INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every 5 seconds.
INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 107033
INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
INFO:     synchronous_engine.hpp(start:1373): 1 iterations completed.
Updates: 107033
Counted in 0.103243 seconds
20 Triangles
Metrics server stopping.
```

with these results we can conclude that both networks present low number of triangles and therefore have low density and moreover, triangle detection gained recently much practical importance since they are central in so-called complex network analysis. First, they are involved in the computation of one of the main statistical property used to describe large graphs met in practice and that is the clustering coefficient of the node as mentioned from Latapy (2008). The expected clustering coefficient for both graphs in study in this section is expected to be low due the low number of triangles presented on them.

3.4.4. Connected Components with *Graph Analytics* Graphlab Toolkit

For the next experiences with the data and Graphlab we followed the website relative to the algorithm available on [10].

The following command was used on graph analytics toolkit directory and for the Network A studied:

```
$./connected_component --graph=/home/110414015/Relationships-Companies-FinancialOrg.tsv --format=tsv
```

The raw output of this command was:

```
Connected Component

INFO:     mpi_tools.hpp(init:63): MPI Support was not compiled.
TCP Communication layer constructed.
Loading graph in format: tsv
INFO:     distributed_graph.hpp(load_from_posixfs:1823): Loading graph from file: /home/110414015/Relationships-Companies-FinancialOrg.tsv
INFO:     distributed_ingress_base.hpp(finalize:166): Finalizing Graph...
INFO:     distributed_ingress_base.hpp(exchange_global_info:493): Graph info:
          nverts: 16339
          nedges: 30313
          nreplicas: 16339
          replication factor: 1
INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every 5 seconds.
INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 16339
INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
INFO:     synchronous_engine.hpp(start:1373): 14 iterations completed.
Updates: 63671
graph calculation time is 0 sec
RESULT:
size    count
2       556
3       113
4       36
5       14
6       6
7       3
8       6
10      1
18      1
```

The following command was used on graph analytics toolkit directory and for the Network B studied:

[10] http://docs.graphlab.org/graph_analytics.html#graph_analytics_connected_component

```
$ ./connected_component --graph=/home/110414015/Relationships-Persons-Companies.tsv --format=tsv
```

The raw output of this command was:

```
Connected Component

INFO:     mpi_tools.hpp(init:63): MPI Support was not compiled.
TCP Communication layer constructed.
Loading graph in format: tsv

INFO:        distributed_graph.hpp(load_from_posixfs:1823): Loading graph from file: /home/110414015/Relationships-Persons-Companies.tsv
INFO:     distributed_ingress_base.hpp(finalize:166): Finalizing Graph...
INFO:     distributed_ingress_base.hpp(exchange_global_info:493): Graph info:
          nverts: 107033
          nedges: 128746
          nreplicas: 107033
          replication factor: 1
INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every 5 seconds.
INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 107033
INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
INFO:     synchronous_engine.hpp(start:1373): 21 iterations completed.
Updates: 801608
graph calculation time is 1 sec
RESULT:
size    count
2       1086
3       573
4       306
5       150
6       108
7       61
8       42
9       22
10      22
11      11
12      2
13      6
14      4
15      3
16      3
18      1
19      1
21      1
23      1
98886   1
```

with these results we can conclude that both networks present one main weakly connected component composed by almost all nodes from the network evidencing that both networks A and B have almost all nodes inter-connected by some defined path

between them. This represents that both networks have few nodes isolated from the rest of the network. The study of connected components in social network analysis has several applications including a key role in the chemistry investigations for organic compounds derived from Tutte Theorem as cited on [11].

3.4.5. KCore decomposition with *Graph Analytics* Graphlab Toolkit

For the next experiences with the data and Graphlab we followed the website relative to the algorithm available on [12].

The following command was used on graph analytics toolkit directory and for the Network A studied:

```
$./kcore --graph=/home/110414015/Relationships-Companies-FinancialOrg.tsv --format=tsv
```

The raw output of this command was:

```
Computes a k-core decomposition of a graph.


INFO:     mpi_tools.hpp(init:63): MPI Support was not compiled.
TCP Communication layer constructed.
INFO:        distributed_graph.hpp(load_from_posixfs:1823): Loading graph from file:
/home/110414015/Relationships-Companies-FinancialOrg.tsv
INFO:     distributed_ingress_base.hpp(finalize:166): Finalizing Graph...
INFO:     distributed_ingress_base.hpp(exchange_global_info:493): Graph info:
          nverts: 16339
          nedges: 30313
          nreplicas: 16339
          replication factor: 1
Number of vertices: 16339
Number of edges:    30313
INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 0
INFO:     synchronous_engine.hpp(start:1373): 0 iterations completed.
Updates: 0
K=0:  #V = 16339   #E = 30313
INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 0
INFO:     synchronous_engine.hpp(start:1373): 0 iterations completed.
Updates: 0
K=1:  #V = 16339   #E = 30313
```

[11] http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29
[12] http://docs.graphlab.org/graph_analytics.html#graph_analytics_kcore

```
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 6685
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
  INFO:     synchronous_engine.hpp(start:1373): 8 iterations completed.
  Updates: 10212
  K=2:  #V = 8645   #E = 23354
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 2860
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
  INFO:     synchronous_engine.hpp(start:1373): 10 iterations completed.
  Updates: 16232
  K=3:  #V = 5037   #E = 16613
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 1683
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
  INFO:     synchronous_engine.hpp(start:1373): 13 iterations completed.
  Updates: 20965
  K=4:  #V = 2578   #E = 9684
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 929
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators

  INFO:     synchronous_engine.hpp(start:1373): 35 iterations completed.
  Updates: 25433
  K=5:  #V = 645   #E = 2479
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 273
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
  INFO:     synchronous_engine.hpp(start:1373): 6 iterations completed.
  Updates: 26318
```

The following command was used on graph analytics toolkit directory and for the Network B studied:

```
  $ ./kcore --graph=/home/110414015/Relationships-Persons-Companies.tsv --format=tsv
```

The raw output of this command was:

```
  Computes a k-core decomposition of a graph.

  INFO:     mpi_tools.hpp(init:63): MPI Support was not compiled.
  TCP Communication layer constructed.
  INFO:        distributed_graph.hpp(load_from_posixfs:1823): Loading graph from file:
/home/110414015/Relationships-Persons-Companies.tsv
```

```
  INFO:     distributed_ingress_base.hpp(finalize:166): Finalizing Graph...
  INFO:     distributed_ingress_base.hpp(exchange_global_info:493): Graph info:
           nverts: 107033
           nedges: 128746
           nreplicas: 107033
           replication factor: 1
  Number of vertices: 107033
  Number of edges:    128746
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 0
  INFO:     synchronous_engine.hpp(start:1373): 0 iterations completed.
  Updates: 0
  K=0:  #V = 107033   #E = 128746
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 0
  INFO:     synchronous_engine.hpp(start:1373): 0 iterations completed.
  Updates: 0
  K=1:  #V = 107033   #E = 128746
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 52238
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
  INFO:     synchronous_engine.hpp(start:1373): 11 iterations completed.
  Updates: 89208

  K=2:  #V = 40460   #E = 64567
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 22127
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
  INFO:     synchronous_engine.hpp(start:1373): 20 iterations completed.
  Updates: 138797
  K=3:  #V = 2437   #E = 5532
  INFO:     synchronous_engine.hpp(start:1248): Iteration counter will only output every
5 seconds.
  INFO:     synchronous_engine.hpp(start:1263): 0: Starting iteration: 0
  INFO:     synchronous_engine.hpp(start:1312):   Active vertices: 1278
  INFO:     synchronous_engine.hpp(start:1361):    Running Aggregators
  INFO:     synchronous_engine.hpp(start:1373): 9 iterations completed.
  Updates: 141918
```

A k-core of a graph G is a maximal connected subgraph of G in which all vertices have degree at least k. Equivalently, it is one of the connected components of the subgraph of G formed by repeatedly deleting all vertices of degree less than k. If a non-empty k-core exists, then, clearly, G has degeneracy at least k, and the degeneracy of G is the largest k for which G has a k-core.

The concept of a k-core was introduced to study the clustering structure of social networks from Seidman (1983) and to describe the evolution of random graphs from Luczak (1991), it has also been applied in bioinformatics by Bader and Hogue (2003) and network visualization by Alvarez-Hamelin *et al.* (2005).

3.4.6. Measuring 'Friends of Friends' with Hadoop Map-Reduce

The algorithm to be explored by us was "friends of friends" which is basically an algorithm for searching the friends of friends which have more friends in common with the iteration origin node.

The book material was downloaded with the following commands available on the book from Holmes (2012):

```
$ git clone git://github.com/alexholmes/hadoop-book.git
```

Then we built the code:

```
$ cd hadoop-book
$ mvn package
```

The results were obtained first by putting (to HDFS) the prepared file with the data of the networks in the form of an adjacency list (a .txt file prepared with R code and as previously documented)

```
$ hadoop fs -put adjency_list.txt .
$ bin/run.sh com.manning.hip.ch7.friendsofafriend.Main \adjency_list.txt calc-output sort-output
```

For the Network B, the one with relations between persons and companies the following similar commands were used:

```
$ hadoop fs -put adjency_list_persons.txt .
$ bin/run.sh com.manning.hip.ch7.friendsofafriend.Main \adjency_list_persons.txt calc-output sort-output
```

The result files of these commands were retrieved from HDFS with the get command similar to previous calls of this command on this document. The results are not made available on this document because of space reasons. Here is a small sample of the results achieved with this algorithm and for the network of relations between companies and financial organizations:

```
10077    8507:2,17745:1,11077:1,24814:1,85008:1,24937:1,2569:1,2599:1,15721:1,26176:1
1008     73285:1,1469:1,35600:1,247:1,213:1,58475:1,51474:1,7522:1,1991:1,1010:1
1009     14833:1,35600:1,2050:1,11160:1,184:1,2474:1,7313:1,142:1,247:1,73285:1
10099    7613:1,7466:1,109:1,2474:1,12:1,357:1,27658:1,15:1,1135:1,26915:1
101      36:8,15:3,7293:3,26:2,7434:2,513:2,53:2,87:2,6:1,6319:1
1010     7490:4,1875:2,607:2,247:1,35509:1,100:1,1:1,57:1,1008:1,1009:1
1011     939:3,15:3,54:2,7279:2,7377:2,51820:1,5136:1,507:1,5:1,483:1
10116    55775:2,2870:2,39005:2,18924:2,72017:2,26185:1,25966:1,25866:1,25794:1,24768:1
1012     10996:1,1523:1
10120    35585:1,3192:1,31255:1,30752:1,30748:1,30663:1,27754:1,26857:1,26789:1,2665:1
10121    13289:1,11617:1,671:1,18956:1
10127    81082:1,9417:1,813:1,7542:1,7541:1,7227:1,27141:1,24898:1,15759:1,12134:1
10128    59502:1,5822:1,5739:1,56896:1,5344:1,4746:1,4410:1,43497:1,43350:1,4314:1
```

From the previous results sample and as an example the institution with ID 101 has a good chance of connecting with the one with ID 36 because both have 8 connections in common although they are not directly interconnected in the input network.

With these results we can conclude that this algorithm is of good application in the commercial data networks where the results could serve as basis for a recommender system. In the case of our network A and B the hypothetical recommender would recommend connections between companies and financial organizations and for network B it would recommend connections between persons and companies regarding consulting services for example.

3.4.7. Centrality Measures with Snap

Several algorithms were used in Snap software, we will write about the results on the next pages:

The command centrality was used on Snap's /examples/centrality directory and for the Network A studied, the usage of the command is as outputted in Snap software:

```
usage: centrality
   -i:Input un/directed graph (default:'../as20graph.txt')
   -o:Output file (default:'node_centrality.tab')
```

The command is the following:

```
./centrality       -i:/home/110414015/Relationships-Companies-FinancialOrg.txt       -
o:centrality.tab
```

The output from Snap is very extensive so we present just a small sample example:

```
#Network: /home/110414015/Relationships-Companies-FinancialOrg.txt
#Nodes: 16339 Edges: 30313
```

#Node Id	Degree	Closeness	Betweennes	EigenVector	Network Constraint	Clustering Coefficient	PageRank	HubScore	Authority Score
3	80.00	0.233747	1139257.192383	0.000461	0.016776	0.000633	0.001181	0.000094	0.029831
843	14.00	0.193071	164648.965528	0.000028	0.083915	0.000000	0.000798	0.000000	0.000021
844	16.00	0.207691	287289.050309	0.000061	0.071393	0.000000	0.000907	0.000000	0.001772
9	33.00	0.213657	310964.724490	0.000223	0.039056	0.000000	0.000361	0.000008	0.015517
1352	9.00	0.181062	96242.573356	0.000015	0.118590	0.000000	0.000539	0.000000	0.000147

The command centrality was used on Snap /examples/centrality directory and also for the Network B studied, the usage of the command is the same as used before. The results are of similar format also.

With these results we can, among other conclusions, inspect the role each node plays on the network regarding its connectivity. Centrality measures allows us to find the principal actors in a network i.e. the nodes that present strong centrality or betweeness centrality are nodes of greater importance has they are central in the path of connection between many nodes of the network.

3.4.8. Communities with Snap

The command community was used on Snap's /examples/community directory and for the Network A studied. The usage of the command is as outputted in Snap software:

```
usage: community
   -i:Input graph (undirected graph) (default:'graph.txt')
   -o:Output file (default:'communities.txt')
   -a:Algorithm: 1:Girvan-Newman, 2:Clauset-Newman-Moore (default:2)
```

The command is the following:

```
./community -i:/home/110414015/adjency_list.txt -o:adjency_list_communities.txt
```

The output from Snap is very extensive so we present just a small sample example:

```
# Input: /home/110414015/adjency_list.txt
# Nodes: 16339    Edges: 14417
# Algoritm: Cluset-Newman-Moore
# Modularity: 0.994151
# Communities: 1943
#NId  CommunityId
3     0
843   0
922   0
1036  0
1268  0
7485  0
1371  0
1744  0
1829  0
2570  0
4346  0
.
.
.
```

The command community was used on Snap's /examples/community directory and also for the Network B studied, the usage of the command is the same as used before and the results are similar in format but for other graph subject of study.

With these results we can conclude that all nodes belong to the same community with Id 0 and for the output chunk listed.

The community detection algorithms have large application in several areas including Psychology, Anthropology, Business and communications, Ecology among many others as mentioned in [13] .

3.4.9. Connected Components with Apache Giraph

The algorithm to be explored by us with Apache Giraph was "Connected Components" which is basically an algorithm available in the examples section of Giraph.

The results were obtained using a specially prepared JSON kind of file graph input and by putting (to HDFS) the prepared files (.txt files prepared with R code available in APPENDIX B on page 84) with the data of the used networks.

For a Network C, the one with Amazon data, the following command was used on the Giraph binary folder:

[13] http://en.wikipedia.org/wiki/Community

```
$hadoop jar target/giraph-0.2-SNAPSHOT-for-hadoop-0.20.203.0-jar-with-dependencies.jar
org.apache.giraph.GiraphRunner org.apache.giraph.examples.ConnectedComponentsVertex -if
org.apache.giraph.io.JsonBase64VertexInputFormat -ip Amazon-Giraph.txt -of
org.apache.giraph.io.JsonBase64VertexOutputFormat -op CC-Amazon -w 1
```

Some of these parameters are self explaining but we must now address the `-w` parameter. This parameter defines the total number of workers available to handle graph partitions. Since for this particular test we are running a pseudo-distributed cluster (single host), it is safe to limit this to one. In a fully-distributed cluster, we would want multiple workers spread out across different physical hosts.

Unfortunately, at the time of closing this thesis document it was not possible to output results of this computation. The process of discovering the reason why the Giraph/Map Reduce task did not complete has not yet finished and we do not have a conclusion about the reason it was impossible to achieve results. Although the installation was tested and apparently it worked, the reason it failed might be related to many reasons inclusively to our test environment and physical architecture. As Giraph is an important tool in big graph analysis and because we feel it might fulfill some of the readers needs and because it might work with the user resources we felt it would be logical to refer it in this document. We did write about the installation procedure, its features, compared its advantages and disadvantages with other tools but we will not continue its exploration further in this thesis.

3.5. Processing Time for Graph Analysis

To give the reader a notion of the processing time that takes we run the previous mentioned algorithms with some networks with different sizes. Networks used in this section were already described in section 3.4.1, resuming the networks we used for these tests section have the following characteristics as number of nodes and number of edges:

- Network A with 16.339 vertexes and 30.313 edges.
- Network B with 107.033 vertexes and 128.746 edges.
- Network C with 334.863 vertexes and 925.872 edges.
- Network D with 1.134.890 vertexes and 2.987.624 edges.
- Network E with 3.997.962 vertexes and 34.681.189 edges.

Have a look on the following table:

Table 4: Processing Time (in seconds)

	Hadoop MR "Friends of Friends"	**Pegasus** Degree Measures	**Graphlab** Triangles Counting	**Snap** Centrality Measures
Network **A**	16,040s	5,380s	0,048s	374s (06m14s)
Network **B**	23,880s	7,070s	0,103s	17400s(4h50m)
Network **C**	138,980s	11,050s	0,305s	-[14]
Network **D**	430,420s	23,330s	1,211s	-[15]
Network **E**	1516,257s	35,680s	16,211s	-[16]

Snap expectedly presents processing times of higher magnitude especially due to the great amount of centrality measures available as results outputted for each network and because although is an optimized tool for graph analysis it doesn't belong to the parallel processing group of tools. Generally the computation is of relatively high speed for all the algorithms and on parallel processing tools even with networks with millions of nodes. For these previous results we do some graphics where the evolution of processing time with higher number of nodes is visible for the networks:

[14] * Value too high
[15] * Value too high
[16] * Value too high

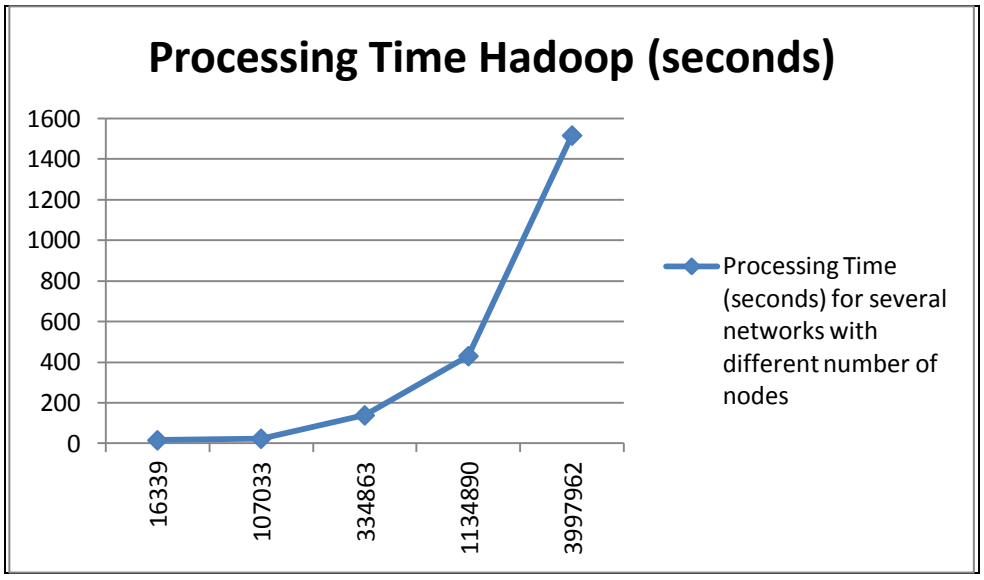


Figure 9: Processing time variation for Hadoop Map-Reduce FoF algorithm

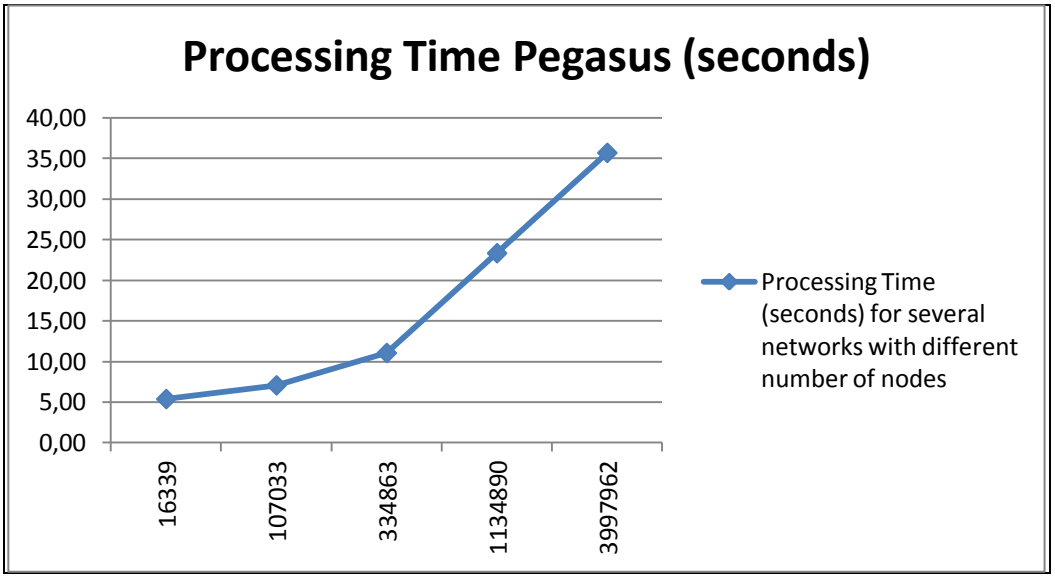


Figure 10: Processing time variation for Pegasus Degree algorithm

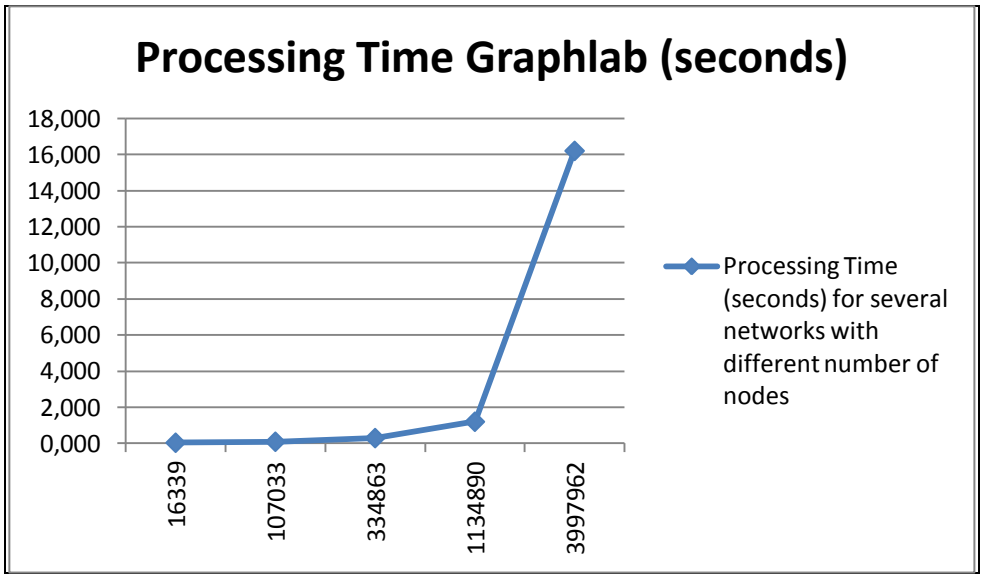


Figure 11: Processing time variation for Graphlab Triangles detection algorithm

The previous figures give some insight on the processing time consumption variation with node degree but we cannot assure that they are good comparison for tools efficiency because the algorithms are different in complexity; the implementation of the tools is variable in terms of architecture or language and the only fixed assumption is that the machine where the computation took place is the same. Our intention is to give a visual insight on the variation of processing time for each tool.

4. Communities Detection and Similarity Ranking algorithms

This Chapter gives an introduction on communities and what it represents to study communities of graphs/social networks. It discusses some algorithm implementations and the issues to take care of on this type of community detection algorithm. Then we introduce the SimRank algorithm and the tools used for development of both algorithms. Finally we present experimental results and a brief comparison with other similar algorithms regarding results and processing time.

4.1. Case Studies

The experiments in this chapter 4 use several datasets. Three of the datasets represent the relationships between technological companies spread around the world and financial organizations but are truncated so that the number of nodes and edges approximately doubles from one network to the next network. These three undirected networks will be used for similarity ranking algorithm comparisons regarding processing time on section 4.6.2. Resuming, this three truncated networks will be throughout this chapter and from now on designated by Network F, G and H and have the following characteristics considering the number of nodes and edges:

- Network F with 471 vertexes and 250 edges.
- Network G with 892 vertexes and 500 edges.
- Network H with 1.659 vertexes and 999 edges.

We used other networks for the community detection algorithms, to compare their results regarding modularity results and processing time. For comparison of modularity results (4.5.2) we used three undirected networks downloaded from [17] and compiled by Newman (2013) for this task, these were the Zachary's Karate Club, Dolphin Social Network and the American Colleague Football. The characteristics of these networks regarding number of nodes and edges are the following:

- Zachary's Karate Club with 34 vertexes and 78 edges.
- Dolphin Social Network with 62 vertexes and 159 edges.

[17] http://www-personal.umich.edu/~mejn/netdata/

- American Colleague Football with 115 vertexes and 615 edges.

For comparison of the community detection algorithms and regarding processing time (4.5.3) consuming we used the undirected networks A, B and C previously used for computing metrics with available graph analysis tools, refreshing the reader's memory their characteristics are the following:

- Network A with 16.339 vertexes and 30.313 edges.
- Network B with 107.033 vertexes and 128.746 edges.
- Network C with 334.863 vertexes and 925.872 edges.

4.2. Introduction to Community Detection

In a social network a community represents individuals that form a group distinguishable by its properties or characteristics. In other words when we say we encountered a community it might be for example a group of friends, family, work colleagues or other group of individuals with same characteristics and label inside the context of a network.

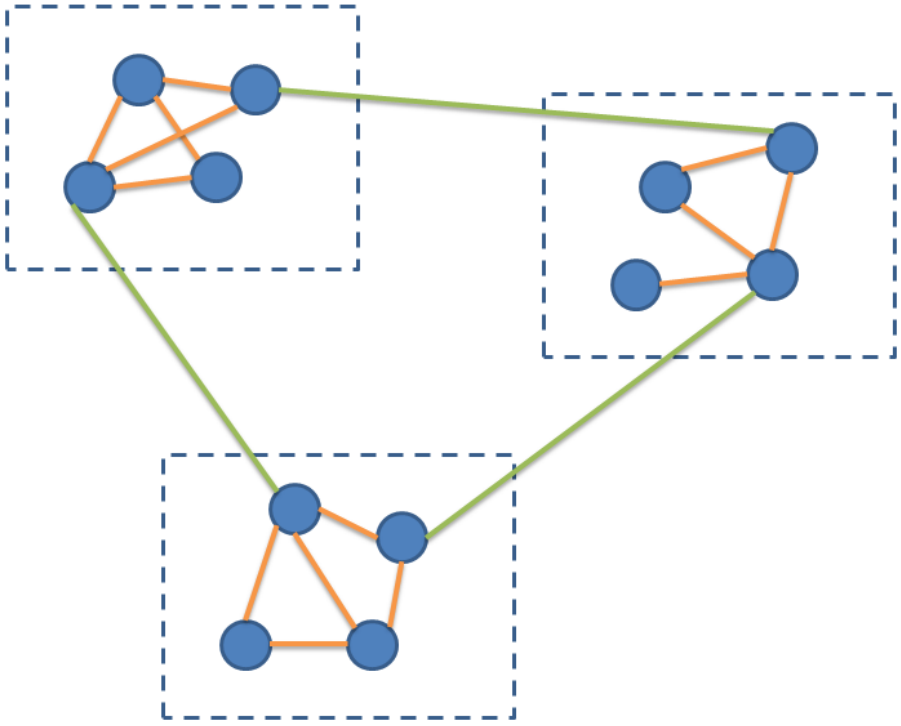


Figure 12: Simple Graph with 3 communities surrounded with dashed squares.

Detection of communities on a network has many applications, for example clients that have the same interests and are geographically near each other might be beneficiary of the implementation of mirror servers for faster services on the World Wide Web. The identification of retail clients with similar interests in products enables the retailer to give better recommendation services and therefore augment the probability of rising profits and service quality. On telecommunications and computer networks community structures of nodes can help to improve compactness of routing tables maintaining efficient choice of communication paths.

Regarding community structure several areas give much importance if the node lives inside a community or on the boundaries of the community. On the first case the node might be important as a control and stability function within the community and in the second case the node might have functions of information exchange between communities. This seems to have high importance for example in social an metabolic networks as mentioned from Fortunato (2010).

4.2.1. Community Detection Algorithms

Community detection in graphs has been generally defined but multiple methods of estimating quality of the detection exist. The majority of current works on community detection relies on improving the modularity value Newman (2006). Modularity can therefore be used to compare different approaches to community detection. There is a good compilation of approaches to communities detection in Fortunato (2010) which resumes saying that the majority of techniques can be divided into two different approaches: agglomerative and divisive.

Community detection is known to be a NP-complete problem. Community detection can be related to graph partitioning and there are good parallel algorithms for graph partitioning but for community detection it is a usual problem that relies on parallelism achievable from sequential algorithms. The top-down approach (divisive approach) or bottom-up approach (agglomerative approach) have inherent sequential flow with possibility of being parallelized on a higher amount on the first stages than the later stages.

Community detection algorithms usually show bad reliance with parallel graph partitioning algorithms and although they show scalability, because of the high

computational overhead of community detection algorithms one cannot usually apply such algorithms to networks of hundreds of millions of nodes or edges. Thus, an efficient and high quality algorithm (modularity) for community detection is hard to achieve and a challenging problem as mentioned by Soman and Narang (2011).

4.3. Similarity Ranking Algorithm

SimRank proposed by Jeh and Widom (2002) has become a measure to compare the similarity between two nodes using network structure. Although SimRank is applicable to a wide range of areas such as social networks, citation networks, link prediction and others, it suffers from heavy computational complexity and space requirements. The basic recursive intuition behind SimRank approach is "two objects are similar if they are referenced by similar objects." As the base case, it is considered that an object is maximally similar to itself, to which we can assign a similarity score of 1.

The similarity between objects *a* and *b* can be designated by *s(a, b)* ϵ [0, 1]. The authors of SimRank wrote a recursive equation for *s(a, b)*. If *a = b* then *s(a, b)* is defined to be 1 as told before. Otherwise,

$$s(a,b) = \frac{C}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} s(I_i(a), I_j(b)) \qquad (4.1)$$

where *C* is a constant between 0 and 1. A slight technicality here is that either *a* or *b* may not have any in-neighbors. Since we have no way to infer any similarity between *a* and *b* in this case, we should set *s(a, b)* = 0, so we define the summation in equation (6.1) to be 0 when *I(a)* = ø; or *I(b)* = ø;

One SimRank equation of the form (5.1) is written for each (ordered) pair of objects *a* and *b*, resulting in a set of n^2 SimRank equations for a graph of size *n*. Let us defer discussion of the constant *C* for now. Equation (5.1) says that to compute *s(a, b)*, we iterate over all in-neighbor pairs ($I_i(a)$, $I_j(b)$) of (a, b), and sum up the similarity $s(I_i(a), I_j(b))$ of these pairs. Then we divide by the total number of in-neighbor pairs, $|I(a)||I(b)|$, to normalize. That is, the similarity between *a* and *b* is the average similarity between in-neighbors of *a* and in-neighbors of *b*. From equation (5.1), it is easy to see that SimRank scores are symmetric, i.e., *s(a, b) = s(b, a)*.

We must also explain the purpose of the constant *C*, which according to the authors of the algorithm can be thought of either as a confidence level or a decay factor. Considering a simple example scenario where person *x* references both persons *c* and *d* as connections in a network, so we conclude some similarity between *c* and *d*. The similarity of *x* with itself is 1, but we probably do not want to conclude that $s(c, d) = s(x, x) = 1$. Rather, we let $s(c, d) = C \cdot s(x, x)$, meaning that we are less confident about the similarity between *c* and *d* than we are between *x* and itself.

4.4. Green-Marl Language

For the purpose of development of both algorithms we used Green-Marl to explore the fact that it is a DSL (domain-specific language) designed specifically for graph analysis algorithms. Users of Green-Marl can describe their graph algorithm using high-level graph constructs which expose the inherent parallelism in the algorithm. A compiler for Green-Marl can exploit this high-level information by applying a series of high-level optimizations and parallelizing the algorithm, and finally producing a parallel implementation of the given algorithm. The Green-Marl compiler final output is an implementation written in a general-purpose language, e.g. C++. Green-Marl specific contributions are as follows from Hong *et al.* (2012):

- Green-Marl, a DSL in which a user can describe a graph analysis algorithm in a intuitive way. This DSL captures the high-level semantics of the algorithm as well as its inherent parallelism.
- The Green-Marl compiler which applies a set of optimizations and parallelization enabled by the high-level semantic information of the DSL and produces an optimized parallel implementation targeted at commodity SMP machines.
- An interdisciplinary DSL approach to solving computational problems that combines graph theory, compilers, parallel programming and computer architecture.

Green-Marl is a tool developed by a Stanford team and it was made available recently. It allows the export of code reusable on other tools like Giraph for example. Leveraging these exportation characteristics we opted to use the C++ and OpenMP Green-Marl

output. We follow the installation procedure available on [18] to install Green-Marl on the hardware; it is a simple and direct process and we have no issues or difficulties to report.

4.4.1. What does Green-Marl offer from start?

Green-Marl offers several algorithms right after install. Some of these algorithms are translatable within Green-Marl to C++ code with OpenMP, therefore directed to multiprocessor computational environments and/or directed to Apache Giraph for cluster computational environments based on Hadoop Map Reduce. The following table resumes algorithms available and compatibility with mentioned tools:

Table 5: Green-Marl Algorithms

Green-Marl Software Algorithms	Brief Description	OpenMP C++ compatible	Giraph/GPS compatible
avg_teen_count	Computes the average teen count of a node	YES	YES
bc	Computes the betweenness centrality value for the graph	YES	NO
bc_random	Computes an estimation for the betweenness centrality value for the graph	YES	YES
communities	Computes the different communities in a graph	YES	NO
kosaraju	Finds strongly connected components using Kosaraju's Algorithm	YES	NO
pagerank	Computes the pagerank value for every node in the graph	YES	YES
potential-friends	Computes a set of potential friends for every node using triangle closing	YES	NO
sssp	Computes the distance of every node from one destination node according to the shortest path	YES	YES
sssp_path	Computes the shortest paths from one destination node to every other node in the graph and returns the shortest path to a specific node.	YES	NO
triangle_counting	Computes the number of closed triangles in the graph	YES	NO

4.5. Communities Detection algorithm with Green Marl

For communities detection implementation with Green-Marl we followed the paper from Soman and Narang (2011). The pseudo code for this algorithm is also available on the mentioned paper and is as follows:

[18] https://github.com/stanford-ppl/Green-Marl

Input: Graph *(V)*
Output: community of each node
foreach *Edge e(i, j)* **do**
    Find weight of *e(i, j) = w(i, j)*
end
foreach *Node n* **do**
    community(n)=n
end
foreach *Node n* **do**
    Find Maximum weighted edge in adjacency list;
    Store weight in *weight(n)*
end
G'=ϕ;
foreach *Node n* **do**
    foreach *edge e(n,v)* **do**
        if *weight(v)=weight(n)* **then**
        Add edge (v,n) to G'
        end
    end
end
Find connected components in G';
foreach *Node n* **do**
    community(n)=smallest label in component containing n in G'
end
while *All nodes are not stably labeled* **do**
    foreach *Node n* **do**
        $community'_n = argmax\Sigma_{i' \in N_n} (community_{i'}).\ w(i', n).W(L_{i'})$
    end
Exchange community and community';
End

Algorithm 2: The weighted label propagation algorithm

Although we have followed the paper algorithm there were some alterations we did which represented ending in not replicating the exact results of the algorithm but obtaining better modularity results for some test networks, this process is described in detail throughout the 4.5.1 section. This original algorithm has essentially 4 main phases that will be from now on declared sequentially as phases A, B, C and D and will be described in this chapter:

A. Weight Assignment & Propagation Function

B. Core edge detection

C. Epidemic spread Control

D. Overlapping Community extraction

A. Weight Assignment & Propagation Function

For label propagation, the algorithm tries to generate a community structure assigning weights to edges and determining how the labels propagate through the network. Edge

weights that implicitly represent accurate topological structure of the inherent communities in the network are desirable. As such prior knowledge of the inherent communities is not available; it is considered that the weight of an edge represents a measure of the importance of that edge to the nodes at the endpoints of that edge. In case of an undirected graph, each edge is replaced by two directed edges.

The weight of a directed edge, *e = (i, j)* (from vertex *i* to vertex *j*), is defined as the ratio between the number of triangles that the edge participates in and the total number of triangles the node *i* participates in. For an edge *e = (i, j)*, let edge *e* represent the highest weighted edge in the locality of *i*, then *i* has a higher chance of being assigned the same label as *j*, as compared to any other label in the vicinity. The directed edges with large weights correspond to connections that have a stronger importance to a node. Also, the edges with low weights represent weak relations, hence the chance of both nodes being in the same community is lower. Therefore, weight of an edge *e = (i, j)* is given by:

$$w_t(i,j) = z_{(i,j)} / \Sigma_{(i,k)}(z_{(i,k)}) \; ; k \in N(i) \qquad (4.2)$$

where, $z_{(i,j)}$, represents the number of triangles with edge *(i, j)* as one of the edges in the triangle.

In case of a weighted graph given as input, the authors suggest the product of the given weight and topological loops based weight mentioned above. Thus, if an edge *e = (i, j)* has weight given by the user as (i, j), then the weight of the edge considered for the label propagation algorithm is:

$$w(i,j) = w_u(i,j) * w_t(i,j) \qquad (4.3)$$

The propagation function to transfer labels from one node to another is then defined as:

$$L(i) = argmax\Sigma_{j \in Ni}\, s(L_j) \qquad (4.4)$$

where, N_i is the set of neighboring vertices of vertex i; $s(L_j)$ is the total weight for the label $L(j)$ in the neighborhood of vertex i.

B. Core edge detection

For a given weighted graph, for each node *i* there exists node *j** such that for node *i*, edge *(i, j*)* has the maximum weight in its neighborhood. There will exist node pairs (v_1, v_2) such that v_1 is paired to v_2 using the maximum edge weight criterion and also conversely, v_2 is paired to v_1 using the maximum edge weight criterion. One can see that using the propagation function defined by the equation (4.4), the labels on two such nodes within a pair can oscillate without ever converging. The oscillatory behavior weakens community detection, as meaningful communities are not formed. This will lead to low modularity output as well as higher number of iterations in the algorithm. Such node pairs forms a local maxima and have the tendency to form the cores of communities. This oscillation problem needs to be addressed meaningfully. Labeling such local maxima pairs with the same label will improve the qualitative performance of the algorithm as well as the overall running time. Hence, the authors propose to find such pairs before the label propagation iterations, and the same label is given to both the nodes in each pair. An extension of this issue is the presence of multiple overlapping pairs, where a single node can form such pairs with multiple nodes. Such overlapping nodes represent local communities in the graph. Hence, such pairs should be part of the same community. In the author's algorithm, it is first found the connected components over such overlapping pairs, and assigned the same label to all the nodes within each component.

On the pseudo code previously written the lines 13 to 24 represent the core edge detection and also the measures to avoid oscillation that prevents converging. It essentially uses one auxiliary generated graph G' that features manipulation of the nodes connections for the nodes originally present in the input graph. This graph G' is then used to apply Kosaraju Connected Components detection algorithm from Sharir (1981) to label the nodes with the initial discovered communities. These initial communities will then be propagated until the final communities labels for every node is discovered but first we will explain the reader other phases of the original algorithm.

C. Epidemic spread Control

Label propagation algorithm has a natural global minima when all the nodes in the graph have the same label. This is caused by a large community dominating over all the other communities. Though, the presence of weak edges between communities can reduce the epidemic spread to a large extent, in graphs with relatively low variation in edge density, the algorithm can still be susceptible to epidemic spread. To tackle epidemic spread, the authors present on the paper two methods that work at node level and as well as use statistics of the spread of the labels in the graph.

The technique proposed by the authors (and also used by us on the programming task) of improving the epidemic resistance is to control the size of a community. We assign a weight to each label based on the total degree of the nodes that have that label. Thus, the weight of a label is given by:

$$\mathrm{W_l}(L_i) = 1 - d_c/2M \qquad (4.5)$$

where, L_c is the label of community c; d_c is sum of the degrees of all nodes inside the community, c; and $2M$ is the total number of edges in the graph. The new propagation function becomes:

$$(i) = argmax\Sigma_{j\in Ni}[\mathrm{s}(L_j) * \mathrm{W_l}(L_j)] \qquad (4.6)$$

where, N_i is the set of neighboring vertices of vertex i; $s(L_j)$ is the total weight for the label $L(j)$ in the neighborhood of vertex i. As the size of a community (number of nodes with same label) increases, the weight of that label decreases. Thus, the ability of a label to propagate reduces with its size. The weight attached with each label thus acts as a global objective function and helps in controlling the size of the communities.

D. Overlapping Community extraction

For this task of extraction of overlapping communities we choose to test the variation of 3 sequential iterations of the code and in the case of having communities labels not converging i.e. the community label changes continuously between two distinct labels, from step 1 to 2 but on step 3 it changes again to the same label of step 1 then the algorithm stops iterating.

4.5.1. Development details and variations of the original algorithm

We started the development of the algorithm respecting phase A. which consisted on the edge weight finding as described in the last section. Following the paper was not enough to get some small test networks examples with the right results. On the next figure the reader can see the network that led to a small alteration of the algorithm.

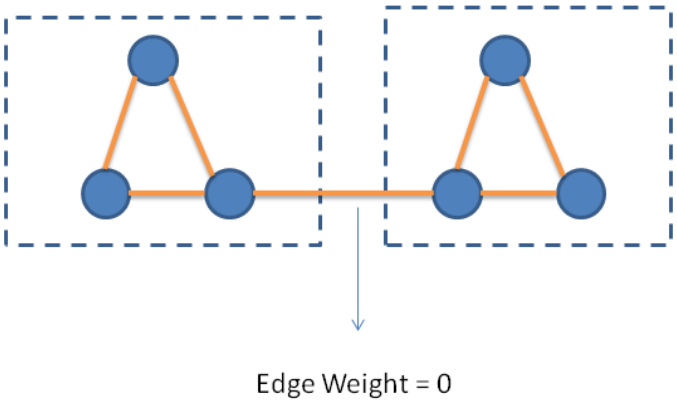


Figure 13: Network used in the development of the algorithm phase A.

Intuition says that there are two different communities and at an initial point in developing the algorithm resulted in the propagation of the same community through all the nodes ending in a graph with only one community as result. At this phase and with test networks also used in the followed paper from Soman and Narang (2011) the results were similar but we added a small condition in the code that determined the propagation of label to be possible only if the edge value was different from 0. In this case the edge weight is 0 because the number of triangles the edge participates in is 0. This small alteration implied that the auxiliary graph G' used in the process did not have a edge connection between the connected nodes that belong to the two different triangles (note that both nodes have the same associated weight) and therefore the connected components obtained for G' would be not one but two in this particular test graph exposed in the previous figure, resulting in better modularity results for this graph and more coherent results also on other tested networks by the authors on the followed paper. An example of better modularity results is the Karate club undirected network from Girvan and Newman (2002), please see the next section where a comparison of our version of the algorithm with other algorithms is made.

4.5.2. Modularity Results - Comparison with other algorithms

For the community detection algorithm comparison we used two other algorithms and the Snap sequential tool. The algorithms selected were the available from Snap, Girvan-Newman algorithm from Girvan and Newman (2002) and the Clauset-Newman-Moore algorithm from Clauset *et al.* (2004). The networks used for these comparisons were some known small networks already described in chapter 4.1.

The modularity results for the algorithms with small test networks are visible on the table 6 and the comparison of processing time is available on the next section.

Table 6: Modularity Comparison for Community Detection Algorithms

	Girvan – Newman Algorithm with Snap	Clauset-Newman-Moore Algorithm with Snap	Developed Algorithm with GM
Zachary's Karate Club	0.401	0.381	**0.436**
Dolphin Social Network	**0.519**	0.515	0.333
American College Football	**0.599**	0.549	0.339

The results obtained from the developed algorithm as we can see from previous table and for the test networks are significant modularity for every use case i.e. the value for this metric is above 0.3 and this a significant division in community structure for the algorithm as mentioned from Clauset *et al.* (2004). The modularity value is indeed superior to other algorithms for the Zachary's Karate Club network for example. The Girvan-Newman algorithm presents results generally superior to Clauset-Newman-Moore but as we will see in the next section has a much slower running time and therefore might be inadequate for larger networks.

4.5.3. Processing Time Results - Comparison with other algorithms

For the developed community detection algorithm and for its comparison regarding consumed processing time we also used the two algorithms already mentioned in the previous section and the sequential tool Snap. The undirected networks selected for this

comparison were the networks already mentioned before, the network A, network B and network C:

Table 7: Processing Time comparison for Community Detection Algorithms

	Girvan – Newman Algorithm with Snap	Clauset-Newman-Moore Algorithm with Snap	Developed Algorithm with GM
Network **A**	288 (hours)	6s	**4s**
Network **B**	300+ (hours)	**53s**	133s
Network **C**	400+ (hours)	*[19]	**45659s**

It is visible from the previous table that Girvan-Newman is an algorithm that has much higher processing time consumption than the other algorithms.

It is also to be noticed that Clauset-Newman-Moore is a very fast algorithm and for Network B presents faster computing than the algorithm developed with Green-Marl. The reader must notice Clauset-Newman-Moore achieves this using just one single core. However the reader must also notice that this algorithm has a high consuming rate of RAM and for Network C the amount of memory use was around 20.7GB when eventually failed with *segmentation fault (core dumped)* error after some few hours of computation. This occurrence made impossible to conclude the computation.

Our version of the community detection algorithm concluded the computation for Network C within approximately 12 hours (45k seconds) and with a modularity of 0.34. Although the value of modularity is significant the number of communities detected is sensibly lower than the number of communities considered being ground-truth for this particular network. The number of communities detected was 27864 and the ground-truth communities mentioned on Leskovec (2009) is around 150000.

Finally and as mentioned already the Girvan-Newman is a considerable time consuming algorithm but on the other hand it has very low RAM memory consumption presenting values of 8MB for Network A, 39MB for Network B and 143MB for Network C. The RAM values consumed by our version of Soman and Narang (2011) are similar to Girvan-Newman's algorithm and therefore considerably very low.

[19] Failed with *segmentation fault (core dumped)* error

4.6. SimRank algorithm with Green Marl

For the SimRank implementation with Green-Marl we follow the paper from Jeh and Widom (2002). The pseudo code for this algorithm is as follows:

```
1    Input: Graph (V)
2    Output: Similarity Rank for every pair (u,v) in the network
3    While Any similarity value did not converge do
4    similarity_old()=similarity_new()
5    foreach Node u do
6            foreach Node v do
7                    foreach u in-neighbor do
8                            foreach v in-neighbor do
9                            similarity(u,v) = similarity(u,v) + similarity_old(u in-neighbor, v in-
neighbor)
10                           end
11                   end
12           similarity_new(u,v) = (C * similarity(u,v))/ (u numInNbrs)*(v numInNbrs))
13           end
14   end
15   End
```

Algorithm 3: The SimRank algorithm

The code written in Green-Marl language for this pseudo-algorithm is available on APPENDIX B starting from page 94. The output is a matrix with the similarity between nodes in the network. Added care was taken to create the empty matrices on the heap to avoid memory issues like memory segmentation faults with larger networks. Being an algorithm with $O(n^2)$ time complexity where n is the number of nodes in the graph, it is a good choice to develop it in distributed computing environments. Leveraging the advantages of multicore hardware lower processing time for similar networks can be achieved. On the 4.6.2 section we write a small comparison between single core processing with R code and the multicore Green-Marl code (translated to C++ and OpenMP) developed by us but first we will explain the development details for this algorithm particularly discussing memory estimations.

4.6.1. Development details – Memory use estimation

The algorithm developed (available in section 5 of APPENDIX B starting from page 94) depends of the creation of two similarity matrices, one with the current iteration results and one with the previous iteration results. Since we are considering a float

number for similarity between any pair of nodes in the graph we can say that the maximum memory that will be used by the program will be approximately given by:

$$\text{MEM MAX} = 4\text{bytes}*2*(\text{number of graph nodes})^2 \qquad (4.7)$$

In expression (4.7) the space occupied in memory for a float number is 4bytes and as we have the creation of two matrices (bi-dimensional arrays) with the size of the number of nodes for each array, therefore the number of nodes is squared for our estimation of memory used by the program. As an example, for a graph with 40000 (40k) nodes, the estimated memory use will be around 12 GB.

Due to the considerable use of memory for this algorithm and limitations of resources available we will be using networks with smaller sizes to take conclusions, and therefore compare the algorithm behavior in sequential single core machines and this multicore version we developed with a machine with 12 cores.

4.6.2. Simrank Single Core Vs Multicore

In this section of Chapter 4 we will give an example output retrieved from the developed algorithm with some small networks. This choice for small networks was done to make it possible to compare the processing time with sequential processing on the same machine and with R software. Starting with the test edge list with 7 nodes and 7 undirected edges on the next figure:

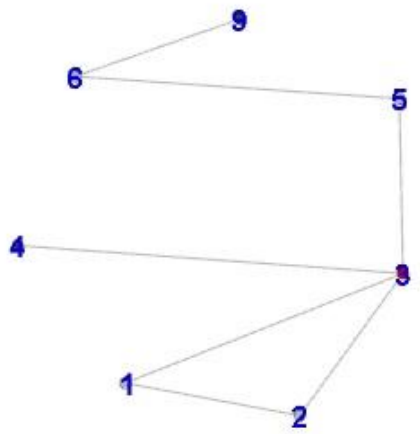


Figure 14: Test Network used in the development of the similarity algorithm.

and using constant C = 0.6 as mentioned from Lizorkin *et al.* (2008) and k = 50 iterations the output matrix is the following:

	1	2	3	4	6	5	9
1	1.000000	0.235798	0.168164	0.350434	0.051199	0.209529	0.068624
2	0.235798	1.000000	0.168164	0.350434	0.051199	0.209529	0.068624
3	0.168164	0.168164	1.000000	0.066980	0.177689	0.043468	0.019956
4	0.350434	0.350434	0.066980	1.000000	0.018981	0.353290	0.106580
6	0.051199	0.051199	0.177689	0.018981	1.000000	0.012027	0.005073
5	0.209529	0.209529	0.043468	0.353290	0.012027	1.000000	0.353290
9	0.068624	0.068624	0.019956	0.106580	0.005073	0.353290	1.000000

For the similarity ranking processing time comparisons the following networks were also used (see previous section 4.1 for data details):

- Network F with 471 vertexes and 250 edges.
- Network G with 892 vertexes and 500 edges.
- Network H with 1.659 vertexes and 999 edges.
- Network A with 16.339 vertexes and 30.313 edges.

Table 8: Processing Time for Similarity Algorithms (in seconds)

	Parallel Simrank with Green-Marl	Sequential Simrank with R
Network **F**	480s	**25s**
Network **G**	1073s	**491s**
Network **H**	**2716s**	7560s
Network **A**	**26851s**	1022000+ s

with these results we can conclude that the processing time is not clearly smaller for every comparison available in table 8 and on the same machine. The sequential operation consumes 1 core with 100% use and the parallel execution of the parallel implementation of the algorithm consumed in average 1095% of the processing resources of the machine with 12 cores which is the same to say that it used

approximately 11 cores for the task. The results obtained allow us to conclude that the processing time for sequential execution is lower for the Networks F and G and higher for the larger networks H and A which is the larger network of this group of networks. For these last two networks the parallel execution ended in much less amount of time than the time value for sequential execution of the algorithm. This is a reasonably expected result and as the size of the network rises it is expected that the difference from parallel execution to sequential execution to be bigger and bigger due to our algorithm complexity and due to diluting of importance for overhead generated by communications between processors and memory accesses on the OpenMP parallel implementation. This causes the parallel algorithm to run slower than the sequential version with smaller networks. This is true for networks with approximately less than 1000 nodes.

The following figure represents the variation of parallel and sequential processing time in respect to the number of nodes of the networks F, G and H:

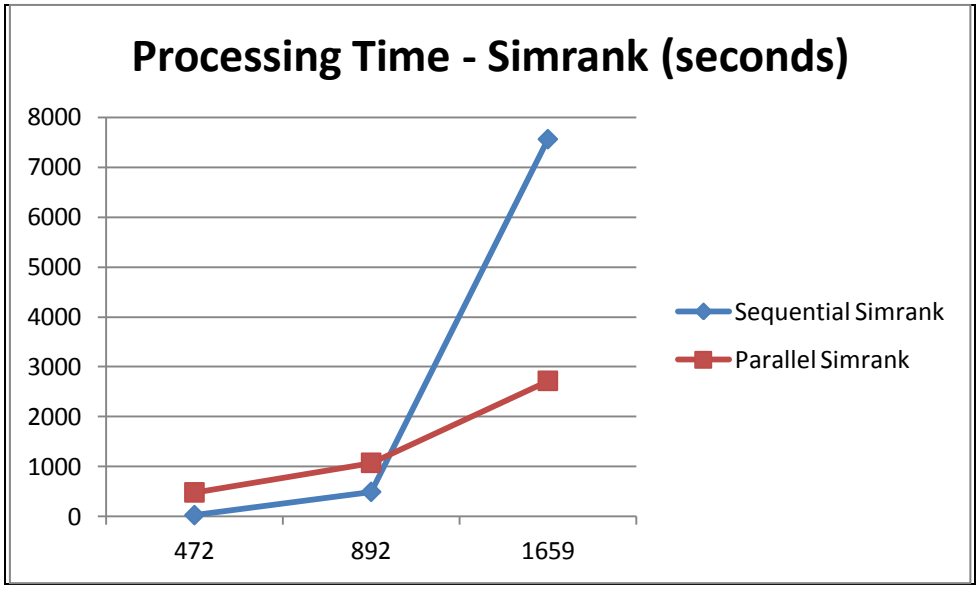


Figure 15: Processing time for parallel/sequential execution of the similarity algorithm.

It is visible from the previous figures that the parallel execution of the algorithm - for the networks doubling the number of nodes - appears to have a less pronounced rise of time with node doubling. This makes the parallel algorithm expected behavior to be much faster than the sequential execution with larger networks.

The previous figures give some insight of the processing time consumption and its variation with node degree but because the algorithms are different in the implementation language the only fixed assumption is that the machine where the computing took place is the same and therefore the processor speed is the same. Our intention is to give a visual insight on the variation of processing time for both algorithm implementations and with previous assumptions.

5. Conclusions

We have been witnessing a very big proliferation of software tools aimed at the analysis of large graphs during the last few years. One of thesis goals was to expose which tools to look for when dealing with big graphs studies. The amount of algorithms and tools available make it reasonable to achieve fast processing of general big data problems and also specifically with graphs studies. We started the thesis with the state of the art regarding parallel computing for graph analysis and its recent evolution, then we made the introduction to the tools used nowadays for distributed graph analysis and then we wrote some practical examples of computing algorithms that leverage the tools potential for big scale graphs studies. We hope to have gathered and provided sound information about the tools with this document, we think by reading this work the reader is incentivized for further exploration of the tools available to use with his/her big graph data problems.

Other thesis goal was to prove the utility and diversity of the tools and algorithms available for graph studies and also prove the relatively easy way to achieve a good approach to large scale social network analysis. We think that this goal was also achieved and the use of an SDL tool like Green-Marl and the help of C++ programming made possible the development of two different algorithms that in a way served to prove that we have huge gains in efficiency and scalability with the use of the parallel computing paradigm.

The novelty of some tools and subjects approached throughout this Thesis make the future even more promising and compelling. There is a good chance that the tools mentioned in this document might evolve to have even more intuitive user interfaces, new and more complex algorithms and also better use of hardware resources. The future is also time to develop higher expectations and therefore we also have some thoughts about future work we would like to write in one of the next sections but first we will resume what we have learned with the writing of this document.

5.1. Lessons Learned

Writing this document was conclusive about the importance parallel paradigm has in solving big data problems. The particular problems addressed with big graphs were approached in this document with the right tools discovered after heavy research. Some

tools evolved over time and eventually were substituted by others since new tools and technologies are constantly appearing nowadays. This positive growing situation we learned to be related to the increasing importance given to social network analysis in modern world science. Many areas of research make use of social network analysis on their daily tasks.

We learned also that the increasing number of SDLs for big graph analysis make the choice of languages for programming tasks essentially between two generic languages, C++ and Java. Both are viable and the choice the user does will dictate the compatible tool he will use for the specific task.

The programming tasks we have done clearly exposed some characteristics we were not aware before for multicore OpenMP programming. The Green-Marl language was also a great and previously unknown tool in the set of tools available. As a very recent SDL for graph analysis with all the expected immaturity nevertheless proved to be a very intuitive approach and also with a very effective use of the parallel computation paradigm therefore successfully reducing its implicit programming complexity.

5.2. Future Work

Considering potential evolution of this work we think the following comments in this section might be of reader's interest.

Due to the novelty of some of the tools available nowadays and also given the fact that some are very recent, further exploration in the future might be useful and important. For example the Apache Giraph tool revealed to be somewhat difficult to use due to the fast and less mature developing process. The tool evolved in a way that sometimes was not very clear to us and frequently we and other users felt the support documentation available did not accompany on these modifications. It was frequent to have console commands working in one week and not in the next week, specifically following a version update or other kind of changes the same command would not work anymore.

Also for future work we are planning to do an update to the developed community detection algorithm. We would like to update it in a way that it features the possibility of support weighted edge lists as inputs.

As future work we would also like to update the similarity ranking algorithm and develop it in a way that it would output a file (output is currently a matrix with

similarity results) presented as a list of nodes and a top-k set of the most similar nodes to each node in the network.

Other development that we might be interested in doing would be to develop Clauset-Newman-Moore algorithm with Green-Marl. We would like to do it leveraging features of the language like the translation to Java/Giraph language/framework. That would make possible and interesting to observe the behavior of such a fast algorithm regarding its memory use in a computing cluster environment. The use of HDFS and a cluster with good RAM resources would make it a very powerful algorithm for community detection even with very large scale social networks with billions or even trillions of nodes.

References

Alvarez-Hamelin, J. I., L. Dall'Asta, A. Barrat and A. Vespignani (2005). "k-core decomposition: a tool for the visualization of large scale networks". CoRR.

Anderson, W., P. Briggs and C. S. Hellberg (2003). "Early experiences with scientific programs on the Cray MTA-2". In Proc. SC'03.

Apache. (2012). "Apache Giraph." from http://incubator.apache.org/giraph/.

Backstrom, L., D. Huttenlocher, J. M. Kleinberg and X. Lan (2006). "Group Formation in Large Social Networks: Membership, Growth, and Evolution". KDD, page 44-54. ACM.

Bader, D. A. and K. Madduri (2008). "SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks". IPDPS, page 1-12. IEEE.

Bader, G. D. and C. W. Hogue (2003). "An automated method for finding molecular complexes in large protein interaction networks". BMC Bioinformatics.

Berry, J. W., B. Hendrickson and S. Kahan (2006). "Graph software development and performance on the MTA-2 and Eldorado". In Cray User's Group.

Borchers, B. and D. Crawford (1993). "MPI: A Message Passing Interface". SC, page 878-883. IEEE Computer Society / ACM.

Clauset, A., M. E. J. Newman and C. Moore (2004). "Finding community structure in very large networks". Physical review E 70(6):066111.

Dagum, L. and R. Menon (1998). "Openmp: An industry-standard api for shared-memory programming.", IEEE Computational Science and Engineering.

El-Ghazawi, T. A., W. W. Carlson and J. M. Draper (2003). "UPC Language Specification, 1.1 edition".

Fortunato, S. (2010). "Community detection in graphs". Physics Reports 486(3–5):75 - 174, Physics Reports.

Girvan, M. and M. E. J. Newman (2002). "Community structure in social and biological networks". Proceedings of the National Academy of Sciences 99(12):7821-7826.

Graphlab. (2012). "Graph Analytics Toolkit." 2012, from http://graphlab.org/toolkits/graph-analytics/.

Graphlab. (2012). "Graphlab The Abstraction." 2012, from http://graphlab.org/home/abstraction/.

Gregor, D., N. Edmonds and B. Barrett (2005). "The Parallel Boost Graph Library", The Trustees of Indiana University.

Holmes, A. (2012). Hadoop In Practice, Manning.

Hong, S., H. Chafi, E. Sedlar and K. Olukotun (2012). "Green-Marl: A DSL for Easy and Efficient Graph Analysis". ASPLOS, page 349-362. ACM.

Jeh, G. and J. Widom (2002). "SimRank: A Measure of Structural-Context Similarity". Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, page 538--543. New York, NY, USA, ACM.

Kang, U. (2012). "PEGASUS: Peta-Scale Graph Mining System." Retrieved 11-2012, from http://www.cs.cmu.edu/~pegasus/.

Kang, U., D. H. Chau and C. Faloutsos (2010). "PEGASUS User's Guide", Carnegie Mellon University.

Kang, U. and C. E. Tsourakakis (2009). "PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations". Proceeding ICDM '09 Proceedings of the 2009 Ninth IEEE International Conference on Data Mining.

Latapy, M. (2008). "Main-memory Triangle Computations for Very Large (Sparse (Power-Law)) Graphs". Theor. Comput. Sci. 407(1-3):458-473.

Leo, S. (2012, 2012-12-20 16:00:03). "Hadoop Wiki." Retrieved 16-01-2013, 2013, from http://wiki.apache.org/hadoop/PoweredBy.

Leskovec, J. (2009). "Stanford Large Network Dataset Collection." Retrieved 25-02-2013, 2013, from http://snap.stanford.edu/data/index.html.

Leskovec, J. (2012). "Stanford Network Analysis Platform." Retrieved 12-2012, 2012, from http://snap.stanford.edu/snap/.

Leskovec, J., L. A. Adamic and B. A. Huberman (2005). "The Dynamics of Viral Marketing". CoRR.

Lizorkin, D., P. Velikhov, M. Grinev and D. Turdakov (2008). "Accuracy Estimate and Optimization Techniques for SimRank Computation". VLDB J. 19(1):45-66.

Luczak, T. (1991). "On the size and connectivity of the k-core of the random graph".

Lumsdaine, A., D. Gregor, B. Hendrickson and J. Berry (2007). "Challenges in Parallel Graph Processing". Parallel Processing Letters 17(1):5-20, World Scientific Publishing Company.

Malewicz, G., M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski (2010). "Pregel: A System for Large-Scale Graph Processing". Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, page 135--146. New York, NY, USA, ACM.

Martella, C. (2012). "Apache Giraph: Distributed Graph Processing in the Cloud". FOSDEM 2012, Graph Processing Room.

Mazza, G. (2012, 2012-11-30 19:22:49). "FrontPage - Hadoop Wiki." Retrieved 11-2012, from http://wiki.apache.org/lucene-hadoop/.

Newman, M. (2006). "Modularity and community structure in networks". Proceedings of the National Academy of Sciences of the United States of America 103(23):8577--82.

Newman, M. (2013). "Network Data." Retrieved 04-2013, from http://www-personal.umich.edu/~mejn/netdata/.

Noll, M. G. (August 5, 2007, June 29, 2012). "Running Hadoop On Ubuntu Linux (Single-Node Cluster)." Retrieved 06-11-2012, from http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/.

Owens, J. R. (2013). "Hadoop Real-World Solutions Cookbook", PACKT Publishing.

Rajaraman, A., J. Leskovec and J. D. Ullman (2012). "Mining of Massive Datasets". Cambridge University Press, Cambridge.

Science, C. M. U.-S. o. C. (2012). "Getting Started with PEGASUS." Retrieved 11-2012, from http://www.cs.cmu.edu/~pegasus/getting%20started.htm.

Seidman, S. B. (1983). "Network structure and minimum degree". Social Networks 5(3):269 - 287.

Sharir, M. (1981). "A strong-connectivity algorithm and its applications in data flow analysis", NEW YORK UNIVERSITY.

Society, I. C. (1990). "System Application Program Interface (API) [C Language]. Information technology—Portable Operating System Interface (POSIX)", IEEE Press, Piscataway,NJ.

Soman, J. and A. Narang (2011). "Fast Community Detection AlgorithmWith GPUs and Multicore Architectures". 2011 IEEE International Parallel & Distributed Processing Symposium.

Thanedar, V. (2012). "API Documentation." Retrieved 04-2012, 2012, from http://developer.crunchbase.com/docs.

Valiant, L. G. (1990). "A bridging model for parallel computation", Commun. ACM 33(8):103-111.

Washington, U. o. (2011). "What is Hadoop?" Retrieved 05-03-2013, 2013, from http://escience.washington.edu/get-help-now/what-hadoop.

Zinn, D. (2010). "MapReduce". Amazon Cloud Computing Workshop in conjunction to the Bioinformatics Next Generation Sequencing Data Analysis Workshop.

Appendix A

1. Hadoop Installation - Implementation Procedures

Due to Hadoop's high importance for the treatment of the thesis data we will write the steps we follow for the example installation of a small cluster with a set of 3 virtual machines with Linux Ubuntu Server OS.

Cluster Architecture

1rst machine, Master:
Name: master
IP: 192.168.0.1/24

2nd machine, Slave:
Name: slave
IP: 192.168.0.2/24

3rd machine, Slave 2:
Name: slave2
IP: 192.168.0.3/24

1. Install **master** in *Single-Node* mode
2. Make a **master** mirror image, it will be our **slave** machine
3. Configure machines **master** and **slave** in *Multi-Node* mode
4. Make a **slave** mirror image, it will be our **slave2**
5. Configure machine **slave2**

Implementation procedure 1(Install **master** in *Single-Node* mode**)**

- **Install JAVA (1st task to do)**

Add the following to **/etc/apt/sources.list.d/** file

```
#JAVA
```

```
deb http://archive.canonical.com/ lucid partner
```

Install java on the machine:

```
$ sudo apt-get update
$ sudo apt-get install openjdk-6-jdk
```

- **Test JAVA (2nd task to do)**

```
user@ubuntu:~# java -version
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) Client VM (build 16.3-b01, mixed mode, sharing)
```

- **Create a group and an user for Hadoop creation (3rd task to do):**

e.g.: create user **hduser** and the group **hadoop**.

```
$ sudo addgroup hadoop
$ sudo adduser --ingroup hadoop hduser
```

- **Configure SSH (4th task to do)**

Generate key for **hduser**

```
user@ubuntu:~$ su - hduser
hduser@ubuntu:~$ ssh-keygen -t rsa -P ""

Generating public/private rsa key pair.
Enter file in which to save the key (/home/hduser/.ssh/id_rsa):
Created directory '/home/hduser/.ssh'.
Your identification has been saved in /home/hduser/.ssh/id_rsa.
Your public key has been saved in /home/hduser/.ssh/id_rsa.pub.
The key fingerprint is:
9b:82:ea:58:b4:e0:35:d7:ff:19:66:a6:ef:ae:0e:d2 hduser@ubuntu
The key's randomart image is:
[...snipp...]
hduser@ubuntu:~$
```

- **Let SSH access file system with the previously created key (5th task to do)**

```
hduser@ubuntu:~$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

- **Test SSH (6th task to do)**

```
hduser@ubuntu:~$ ssh localhost

The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is d7:87:25:47:ae:02:00:eb:1d:75:4f:bb:44:f9:36:26.
Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
Linux ubuntu 2.6.32-22-generic #33-Ubuntu SMP Wed Apr 28 13:27:30 UTC 2010 i686
GNU/Linux
Ubuntu 10.04 LTS
[...snipp...]
hduser@ubuntu:~$
```

- **Disable IPv6 (7th task to do)**

This is done by editing **sysctl.conf** file the following way:

```
hduser@ubuntu:~$nano /etc/sysctl.conf

#disable ipv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

Warning: Computer must be restarted now

- **Install Hadoop (assuming download made to /usr/local/) (8th task to do)**

```
$ wget http://mirrors.fe.up.pt/pub/apache/hadoop/core/stable/hadoop-1.0.4.tar.gz
```

After download:

```
$ cd /usr/local
$ sudo tar xzf hadoop-1.0.3.tar.gz
$ sudo mv hadoop-1.0.3 hadoop
$ sudo chown -R hduser:hadoop hadoop
```

So the directory where **Hadoop** is installed will be: /usr/local/hadoop/bin

- **Update $HOME/.bashrc (9th task to do)**

Add the following text to the end of **$HOME/.bashrc** file of the user **hduser**.

```
# Set Hadoop-related environment variables
export HADOOP_HOME=/usr/local/hadoop
# Set JAVA_HOME (we will also configure JAVA_HOME directly for Hadoop later on)
export JAVA_HOME=/usr/lib/jvm/java-6-sun
# Some convenient aliases and functions for running Hadoop-related commands
unalias fs &> /dev/null
alias fs="hadoop fs"
unalias hls &> /dev/null
alias hls="fs -ls"
# If you have LZO compression enabled in your Hadoop cluster and
# compress job outputs with LZOP (not covered in this tutorial):
# Conveniently inspect an LZOP compressed file from the command
# line; run via:
#
# $ lzohead /HDFS/path/to/lzop/compressed/file.lzo
#
# Requires installed 'lzop' command.
# lzohead () {
hadoop fs -cat $1 | lzop -dc | head -1000 | less
}
# Add Hadoop bin/ directory to PATH
export PATH=$PATH:$HADOOP_HOME/bin
```

- **Configure Hadoop (10th task to do)**

hadoop-env.sh

The only required environment variable we have to configure for Hadoop in this tutorial is JAVA_HOME. Open **/conf/hadoop-env.sh** file in the editor of your choice (if you used the installation path in this tutorial, the full path is

/usr/local/hadoop/conf/hadoop-env.sh) and set the JAVA_HOME environment variable to the Sun JDK/JRE 6 directory.

Alter

```
# The java implementation to use. Required.
# export JAVA_HOME=/usr/lib/j2sdk1.5-sun
```

to

```
# The java implementation to use. Required.
  export JAVA_HOME=/usr/lib/jvm/java-6-sun
```

Configure the directory where Hadoop will keep the Data files

```
$ sudo mkdir -p /app/hadoop/tmp
$ sudo chown hduser:hadoop /app/hadoop/tmp
$ sudo chmod 750 /app/hadoop/tmp
```

Hadoop config files (xml)

Add the following snippets between the <configuration> ... </configuration> tags in the respective configuration XML file.

- **conf/core-site.xml**

```
<!-- In: conf/core-site.xml -->
<property>
<name>hadoop.tmp.dir</name>
<value>/app/hadoop/tmp</value>
<description>A base for other temporary directories.</description>
</property>
<property>
<name>fs.default.name</name>
<value>HDFS://localhost:54310</value>
<description>The name of the default file system. A URI whose scheme and authority
determine the FileSystem implementation. The
uri's scheme determines the config property (fs.SCHEME.impl) naming
the FileSystem implementation class. The uri's authority is used to
determine the host, port, etc. for a filesystem.</description>
</property>
```

- **conf/mapred-site.xml**

```
<!-- In: conf/mapred-site.xml -->
<property>
<name>mapred.job.tracker</name>
<value>localhost:54311</value>
<description>The host and port that the MapReduce job tracker runs
at. If "local", then jobs are run in-process as a single map
and reduce task.
</description>
</property>
```

- **conf/HDFS-site.xml**

```
<!-- In: conf/HDFS-site.xml -->
<property>
<name>dfs.replication</name>

<value>1</value>
<description>Default block replication.
```

```
The actual number of replications can be specified when the file is created.
The default is used if replication is not specified in create time.
</description>
</property>
```

Format FileSystem (HDFS)

```
hduser@ubuntu:~$ /usr/local/hadoop/bin/hadoop namenode -format
10/05/08 16:59:56 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = ubuntu/127.0.1.1
STARTUP_MSG: args = [-format]
STARTUP_MSG: version = 0.20.2
STARTUP_MSG: build = https://svn.apache.org/repos/asf/hadoop/common/branches/branch-0.20
-r 911707; compiled by 'chrisdo' on Fri Feb 19 08:07:34 UTC 2010
************************************************************/
10/05/08 16:59:56 INFO namenode.FSNamesystem: fsOwner=hduser,hadoop
10/05/08 16:59:56 INFO namenode.FSNamesystem: supergroup=supergroup
10/05/08 16:59:56 INFO namenode.FSNamesystem: isPermissionEnabled=true
10/05/08 16:59:56 INFO common.Storage: Image file of size 96 saved in 0 seconds.
10/05/08 16:59:57 INFO common.Storage: Storage directory .../hadoop-hduser/dfs/name has
been successfully formatted.
10/05/08 16:59:57 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at ubuntu/127.0.1.1
************************************************************/
hduser@ubuntu:/usr/local/hadoop$
Starting your single-node cluster
```

Initialize created node

```
hduser@ubuntu:~$ /usr/local/hadoop/bin/start-all.sh
starting namenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-namenode-
ubuntu.out
localhost: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-
datanode-ubuntu.out
localhost: starting secondarynamenode, logging to /usr/local/hadoop/bin/../logs/hadoop-
hduser-secondarynamenode-ubuntu.out
starting jobtracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-jobtracker-
ubuntu.out
localhost: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-
tasktracker-ubuntu.out
hduser@ubuntu:/usr/local/hadoop$
```

To visualize Hadoop processes running use **jps** command.

```
hduser@ubuntu:/usr/local/hadoop$ jps
2287 TaskTracker
2149 JobTracker
1938 DataNode
2085 SecondaryNameNode
2349 Jps
1788 NameNode
```

How to Stop Node

```
hduser@ubuntu:/usr/local/hadoop$ bin/stop-all.sh
stopping jobtracker
localhost: stopping tasktracker
stopping namenode
localhost: stopping datanode
localhost: stopping secondarynamenode
hduser@ubuntu:/usr/local/hadoop$
```

Note: This ends our Pseudo-Distributed (only one node) Hadoop installation

Implementation procedure 2 (Make a **master** mirror image, it will be our **slave** machine**)**

To have *multi-node* platform we made a copy of **master** machine and this copy will be the first **slave** machine that we will configure. This description starts with the assumption a copy was made.

Implementation procedure 3 (configure **master** and **slave** machines in *Multi-Node* mode**)**

- **Configure platform in Multi-Node mode (11th task to do)**

First changes to do:

Change **/etc/hostname** file of the copy machine to have the name **slave**

Change **/etc/network/interfaces** file of the copy machine to have the *IP* **192.168.0.2**

Change **/etc/hosts** file and add the names/IP's of **master** and **slave** (we also included **slave2**)

- **Configure Slave machine (12th task to do)**

Configure SSH access

The user **hduser@master** will have to be able to access via SSH to himself **master**, and also the **slave** machine. For that it is necessary to copy the public key existing on **master** to the **slave** machine.

```
hduser@master:~$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub hduser@slave
```

Test connection to both nodes

```
hduser@master:~$ ssh master

The authenticity of host 'master (192.168.0.1)' can't be established.
RSA key fingerprint is 3b:21:b3:c0:21:5c:7c:54:2f:1e:2d:96:79:eb:7f:95.
Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'master' (RSA) to the list of known hosts.
Linux master 2.6.20-16-386 #2 Thu Jun 7 20:16:13 UTC 2007 i686
...
hduser@master:~$
hduser@master:~$ ssh slave

The authenticity of host 'slave (192.168.0.2)' can't be established.
RSA key fingerprint is 74:d7:61:86:db:86:8f:31:90:9c:68:b0:13:88:52:72.
Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'slave' (RSA) to the list of known hosts.
Ubuntu 10.04
...
hduser@slave:~$
```

Configure parameterization files

- **Update conf/masters file (on master)**

This file should have only the name of **master** machine

```
master
```

- **Update conf/slaves file (on master)**

```
master
slave
slave2
```

- **Update conf/core-site.xml file (on both machines)**

```
<property>
<name>fs.default.name</name>
<value>HDFS://master:54310</value>
<description>The name of the default file system. A URI whose
scheme and authority determine the FileSystem implementation. The
uri's scheme determines the config property (fs.SCHEME.impl) naming
the FileSystem implementation class. The uri's authority is used to
determine the host, port, etc. for a filesystem.</description>
</property>
```

- **Update conf/core-site.xml file (on both machines)**

```
<!-- In: conf/mapred-site.xml -->
<property>
<name>mapred.job.tracker</name>
<value>master:54311</value>
<description>The host and port that the MapReduce job tracker runs
at. If "local", then jobs are run in-process as a single map
and reduce task.
</description>
</property>
```

- **Update conf/HDFS-site.xml file (on both machines)**

```
<!-- In: conf/HDFS-site.xml -->
<property>
<name>dfs.replication</name>
<value>3</value>
<description>Default block replication.
The actual number of replications can be specified when the file is created.
The default is used if replication is not specified in create time.
</description>
</property>
```

Formating FileSystem (HDFS)

```
hduser@master:/usr/local/hadoop$ bin/hadoop namenode –format

... INFO dfs.Storage: Storage directory /app/hadoop/tmp/dfs/name has been successfully
formatted.

hduser@master:/usr/local/hadoop$
```

Initiate created platform

Initiate FileSystem:

```
hduser@master:/usr/local/hadoop$ bin/start-dfs.sh

starting namenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-namenode-
master.out
slave: Ubuntu 10.04
slave: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-
datanode-slave.out
```

```
master: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-
datanode-master.out
master: starting secondarynamenode, logging to /usr/local/hadoop/bin/../logs/hadoop-
hduser-secondarynamenode-master.out

hduser@master:/usr/local/hadoop$
```

Initiate Map/Reduce processes

```
hduser@master:/usr/local/hadoop$ bin/start-mapred.sh

starting jobtracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hadoop-jobtracker-
master.out
slave: Ubuntu 10.04
slave: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-
tasktracker-slave.out
master: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-
tasktracker-master.out
hduser@master:/usr/local/hadoop$
```

To stop both processes you should execute them on the following order (Stop Map Reduce first):

```
hduser@master:/usr/local/hadoop$ bin/stop-mapred.sh

stopping jobtracker
slave: Ubuntu 10.04
master: stopping tasktracker
slave: stopping tasktracker
hduser@master:/usr/local/hadoop$


hduser@master:/usr/local/hadoop$ bin/stop-dfs.sh

stopping namenode
slave: Ubuntu 10.04
slave: stopping datanode
master: stopping datanode
master: stopping secondarynamenode
  hduser@master:/usr/local/hadoop$
```

Implementation procedure 4 (make a **slave** image, it will be our **slave2** machine**)**

Having machines **master** and **slave** correctly configured we will add one more *slave* machine to the platform, this will have the name **slave2**. The configuration of this new *slave* will have **slave1** has its base, first thing to do will be to copy **slave1**.

Implementation procedure 5 (Configure machine **slave2)**

- **Configure Slave2 machine (13th task to do)**

On this new copy it is necessary to do following updates:

- Update **/etc/hostname** file to be **slave2**
- Update **/etc/network/interfaces** file on the copy machine to have *IP* **192.168.0.3**

Once **slave2** is a copy of **slave1** the *multi-node* configuration is correctly done and we lack only some minor adjustments on some parameters.

Configure SSH access

The user **hduser@master** will have to access via SSH to itself, **master** and also **slave** machine. For that it is necessary to copy public key existing on **master** to the new **slave.**

```
hduser@master:~$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub hduser@slave2
```

Testing access to the machine

```
hduser@master:~$ ssh slave2

The authenticity of host 'slave2 (192.168.0.3)' can't be established.
RSA key fingerprint is 74:d7:61:86:db:86:8f:31:90:9c:68:b0:13:88:52:72.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'slave2' (RSA) to the list of known hosts.
Ubuntu 10.04
...
hduser@slave:~$
```

Directory /app/hadoop/tmp/dfs/data

Being **slave2** a copy of **slave1**, regarding **Hadoop** there are references to the original machine that avoid this new machine to be integrated correctly on the created platform so now we must remove the following directory:

/app/hadoop/tmp/dfs/data

This directory will be again created on the first time the machine is integrated on the platform.

In case Hadoop is being executed it is possible to add slave2 machine with the following command on the OS:

```
ubuntu@slave2:/usr/local/hadoop/bin$ ./hadoop-daemon.sh start datanode
```

Finally we have a **Hadoop** in *Multi-Node* mode working with the architecture defined on the beginning of this document.

2. Installation Procedures for Pegasus

We followed install procedures available on the project website (Science 2012) fundamentally it was needed to download PEGASUS files, unzip them and it's done.

```
$ wget http://www.cs.cmu.edu/~pegasus/PEGASUSH-2.0.tar.gz
$ tar -xzpf PEGASUSH-2.0.tar.gz
```

Following install we run Pegasus by opening install directory and inputting the following command:

```
$cd PEGASUS
/PEGASUS$ ./pegasus.sh
```

After previous commands Pegasus console is open and an algorithm for retrieval of vertexes degree was used with very fast results obtained for the data used. It took less than two minutes to get the degree of around 130000 vertexes.

To obtain results, first we had to follow Pegasus manual by Kang *et al.* (2010) to prepare graph and transfer it to Hadoop file system (HDFS).

3. Installation Procedures for Giraph

The installation of Giraph was made following the SVN checkout of the latest Giraph source, located at the official Apache site:

```
$ svn co https://svn.apache.org/repos/asf/giraph/trunk
```

After that we changed the folder to /trunk and compiled the code with the following command:

```
$ mvn compile
```

Once the build finishes, we navigated to the target folder created in the trunk folder and could see the JAR file giraph-0.2-SNAPSHOT-jar-with-dependencies.jar.

Then, we tested with the following command:

```
$ hadoop jar target/giraph-0.2-SNAPSHOT-for-hadoop-0.20.203.0-jar-with-dependencies.jar
org.apache.giraph.benchmark.PageRankBenchmark -V 1000 -e 1 -s 5 -w 1 -v
```

If the installation was successful the reader should see the job execute and the Map-Reduce command line output show success. Please pay attention that depending on the version of your Hadoop and Giraph installation, the previous command for testing installation might be different. You will have to change it accordingly.

Note: Some of this procedures for Giraph install were taken from Owens (2013).

4. Installation Procedures for Graphlab

The install of Graphlab was made following the next install procedures, essentially we downloaded Graphlab package from [20] and after uncompressing it, in the directory *graphlabapi* we had to compile the source files, resuming the following commands where used:

```
$ wget http://graphlabapi.googlecode.com/files/graphlabapi_v2.1.4434.tar.gz
$ tar -xzpf graphlabapi_v2.1.4434.tar.gz
```

Running ./configure in the *graphlabapi* directory, will create two sub-directories, release/ and debug/. Then we compiled only the graph analytics toolkit with the following command on the graphlabapi/toolkits/graph_analytics directory:

```
$ make -j 4
```

The command will perform up to 4 build tasks in parallel. There are several toolkits available from Graphlab and more information on the toolkits can be retrieved from webpage [21].

5. Installation Procedures for Hadoop Map Reduce (from book)

For implementation of Map/Reduce algorithms the book **Hadoop In Practice** Holmes (2012) was followed. On this book there are some algorithms developed by the author available for use by the reader of the book. There are also algorithms simply referred by the author but developed by other persons. All the algorithms are written in JAVA language. For further information please consult the book.

[20] http://code.google.com/p/graphlabapi/downloads/list
[21] http://docs.graphlab.org/toolkits.html

6. Installation Procedures for Snap (Stanford Network Analysis Platform)

The install of Snap was made following install procedures, essentially we downloaded Snap package from [22] and after uncompressing it, in the directory Snap we had to compile the source files using instructions for Linux OS available in [23].

7. Installation Procedures for Green-Marl

The install of Green-Marl was made following the install procedures in [24], essentially we downloaded Green-Marl package from Github and after that, in the directory *Green-Marl* we had to compile the source files, resuming the following commands where used:

```
$ git clone git://github.com/stanford-ppl/Green-Marl.git
$ cd /Green-Marl
$ make compiler
```

[22] http://snap.stanford.edu/snap/download.html
[23] http://snap.stanford.edu/snap/install.html
[24] https://github.com/stanford-ppl/Green-Marl

Appendix B

1. Edge List to Adjacency List – R code

```
graph <- read.csv("Relationships-Companies-FinancialOrg.txt", sep=" ",header=FALSE)
relations <- list()
nodes <- unique(c(graph[,1],graph[,2]))

for (k in 1:length(nodes)){
       relations <-
       c(relations,list(unique(c(nodes[k],graph[graph[,1]==nodes[k],2],graph[graph[,2]==
       nodes[k],1]))))
}
lapply(relations, write, "adjency_list.txt", append=TRUE, ncolumns=10000)
```

2. Edge List to Giraph JSON Input Format – R code

```
edges <- readLines("com-amazon.ungraph.tsv");

for (i in 1:length(edges)){
         if(edges[i]!=""){
                  node1 <- strsplit(edges[i],split="\t")[[1]][1]
                  node2 <- strsplit(edges[i],split="\t")[[1]][2]
                  write(paste("[", node1 ,",0,","[[",node2,",0]]]",sep=""), "Amazon-
         Giraph.txt", sep="\n", append=TRUE)
         } else {break}
}
```

3. Community Detection – Green-Marl code (core .gm file)

```
Proc label_node_1(G: Graph, Gaux: Graph, EWN: Node_Prop<Float>(G), COMM:
Node_Prop<Int>(G), EW: E_P<Float>(G)) //: Int
  {
       N_P<Bool>(G) Covered;

      // Compute Edge-Weight
       [printf("\nProcessing...Computing Graph Edges Weight!")];
      Foreach(s: G.Nodes) {
              G.Covered = False;
              Int counting = 0;
              Foreach(x: s.OutNbrs){
                     Foreach(y: x.OutNbrs)(!y.Covered){
                     If(y.HasEdgeTo(s))
                     counting ++;
                     }
              x.Covered = True;
              }

              Foreach (t: s.Nbrs) {
              Int triangles = 0;
              Foreach (u: s.Nbrs) {
                  If (t.HasEdgeTo(u)) {
                      triangles ++;
                                    //[printf("\nOn node %i - neighbour %i has edge to
neighbour %i",$s,$t,$u)];
                  }
              }
              Edge(G) e = t.ToEdge();
              e.EW = (triangles == 0) ? 0 : counting / (Float) triangles;
                     //[printf("\n Node %i to %i - counting = %i - Triangles = %i -
Edge weight = %f",$s,$t,$counting,$triangles,$e.EW)];
          }
      }

      // Initialize
      Edge_Prop<Bool>(G) VC;
       Node_Prop<Int>(G) membership;
       Node_Prop<Bool>(G) Covered2;
      G.Covered2 = False;
      G.EWN = 0;
       Int counter = 0;

       //has to be sequential to be respectful to node sequence
       [printf("\nProcessing...Setting Initial Community Labels for every node!")];
      For(n: G.Nodes) {
          n.COMM = counter; //Community initiation - each node belongs to its community
              counter++;
      }
      G.VC = False;

      // Cover & Compute EWN
      // Sequential Execution
      // (becomes non-deterministic if parallelized)
       [printf("\nProcessing...Computing Graph Maximum Neighbor's Edge Weight and for
every node!")];
```

```
      Foreach (s:G.Nodes)(!s.Covered2) {
          Edge(G) e_sel = NIL;
          Float maxval = -1;
          Node(G) from, to;

          Foreach(t: s.OutNbrs) {
              Edge(G) e1 = t.ToEdge();
              <maxval; from, to, e_sel> max= <e1.EW; s, t, e1> @t;
          }
          // there can be nodes that has no edges
          If (e_sel!= NIL) {
              e_sel.VC = True;
              from.Covered2 = True;
              //to.Covered2 = True;
              s.EWN = maxval;
          }
      }

       //Gaux has no edges yet, just nodes, lets add edges
       //has to be sequential or else it makes segmentation fault
      [printf("\nProcessing...Computing/Creating Auxiliary Graph Edges!")];
       For(n:G.Nodes){
               For(v: n.OutNbrs) (v.EWN==n.EWN){
               //To do
                       Edge(G) e2 = v.ToEdge();
                       If(e2.EW != 0){
                               [Gaux.add_edge($v,$n)];
                               //[Gaux.add_edge($n,$v)];
                               //[printf("\nAdded edge %i to %i, %f =
%f",$v,$n,$v.EWN,$n.EWN)];
                       }
               }
       }

  }

  Proc label_node_2(G: Graph, Gaux: Graph, COMM: Node_Prop<Int>(Gaux)) //: Int
  {

       //To do Find connected components in Gaux
       [printf("\nProcessing...Computing Auxiliary Graph Kosaraju Strong Connected
Components!")];
       //Kosaraju connected components initialization
       Node_Prop<Int>(Gaux) mem;
       // Initialize membership
      Gaux.mem = -1;

      N_P<Bool>(Gaux) Checked;
      Gaux.Checked = False;

      // [Phase 1]
      // Obtain reverse-post-DFS-order of node sequence.
      // Node_Order can be also used here but Node_Seq is faster
      Node_Seq(Gaux) Seq;
      For(t:Gaux.Nodes) (!t.Checked)
      {
          InDFS(n:Gaux.Nodes From t)[!n.Checked]
          {} // do nothing at pre-visit
```

```
            InPost{ // check at post-visit
                n.Checked = True;
                Seq.PushFront(n);
            }
        }

        // [Phase 2]
        // Starting from each node in the sequence
        //   Do BFS on the transposed graph G^.
        //   and every nodes that are (newly) visited compose one SCC.
        Int compId = 0;
         Map<Int,Int> Node_community;
        For(t:Seq.Items)(t.mem == -1)
        {
            InBFS(n:Gaux^.Nodes From t)[n.mem == -1]
            {

                        n.mem = compId;
                        //[printf("\n Node %i member of component %i",$n,$n.mem)];

            }
            compId++;
        }


         //Label each component nodes with the lower label of community
         [printf("\nProcessing...Labeling each node in component with its lower label!")];
         Int comp_aux_label = 0;
         Int mem_aux = -1;
         N_P<Bool>(Gaux) Checked2;
         Gaux.Checked2=False;
         Foreach(s:Gaux.Nodes)(!s.Checked2){
                If(mem_aux != s.mem){
                        comp_aux_label = s.COMM; //New component, from now on this label
will be the same for all nodes in this new component
                        Foreach(t:Gaux.Nodes)(t.mem==s.mem && !t.Checked2){
                                t.COMM=comp_aux_label;
                                //[printf("\nProcedure 2 - Node %i member of community
%i",$t,$t.COMM)];
                                t.Checked2=True;
                        }
                }
         mem_aux = s.mem;
         s.Checked2 = True;
        }

    }

    Proc label_node_3(G: Graph, Gaux: Graph, calc_mod: Int, EWN: Node_Prop<Float>(G),
COMM: Node_Prop<Int>(G), EW: Edge_Prop<Float>(G)) //: Int
    {

    [FILE *myfile];
    [myfile=fopen("results-raw.txt","a")];
    Edge_Prop<Bool>(G) VC;

    // Initialize before converging loop
    Bool Converged = False;
```

```
  N_P<Int>(G) prev_COMM;
  N_P<Int>(G) prev_prev_COMM;
  G.prev_COMM = -1;
  G.prev_prev_COMM = -1;
  Int iter = 0;

  While(!Converged && iter < 10){
  iter = iter + 1;
  [printf("\nAlgorithm Iteration %i", $iter)];
  Converged=True;

  //Calculate total degree of members of the same community
  Map<Int,Int> communityDegree;

  Foreach(n: G.Nodes) {
              Int d = n.OutDegree();
              communityDegree[n.COMM] += d;
              //[printf("\nProcedure 3 - Node %i on community %i", $n,$n.COMM)];
  }

   // Initialize
   Node_Prop<Float>(G) labelWeight;
   Node_Prop<Bool>(G) Covered;
   G.labelWeight = -1;

   Int CommDegree = 0;
   Int nedges = G.NumEdges();
   //[printf("\nNumber of Edges: %i",$nedges)];

   //Calculate labelWeight depending of size of community
   Foreach(s:G.Nodes){
       CommDegree = communityDegree[s.COMM];
       s.labelWeight = (1 - (CommDegree)/(Float)(2*nedges));
       //[printf("\nCD for Node %i and community %i: %i - LabelWeight:
%.2f",$s,$s.COMM,CommDegree,$s.labelWeight)];
   }

   //Initialize vars to final step of algorithm - node final label
   G.Covered = False;
   G.VC = False;
   Map<Int,Float> TEW_COMM; //Total edge weight of intra communities nodes

       // Cover & Compute COMM label
      // Sequential Execution
      // (becomes non-deterministic if parallelized)
       For(s: G.Nodes)(!s.Covered) { // Choose an edge that has maximum edge weight
       Edge(G) e_sel1 = NIL;
      Float maxval1 = -1;
      Node(G) from1, to1;
       TEW_COMM.Clear();

              Foreach(r: s.OutNbrs){
              //Edge(G) e = r.ToEdge();
              TEW_COMM[r.COMM] += r.EWN;
              //TEW_COMM[r.COMM] += e.EW;
              }

              For(t: s.OutNbrs){ // value among remaining nodes
```

```
                Edge(G) e2 = t.ToEdge();
                //Edge(G) e1 = t.ToEdge();

                //<maxval1; from1, to1, e_sel1> max= <t.labelWeight * t.EWN; s, t, e1>
@s;//@s??
                <maxval1; from1, to1, e_sel1> max= <t.labelWeight * TEW_COMM[t.COMM]; s,
t, e2> @t;
                }

        If (e_sel1!= NIL) {
                e_sel1.VC = True;
                from1.Covered = True;
                to1.Covered = True;
                        s.prev_prev_COMM = s.prev_COMM;//save previous to previous
COMMUNITY
                        s.prev_COMM= s.COMM;//save previous COMMUNITY
                s.COMM = to1.COMM;
                        //If((s.COMM != s.prev_COMM) && (s.COMM ==
s.prev_prev_COMM)){Converged=False;}
                        If(s.COMM != s.prev_COMM && s.prev_COMM !=
s.prev_prev_COMM){Converged=False;}
           }
        }

   }

   N_P<Bool>(G) Covered2;
   G.Covered2 = False;
   For(s: G.Nodes)(!s.Covered2){
   [char buffer[100]];
   [if (myfile != NULL){
        sprintf(buffer,"%i\t%i\r\n",$s,$s.COMM);
        fputs(buffer,myfile);
   } else throw("Unable to open file results-raw.txt")];
   s.Covered2=True;
   }
   [fclose(myfile)];


        If (calc_mod == 1){
        //Calculate Modularity - Modularity algorithm
        [printf("\nCalculating Modularity. Please Wait...")];
        // Initialize
       Node_Prop<Bool>(G) Covered3;
        G.Covered3 = False;
        Float Mod = 0.0;

                Foreach(u:G.Nodes)(!u.Covered3)
                {
                Foreach(v:G.Nodes)(v.COMM == u.COMM && v!=u){

                                If (u.HasEdgeTo(v)){
                                        Mod += 1 -
(u.NumNbrs()*v.NumNbrs())/(2*G.NumEdges()); //New_deg[u]
                                } Else {
                                        Mod +=  -
(u.NumNbrs()*v.NumNbrs())/(2*G.NumEdges()); //New_deg[u]
                                }
```

```
                    }
                u.Covered3 = True;
                }
        Mod = Mod/(2*G.NumEdges()); //Because we duplicated number of edges on graph
input, other way it would have to be 2*G.NumEdges???
        [printf("\nModularity: %f", Mod)];
       //print "Modularity: %f" % Mod
        }
   }
```

4. Community Detection – Main File (C++) code (core .cc file)

```
#include "communities-algo.h"         // header generated by gm_comp
#include <sys/time.h>
#include <iostream>
#include <fstream>
#include <string>
#include <stdio.h>
#include <map>
#define WONT_OPEN 20

#include <sys/types.h>
#include <dirent.h>
using namespace std;

//todo - convert to hash_map as desired.
typedef map<long, string> NodeMap;
typedef map<string, long> NameMap;

void add_node(gm_graph *G, gm_graph *Gaux, NameMap *names, NodeMap *nodes, long id,
string name) {
  G->add_node();
  Gaux->add_node();
  (*names)[name] = id;
  (*nodes)[id] = name;
}

//void load_edge_list(gm_graph *G, gm_graph *Gaux, NameMap *names, NodeMap *nodes,
string filename, char separator, string directed,string weighted) {
void load_edge_list(gm_graph *G, gm_graph *Gaux, NameMap *names, NodeMap *nodes, string
filename, char separator, string directed) {
  ifstream file;
  file.open(filename, fstream::in);
  cout << "\nOpened File " << filename;
  if (!(file.is_open())) {
         cout << "\nFile is not open... ";
    throw WONT_OPEN;
  }

  cout << "\nInitializing Variables... ";
  if(directed.compare("n")==0){
  cout << "\nGraph is undirected!";
  } else if(directed.compare("y")==0){
  cout << "\nGraph is directed!";
  }
  //TO DO - PREPARE CODE FOR WEIGHTED GRAPHS
  /*
  if(weighted.compare("n")==0){
  cout << "\nGraph is not weighted!";
  } else if(weighted.compare("y")==0){
  cout << "\nGraph is weighted!";
  }
  */
  string line;
  long int node_counter = 0;
  long int edge_counter = 0;
  cout << "\nBegining While Loop to read the edge list file... ";
  while(file.good()) {
         getline(file, line);
    if (line.find('#') != std::string::npos) continue;
    if(file.eof()) break;
    size_t split = line.find(separator);
    string u = line.substr(0, split);
    string v = line.substr(split+1);
    if(names->count(u) == 0) {
      add_node(G, Gaux,names, nodes, node_counter++, u);
    }
    if(names->count(v) == 0) {
      add_node(G, Gaux,names, nodes, node_counter++, v);
    }
         if (directed.compare("n")==0){//graph is undirected
         G->add_edge((*names)[u], (*names)[v]);
         G->add_edge((*names)[v], (*names)[u]);
         edge_counter++;
         } else if (directed.compare("y")==0) {//graph is directed
         G->add_edge((*names)[u], (*names)[v]);
```

```
          edge_counter++;
          }
  }
  cout << "\nGraph has "<< node_counter << " Nodes!";
  cout << "\nGraph has "<< edge_counter << " Edges!";
  cout << "\nClosing Edge List file!";
  file.close();
}


//function to translate internal green-marl nodes Ids to edge list nodes
void compile_results(NameMap *names, NodeMap *nodes) {
  //for reading raw results file
  ifstream file;
  //for writing final results file
  ofstream resultsfile;
  resultsfile.open("results-communities.txt",fstream::in | fstream::out | fstream::app);


  //for reading raw results file
          long size;
          char *buf;
          char *ptr;
          size = pathconf(".", _PC_PATH_MAX);
          if ((buf = (char *)malloc((size_t)size)) != NULL)
          ptr = getcwd(buf, (size_t)size);
          file.open(string(buf).append("/results-raw.txt"), fstream::in);
          cout << "\nOpening Raw Results File ";
  if (!(file.is_open())) {
          cout << "\nFile is not open... ";
    throw WONT_OPEN;
  }
  string line;
  long int node_counter = 0;
  long int edge_counter = 0;
  cout << "\nBegining While Loop to read the Raw file... ";
  while(file.good()) {
          getline(file, line);
    if (line.find('#') != std::string::npos) continue;
    if(file.eof()) break;
    size_t split = line.find('\t');
    string u = line.substr(0, split);
    string v = line.substr(split+1);
          //for writing final results file
          char buffer[100];
          if (resultsfile !=NULL){
          string s_u = (*names).find((*nodes).find(stol(u))->second)->first;
          string s_v = (*names).find((*nodes).find(stol(v))->second)->first;
          resultsfile << s_u.c_str() << "\t" << s_v.c_str() << "\r\n";
          } else
          {
          printf("Unable to open file results-communities.txt to write results");
          throw WONT_OPEN;
          }
}
//fclose(resultsfile);
resultsfile.close();
}

int main(int argc, char** argv) {
gm_graph G, Gaux;
NameMap names;
NodeMap nodes;
string directed;
string weighted;
string file_name;
string calc_mod_aux;
time_t timer, timer_end;
struct tm * ptm_start;
struct tm * ptm_end;
float ptm_interval;
int calc_mod;

 puts("\n#########################################");
 puts("#####  Community Detection Algorithm   #####");
 puts("#########################################\n");
```

```
if( remove( "results-raw.txt" ) != 0 )
    puts( "No need for cleaning tasks...continuing..." );
  else
    puts( "1st Cleaning Task Successfully Done" );
if( remove( "results-communities.txt" ) != 0 )
    puts( "No need for cleaning tasks...continuing..." );
  else
    puts( "2nd Cleaning Task Successfully Done" );

printf("Is the graph directed? Answer y (yes) or n (no): ");
cin >> directed;
//printf("Is the graph weighted? Answer y (yes) or n (no): ");
//cin >> weighted;
printf("Input graph file name (only unweighted edge list is accepted!!): ");
cin >> file_name;
printf("Do you want to calculate Modularity? It can make the algorithm slow! Answer y
(yes) or n (no): ");
cin >> calc_mod_aux;
if(calc_mod_aux.compare("n")==0){calc_mod=0;} else
if(calc_mod_aux.compare("y")==0){calc_mod=1;}
time(&timer);  /* get current time; same as: timer = time(NULL)  */
ptm_start = gmtime(&timer);
cout << "Started Computation of Communities Algo at: " << ptm_start->tm_hour << ":" <<
ptm_start->tm_min << "\n";
cout << "Loading Edge List...";

//load_edge_list(&G, &Gaux ,&names, &nodes, file_name, '\t', directed, weighted);
load_edge_list(&G, &Gaux ,&names, &nodes, file_name, '\t', directed);
cout << "\nCalculating Communities Labels for every node...";

//Variables for .gm procedures
// Create an array to hold the node property
int32_t* comm = new int32_t[G.num_nodes()]();
// Create an array to hold the node property
float* ewn = new float[G.num_nodes()]();
float* ew = new float[G.num_edges()]();

label_node_1(G, Gaux, ewn, comm, ew);//1rst Phase - Calculate Edge weights and store
them in each node
Gaux.freeze();
label_node_2(G, Gaux, comm); //2Phase - Build auxiliary graph - calculate connected
components on graph aux

label_node_3(G, Gaux, calc_mod, ewn, comm, ew); //3rd Phase -  final labels for our
original graph
time(&timer_end);  /* get current time; same as: timer_end = time(NULL)  */
ptm_end = gmtime(&timer_end);
cout << "\nEnded Computation of Communities Algorithm at: " << ptm_end->tm_hour << ":"
<< ptm_end->tm_min << "\n";
ptm_interval = difftime(timer_end,timer);
cout << "Processing Time - " << ptm_interval/3600 << " hours, " << ptm_interval/60  << "
minutes OR "<< ptm_interval <<" seconds \n";
cout << "Compiling Results...";
compile_results(&names, &nodes);
cout << "\nFile results-communities.txt has the algorithm results! Enjoy!\n";

 puts("\n###########################################");
 puts("#####  Community Detection Algorithm   #####");
 puts("###########################################\n");

return 0;
}
```

5. SimRank – Green-Marl code (core .gm file)

```
Proc simrank(G: Graph )
{

    [FILE *myfile];
    Float r = 0.9;
          Float s_uv = 0.0;
          Int iter = 100;
          Float eps = 0.0001;
          [bool FLAG_CONV = false];
          Int n_nodes = 0;
          Node_Prop<Bool> Covered;
          n_nodes = G.NumNodes();
          G.Covered = False;

          [float** sim_df = new float*[$n_nodes]];
          [float** sim_df_old = new float*[$n_nodes]];

    [for(int i = 0; i < $n_nodes; i++) {
        sim_df[i] = new float[$n_nodes];
                    sim_df_old[i] = new float[$n_nodes];
    }];

          //initialize matrices
          Foreach(s:G.Nodes){
          Foreach(t:G.Nodes){
          [
          if($s==$t){
          sim_df[$s][$t]=1;
          sim_df_old[$s][$t]=1;
          sim_df[$t][$s]=1;
          sim_df_old[$t][$s]=1;
          }else{
          sim_df[$s][$t]=0;
          sim_df_old[$s][$t]=0;
          sim_df[$t][$s]=0;
          sim_df_old[$t][$s]=0;
          }
          ];
          }
          }


          Node_Prop<Int> numNbrs;
          G.numNbrs = 0;
          Int j = 0;

          While(j <= iter){

          [if (!FLAG_CONV) {FLAG_CONV=true;} else {break;}];//test convergence FLAG

          [for(int k = 0; k < $n_nodes;k++){
          memcpy(sim_df_old[k], sim_df[k], sizeof(float) * $n_nodes);
          }];

                    Foreach(u: G.Nodes) {
                    u.numNbrs = u.NumOutNbrs();
                              Foreach(v: G.Nodes){
                              [if ($u == $v) {
                              continue;
                              } else {$s_uv=0.0;}];
                              v.numNbrs = v.NumOutNbrs();
                                        Foreach(n_u: u.OutNbrs)
                                        {
                                                  Foreach(n_v: v.OutNbrs){
                                                            [$s_uv = $s_uv +
sim_df_old[$n_u][$n_v]];
                                                  }
                                        }
                              [
                                        sim_df[$u][$v] = ($r * $s_uv)/ (float)
(($u.numNbrs)*($v.numNbrs));
                                        sim_df[$v][$u] = sim_df[$u][$v];
```

```
                                        if(sim_df[$u][$v] - sim_df_old[$u][$v] >= (float)
$eps || sim_df[$v][$u] - sim_df_old[$v][$u] >= (float) $eps){
                                        FLAG_CONV=false;
                                                                } //if there is no convergence
                    in any value of simrank then FLAG_CONV=FALSE
                                        ];

                                }
                    }
          j = j+1;
          }

// TO DO - Write file with top 10 of similarity ranking for all nodes
G.Covered=False;
[myfile=fopen("results-simrank-raw.txt","a")];
[char buffer[100]];
Int line = 1;
For(s:G.Nodes){
[if (myfile !=NULL && $line == 1){
          sprintf(buffer,"\t");
          fputs(buffer,myfile);

}];
For(u:G.Nodes){
[if (myfile !=NULL && $line == 1){
          sprintf(buffer,"%i\t",$u);
          fputs(buffer,myfile);

}];
}
If(line == 1){
[sprintf(buffer,"\n")];
[fputs(buffer,myfile)];
}
line = 0;
[if (myfile !=NULL){
          sprintf(buffer,"%i\t",$s);
          fputs(buffer,myfile);

} else {puts("Unable to open file results-simrank-raw.txt");}];
For(t:G.Nodes){
[if (myfile !=NULL){
          sprintf(buffer,"%f\t",sim_df[$s][$t]);
          fputs(buffer,myfile);

} else {puts("Unable to open file results-simrank-raw.txt");}];
}
[sprintf(buffer,"\n")];
[fputs(buffer,myfile)];
}
[fclose(myfile)];
}
```

6. SimRank – Main File (C++) code (core .cc file)

```
#include "simrank.h"         // header generated by gm_comp
#include <sys/time.h>
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <stdlib.h>
#include <unistd.h>

#define WONT_OPEN 20

#include <sys/types.h>
#include <dirent.h>
using namespace std;

//todo - convert to hash_map as desired.
typedef map<long, string> NodeMap;
typedef map<string, long> NameMap;

void add_node(gm_graph *G, NameMap *names, NodeMap *nodes, long id, string name) {
  G->add_node();
  (*names)[name] = id;
  (*nodes)[id] = name;
}

//void load_edge_list(gm_graph *G, NameMap *names, NodeMap *nodes, char filename[256],
char separator, char directed[256]) {
void load_edge_list(gm_graph *G, NameMap *names, NodeMap *nodes, string filename, char
separator, string directed) {
  ifstream file;
  file.open(filename, fstream::in );
  cout << "\nOpened File " << filename;
  if (!(file.is_open())) {
       cout << "\nFile is not open... ";
    throw WONT_OPEN;
  }

  cout << "\nInitializing Variables... ";
  if(directed.compare("n")==0){
  cout << "\nGraph is undirected!";
  } else if(directed.compare("y")==0){
  cout << "\nGraph is directed!";
  }
  string line;
  long int node_counter = 0;
  long int edge_counter = 0;
  cout << "\nBegining While Loop to read the edge list file... ";
  while(file.good()) {
       getline(file, line);
    if (line.find('#') != std::string::npos) continue;
    if(file.eof()) break;
    size_t split = line.find(separator);
    string u = line.substr(0, split);
    string v = line.substr(split+1);
       if(names->count(u) == 0) {
      add_node(G, names, nodes, node_counter++, u);
    }
    if(names->count(v) == 0) {
      add_node(G, names, nodes, node_counter++, v);
    }
       if (directed.compare("n")==0){//graph is undirected
       G->add_edge((*names)[u], (*names)[v]);
       G->add_edge((*names)[v], (*names)[u]);
       edge_counter++;
       } else if(directed.compare("y")==0){//graph is directed
       G->add_edge((*names)[u], (*names)[v]);
       edge_counter++;
       }
  }
  cout << "\nGraph has "<< node_counter << " Nodes!";
  cout << "\nGraph has "<< edge_counter << " Edges!";
  cout << "\nIMPORTANT NOTE: With this Graph, memory use will be approximately around "
<< 2*node_counter*node_counter*4/1000000 << "MB MAX";
```

```
  cout << "\nPLEASE MAKE SURE YOUR MACHINE'S MEMORY IS ENOUGH TO RUN THE ALGORITHM!!";
  cout << "\nClosing Edge List file...";
  file.close();
  cout << "\nClosed Edge List file!";
}

//function to translate internal green-marl nodes Ids to edge list nodes
void compile_results(NameMap *names, NodeMap *nodes) {
  //for reading raw results file
  ifstream file;
  //for writing final results file
  ofstream resultsfile;
  resultsfile.open("results-simrank.txt",fstream::in | fstream::out | fstream::app);

  //for writing final results file
  //resultsfile=fopen("results-communities.txt","a");

  //for reading raw results file
long size;
char *buf;
char *ptr;
size = pathconf(".", _PC_PATH_MAX);
if ((buf = (char *)malloc((size_t)size)) != NULL)
    ptr = getcwd(buf, (size_t)size);
        //cout << string(buf).append("/results-simrank-raw.txt");
  file.open(string(buf).append("/results-simrank-raw.txt"), fstream::in );
  //file.open("/home/110414015/Green-Marl/apps/output_cpp/bin/results-simrank-raw.txt",
fstream::in | fstream::out | fstream::app);
  cout << "\nOpening Raw Results File ";
  if (!(file.is_open())) {
        cout << "\nFile is not open... ";
    throw WONT_OPEN;
  }
  string line;
  long int node_counter = 0;
  long int edge_counter = 0;
  long int line_counter = 0;
  cout << "\nBegining While Loop to read the Raw file... ";
  while(file.good()) {
        //cout << "\nRead Lines started..." ;
    //line = (char) file.get();
        getline(file, line);
        line_counter++;
        //std::stringstream(line);
        //cout << "\nRead Line: " << line;
    if (line.find('#') != std::string::npos) continue;
    if(file.eof()) break;
        if (line_counter==1){
        size_t split = line.find('\t');
        string v = line.substr(split+1);
        size_t split2 = v.find('\t');
        string node1 = v.substr(0, split2);
        v = v.substr(split2+1);
        //cout << "\nNode1: " << node1;
        string s_u = (*names).find((*nodes).find(stol(node1))->second)->first;
        resultsfile << '\t' << s_u.c_str() ;
        do{
        if (v.find('\t')==std::string::npos){
        //cout << "\nNo tab and End of line " << v;
        break;
        } else if (v.find('\t')!=std::string::npos)
        {
        split = v.find('\t');
        string node = v.substr(0, split);
        //cout << "\nTab and Node: "<< node;
        v = v.substr(split+1);
        s_u = (*names).find((*nodes).find(stol(node))->second)->first;
        resultsfile << '\t' << s_u.c_str();
        }
        }while(true);
        //for writing final results file
        if (resultsfile !=NULL){
        resultsfile << "\r\n";
        } else
        {
        printf("Unable to open file results-simrank.txt to write results");
        throw WONT_OPEN;
```

```
        }
        } else {
        size_t split = line.find('\t');
        string node1 = line.substr(0, split);
        //cout << "\n New Line: " << line;
        //cout << "\nNode: " << node1;
        string s_u = (*names).find((*nodes).find(stol(node1))->second)->first;
        resultsfile << s_u.c_str() ;
        string v = line.substr(split+1);
        do{
        if (v.find('\t')==std::string::npos){
        //cout << "\n New Line: " << line;
        //cout << "\nNode: " << v;
        resultsfile << '\t' << v;
        break;
        } else if (v.find('\t')!=std::string::npos)
        {
        split = v.find('\t');
        string value = v.substr(0, split);
        v = v.substr(split+1);
        resultsfile << '\t' << value;
        }
        }while(true);
        //for writing final results file
        if (resultsfile !=NULL){
        resultsfile << "\r\n";
        } else
        {
        printf("Unable to open file results-simrank.txt to write results");
        throw WONT_OPEN;
        }
        }
}
//fclose(resultsfile);
resultsfile.close();
}


int main(int argc, char** argv){
gm_graph G;
NameMap names;
NodeMap nodes;
string directed;
string file_name;
time_t timer, timer_end;
struct tm * ptm_start;
struct tm * ptm_end;
float ptm_interval;

 puts("\n#################################");
 puts("#####  SimRank Algorithm   #####");
 puts("#################################\n");

if( remove( "results-simrank-raw.txt" ) != 0 )
    puts( "No need for 1st cleaning task...continuing..." );
  else
    puts( "1st Cleaning Task Successfully Done" );
if( remove( "results-simrank.txt" ) != 0 )
    puts( "No need for 2nd cleaning task...continuing..." );
  else
    puts( "2nd Cleaning Task Successfully Done" );

printf("Is the graph directed? Answer y (yes) or n (no): ");
cin >> directed;
printf("Input graph file name (only unweighted edge list is accepted!!): ");
cin >> file_name;
time(&timer);  /* get current time; same as: timer = time(NULL)  */
ptm_start = gmtime(&timer);
cout << "Started Computation of Similarity Ranking (Simrank): " << ptm_start->tm_hour <<
":" << ptm_start->tm_min << "\n";
cout << "Loading Edge List...";
load_edge_list(&G, &names, &nodes, file_name, '\t', directed);
cout << "\nCalculating Simrank for every node...";
cout.flush();
simrank(G);
time(&timer_end);  /* get current time; same as: timer_end = time(NULL)  */
ptm_end = gmtime(&timer_end);
```

```
cout << "\nEnded Computation of Simrank at: " << ptm_end->tm_hour << ":" << ptm_end-
>tm_min << "\n";
ptm_interval = difftime(timer_end,timer);
cout << "Processing Time - " << ptm_interval/3600 << " hours, " << ptm_interval/60  << "
minutes OR "<< ptm_interval <<" seconds \n";
cout << "Compiling Results...";
compile_results(&names, &nodes);
cout << "\nFile results-simrank.txt has the algorithm results! Enjoy!\n";

 puts("\n###################################");
 puts("#####  SimRank Algorithm   #####");
 puts("###################################\n");

return 0;
}
```