


A Spark Parallel Betweenness Centrality Computation and its Application to Community Detection Problems


Daniel Gómez González

(Complutense University of Madrid, Spain)

 <https://orcid.org/0000-0001-9548-5781> dagomez@estad.ucm.es)


Luis Llana Díaz

(Complutense University of Madrid, Spain)

 <https://orcid.org/0000-0003-1962-1504> llana@ucm.es)

Cristóbal Pareja

(Complutense University of Madrid, Spain)

 <https://orcid.org/0000-0001-7739-0236> cpareja@ucm.es)

Abstract: The Brandes algorithm has the lowest computational complexity for computing the betweenness centrality measures of all nodes or edges in a given graph. Its numerous applications make it one of the most used algorithms in social network analysis. In this work, we provide a parallel version of the algorithm implemented in Spark. The experimental results show that the parallel algorithm scales as the number of cores increases. Finally, we provide a version of the well-known community detection Girvan-Newman algorithm, based on the Spark version of Brandes algorithm.

Keywords: Spark, MapReduce, Social Network Analysis, Centrality measure, Brandes Algorithm, distributed programming

Categories: G.2 G.2.2 J.4 C.1.4 C.2.4

DOI: 10.3897/jucs.80688

1 Introduction

Social network analysis is currently a popular discipline. It is extensively used in a range of applications and real problems. Some common network analysis applications include data aggregation and mining, network propagation modelling, network modelling and sampling, user attribute and behavior analysis, community-maintained resource support, location-based interaction analysis, social sharing and filtering, recommender systems development, and link prediction and entity resolution. One of the major concerns in network analysis is related to the concept of centrality. Centrality measures the importance of a node's position in a network. In social, biological, communication, and transportation networks, among others, it is important to know the relative structural prominence of nodes to identify the key elements in the network (see for example [Borgatti, 2005, Gómez et al., 2003, Newman, 2003, Gómez et al., 2013, Bravetti et al., 2007, Freeman, 1978]). Numerous studies have proposed and analyzed several centrality measures (for more details, see [Borgatti, 2005, Wasserman and Faust, 1994]).

One significant drawback of some standard centrality measures is the processing time needed. Centrality measures must be calculated for large and complex networks in the era of Big Data. The Brandes algorithm [Brandes, 2001] can solve the betweenness centrality

defined in [Freeman, 1978]. But, its sequential implementations cannot successfully deal with large graphs due to the time this would require. The authors in [Balkir et al., 2015] propose a parallel algorithm to compute an approximation of centrality. This centrality measure often needs to compute the shortest paths between all the pairs of nodes in the network, making it highly complex. The exact computation of the betweenness centrality involves solving the APSP problem. The authors in [Yang and Lonardi, 2007] provide a parallel algorithm for computing betweenness centrality. Their solution requires high-end shared symmetric memory multiprocessor architectures.

Another implementation of betweenness centrality can be found in [Madduri et al., 2009, Proutzos and Pingali, 2013a, Wang et al., 2016], which was subsequently improved in [Jamour et al., 2018]. These work use parallel versions of betweenness centrality based on an incremental betweenness centrality computation. They are variations of the Brandes algorithm, not the algorithm itself. The authors in [AlGhamdi et al., 2017] carried out the first major study to attempt to benchmark betweenness centrality calculations. They used a supercomputer to compute exact betweenness centralities for large graphs. Finally, in [Matta et al., 2019] there is an approximated algorithm for computing the betweenness centrality with very good results.

The major contribution of this paper is to provide a parallel version of the Brandes betweenness centrality algorithm [Brandes, 2001] in the Spark framework (<https://spark.apache.org/>). The Brandes algorithm is the fastest known way to compute the well-known centrality measure for links or nodes in a network. We have studied the performance of this algorithm with large-scale data, and have shown how the computational cost decreases almost linearly as the number of cores grows.

We have applied our parallel version of the Brandes algorithm to the Girvan-Newman algorithm [Girvan and Newman, 2002] (from now on GN algorithm). This algorithm uses the computation of betweenness centrality for links. We have shown how the GN algorithm benefits from the use of our parallel implementation of the Brandes algorithm. Additionally, we have compared this version of the GN algorithm with a recent parallel version of the same algorithm called SPB-MRA [Moon et al., 2016].

We have organized this paper as follows. First, we have dedicated the next Section to some preliminary concepts: parallel computing, Spark programming paradigm, centrality measures and community detection problems. Section 3 details the Brandes algorithm for the parallel computing of the node and edge betweenness centrality. Next, in Section 4 we have shown the results achieved by our parallel implementation. Section 5 contains the Girvan-Newman's algorithm using the parallel version of the Brandes algorithm, and finally there are some conclusions in Section 7.

2 Related work: Preliminaries

2.1 Distributed programming with Spark

Sequential algorithms frequently cannot successfully deal with the problems that involve a large amount of data. To overcome this problem, there are new models of programming that distribute the workload over several processors.

Currently, the most common conceptual model for implementing distributed computing with large datasets is MapReduce. This computational model comes from the functional programming paradigm and allows parallel program execution. In this way, it is possible to process massive datasets in a distributed framework. Processing a large dataset under this model means splitting it into multiple small pieces to be processed

(map) separately, and then combining the partial results (reduce) in a tree-like manner. Both steps, processing single chunks of data and compiling them together, can be run in a distributed manner. Both Hadoop [Hadoop] and Spark [Spark] implement this model. Thanks to them, a programmer does not need to deal with the low-level technical details related to the system, such as partitioning datasets into chunks and the communication between the processing units. Programmers thus only have to focus on how to process single portions of data and how to combine the partial results.

Spark is a free, open-source project for large-scale data processing. Although the implementation language is Scala, one can use Spark with several programming languages, including Python. The computational model of Spark is more abstract than Hadoop's model, exploiting the concept of Resilient Distributed Datasets (RDDs). RDDs allow Spark to work on data stored in memory and not only on disk like Hadoop. Spark performance is therefore usually much better than what can be achieved in Hadoop.

2.2 Social network analysis: Centrality measures

Centrality is a complex notion that has a vague definition. Usually, there is a consensus about the idea that a node has high centrality if it can communicate directly with other nodes or if it performs as an intermediary in a communication among other nodes. These ideas are derived from the three first concepts of centrality:

- Degree centrality. We can define this measure as the number of links (edges) emerging from a node. The simplicity of this measure makes it usable in very large and complex networks. For example, in large social networks such as *Facebook*, the popularity of a person is measured by the number of its friends. The degree of centrality focuses on the communication activity level, that is, the number of direct links from a node to its neighborhood. As the ability to increase the number of direct communications grows, the centrality degree of a node increases [Wasserman and Faust, 1994].
- Closeness centrality. Closeness centrality measures the possibility of a given node to communicate with many others using a minimum number of intermediaries. The degree of centrality is a particular case of closeness centrality because only communications without intermediaries are considered (see [Bavelas, 1948] for more details).
- Betweenness centrality. Betweenness centrality measures the possibility of being an intermediary in communication between other nodes. Betweenness is a type of measure of communication control. This measure was first proposed and studied in [Freeman, 1978].

In this work, we focus on the calculation of betweenness centrality. Although there are different betweenness centrality measures based on how the information flows (see [Borgatti, 2005]), the most used measure is most likely the one based on the shortest paths defined by Freeman et al. [Freeman, 1978].

Definition 2.1 Given a graph (V, E) where $V = \{1, \dots, n\}$ represents the set of vertices and E represents the set of direct links, the betweenness centrality of a node $v \in V$ is expressed as:

$$bet(v) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{\pi_{i,j}(v)}{\pi_{i,j}}$$

where $\pi_{i,j}(v)$ represents the number of shortest paths from i to j through node v , and $\pi_{i,j}$ is the number of shortest paths from i to j .

Calculating the betweenness centralities of all vertices in a network requires obtaining the shortest paths between all pairs of vertices, which takes $\Theta(|V|^3)$ time with the Floyd-Warshall algorithm, modified not only to find one path, but also to count all of the shortest paths between two nodes.

On a sparse graph, Johnson's algorithm may be more efficient. The time complexity is $O(|V|^2 \log |V| + |V||E|)$. On unweighted graphs, calculating betweenness centrality takes $O(|V||E|)$ time using the Brandes algorithm [Brandes, 2001].

Let us observe that, on the calculation of the betweenness centrality of all vertices, one usually assumes that the graph is undirected and connected. In the following section, we describe the sequential version of the *Brandes* algorithm.

2.3 Community detection algorithms: Girvan and Newman

One of the most popular algorithms in community detection problems is the Girvan-Newman (GN) algorithm [Girvan and Newman, 2002]. This algorithm is a hierarchical clustering algorithm based on the betweenness centrality idea defined in [Freeman, 1978] for edges instead of nodes. The betweenness measure of an edge represents the number of shortest paths between pairs of nodes that pass through it. The idea behind the algorithm is that edges with strong betweenness frequently connect communities. Thus, removing these edges breaks the graph into communities. So, communities in the graph are obtained by sequentially eliminating edges with the highest betweenness centrality repeatedly.

There are others more efficient, faster and sophisticated community detection algorithms, such as Louvain CNM or community detection based on optimization algorithms (see [Fortunato, 2010] for more details). Nevertheless, due to its simplicity and intuitiveness, GN algorithm is currently used in non-large networks. However, it is still an open problem to design a scalable and accurate parallel GN algorithm to tackle large graphs using a parallel machine with distributed memory [Moon et al., 2016, Nath and Roy, 2018]. In the next section, we briefly describe some attempts to extend the GN algorithm for large graphs using a distributed architecture.

2.4 Parallel centrality measures and community detection algorithms

As mentioned above, the concept of betweenness in social networks is a crucial element in the identification of communities because this centrality measure between two nodes represents the communication stress suffered by these two nodes. When this stress is very high, we can assume that these two nodes should belong to different communities, being each of them intermediaries between these two communities. The idea of the most common betweenness centrality measures is the computation of the minimum paths between each pair of nodes. Some authors [Tan et al., 2009, Green and Bader, 2013, Prountzos and Pingali, 2013b] have parallelized this. However, most of these parallel versions come from algorithms that are not very efficient in the sequential version. For this reason, we propose in this paper a parallelization of the most efficient sequential algorithm for computing betweenness centrality: the Brandes algorithm.

In community detection problems, there are many authors (for example [Guo et al., 2015, Zhang et al., 2016]) who try to find parallel versions of some well-known community detection algorithms. The authors in [Bahmani et al., 2012] propose an algorithm that

Brandes algorithm for computing betweenness centrality in unweighted graphs

```

1: procedure BetweennessCentrality( $G = (V, A)$ )
2:    $c_B[v] \leftarrow 0, v \in V$ 
3:   for  $s \in V$  do
4:      $S \leftarrow$  empty stack
5:      $P[w] \leftarrow$  empty list,  $w \in V$ 
6:      $\sigma[t] \leftarrow 0, t \in V; \sigma[s] \leftarrow 1$ 
7:      $d[t] \leftarrow -1, t \in V; d[s] \leftarrow 0$ 
8:      $Q \leftarrow$  empty queue
9:     enqueue  $s \rightarrow Q$ 
10:    while  $Q$  not empty do
11:      dequeue  $v \leftarrow Q$ 
12:      push  $v \rightarrow S$ 
13:      foreach neighbor  $w$  of  $v$  do
14:        //  $w$  found for the first time?
15:        if  $d[w] < 0$  then
16:          enqueue  $w \rightarrow Q$ 
17:           $d[w] \leftarrow d[v] + 1$ 
18:        end if
19:        // shortest path to  $w$  via  $v$ ?
20:        if  $d[w] = d[v] + 1$  then
21:           $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$ 
22:          append  $v \rightarrow P[w]$ 
23:        end if
24:      end foreach
25:    end while
26:     $\delta[v] \leftarrow 0, v \in V$ 
27:    //  $S$  returns vertices in order of non-increasing distance from  $s$ 
28:    while  $S$  not empty do
29:      pop  $w \leftarrow S$ 
30:      for  $v \in P[w]$  do  $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$ 
31:      if  $w \neq s$  then  $C_B[w] \leftarrow C_B[w] + \delta[w]$ 
32:    end while
33:  end for
34:  return  $c_B$ 
35: end procedure

```

finds the densest subgraph and implements it using the MapReduce model. The authors in [Li et al., 2011] propose a modified parallel version of the label propagation algorithm. However, there are few works dealing with the problem of finding parallel versions of the well-known GN algorithm. The authors in [Yang and Lonardi, 2007] present a parallel version of the GN algorithm using the MPI library. The authors in [Moon et al., 2016] partially solved some of these issues and developed two algorithms (the SPB-MRA and SPB-VCA) that efficiently divide a graph into subgraphs and store them in distributed file systems (in SPB-MRA) or in local file systems (in SPB-VCA) to support big data graphs.

Nevertheless, these two last algorithms present two significant drawbacks, which

```

def node_betweenness_parameterized(G, weight, endpoints):
    def f(s):
        if weight is None: # use BFS
            S, P, sigma = _single_source_shortest_path_basic(G.value, s)
        else: # use Dijkstra's algorithm
            S, P, sigma = \
                _single_source_dijkstra_path_basic(G.value, s, weight)
        # Calculate new contribution:
        if endpoints:
            return \
                _accumulate_endpoints__contribution_from_node(S,
                                                                P, sigma, s)
        else:
            return \
                _accumulate_basic__contribution_from_node(S,
                                                            P, sigma, s)
    return f

def node_betweenness_centrality(G, spark_context,
                                k=None, normalized=True, weight=None,
                                endpoints=False,
                                seed=None):
    brG = spark_context.broadcast(G)
    ncores = int(spark_context.getConf().get('total-executor-cores'))
    betweenness_contr_due_to_a_node = \
        node_betweenness_parameterized(brG, weight, endpoints)

    paralNodes = spark_context.parallelize(G.nodes_iter(), ncores)
    betweenness = paralNodes.flatMap(betweenness_contr_due_to_a_node).\
        reduceByKey(operator.add).collectAsMap()
    _rescale(betweenness, len(G),
             normalized=normalized,
             directed=G.is_directed(),
             k=k)
    return betweenness

```

Figure 1: Python functions that compute the node betweenness

motivate our work. First, the betweenness centrality algorithm that the authors are using to build their parallel version is only moderately efficient, (whilst the Brandes algorithm is the most efficient known algorithm for this problem). Second, the parallel GN version that they proposed is just an approximation method of the original (it does not calculate the same results of the original GN algorithm). Consequently, the performance of the parallel version is unacceptable in some situations.

3 A parallel version of the Brandes algorithm: PBN and PBE

The Brandes algorithm is currently the most efficient sequential algorithm for computing node betweenness centrality [Fortunato, 2010, Green and Bader, 2013]. Several authors [Proutzos and Pingali, 2013b] have studied the possibilities of parallelizing it. Our approach consists in processing multiple source nodes in parallel. This coarse-grained parallelization strategy is possible due to the independence of the information calculated for each source node. Our starting point was the Python version of the Brandes algorithm in the package Networkx [Networkx]. Networkx is a free open source piece of software

```

def edge_betweenness_parameterized(G, weight):
    def f(s):
        if weight is None: # use BFS
            S, P, sigma = _single_source_shortest_path_basic(G, value,
                                                            s)
        else: # use Dijkstra's algorithm
            S, P, sigma = _single_source_dijkstra_path_basic(G, value,
                                                            s, weight)
        # accumulation
        return _accumulate_edges__contribution_from_node(G, value,
                                                         S, P, sigma, s)
    return f

def edge_betweenness centrality(G, spark_context, normalized=True,
                               weight=None):
    brG = spark_context.broadcast(G)
    ncores = int(spark_context.getConf().get('total-executor-cores'))
    betweenness_contr_due_to_an_edge = \
        edge_betweenness_parameterized(brG, weight)
    paralNodes = spark_context.parallelize(G.nodes_iter(),
                                           numSlices=ncores)
    betweenness = paralNodes.flatMap(betweenness_contr_due_to_an_edge)\
        .reduceByKey(operator.add).collectAsMap()

    _rescale_e(betweenness, len(G),
              normalized=normalized,
              directed=G.is_directed())
    return betweenness

```

Figure 2: Python functions that compute the node betweenness

that is carefully documented and broadly used. In abstract, it is the ideal choice in this context.

In order to synthesize the parallel algorithm, we have transformed the original functions `node_betweenness centrality` and `edge_betweenness centrality`, to adapt them to the Spark MapReduce scheme. The functionality of the new version is the same as the original. We defined three functions (`_accumulate_basic`, `_accumulate_endpoints` and `_accumulate_edges`). Each performs their respective contributions to centrality, updating the shared variable `betweenness`, in an accumulative way that is intrinsically sequential. In the original Brandes algorithm, this variable plays the role of a shared, unique variable, and is passed by reference to each of these functions. In fact, this variable has to be updated in each call to these functions, which forces the computation to be sequential. The key is that this shared role is not necessary because each of these functions can calculate its specific contribution. Because these contributions are additive, commutative, and associative, the order in which single contributions are added is irrelevant, so these contributions can be computed in parallel. This intermediate step allows us to transform these functions into three respective, similar functions, that calculate the centrality without performing accumulation. This is a critical change because it avoids a strong dependency with the variable `betweenness`. The calls to the new functions `betweenness centrality` and `edge_betweenness centrality` also need to be adapted to perform the respective accumulations (of nodes and edges to centrality) after the calculations. Specifically, we have parameterized these functions with simple partial evaluation, that is, they are applied to all parameters except the nodes. This basic

Graph name	Number of vertices	Number of Edges
ca-GrQc	5242	14496
ca-HepTh	9877	25998
ca-AstroPh	18772	198110
cit-HepPh	34546	421578
soc-Slashdot0811	77360	905468
sx-superuser	194085	1443339
com-dblp	317080	1049866

Table 1: Input statistics for real world networks used in our experimental study.

Graph Name	Time (secs.)
ca-GrQc	101.05
ca-HepTh	512.19
ca-AstroPh	4975.47
cit-HepPh	26395.99

Table 2: Time required for the sequential version of the Brandes Algorithm

transformation allows us to express the loop traversing the nodes using a MapReduce scheme: the **reduce** operation can combine the partial results as mapping functions yield them. These functions are shown in Figures 1 and 2. The full code can be found in GitHub: https://github.com/LuisLlana/spark_brandes.

4 Results and discussion

4.1 Networks data set

The network data sets that we use in this paper can be found in standard repositories of network datasets such as the Stanford Large Network Dataset Collection [Stanford] or the UCI Network Data Repository [UCI]. Table 1 shows the datasets used in the experimental setting as some characteristics of the network. We have included the time required by the sequential implementation of the Brandes function in the Networkx module in Table 2 on the smaller graphs. Due to the time required we have not included the time for the bigger ones on. We have executed the Brandes Algorithm in the Networkx package on an Intel Core i5 CPU 3.10GHz.

In Figure 5 and Figure 6 we can see the strong relations between the running time with 200 cores and the graph size.

4.2 Scalability

First, we can compare the obtained results with the sequential version of the algorithm. The sequential algorithm takes 26396 seconds (7 hours and 20 minutes) to compute the Brandes algorithm while the parallel version with 50 nodes takes 2438 seconds (39 minutes). That is an 11.24x speedup. Discussing this comparison is difficult because we have different architectures and Spark introduces an overhead in the computations. Nonetheless, this comparison is not as relevant as the scalability that we discuss next.

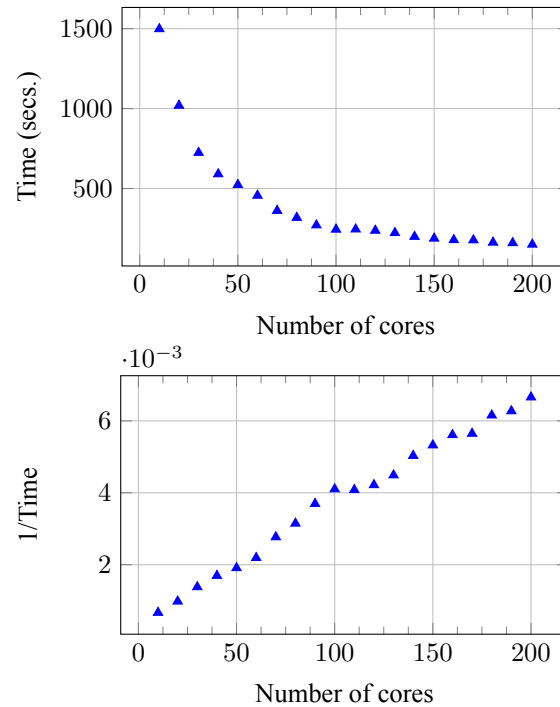


Figure 3: Arxiv Astro Physics graph speed up (ca-AstroPh)

Gaph name	Time (sec)	Vertices	Edges
ca-GrQc	34.70	5242	14496
ca-HepTh	27.67	9877	25998
ca-AstroPh	150.23	18772	198110
cit-HepPh	884.94	34546	421578
soc-Slashdot 905468	3703.87	77360	905468
sx-superuser	40292.89	194085	1443339
com-dblp	112854.35	317080	1049866

Table 3: Results for real world networks in our experimental study (200 cores).

To show the scalability of our algorithm, we have measured the elapsed time for one iteration while varying the number of cores. Table 3 shows the time used by our implementation for the graphs in Table 1 using 200 cores. The full study of scalability has been carried out with the graphs ca-AstroPh and cit-HepPh. We have run our implementation on these graphs with the number of cores varying from 10 to 200 in the case of the ca-AstroPh and from 50 to 200 in the case of cit-HepPh. The results are shown in Figures 3 and 4.

In both figures, the left graph shows the time required to complete the programs while the right one shows the inverse, which gives an idea of the speedup. In both cases, we can observe an almost linear speedup. Figure 7 compares the execution time between

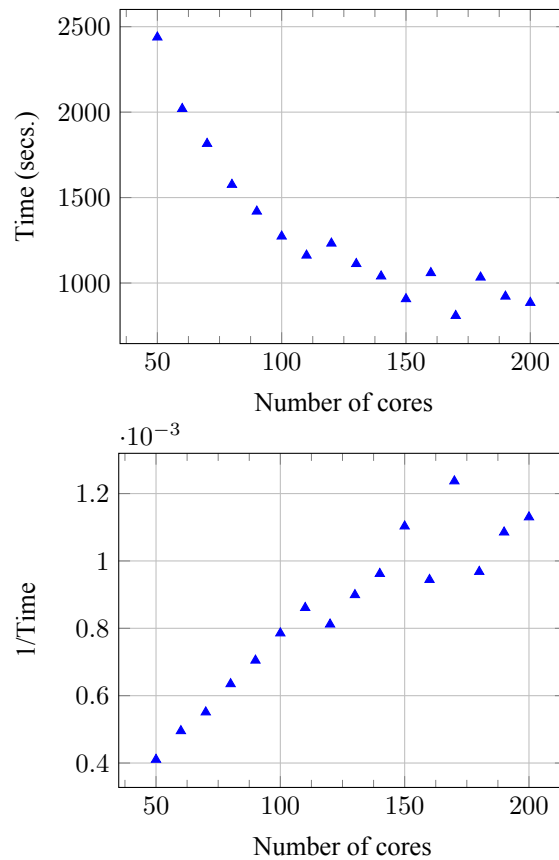


Figure 4: Arxiv High Energy Physics paper citation network (cit-HepPh)

different sized graphs: ca-GrQc (14496 edges), ca-HepTh (25998 edges), ca-AstroPh (198110 edges) and cit-HepPh (421578 edges).

At this point we want to quantify scalability of the parallelization and the ratio between the execution time (Y) and the number of cores (X). In order to achieve this, we present four regression models of the form $Y = f(X)$ for the considered examples.

- $Y = \alpha_0 + \alpha_1 X$. Linear regression.
- $Y = \alpha_0 e^{\alpha_1 X}$. Exponential regression.
- $Y = \alpha_0 + \alpha_1 \log(X)$. Logarithmic regression.
- $Y = \alpha_0 X^{\alpha_1}$. Power regression.

Table 4 shows the different regression results in terms of determination coefficient R^2 . As we can see, the best adjustment for the two examples is the power regression model (both of them with a α_1 negative value). Figure 8 shows the potential regression model and the error function for the ca-AstroPh example.

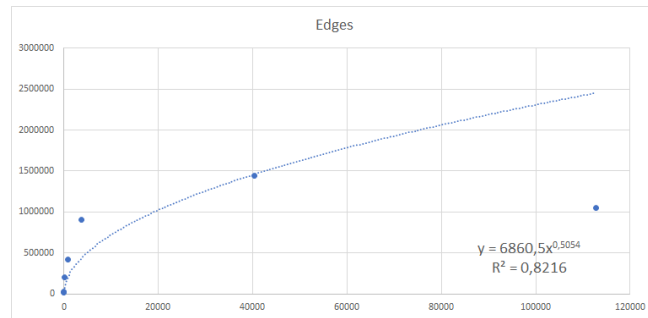


Figure 5: Relation (power regression) between running time with 200 cores and edges sizes of the network.

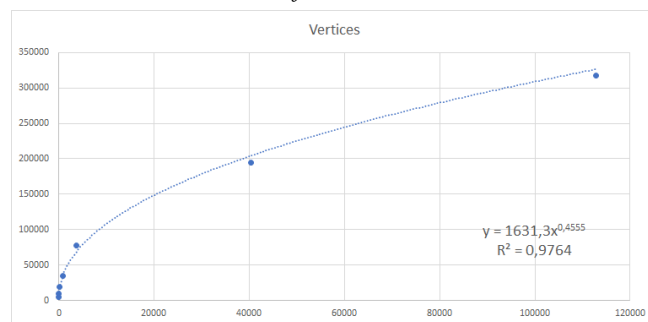


Figure 6: Relation (power regression) running time with 200 cores and edges sizes of the network

To illustrate the reduction in time depending on the number of cores, the next regression model shows us the relationship between the inverse of the execution time ($Z = \frac{1}{Y}$) and the number of cores X . As done previously, the same four regression models were studied for the two examples analyzed, now for $Z = f(X)$.

In Table 5, we show the regression results in terms of determination coefficient R^2 . The best adjustment for the two examples is now the linear regression model (both with a positive α_1 value, but lower than 1).

Figure 9 shows the linear adjustment between the inverse of time and the number of cores for the ca-AstroPh example. It clearly shows that the time used by the algorithm decreases as the number of cores increases. Additionally, the asymptotic relationship within the number of cores and the elapsed time used is just an inverse function. In other words, if we double the number of cores, the time request is halved, asymptotically. According to the experiments, the scalability of our parallel algorithm is almost optimal.

Moreover, Figure 8 shows that the error estimated, clearly and very quickly, tends to zero, underlining the optimality of the parallelization.

4.3 Software and hardware

Our implementation was developed using Python 3.4, Spark 2.1 and the pyspark module 2.1. We ran our programs in an HP ProLian SL390 G7 with 40 nodes running Linux

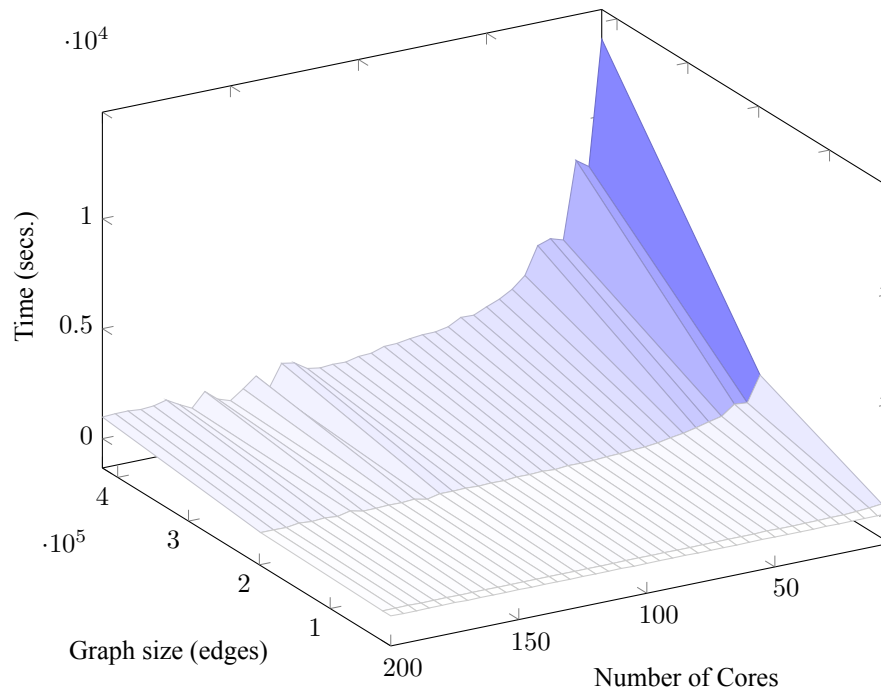


Figure 7: Time comparison between graphs

Example	Regression Model			
	Linear	Exponential	Logarithmic	Power
ca-AstroPh	0,64	0,88	0,91	0,99
cit-HepPh	0,7864	0,85	0,9069	0,9361

Table 4: Determination coefficient for different considered regression models between.

Redhat Enterprise 5.7. Each node had 6 cores; the total RAM of the system was 1200Gb. We used a maximum of 200 executors, each one with limited memory of 2Gb.

5 Application of PB algorithm: a new parallel version of the GN algorithm

In the previous sections, we have described a parallel version of the Brandes algorithm that computes the betweenness centrality for all nodes and edges. Based on it, in this section, we show a possible application to this efficient computational in the field of community detection problems. One of the most well-known algorithms in community detection problem is the Girvan-Newman algorithm. We modified the Jahanbakhsh version [Jahanbakhsh].

Our version of the Girvan-Newman (Figure 10) algorithm stops when there are only small communities. A community is a connected component of the graph. The size of

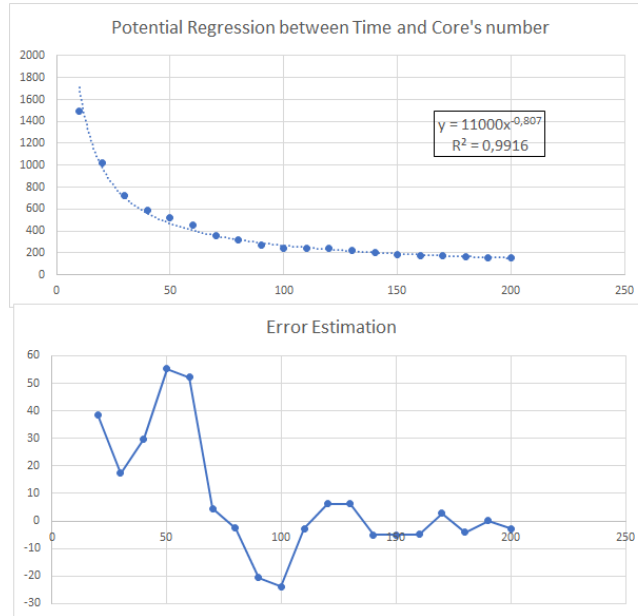


Figure 8: Power regression $Y = 11000 x^{-0,807}$ with $R^2 = 0,9916$.

Example	Regression Model			
	Linear	Exponential	Logarithmic	Power
ca-AstroPh	0,9904	0,87	0,89	0,9901
cit-HepPh	0,9754	0,8338	0,8504	0,931

Table 5: Determination coefficient for different regression models between inverse time (Z) and number of cores (X).

the desired largest community is controlled by the `comm_size` parameter. Similar to the previous algorithms, the full code is in GitHub: https://github.com/LuisLlana/spark_brandes.

5.1 Performance of GN-PBE results

We have conducted some experiments to evaluate the performance of the proposed algorithm. This section describes the details of the experimental setting and its analysis.

We have carried out two studies to show the effectiveness of our parallel version of the GN algorithm (GN-PBE). We have used two medium/large size examples: the ca-GrQc and ca-HepTh networks. It is important to observe that trying to obtain this amount of iterations with undistributed programming would have increased the time considerably, so only the results for 200 cores were taken into account. The GN algorithm has to compute the betweenness centrality iteratively for all edges of the network and then choose the maximum value and eliminate it from the graph.

Figures 11 and 13 show the time of execution in seconds of the entire process during the first 2500 and 9000 iterations of the algorithm for the ca-GrQc and ca-HepTh networks.

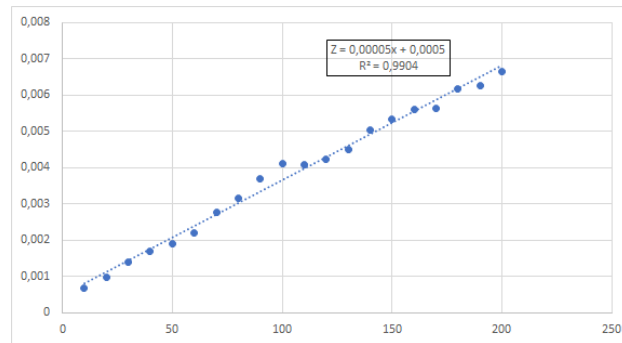


Figure 9: Linear regression $Z = 0.0005 + 0.0005x$ with $R^2 = 0,9916$.

```

def girvan_newman_step(G, spark_context=None):
    from betweenness_centrality_spark \
        import edge_betweenness_centrality

    bw = edge_betweenness_centrality(G, spark_context=spark_context)
    edges = list(bw.keys())
    edges_bw = list(bw.values())
    bw_max = max(edges_bw)
    pos_max = edges_bw.index(bw_max)
    u, v = edges[pos_max]
    G.remove_edge(u,v)
    return u, v, bw_max

def girvan_newman(G, comm_size, spark_context=None, outfile=sys.stdout):
    total_edges = G.number_of_edges()
    ini_time = time.time()
    components, num = find_big_components(comm_size, G)
    show_info(outfile, ini_time, 0, comm_size, components, num)
    while len(components) > 0:
        removed = girvan_newman_step(G, spark_context)
        total_edges -= 1
        components, num = find_big_components(comm_size, G)
        show_info(outfile, ini_time, removed, comm_size, components, num)
    return G

```

Figure 10: Girvan-Newman algorithm using the parallel version of the Brandes algorithm

In both graphs, we can see how time elapses more slowly from one iteration to another as we increase the number of iterations. A more in-depth analysis of this phenomenon can be observed in the three graph in Figure 12 and Figure 14.

Figure 12 shows a notable time reduction around the 600th iteration. This is because the graph has been disconnected (when the edge with most betweenness is removed) into two large and homogeneous communities (that is, connected components). After this iteration, the calculation of betweenness centrality is drastically reduced since the centrality measure of a particular edge depends on the connected component to which it belongs.

Another aspect that should be addressed is the observed variability that seems to be

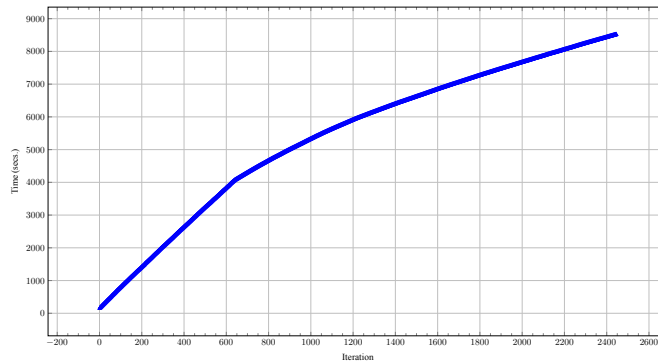


Figure 11: Girvan-Newman iterations for ca-grqc

Groups	N	Clusters													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
24	100	1.84													
25	43	1.84													
23	100	1.92	1.92												
22	100	1.93	1.93												
21	100	1.94	1.94												
20	100	1.97	1.97												
19	100		2.01	2.01											
18	100			2.11	2.11										
17	100			2.13	2.13										
15	100				2.23										
16	100				2.24										
14	100					2.38									
13	100					2.48									
12	100						2.78								
11	100							3.03							
10	100								3.21						
9	100									3.39					
8	100										3.65				
7	100											4.60			
6	100												5.92		
5	100													6.01	
4	100														6.04
2	100														6.16
3	100														6.20
1	100														6.46
p-value		.13	.44	.08	.07	.07	1.00	1.00	1.00	1.00	1.00	1.00	.05	.48	1.00

Table 6: Student-Newman Keuls contrast of GN-PBE algorithm on ca-grqc graph group by 100 iterations

noise in the curve. This variability in the execution-time around an iteration is explained in the box-plot chart. In general, for a set of close iterations, the computation of an iteration is quite stable. However, there are certain outlier iterations in which the execution time is increased (for example, during the first 100 iterations the average execution time is 6.5 seconds, but we can see other outlier iterations close to 8 seconds). This phenomenon mainly occurs in dense graphs where the computation of all the minimum paths is constant. Sometimes, the algorithm removes an edge that does not disconnect. In these cases, the time increases because the minimum paths are longer.

Figure 14 shows the execution-time behavior of the GB-SBE algorithm for the first 9000 iterations. First, there is no step in which the time is drastically reduced as was the case in the previous example. The tendency in this example is less pronounced than in the previous one. We can identify the outlier process around a controlled group of iterations. We observe that the execution time of an iteration in a group of iterations is very

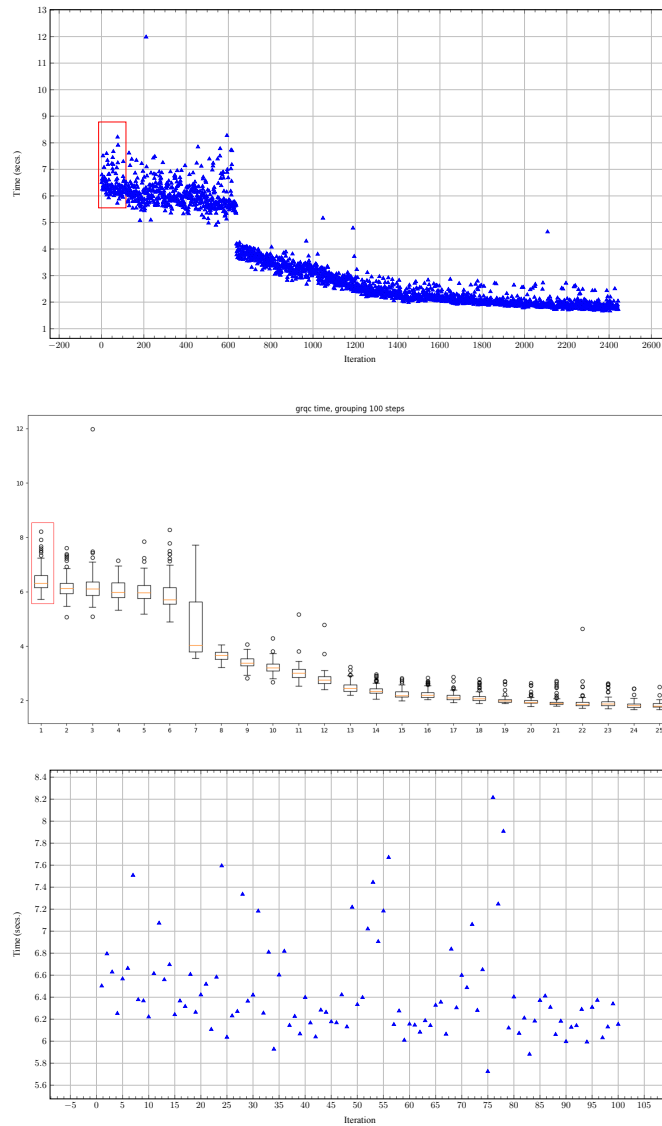


Figure 12: Girvan-Newman delta times for ca-grqc

stable. There are also anomalous increases of time produced by the same phenomenon mentioned above, which are related to the greater complexity of the calculation of the minimum paths in less dense graphs than in dense ones. We can also observe a reduction in execution time as the number of iterations increases.

From both experiments, we can guarantee that time reduction tends to drop as we increase the number of iterations in both problems. To verify this, we performed a Student-Newman-Keuls contrast between groups of 100 and 500 iterations for the two

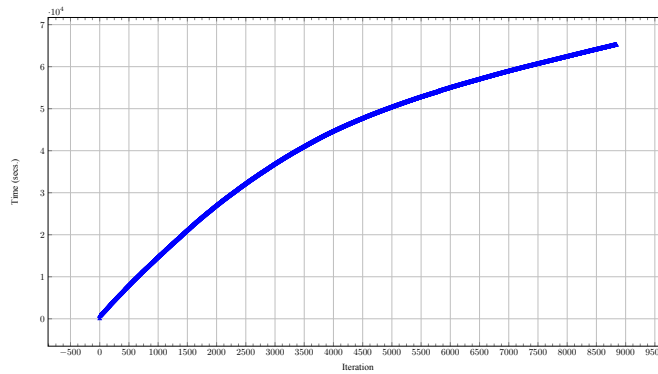


Figure 13: Girvan-Newman iterations for ca-hepth

Group	N	Clusters													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
18	338	3.33													
17	500	3.39													
16	500	3.42													
15	500	3.54													
14	500		3.81												
13	500		3.99												
12	500			4.45											
11	500				4.81										
10	500					5.36									
9	500						6.11								
8	500							7.26							
7	500								8.24						
6	500									9.35					
5	500										10.37				
4	500											11.76			
3	500												12.72		
2	500													13.45	
1	500														14.75
p-value		.14	.07	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 7: Student-Newman Keuls contrast of GN-PBE algorithm on ca-hepth graph group by 500 iterations

examples to see if there is a significant difference between the average execution times in each group. Tables 6 and 7 show the results and also the p-value of each group for both problems. We can observe that the decrease in groups of 100 is significant. This can also be seen in the second example when we consider groups of 500 iterations.

6 Threats to validity

There are several issues in the experiments we have carried out. First, the comparison of the sequential and the parallel versions of the algorithm may be not accurate because the architectures they have been executed are different. This is because our institution policies did not allow us the execution of the sequential version in the cluster. Anyway, the purpose of this research has been to show the scalability of the parallel version.

Another problem with this version is that we need a copy of the graph in each node of the cluster. So, the size of the graph we can consider is limited by the amount of RAM memory of the nodes in the cluster. Assuming that each node can have 8Gb of memory for storing the graph, we could deal with graphs up to 40 million nodes and edges. For bigger graphs we could use GraphX. This library has two problems with respect our

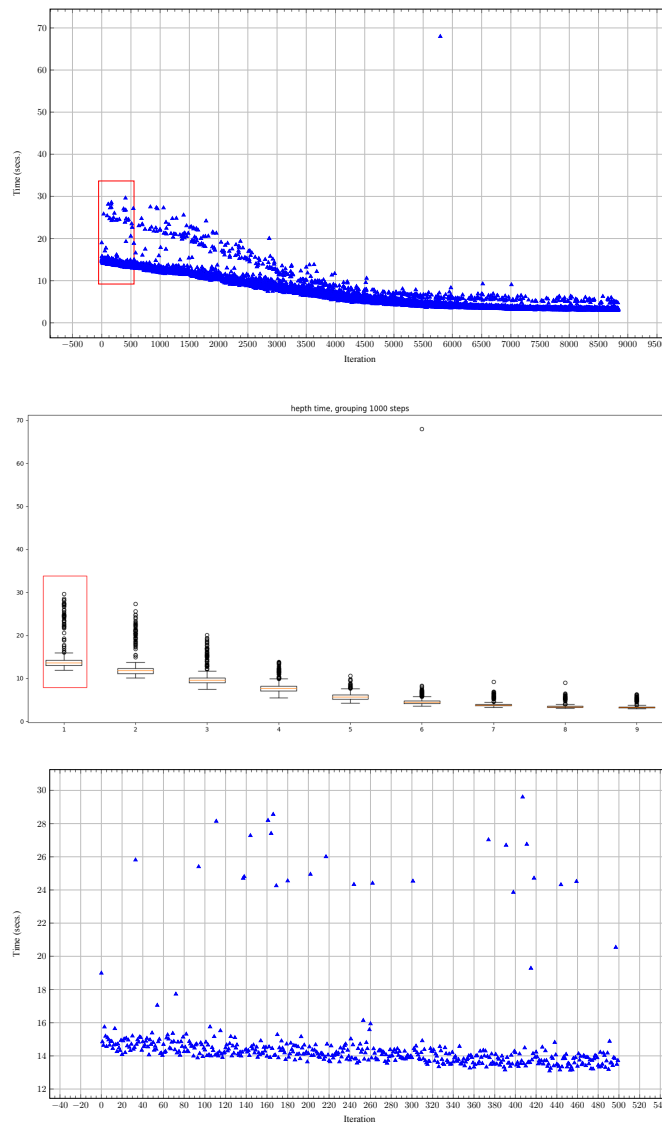


Figure 14: Girvan-Newman delta times for ca-hepth

approach. Firstly, it is only implemented in Scala, and it does not have support for Python yet. Secondly, and more important, it does not implement the Djisktra Algorithm, that is in the base of the Brandes Algorithm.

7 Conclusions and Future Work

The core of this work is the development of a distributed algorithm for the well-known Brandes algorithm, the most efficient algorithm for calculating the betweenness centrality in terms of computational complexity [Fortunato, 2010, Green and Bader, 2013]. The parallel version of this algorithm was implemented and the open source code is freely available in Python. Brandes algorithm parallelization was possible because the contributions to centrality was additive, and these can then be calculated using the MapReduce scheme. We synthesized the respective functions for each contribution by partial evaluation and straightforward functional transformation techniques. The experimental results showed that the performance of the algorithm increases with the number of cores. Specifically, the time decreases proportionally to the number of cores, so we can state that the parallelization achieved is nearly optimal.

In addition to the parallelization of the Brandes algorithm, we provided in this paper a parallel version of one of the best-known community detection algorithms: the Girvan-Newman (GN) algorithm. The GN algorithm is based on the calculation of the betweenness centrality measure. As far as we know, few attempts have been made to parallelize the GN algorithm. One of these attempts was provided in the SPB-MRA algorithm developed some years ago. It is important to observe that although the scalability properties of the SPB-MRA algorithm are excellent, it is not fair to compare the GN-PBE algorithm provided here with the SPB-MRA for different reasons. The SPB-MRA algorithm does not reproduce the results obtained by the GN algorithm, and as the authors point out, only a high percentage of similar edges are removed from the graph. Therefore, the results provided by the SPB-MRA algorithm could be unacceptable in terms of modularity or in detecting good communities. In this sense, the algorithm developed here (GN-PBE) reproduces the same results as the original GN algorithm, but drastically reduces the execution time. In this article, the first thousand iterations of the GN algorithm were obtained for two classic, medium/large size examples. These results were possible due to the GN-PBE version, since the execution time of the original GN algorithm would have been unacceptable.

Acknowledgments

This work has been supported by State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant RTI2018-093608-B-C31 (FAME), the Comunidad de Madrid under grant S2018/TCS-4314 (FORTE-CM) co-funded by EIE Funds of the European Union, and the Madrid Government (Comunidad de Madrid-Spain) under the Multiannual Agreement with the Complutense University as part of the Program to Stimulate Research for Young Doctors in the context of the V PRICIT (Regional Programme of Research and Technological Innovation) under grant PR65/19-22452.”

References

- [AlGhamdi et al., 2017] AlGhamdi, Z., Jamour, F. T., Skiadopoulos, S., and Kalnis, P. (2017). A benchmark for betweenness centrality approximation algorithms on large graphs. In Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017, pages 6:1–6:12. ACM.
- [Bahmani et al., 2012] Bahmani, B., Kumar, R., and Vassilvitskii, S. (2012). Densest subgraph in streaming and mapreduce. *Proc. VLDB Endow.*, 5(5):454–465.

- [Balkir et al., 2015] Balkir, A. S., Oktay, H., and Foster, I. T. (2015). Estimating graph distance and centrality on shared nothing architectures. *Concurr. Comput. Pract. Exp.*, 27(14):3587–3613.
- [Bavelas, 1948] Bavelas, A. (1948). A mathematical model for group structures. *Human organization*, 7(3):16–30.
- [Borgatti, 2005] Borgatti, S. P. (2005). Centrality and network flow. *Soc. Networks*, 27(1):55–71.
- [Brandes, 2001] Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177.
- [Bravetti et al., 2007] Bravetti, M., Gilmore, S., Guidi, C., and Tribastone, M. (2007). Replicating web services for scalability. In Barthe, G. and Fournet, C., editors, *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*, pages 204–221. Springer.
- [Fortunato, 2010] Fortunato, S. (2010). Community detection in graphs. *Physics reports*, 486(3-5):75–174.
- [Freeman, 1978] Freeman, L. C. (1978). Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239.
- [Girvan and Newman, 2002] Girvan, M. and Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826.
- [Gómez et al., 2013] Gómez, D., Figueira, J. R., and Eusébio, A. (2013). Modeling centrality measures in social network analysis using bi-criteria network flow optimization problems. *Eur. J. Oper. Res.*, 226(2):354–365.
- [Gómez et al., 2003] Gómez, D., González-Arangüena, E., Manuel, C., Owen, G., del Pozo, M., and Tejada, J. (2003). Centrality and power in social networks: a game theoretic approach. *Math. Soc. Sci.*, 46(1):27–54.
- [Green and Bader, 2013] Green, O. and Bader, D. A. (2013). Faster betweenness centrality based on data structure experimentation. In Alexandrov, V. N., Lees, M., Krzhizhanovskaya, V. V., Dongarra, J. J., and Sloot, P. M. A., editors, *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, volume 18 of *Procedia Computer Science*, pages 399–408. Elsevier.
- [Guo et al., 2015] Guo, K., Guo, W., Chen, Y., Qiu, Q., and Zhang, Q. (2015). Community discovery by propagating local and global information based on the mapreduce model. *Inf. Sci.*, 323:73–93.
- [Hadoop] Hadoop. Hadoop. <https://hadoop.apache.org/>.
- [Jahanbakhsh] Jahanbakhsh, K. Python implementation of the girvan newman algorithm. <https://github.com/kjahan/community/blob/master/cmt.py>.
- [Jamour et al., 2018] Jamour, F. T., Skiadopoulos, S., and Kalnis, P. (2018). Parallel algorithm for incremental betweenness centrality on large graphs. *IEEE Trans. Parallel Distributed Syst.*, 29(3):659–672.
- [Li et al., 2011] Li, Q., Wang, Z., Wang, W., Liu, Y., Wang, P., and Yu, T. (2011). LI-MR: A local iteration map/reduce model and its application to mine community structure in large-scale networks. In Spiliopoulou, M., Wang, H., Cook, D. J., Pei, J., Wang, W., Zaïane, O. R., and Wu, X., editors, *Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on, Vancouver, BC, Canada, December 11, 2011*, pages 174–179. IEEE Computer Society.
- [Madduri et al., 2009] Madduri, K., Ediger, D., Jiang, K., Bader, D. A., and Chavarría-Miranda, D. G. (2009). A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–8. IEEE.
- [Matta et al., 2019] Matta, J., Ercal, G., and Sinha, K. (2019). Comparing the speed and accuracy of approaches to betweenness centrality approximation. *Computational Social Networks*, 6(1):2.

- [Moon et al., 2016] Moon, S., Lee, J., Kang, M., Choy, M., and Lee, J. (2016). Parallel community detection on large graphs with mapreduce and graphchi. *Data Knowl. Eng.*, 104:17–31.
- [Nath and Roy, 2018] Nath, K. and Roy, S. (2018). A parallel approach to detect communities in evolving networks. In Mondal, A., Gupta, H., Srivastava, J., Reddy, P. K., and Somayajulu, D. V. L. N., editors, *Big Data Analytics - 6th International Conference, BDA 2018, Warangal, India, December 18-21, 2018, Proceedings*, volume 11297 of *Lecture Notes in Computer Science*, pages 188–203. Springer.
- [Networkx] Networkx. Networkx: Software for complex networks. <https://networkx.github.io/>.
- [Newman, 2003] Newman, M. E. J. (2003). The structure and function of complex networks. *SIAM Rev.*, 45(2):167–256.
- [Prountzos and Pingali, 2013a] Prountzos, D. and Pingali, K. (2013a). Betweenness centrality: algorithms and implementations. In Nicolau, A., Shen, X., Amarasinghe, S. P., and Vuduc, R. W., editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, February 23-27, 2013*, pages 35–46. ACM.
- [Prountzos and Pingali, 2013b] Prountzos, D. and Pingali, K. (2013b). Betweenness centrality: algorithms and implementations. In Nicolau, A., Shen, X., Amarasinghe, S. P., and Vuduc, R. W., editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, February 23-27, 2013*, pages 35–46. ACM.
- [Spark] Spark. <https://spark.apache.org/>.
- [Stanford] Stanford large network dataset collection. <https://snap.stanford.edu/data/>.
- [Tan et al., 2009] Tan, G., Tu, D., and Sun, N. (2009). A parallel algorithm for computing betweenness centrality. In *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*, pages 340–347. IEEE Computer Society.
- [UCI] UCI. The uci network data repository. <http://networkdata.ics.uci.edu/>.
- [Wang et al., 2016] Wang, Y., Davidson, A. A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. (2016). Gunrock: a high-performance graph processing library on the GPU. In Asenjo, R. and Harris, T., editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 11:1–11:12. ACM.
- [Wasserman and Faust, 1994] Wasserman, S. and Faust, K. (1994). *Social Network Analysis: Methods and Applications*. Cambridge University Press.
- [Yang and Lonardi, 2007] Yang, Q. and Lonardi, S. (2007). A parallel edge-betweenness clustering tool for protein-protein interaction networks. *Int. J. Data Min. Bioinform.*, 1(3):241–247.
- [Zhang et al., 2016] Zhang, Q., Qiu, Q., Guo, W., Guo, K., and Xiong, N. (2016). A social community detection algorithm based on parallel grey label propagation. *Comput. Networks*, 107:133–143.