Journal of the Brazilian Computer Society manuscript No.
(will be inserted by the editor)

An Efficient Parallel Algorithm for Graph Centrality Metrics*

Leonardo Carlos da Cruz · Cristina Duarte Murta

Received: date / Accepted: date

Abstract Computation of large scale complex networks are useful to extract some insight from these intricate systems and increase the knowledgement about them. However, using deterministic methods instead of statistical approaches to carry out this type of computation is very difficult in terms of execution time and efficient resource usage. In this paper we present a new parallel algorithm for the exact calculation of metrics of centrality in graphs representing this large systems, under the MapReduce/Hadoop paradigm. The goal is to compute the distance between vertices and some centrality metrics derived from graph distances, in large graphs, by managing the computing resources efficiently. The proposed algorithm has been tested in several graphs and the results compared with sequential and parallel algorithms. The experiments show that it makes efficient use of memory and disk space, resulting in faster execution time, when compared to other implementations. The techiniques implemented were able to reduce about 70% of memory usage, more than 34% of disk space and at least 30% reduction in execution time.

Keywords parallel processing · graphs · complex networks

* submitted to Special Issue on High-Performance Computing Systems.

L. C. da Cruz
Laboratório de Computação Científica
Universidade Federal de Minas Gerais,
Belo Horizonte, MG, Brasil.
E-mail: leonardocruz@ufmg.br

C. D. Murta
Departamento de Computação. Centro Federal de Educação Tecnológica de Minas Gerais,
Belo Horizonte, MG, Brasil.
E-mail: cristina@decom.cefetmg.br

1 Introduction

A large number of systems as diverse as the Internet and Web, biological systems, social and transportation networks, and many others, have in common the fact that they can be modeled by graphs. The number of elements involved in these complex networks has achieved increasingly larger scales, which requires parallel processing for analysis. The ever-growing data acquisition routines associated with the widespread availability of data storage devices made possible the surge of large masses of data in different contexts. Thus, in many areas of knowledge, from collecting samples of DNA molecules capable of generating millions of fragments out a simple bacterium, to image acquisition and radiation measurements captured by telescopes that scan the universe [2], there is an increasing amount of data to process and store. A model commonly used to represent such data sets are graphs. So explore models based on graphs is an important task to assist scientific discovery. However, developing efficient techniques capable of extracting properties from graphs is a nontrivial task, and the challenge becomes even greater when working with large graphs, in which the number of edges and vertices reach scales of millions or higher.

In the context of parallel processing of large amounts of data, MapReduce [5] is a parallel programming model that has been adopted by information technology companies and by the academia. One of the factors that allowed its use was the development of the Hadoop [1] platform, an open source implementation of the model, which successfully computes data on the scale of petabytes. Thus, it is desirable to apply this model to the processing of large graphs.

In this context, we present the AHEAD algorithm (*Advanced Hadoop Exact Algorithm for Distances*), a parallel algorithm for processing large graphs, implemented in MapReduce/Hadoop. Our goal is to investigate the applicability of the MapReduce model to process graphs, particularly the processing of the distances between the vertices, as well the scalability of this processing of graphs with quantities of vertices and edges in various scales, and also the efficient use of computing resources in the procedure.

The computation of measures of centrality in graphs presents barriers regarding its complexity [8]. Those measures, that include the *diameter* and *radius* of the graph, are expensive, if not prohibitive, to be calculated in large scale graphs [11]. Thus, the question to be evaluated is what sizes of graphs we can analyze, using parallel algorithms, to obtain its exact centrality measures. For larger graphs, we may have an approximate calculation of these measures only [11,12].

This work follows a previous effort developed in [16], in which was proposed the HEDA algorithm (*Hadoop based Exact Diameter Algorithm*), that finds exact centrality measures for graphs of moderate size. However, the HEDA algorithm exhibit limitations on the use of data storage space [15]. These limitations are related to the use of the main memory by the algorithm and to how Hadoop manage the data movement in the distributed file system. To overcome these limitations, the AHEAD algorithm apply the techniques proposed in the work [14], enabling advances in obtaining accurate measurements

of *diameter* and *radius* on graphs of larger scales. The results presented in this paper show significant improvements in the use of main memory, disk space and execution time compared to other implementations.

This paper is structured as following. Sect. 2 describes the background and literature review connected to this work. Sect. 3 presents the proposed algorithm. Sect. 4 describes the experiments made in order to evaluate AHEAD performance and discusses the results. Sect. 5 concludes.

2 Related work

2.1 The MapReduce model

The MapReduce programming model aims to describe a way to process large amounts of data in parallel [5]. In this model it is assumed that the volume of data to be processed is not compatible with the storage capacity of a single compute node. Thus, the model include procedures for communication between the computational resources, whose goal is to distribute consistently the data and the processing. Additionally, MapReduce is designed to be used on clusters of commodity machines [5]. This condition allows flexibility to reduce costs, which is especially interesting when the expectation is to use thousands of compute nodes. Given this perspective, the model also provides maintenance of reliability during the parallel processing, even if the probability of failure increases not only due to a less robust machines, but also due to the number of machines in the system.

From the point of view of parallel programming, MapReduce specifies only two functions that must be written by the programmer: the *Map* function and the *Reduce* function. The programming needed for task synchronization, data distribution and the adequate use of computational resources is present in model's layers that are not visible to the programmer, being only accessible by setting the various parameters of the MapReduce/Hadoop implementation. This programming paradigm, therefore, seeks to reduce the complexity of parallel programs development aimed to large volumes of data, typically in the range of petabytes, providing reliable processing through fault management of the clustered machines in the distributed system [6].

A scheme depicting how the MapReduce model work is shown in Figure 1. The model's basic idea is to process in parallel many input files using mutiple copies of a *Map* function and multiple copies of a *Reduce* function. In the example shown in Figure 1, the goal is to count words from the documents stored in the distributed file system HDFS (*Hadoop Distributed File System*). Initially, the *Map* function takes as input the pair *¡file,content¿*, processing a line from the file at a time. The output of the processing is the pair consisting of a word taken as *key*, and the numeral 1 taken as *value* . Each *¡key,value¿* pair is stored in partitions on local disks of the machine running the *Map* function, where the destination partition is determined by a partitioner function. After

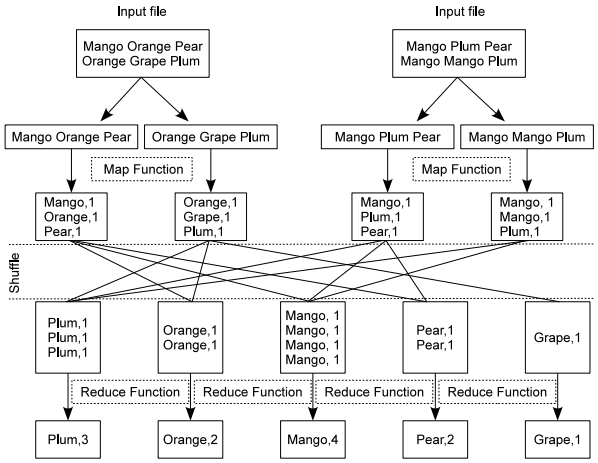


Fig. 1 A word counter implemented in MapReduce.

all input files have been processed in parallel, all outputs of the *Map* function copies are redistributed by the system, in a procedure called *shuffle* [13].

In this process, the outputs of the *Map* function with identical words or *keys* are grouped on the same partition, which are stored on the machine file system that will perform the *Reduce* function. At the end of the redistribution, the next step is performed by the execution of the *Reduce* function, applied in parallel on each partition. As the function is applied, the output is written to HDFS. In Figure 1, the *Reduce* function sum all the values associated with the same key, resulting in the count of the words in the input files.

2.2 Calculation of Graph Centrality Measurements

In this work we consider directed graphs $G = (V, E)$, being V the set of vertices and E the set of directed edges, where edges represent some relationship between two vertices. Undirected graphs can be represented by replacing the undirected edges with two directed edges in opposite directions. Graphs treated here are simple, that is, meaning that they have no multiple edges, no loops and no weights on the edges.

In many mathematical models that use graphs, the notion of the *center* of a network is quite explored, since it is often desired to determine the location of elements in the graph that meet some criteria. For example, one may want to determine a point in the network where the access time to the remaining points is minimized. In other cases, it is desired that this time has an upper limit, that is, it is desired that the response time between the point and the others do not surpass a pre-determined value.

In this particular work, we are interested in to calculate the *diameter* and *radius* of a graph by calculating the eccentricities of each vertice, which in turn, are calculated from the distances between the vertices of the graph. The

distance $d(s,t)$ from a vertice s to a vertice t is the length of the shortest path between s and t. The eccentricity of a vertice is the longest distance found between that vertice to the others reachable vertices of the graph. Thus, we denote the eccentricity of a vertice v by $e(v) = \max_{t \in V}\{d(v,t)\}$.

The largest eccentricity among the vertices is defined as *diameter* of the graph, that is, the longest distance between two vertices present in the graph. Mathematically we express the *diameter* of a graph $G = (V, A)$ by $D(G) = \max_{v \in V}\{e(v)\}$. The *radius* of the graph, denoted by $R(G) = \min_{v \in V}\{e(v)\}$, is the smallest eccentricity among the vertices of the graph. A simple analysis reveals that vertices with low eccentricity values are in the *centre* of the graph, while high values of eccentricity indicate the vertices in the *periphery* of the graph. Note that this centrality measure has high computational cost, since the number of shortest paths is upper bounded by $\binom{|V|}{2} = \frac{|V|^2-|V|}{2}$ for graphs with $|V|$ vertices. Moreover, one should take into account the computational cost needed to find a single shortest path.

2.3 Parallel Processing of Graphs using MapReduce Model

Before processing data using the MapReduce model is necessary to analyze the suitability of this programming paradigm to the target application. Some more complex algorithms are not easily adaptable to the model, while others are inherently parallelizable, with the best performance of MapReduce being evident in the latter group [18]. In the case of graph processing, it is possible to implement operations over graphs in a simple and efficient way, provided that these operations make distinctive use of local processing [3].

In the work developed in [14], the authors describe a set of best practices for the design of graph algorithms using MapReduce. Those practices consist of three techniques, called *In-Mapper Combiner*, *Schimmy* and *Range Partitioning*, all inspired by the model's workflow. The technique *In-Combiner mapper* consist to pre-process the output of *Map* phase (i.e., before it is written in the disk), minimizing the amount of data transferred through the phase *Shuffle* and reducing the work done by the *Reduce* phase. In other words, the objective of this technique is to minimize the amount of the processing done in the *Reduce* phase by decreasing the size of the output at the *Map* phase. When using the *Schimmy* technique, the intention is to transmit in the most efficient manner possible, between one phase and another, the information about the structure of the graph. This means that the data traffic should be mostly made of calculation data. Finally, the technique *Range Partitioning* consists of designing a partition function that takes advantage from information about the topology of the graph. For example, it is more interesting to create a partitioning function that allocates neighboring nodes in the same partition at the output of the *Map* phase, so any processing involving both vertices is done locally, i.e., on the same computing node memory. Combining these three design practices, the authors managed to performance gains up to 70% compared to other implementations of the *PageRank* algorithm [14,17].

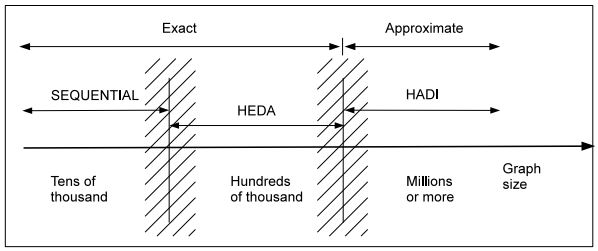


Fig. 2 Range of work for HEDA, HADI and sequential algorithms, in the computation of graph diameters on graphs of various sizes.

2.4 The HADI Algorithm

The algorithm HADI *(Hadoop diameter and radii estimator)* [12] is an algorithm implemented in Hadoop to estimate the diameter of graphs with sizes on the order of petabytes. To calculate the approximate diameter of the graph, the algorithm makes use of the concept of *effective diameter*, which is defined as the minimum length at which 90% of the connected pairs of vertices in the graph are reachable to each other. The main operation of the algorithm is to estimate, in parallel, the counting of the number of connected pairs of vertices that reach each other within a given path length, using for this purpose the *Flajolet-Martin algorithm* [7]. When estimating the count of connected pairs, it is possible to calculate the approximate diameter of the graph. The algorithm achieves good results in terms of scalability with the number of machines and in terms of run time with the number of vertices. However, the estimated diameter may differ significantly from its actual value [16].

2.5 The HEDA Algorithm

Also developed for MapReduce/Hadoop platform, the HEDA algorithm *(Hadoop-based Exact Diameter Algorithm)* [16] is based on the expansion of the borders between known and unknown vertices, discovering new neighbors at each expansion (breadth-search first). The algorithm makes a parallel traversal, calculating the distance between a source node and all others simultaneously and accurately, as it progress in the discovery of new vertices.

Fig. 2 displays the ranges of sizes of graphs in which HEDA, HADI and sequential algorithms operate, in their exact and approximate calculations of the graph diameter. The shaded region indicates that the limits of operation of each algorithm have some flexibility, depending on the graph size and on the computational resources applied in the processing. There is a space for HEDA algorithm to operate, which is explored through parallelism in order to obtain deterministic results for graphs that are impossible to be processed by sequential solutions.

Given the current inability to calculate the exact value of centrality metrics in very large graphs, the question that arises is: for which graph sizes (i.e,

number of vertices and edges) is it possible to compute the exact value of radius and diameter? This question is important because for those interested in calculating the diameter of large graphs, it is up to them to choose what type of computation - exact or approximate - best meets their needs.

3 Description of the AHEAD Algorithm

3.1 Overview

In the case of parallel processing of graphs using the MapReduce model, splitting data input means the division of the structure of the graph in the distributed system. Therefore, given a number of *mappers*, which are Java virtual machines instantiated by the Hadoop, the structure of the graph should be distributed among them in the form of exclusive subsets. However, the *mapper* that performs the *Map* function in a subset does not share the data in its memory with other *mappers* of the distributed system. Thus, to calculate and update the distances between vertices of different partitions or machines, it is necessary to transmit the local results between the partitions of the graph. To achieve that, the *mappers* temporarily store intermediate results on the local file system to pass them to the following iterations of processing.

The reception and synthesis of partial results is done by *reducers*, that are also Java virtual machines instantiated and managed by the Hadoop platform. The synthesis deals with the need of filtering the information received by the *reducers*, to select those that meet the criteria related to the proposed problem. In the case of the AHEAD algorithm, the criterion is the shortest distance calculated so far between two vertices of the graph. Moreover, the *reducer* should access the partitions which such vertices belong, in order to perform proper update of information on the distance between them.

3.2 The Algorithm operation

The AHEAD algorithm implements the ideas detailed in the work [14], with the implementation of techniques *In-Mapper Combiner* and *Schimmy*. The technique *Range Partitioning* is not implemented because it is effective only for graphs that represent WWW domains. However, the action of partitioning the graph is actually implemented in AHEAD, using the Hadoop *Default Partitioner* function. This function partitions the input data based on a *hash* function, which provides homogeneous partitions sizes. Equal-sized partitions allow its processing to be finished in approximately equal time among them, thus avoiding contention on the parallel execution due to a poor balancing of the data distribution. We will refer to this type of partitioning as *Hash Partitioning*.

The goal in implementing the techniques mentioned was to optimize the use of disk space and memory. This optimization enables the advancement in

```
1: G ← adjacency_list;
2: R ← number_of_reducers;
3: for i=1 to R do in parallel
4:     G_i ← CreatePartitionMatrix(G);
5:     HDFS ← G_i;
6: end for
7: while (stop condition = false) do
8:     for all G_i ∈ G = {G_1 ∪ G_2 ∪ ... ∪ G_R} do in parallel
9:         ComputeShortestPaths(G_i);
10:        UpdateDistances(G_i);
11:    end for
12:    Update(stop condition);
13: end while
14: for all G_i ∈ G = {G_1 ∪ G_2 ∪ ... ∪ G_R} do in parallel
15:    for all vertice v ∈ G_i do
16:        ComputeEccentricity(v)
17:        D_i ← LargestEccentricity();
18:        R_i ← SmallestEccentricity();
19:    end for
20: end for
21: repeat in parallel
22:    Diameter ← RemoveSmallest({D_1, D_2, ..., D_R});
23:    Radius ← RemoveLargest({R_1, R_2, ..., R_R});
24: until {D_1, D_2, ..., D_R} = ∅ and {R_1, R_2, ..., R_R} = ∅
```

Fig. 3 Main program with three phases of the AHEAD algorithm.

the range of graph sizes that can be processed deterministically, according to the direction shown in Fig. 2. The pseudo-code algorithm AHEAD, shown in Fig. 3, is composed of three phases listed below:

1. Lines (1-6): Parallel creation in HDFS of the distance matrix of the graph. The matrix is partitioned into R partitions, where R is the number of *reducers*;
2. Lines (7-13): Parallel computation of the distances between all vertices and parallel update in HDFS of the distance matrix partitions of the graph;
3. Lines (14-24): Parallel computation of eccentricity of all vertices. The eccentricities are used to calculate the radius and the diameter of the graph.

The matrix representation of the graph informs that the line position i and column j of the matrix is occupied by the value of the distance between the destination vertex i and the source vertex j. The matrix is created in phase 1 from the reading of all records of the adjacency list of the graph. The adjacency list contains in each record a vertex of the graph and its neighbors list. In line 4, *mappers* send in parallel the records of the adjacency list for a given partition G_i. In line 5, *reducers* receive and make the expansion of the records with columns corresponding to distances to other vertices of the graph. The partition to which the record is sent and subsequently extended is determined by the function *Hash Partitioning*. At the end of this phase, there will be R files in HDFS, each containing a portion of the distances matrix of the graph. This is the only phase where the structure of the graph is transmitted

via a process *shuffle*, which makes the AHEAD algorithm to comply with the design pattern *Schimmy* proposed in [14].

In line 9 of Fig. 3, *mappers* working in parallel perform the computation of for distances between vertices of the graph, according to the information available in the partition G_i. The computation of distances is based on the breadth-first search algorithm, where the accumulated distance of a vertex with respect to an origin serves as a reference for calculating the distance of its neighbors based on the same origin. Therefore, to calculate the distance to a neighboring vertex, it is enough to add one to the accumulated distance. The *Map* function that computes the paths between different origins and destinations was implemented so that only the lowest values are recorded locally before being sent to *reducers*. Furthermore, when there are different paths between same source and destination with equal distances values, only one of these values is made available, avoiding redundancy of information in the local disk. These two techniques of implementation of the function *Map* contemplate the design pattern *In-Mapper Combiner* [14].

In line 10, *reducers* accesses the partition of the graph, i.e., the graph file, to make updates of distances. Each *reducer* is responsible for a partition. The update of R partitions occurs simultaneously, i.e., the *reducers* work in parallel as well as the *mappers* do when calculating the distances. Whenever the distance of a vertex is updated, the calculation of the distance of its neighbors vertices regarding the origin considered is triggered in the next iteration. The stop condition on the line 12 is based on this fact, because the *reducer* that updates any partition also increments a global update counter. A complete iteration of the loop in line 7 corresponds to a cycle *map-shuffle-reduce* or *Job*. When there is no update during a *Job*, the condition in line 12 changes, finishing the loop. Note that inside the loop, only distances values are transferred between *mappers* and *reducers*. There is no transfer of the structure of the graph. This fact saves the space utilized for local storage of partial results as well it saves network resources employed for transferring the data between the Java virtual machine instances.

3.3 Number of *Mappers* and *Reducers*

The amount of *mappers* instantiated by the Hadoop platform depends on the size of the input data. The number of *mappers* M is the result of the division:

$$M = \frac{T_e}{B_{\text{hdfs}}}, \tag{1}$$

being T_e the amount of bytes in the input of the *Map* phase and B_{hdfs} a configurable parameter of Hadoop that specifies the size in bytes of the data block used by HDFS. A file stored in HDFS consists of one or more blocks depending on the value of B_{hdfs} and, according to the equation (1), Hadoop plataform allocates by *default* one *mapper* for each data block. The number of R *reducers* instantiated by Hadoop is specified via parameter in Hadoop, so

also it is in AHEAD, and the value used in this work is always equal to the amount of cores available in the system. Thus, considering c the number of cores available in the cluster and $k = \frac{M}{R}$ the relationship between the number of *mappers* and *reducers* desired, dividing both sides of the equation (1) by R, we obtain:

$$B_{\text{hdfs}} = \frac{T_e}{k \cdot c} \cdot \tag{2}$$

If $k = 1$ and T_e is known, the estimated value of B_{hdfs} is such that the number of *mappers* or *reducers* allocated by the Hadoop platform equals the number of cores available in the system. The modular implementation of AHEAD allows the phase 1 (lines 1-6) to be executed separately, which enables the prior verification of the value of T_e, that is, the size of all matrix partitions. Thus it is possible to estimate the value of B_{hdfs} so that the relations *cores/ mappers* and *cores/reducers* are 1/1. The value B_{hdfs} is estimated because the size of matrix partitions (i.e., T_e) may vary during the phase 2 (lines 7-13) as the columns of the matrix are updated with values of distances which may increase the size of the matrix in bytes.

4 Description of the Experiments and Results

4.1 Planning of experiments

The AHEAD algorithm was evaluated by processing both synthetic and real graphs. The Real graphs were obtained from the database *Stanford Large Network Dataset Collection*[1] maintained by the research group *Stanford Network Analysis Project*. Three graphs obtained from real data were tested: (1) the social network *Epinions*, with 75,879 vertices and 508,837 edges, (2) a graph representing the e-mail communication network used by company employees of *Enron*, with 36,692 vertices and 367,662 edges, a scientific collaboration network between co-authors of articles in the category *Astro Physics* of the eletronic magazine *arXiv*, with 18,772 vertices and 396,160 edges.

Additionally to this set, synthetic graphs were created with fixed number of 50,000 (50K) vertices and number of edges ranging between 150K and 1.600K, in order to evaluate the scalability of AHEAD algorithm with the variation of the number of edges. Synthetics graphs with number of vertices varying between 25K and 200K and number of edges varying between 75K and 600K were also generated. The library *NetworkX* [10] was used to generate synthetic graphs, which was also the source of one of the two sequential algorithms utilized in the test. The second sequential algorithm tested in the experiment is cited in [9]. The results of both are compared with AHEAD in order to verify the correctness of the results (radius and diameter of the graph). A comparison between HEDA and AHEAD is made with the purpose of evaluating the use of disk space and memory. The values presented in graphs and tables are the

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

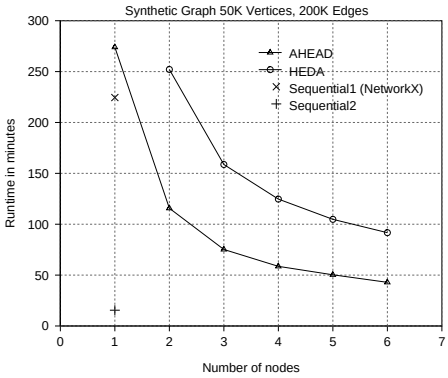


Fig. 4 Comparison between AHEAD, HEDA and sequential algorithms.

average results of three runs. We note that the variation around the mean of the measurements is small. All data collection can be found in [4].

The tests were performed on a cluster with 6 nodes interconnected via 1Gb/s switch, where each node has two Intel Xeon X5660 processors at 2.80GHz, 98GB RAM and 300GB of disk. Altogether there were 72 physical cores or 144 virtual cores (using *hyperthreading*) present in the cluster. The Hadoop version installed is 1.1.2 and the operating system is Gentoo Linux Kernel 3.6.11. Only during the comparison between HEDA and AHEAD algorithms, described in Section 4.6, the capacity of local disks were changed to the maximum amount available, which was 2TB per node.

4.2 Algorithm Validation

The results of radius and diameter generated by AHEAD are equal to the results provided by the sequential algorithms and also to the HEDA results, for various graphs tested. For example, for the synthetic graph with 50K vertices and 200K edges, all algorithms tested showed the same results, namely, $Diameter = 15$ and $radius = 11$. Fig. 4 shows the execution times of the sequential algorithms (on a single machine), and the algorithms HEDA and AHEAD, when varying the number of machines. The results shown in Fig. 4 indicate that the MapReduce/Hadoop model has a significant overhead, and its use is only justified to quantities of data that can not be performed on a single machine.

Table 1 Variation in the number of edges in synthetic undirected graphs with 50K vertices.

Edges	Radius	Diameter	Average Time(min.)	Standard Deviation	Number of edges variation	Time variation
200.000	11	15	46,6	0,4	-	-
400.000	7	9	56,1	0,0	2x	1,2x
800.000	5	6	96,7	1,9	4x	2,1x
1.600.000	4	5	233,1	1,0	8x	5x

4.3 Evaluation of the scalability of the AHEAD

4.3.1 Variation in the number of edges

The behavior of the AHEAD algorithm due to variation in the number of edges is shown in Fig. 5 and 6. In this experiment, the graph has 50K vertices and number of edges varies between 200K and 1.600K. The results of diameter and radius are shown in Table 1. Increasing the number of edges represents the increase in the number of relationships between a fixed number of entities. In the tested range, the AHEAD runtime was multiplied about four times while the number of edges was multiplied by eight.

Fig. 5 shows that, starting from 400K edges, the *map* phase, which makes the breadth-first search in the graph, dominates the processing time spent on the *Job* with longer duration. In Fig. 5, the processing time of the *reduce* phase includes the time spent in the data transmission process (*shuffle*). Fig. 6 shows that the rate of growth of the execution time is slow. In Figures 4 and 6, respectively, HEDA algorithm can not process the graph with 200K edges on a single machine and can not process a graph with 1.600K edges in six machines. In both cases, the system sets off messages about the lack of disk space on the cluster during the processing.

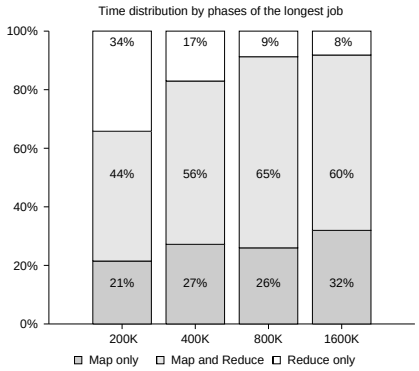


Fig. 5 Time spent in each phase with increasing number of edges. Data are collected from the longest job.

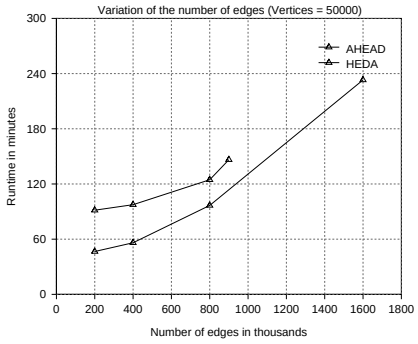


Fig. 6 Variation in the number of edges on graphs with 50K vertices.

Table 2 Effects of variation in the number of vertices and edges on synthetic directed graphs.

Vertices / Edges	V + A	Radius	Diameter	Average time(min.)	Standard Deviation	Graph size variation	Time variation
25.000 /75.000	100.000	13	21	19,3	0,1	-	-
50.000 /150.000	200.000	13	22	44,1	0,5	2x	2,3x
100.000 /300.000	400.000	15	22	170,5	3,6	4x	8,8x
200.000 /600.000	800.000	15	24	1212,8	9,4	8x	62,8x

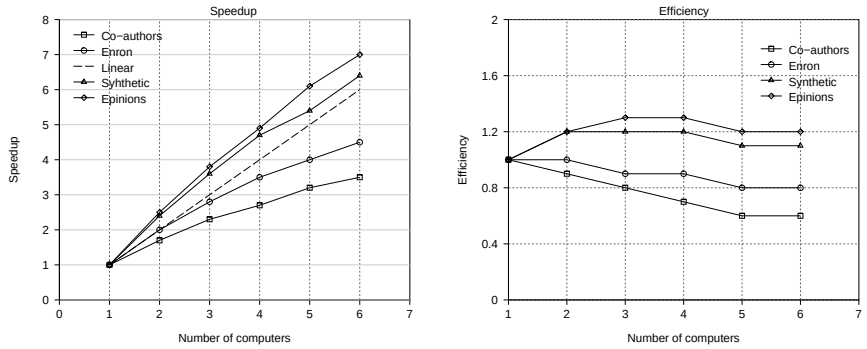
4.3.2 Variation in the number of edges and vertices

Table 2 shows the values of diameter and execution time when processing directed synthetic graphs varying the number of edges and vertices. Increasing the number of vertices and edges in this case represents the increase of the number of entities and relations among them. In practice, the insertion of vertices is only effective in the computation of distances if they have degree higher or equal to one. The graphs shown in Table 2 are connected, which ensures the existence of at least one new edge to each new vertex inserted. In this table, it is notable that the variation of execution time is larger than the variation of the size of the graph compared to the variation shown in Fig. 6. By increasing the amount of vertices and edges ($V + E$), keeping the relation edges/vertices relatively low ($E/V = 3$), there are an increasing amount of vertices distant from each other, which makes harder the algorithm's traverse on the graph. This fact affects the behavior of the algorithm because when the relation edges/vertices is low, there will be fewer options of paths between vertices. The reduced number of paths makes the algorithm reach more isolated or far away vertices from each other, resulting in increasingly larger diameters shown in Table 2. In the other hand, the relation edges/vertices seen in Table 1 are high, reaching the highest value as much as ($E/V = 32$). The execution time variation is therefore low as well are the diameters of the graphs.

In this experiment, undirected synthetic graphs were also processed. The results obtained are given in Table 3. The analysis made for directed graphs is also valid in this case. The small difference that appears in the values of time variation (last column), if compared to the same measurement for the directed graphs in Table 2, is due to the fact that bidirectional edges provide the existence of alternative pathways, facilitating the exploration of the graph by the algorithm.

Table 3 Effects of variation in the number of vertices and edges on synthetic undirected graphs.

Vertices / Edges	V + E	Radius	Diameter	Average Time(min.)	Standard Deviation	Graph size variation	Time variation
25.000 /75.000	100.000	15	21	19,5	0,2	-	-
50.000 /150.000	200.000	15	23	47,1	0,8	2x	2,4x
100.000 /300.000	400.000	16	24	163,3	0,9	4x	8,4x
200.000 /600.000	800.000	18	26	1129,8	1,7	8x	57,9x



(a) Speedup. (b) Efficiency.

Fig. 7 Performance test using one synthetic graph and three real graphs.

4.4 Speedup and Efficiency

For the performance test, the real graphs described in Section 4.1 were used, together with a synthetic graph containing 50K vertices and 200K edges. Fig. 7 shows the results of *speedup* and efficiency with the variation in the number of machines present in the cluster. In Fig. 7a we note that, for larger graphs, the AHEAD algorithm presents a *superlinear* speedup. This can be explained by the fact that, when adding machines in the cluster, the amount of data that a compute node must process is smaller, which allows the data processing to be better adjusted in the cache memory of each processor. Thus, the memory operations will be performed on each compute node more efficiently. This effect also appears in Fig. 7b where there are efficiency values greater than 1. For smaller graphs, we believe that this effect is superposed by the system

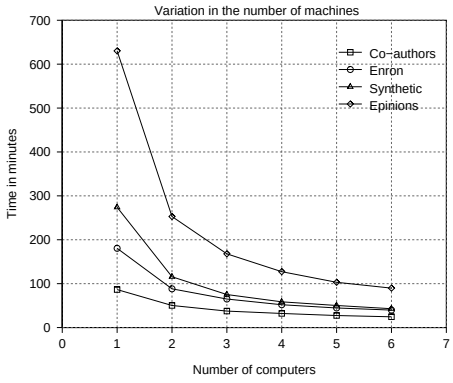
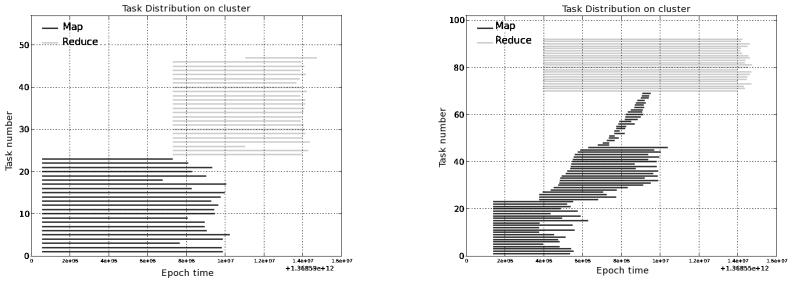


Fig. 8 Execution time with variation in the number of machines.



(a) (k = 1) (b) (k = 2)

Fig. 9 Variation in the ratio between the number of *mappers* and *reducers* in processing the graph Epinions with one machine. For $k = 1$ the number of *mappers* is equal to the number of *reducers*. For $k = 2$ the number of *mappers* is expected to be the double of the number of *reducers*, but the value of T_e during the longest job was slightly above the estimated, causing the instantiation of *map* tasks 47-69 with small processing time.

overhead, which is due to several factors related to the distributed processing of *mappers* and *reducers*, such as the instantiation of virtual machines, fault management and data communication on the ethernet network during *shuffle* phase.

Fig. 8 shows the variation of the average execution time for various graphs when varying the number of cores presented in the cluster. The plot in Fig. 8 shows that, improving execution time with the addition of cores is more evident in the processing of the graph with the largest number of vertices and edges, Epinions. For this graph, the processing load that just one machine (23 cores) experiences is better distributed in six machines (138 cores) than when the same occurs in the smaller graph, Co-authors. This result shows that the AHEAD algorithm gives good results for its primary purpose, which is to process graphs on larger scales.

4.5 Variation of the HDFS block size

According to Equation 2 of Section 3.3, the relationship between the number of *mappers* and *reducers*, k, can be adjusted by the proper selection of the value of B_{hdfs}, which is the block size used in HDFS. Fig. 9 shows the distribution of *mappers* and *reducers* in the cluster for $k = 1$ and $k = 2$, on a machine with 23 cores. The plots were generated from the sequence *map-shuffle-reduce* (*Job*) of longest duration, during processing of the graph Epinions. A horizontal bar on the graph has the length equivalent to the duration of a task, considering its dates of beginning and end in the Unix *epoch* format. In Fig. 9a, we note the predominance of parallelism between *map* tasks and between *reduce* tasks, in isolation, i.e., there is little concurrency between *mappers* and *reducers*. Fig. 9b shows predominance in concurrency between tasks of different nature. We also note that for $k = 2$, the instantiation of *map* tasks occurs in groups

Table 4 Resources usage by the algorithms HEDA and AHEAD.

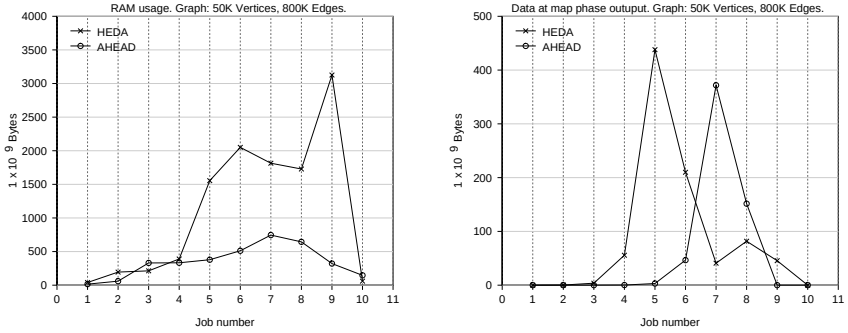
Resource	Graph	HEDA	AHEAD	Reduction
Memory (GB)	Synthetic	11161,6	3487,2	68,8%
	Enron	11152,7	3714,1	66,7%
	Co-authors	11157,6	3075,5	72,4%
Disk (GB)	Synthetic	875,7	573,2	34,5%
	Enron	399,4	142,4	64,3%
	Co-authors	182,2	90,5	50,3%
Time (Min.)	Synthetic	147,9	102,8	30,5%
	Enron	67,1	40,2	40,0%
	Co-authors	46,7	24,5	47,5%

of 23 *mappers*, which is the number of cores. On the Y axis, the last group of *mappers* is composed of the tasks 47-69. This group appears because the value of T_e was above the estimated value. Based on the experiments, we noticed that the best runtimes of AHEAD, for many graphs, occur for values of k in the range $1 \leq k \leq 2$. More specifically, for this work, the adjustments of B_{hdfs} were made to maintain the value of k close to 1. However, the study of the effects of the variation of k should be made in more depth and in different contexts of use of the Hadoop platform.

4.6 Comparison between HEDA and AHEAD

For this comparison test the graphs Co-authors, Enron and one synthetic graph with 50K vertices and 800K edges were used. In this test, the disk space capacity of each of the six compute nodes was increased to 2TB. However, each 2TB disk has a 7.2K RPM of spin rate and 3 GB/s of data transfer rate, making them slower than the 300GB drives used in the previous tests, which especifications are 15K RPM of spin rate and 6 GB/s of data transfer rate. Correctness tests of the version of HEDA algorithm provided by the author [15] were also made. For the HEDA version used, the algorithm runs until the number of iterations provided as an input parameter is achieved. It is expected that the number of iterations provided at the input is greater than the diameter of the graph. The HEDA algorithm returns values of radius and diameter equal to the values returned by sequential algorithms and AHEAD, when the number of iterations passed as parameter is sufficient for the calculation. The minimum number of iterations required is precisely the diameter of the graph, because the HEDA algorithm also makes use of the breadth-first procedure for calculating distances.

Both algorithms were run using the same value of B_{hdfs}. A summary of the measurements collected from resources usage during the processing is shown in Table 4. Fig. 10a, displays the values of memory usage in the processing of a synthetic graph with 50K vertices and 800K edges. The value provided by the Hadoop platform is the sum of the bytes used by all tasks instantiated during a job. For example, in job number 6, AHEAD used 500 GB of memory, this value being the sum of bytes used by each task that has been instantiated during



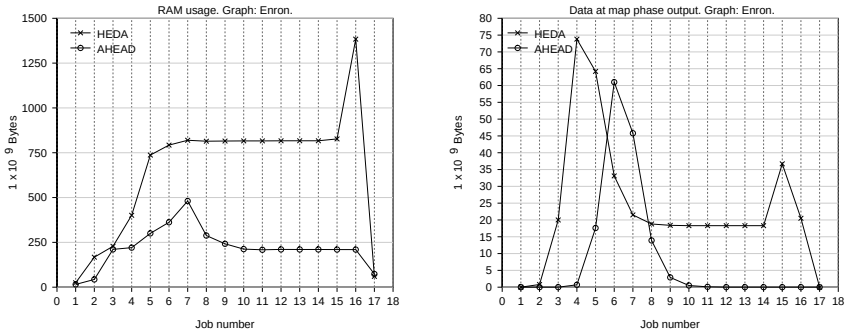
(a) RAM usage per job on hadoop cluster. (b) Disk space usage per job on hadoop cluster.

Fig. 10 Comparison of the use of the aggregated file system space and RAM by the algorithms HEDA and AHEAD during the processing of a synthetic graph.

a cycle *map-shuffle-reduce*, i.e., a job. The bytes used includes not only those used for the creation of a Java virtual machine (*mapper* or *reducer*), but also the memory that the virtual machine itself uses to perform processing. The AHEAD algorithm reduces by 68.8% memory usage compared to the HEDA algorithm when aggregating the values of all jobs. Fig. 10b shows the use of the aggregate disk space on the cluster. The measured values are related to the amount of data in the output of the *map* phase, which is stored on the local disks before the transfer (*shuffle*) to *reducers*. AHEAD algorithm reduces by 34.5% the disk space usage, considering the amount of data stored in each job.

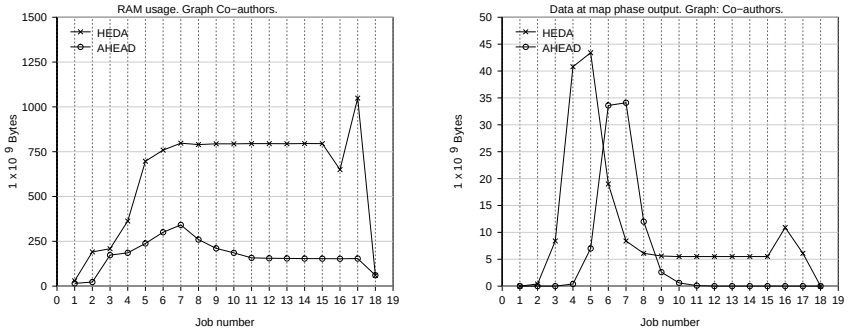
AHEAD and HEDA algorithms use multiple cycles/jobs to process a graph, being the output of a job the input of the next. Therefore the analysis takes into account the sum of the resource usage of all cycles/jobs. The usage of a smaller amount of RAM by the AHEAD compared to the HEDA algorithm was due to a smaller number of Java virtual machines instantiated to perform the same work. According to Equation 2 discussed in subsection 3.3, the smaller number of *mappers* is due to smaller amount of data input. In turn, the smaller amount of data input is a consequence of the implementation of *In-mapper combiner*, *Schimmy* and *Hash Partitioning* techiniques. This result shows that it is possible to process graphs with larger connectivity and number of vertices while using the techniques implemented in the AHEAD algorithm, under the same computational resource.

Figures 11 and 12 show similar results obtained when processing the Enron and the Co-authors graphs, respectively. Besides being able to handle larger graphs, the execution time is reduced, as shown in Table 4, due to the smaller number of input/output operations on the disks of the computing nodes.



(a) RAM usage per job on hadoop cluster. (b) Disk space usage per job on hadoop cluster.

Fig. 11 Comparison of the use of the aggregated file system space and RAM by the algorithms HEDA and AHEAD during the processing of Enron graph.



(a) RAM usage per job on hadoop cluster. (b) Disk space usage per job on hadoop cluster.

Fig. 12 Comparison of the use of the aggregated file system space and RAM by the algorithms HEDA and AHEAD during the processing of Co-authors graph.

5 Conclusions and Future Work

This paper has presented the parallel algorithm AHEAD, developed for efficient and exact computation of centrality measures in large graphs. Experiments were performed to analyze the scalability, speedup and efficiency of the algorithm. Correctness tests were also performed as well as the comparison of its results with the results of sequential and parallel algorithms. Through the results obtained by the proposed experiments, we observed that the AHEAD algorithm showed better performance in the computation of larger graphs, showing superlinear speedup and efficiency greater than unity in these cases. Regarding the behavior of the AHEAD algorithm when varying the size of

the input graph, the results indicate that it has a slow variation in execution time with the increase in number of edges in graphs with high number of connections, namely graphs where the average degree of vertices is higher. Through the implementation of techniques *In-Mapper Combiner* and *Schimmy* for parallel algorithms implemented in the Hadoop enviroment model it was able to reduce up to 72% the use of RAM and up to 64% the use of disk space, compared to HEDA algorithm. The main contribution of this work is the possibility to expand the range of sizes of graphs that can be processed, considering the exact computation and the computational resources available.

The challenge of adapting MapReduce/Hadoop model in the graph context is the need to overcome the costs involved in producing partial results in local disks of the machines belonging to the distributed system. The *mappers* and *reducers* managed by the Hadoop platform do not share globally their main memory addressing, and therefore the distributed computing in this platform is unable to share information about the computation of partitions without temporarily store them on disk. In this sense, the individual memories of each compute node are extended through intensive use of disk, that is, input/output operations. Thus, the disks and the network media are responsible to share data, an activity that must occur in any distributed computing. This cost can be high, especially if the processing involves several cycles *map-shuffle-reduce*. In a system in which the exchange of information on distributed computing occurs with direct memory access between the nodes, performance is certainly better.

The question is therefore to understand in which situations it is justified to use the model MapReduce/Hadoop. The answer lies in the fact that this type of distributed system offers low operation costs to increase its structure. At the programming level, there are simplifications when developing software involving parallel processing, with the developer focusing only on the functions *map* and *reduce* even in activities that involve processing data on the petabyte scale. Furthermore, the platform provides automatic fault management with the reallocation of the computation in other parts of the distributed system.

In the particular case of processing graphs, the advantage is precisely the possibility of working with graphs modeling complex networks. These networks contain a high number of participating members, with equally high number of connections, so that the amount of memory required to process such graphs can justify the use of models as proposed in MapReduce/ Hadoop. Even if the processing of these networks last hours or even days, due to its related costs, we cannot say that the data obtained from the computation is obsolete, since networks of this size change very slowly.

Future work should investigate what is the best ratio between the number of *mappers* and *reducers* in a Hadoop cluster running AHEAD. Moreover, the implementation of the AHEAD can be improved by partitioning the graph in a manner that minimize the number of neighboring vertices on different partitions, at the same time maintaining the partition's sizes relatively equal.

Acknowledgements The authors thank to INCT InWeb, CNPQ and FAPEMIG for their support.

References

1. *Apache Hadoop*. `http://hadoop.apache.org`. Accessed: 2014-02-06
2. Bryant, R.: Data-intensive scalable computing for scientific applications. Computing in Science & Engineering **13**(6), 25–33 (2011)
3. Cohen, J.: Graph Twiddling in a MapReduce World. Computing in Science & Engineering **11**(4), 29–41 (2009)
4. da Cruz, L.C.: An Efficient Parallel Algorithm for Computation of Graph Centrality Metrics. Master's thesis, PPGMMC, CEFET-MG (2013). In portuguese only.
5. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: USENIX Association Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDE'04), pp. 137–149. USENIX ASSOC (2004)
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
7. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci. **31**(2), 182–209 (1985)
8. Freeman, L.C.: Centrality in social networks conceptual clarification. Social Networks **1**(3), 215 – 239 (1979)
9. Gonçalves, M.R.S., Maciel, J.N., Murta, C.D.: Generation of Internet Topology by Reduction of the Original Graph. In: Anais do XXVIII SBRC, pp. 959–972 (2010). In portuguese only.
10. Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy2008), pp. 11–15 (2008)
11. Kang, U., Papadimitriou, S., Sun, J., Tong, H.: Centralities in large networks: Algorithms and observations. In: SIAM International Conference on Data Mining, pp. 119–130. SIAM / Omnipress (2011b)
12. Kang, U., Tsourakakis, C.E., Appel, A.P., Faloutsos, C., Leskovec, J.: Hadi: Mining radii of large graphs. ACM Trans. Knowl. Discov. Data **5**(2), 8:1–8:24 (2011a)
13. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'10, pp. 938–948. Society for Industrial and Applied Mathematics (2010)
14. Lin, J., Schatz, M.: Design patterns for efficient graph algorithms in MapReduce. In: Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG '10, pp. 78–85. ACM (2010)
15. Nascimento, J.P.B.: A Parallel Algorithm for Computation of Centrality in Large Graphs. Master's thesis, PPGMMC, CEFET-MG (2011). In portuguese only.
16. Nascimento, J.P.B., Murta, C.D.: A Parallel Algorithm for Computation of Centrality in Large Graphs. In: Anais do XXX SBRC, pp. 393–406 (2012). In portuguese only.
17. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab (1999)
18. Srirama, S.N., Jakovits, P., Vainikko, E.: Adapting scientific computing problems to clouds using MapReduce. Future Gener. Comput. Syst. **28**(1), 184–192 (2012)