Distributed and Parallel Algorithm for Computing Betweenness Centrality

Mirlayne Campuzano-Alvarez(✉) and Adrian Fonseca-Bruzón

Center for Pattern Recognition and Data Mining, Santiago de Cuba, Cuba
{mirlayne,adrian}@cerpamid.co.cu

Abstract. Today, online social networks have millions of users, and continue growing up. For that reason, the graphs generated from these networks usually do not fit into a single machine's memory and the time required for its processing is very large. In particular, computing a centrality measure, like betweenness, is expensive on these graphs. For addressing this challenge, in this paper we present a parallel and distributed algorithm for computing betweenness. Also, we develop a heuristic for reducing the overall time, obtaining a speedup over 80x in the best cases.

Keywords: Online social network · Betweenness centrality · MPI

1 Introduction

Nowadays, online social networks have gained a huge popularity among people all around the world. An important task in Social Networks Analysis (SNA) is to discover the most prominent users into a social network by using betweenness centrality. This measure determines the frequency that a node or an edge acts as a bridge in the shortest paths in the network.

However, in spite of the massive importance of betweenness, its computation is expensive in large networks. For that reason, several strategies have been proposed in the literature to accelerate this process. Some approaches make a preprocessing of the graph by identifying vertexes with *strategical positions* which can be removed or merged, thus reducing the number of nodes to visit when the shortest paths are calculated [6].

However, due to the fast grow up of online social networks, nowadays previous techniques are not enough, because the time cost can reach several days and the main memory of a single machine is not enough to support the network representation.

To tackle those problems, in this paper we propose a parallel and distributed algorithm for computing betweenness using MPI. Also, we propose a modification that allows our algorithm speed up when it is used in sparse networks. The experimental results show that our proposal is suitable to process large networks and it can be adapted according to the available resources.

DOI: 10.1007/978-3-319-52277-7_23

The rest of the paper is organized as follows. First, we expose some definitions related with graph and betweenness concepts. Then, is given a brief description of previous works, including an explanation of the sequential algorithms used. Next, we explain our proposal. Finally, we describe the experimental environment, present the results and make an analysis of them.

2 Background

Commonly, social networks are represented by a graph $G = (V, E)$ where V is a set of nodes, $n = |V|$ and E a set of edges, $m = |E|$. In this paper we use unweighted and undirected graphs. Let d_s be the distance array of the shortest paths from the source vertex s to each node of the graph. We denote $\sigma_{st}(v)$ as the number of the shortest paths from s to t that pass through v and σ_{st} the number of the shortest paths from s to t, $\forall t \in V$.

Betweenness is considered a medial centrality measure, because all walks passing through a node are considered [1]. Formally, it is defined as:

$$Cb(v) = \sum_{\substack{s \neq v, v \neq t \\ s,v,t \in V}} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

The betweenness computation involves two main steps: determining the shortest paths and computing the betweenness values. The first step implies to determine a Directed Acyclic Graph (DAG) for a selected source node s. The second step, consists in the computation of the betweenness value for each node in the DAG previously computed. This process must be repeated taking every node of graph as source node.

According to Eq. 1, betweenness is costly to compute because requires $\mathcal{O}(n^3)$ run-time. In large networks this implies that compute exact betweenness of all nodes can take hours, even days. For that reason, have been proposed algorithms and methods to reduce the time required for its computing.

The best known sequential algorithm was proposed by Brandes in [3]. The author defined the dependency of a node v for a given source s as is shown in Eq. 2, where $P_s(w)$ is the set of predecessors of w in the DAG rooted by s:

$$\delta_{s\bullet}(v) = \sum_{w: v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * (1 + \delta_{s\bullet}(w)) \tag{2}$$

Then, the betweenness value of a node v will be the sum of $\delta(v)$ for each $s \in V, s \neq v$. Using Eq. 2, the running time of the algorithm was reduced to $\mathcal{O}(nm)$ for unweighted graphs. But this reduction in the time cost is not enough for large social networks.

In [2] was proposed a new algorithm which speed up the Brandes's. This method take advantage of the sparsity of social networks identifying the nodes of degree 1. The authors point out that those that had none value of betweenness, however their presence is influential in the betweenness value of their only neighbor. Thus, they compute the betweenness value of the neighbors of the "hanging" nodes once. Then, those nodes can be removed and the process is repeated for residual graph until there are not more nodes with degree 1. The betweenness value of the removed nodes is computed as follows: $Cb(v) = Cb(v) + (n - p(v) - p(u) - 2) * (p(u) + 1)$, where p is an array which store in the position $p(i)$ the number of nodes removed from the sub-tree rooted i.

To keep correctness of Brandes's algorithm they modified the dependency formula as is shown below:

$$\delta_{s\bullet}(v) = \sum_{w:v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * (1 + \delta_{s\bullet}(w) + p(w)) \tag{3}$$

Finally, the betweenness value of the remaining nodes of the graph is computed as follows:

$$Cb(w) = Cb(w) + \delta(w) * (p(s) + 1), w \neq s \tag{4}$$

In [4] was proposed a distributed algorithm to compute betweenness using successor sets instead of predecessor sets. They used the Δ-stepping algorithm during the shortest path traversal [5], allowing use their proposal for weighted and unweighed graphs as well. This is a space efficient approach incorporated at present in the Parallel Boost Graph Library. However, as the authors themselves point out on the experimental results, the algorithm is faster than the sequential version when are available 16 processors or more. Besides, they do not propose any heuristic to distributed nodes, so they need to search across processors where are store vertexes every time they send a message.

3 Our Proposal

In this paper, we propose a solution to reduce the time required for computing the shortest paths using a distributed schema. First, we distribute the nodes of the graph using a round robin strategy across the available processors. Then, we determine the DAGs rooted by the subset of nodes which belong to every processor. Next, we determine the number of the shortest paths that pass through the nodes that lie in the DAGs previously determined. Finally, we compute the dependency values and the betweenness contribution of the nodes reached in the DAGs.

In our case, we opted for distributing the network nodes across the p available processors. Each node is tagged with a unique id $(0 \leq id < n)$ and they will be allocated to the processor r if $id \mod p = r$. Using this strategy, each processor stores a subset of nodes of the graph and we are able to know in which processor the nodes are allocated.

When the shortest paths are computed, it is necessary to visit all the reachable nodes. Due to the vertexes are distributed in multiple processors, it is necessary to send messages requesting the needed nodes and receive those nodes and their neighbors. This situation might imply a lot of communications and thus an increase of the time cost compared with the sequential version. To avoid that problem, we propose the use of two threads on each processor. The first one do the real computation, and also it demands and receives the needed nodes. The second thread attends requests, builds the array of neighbors and send the nodes in demand along its neighbors to the processor which need them. Thus, we overlap some computations and communications. In this way the over all time cost is reduced because the overlapping of both the computation and communications process.

As we model the network as an unweighted and undirected graph, we use the BFS algorithm to determine the shortest paths. This algorithm performs a breadth search starting at a root node s_i, by means, it visits the nodes by levels and do not pass to next level until are visited all nodes at the current level. This process must be repeated taking each node of the graph as root. As we have the graph distributed across multiple processors, we are able to discover in parallel the DAG corresponding to a source node s on each processor. However, we notice that various s_i at the same processor might need the same nodes in the same level of BFS algorithm. For that reason, we propose to expand the DAGs corresponding to several s_i at the same time. The number of DAGs simultaneously expanded is controlled by the parameter ch. This strategy reduces communications which leads thrift of time.

Nevertheless, we need to store the DAGs corresponding to each source node and this implies keep loaded in memory a lot of information, because apart from the $S_s(v)$, we need the σ and d_s arrays and a stack S, which stores the nodes in non-increased order from the root, for each source node. Due to the amount of information we need to store, is necessary to minimize the number of structures used in our algorithms. For that reason, we opted for obtaining the number of the shortest paths σ later, because this structure is not needed to discover the shortest paths and it can be deduced from the successors array.

We can deduce the nodes that will be visited at next level from the array d_s of the following way: the first thread only requests those nodes which are not visited yet and that are going to be reachable at next level. This is made by checking the arrays d_s, which are updated before a node is visited. And this process is repeated until there are no more vertexes to process.

Once there have been determined the DAGs, the dependency value of each node is calculated. Then, we compute the dependency value and finally a partial betweenness. That procedure is made in each processor in parallel.

Finally, when the contribution to betweenness of all nodes of the graph has been computed and stored in the δ arrays, we are able to compute the final score. To get the betweenness value of each node, we perform a reduction of the partial values.

3.1 Modification for Sparse Networks

As social networks often are distinguished by their sparsity, this property can be taking in advantage to speed up the betweenness computation. This was made by using the idea of the SPVB algorithm, in [2]. In our approach, we modified our previous distributed algorithm by adding a preprocessing step. First, we look up, in parallel, for all connected components, the number of vertex in each component and which of them belongs to each vertex. Then, we determine, in parallel the nodes of degree 1 for each connected component. For those nodes, we compute the betweenness value that they aport to its unique neighbor and finally those nodes are removed. With the residual graph, we repeat this step, until there are not more nodes of degree 1. Also, we removed the nodes of degree 0. When is finished the pre-processing step, we compute the shortest paths as we explained before and employed Eqs. 3 and 4 to keep correctness of betweenness values.

3.2 Analysis

We consider the worse of the cases to make our analysis, and we focus in the stage of computing the shortest paths, because is there where we propose the major changes. As we process several source nodes at the same time, this process depends of the parameter ch, and the minimum value that it can take is 1, which means that the process must be repeated by taking a single source node s each time, i.e. n/p times for each processor. Then, to select the nodes in the shortest paths, visit the n nodes of the graph at most $diam$ times, where $diam$ is the diameter of the network.

The amount of memory consumed on each processor depends of the parameter ch, because we almost use the same structures than the Brandes's algorithm, but for several source nodes at the same time. Also, as we employ threads, the processes share all the data.

4 Experiments

For validating our proposal, we conducted several experiments with different datasets corresponding to real networks. We analyze the performance of our proposal in comparison with Brandes's algorithm. Also, we make an analysis of the rate between computing time and communications time in our method in relation to the number of processor employed.

Table 1. Datasets from standford large network collection

Dataset	Nodes	Edges	Average	% Hang
soc-Slashdot0922	82168	948464	11.54	2.19
Email-Enron	36692	183831	10.02	31.09
soc-Epinions	75879	508837	6.71	50.84
Email-EuAll	265214	420045	1.58	86.34

To implement our algorithms we use the C++ language programming. To run the experiments, we employed a cluster of 13 machines, each of them with: 4 GB of RAM DDR2, processor DualCore Opteron 2.66 GHz.

We use four well known datasets from Stanford Large Network Dataset Collection[1]. These datasets were selected attending their features, especially the density of the network, as is shown in Table 1. The column Average indicates the average degree of each node. As can be observed, the datasets present different density of edges per nodes. The column % Hang indicates the percent of nodes that can be removed from the graph when the modification for sparse networks is applied.

Table 2. Speedup

Datas	soc-Slashdot0922		Email-Enron		soc-Epinions		Email-EuAll	
Procs	DBB	DSPVB	DBB	DSPVB	DBB	DSPVB	DBB	DSPVB
2	0.75	2.06	0.99	1.86	0.57	2.78	0.38	20.92
3	1.07	2.99	1.36	2.65	0.80	4.00	0.58	26.82
4	1.41	3.98	1.74	3.39	1.00	5.08	0.64	33.60
5	1.68	4.78	2.14	4.10	1.20	6.17	0.70	39.51
6	1.99	5.75	2.49	5.01	1.36	7.58	0.77	45.43
7	2.22	6.50	2.86	5.60	1.55	8.59	0.83	53.45
8	2.59	7.49	3.16	6.25	1.71	9.61	0.92	60.29
9	2.87	8.55	3.56	6.99	1.92	11.01	0.99	61.59
10	3.15	9.28	3.86	7.60	2.10	11.89	1.03	71.94
11	3.41	10.19	4.14	8.36	2.33	12.94	1.16	77.83
12	3.66	10.99	4.48	8.99	2.50	14.11	1.27	80.58
13	3.96	11.72	4.86	9.59	2.68	15.07	1.34	86.29

To evaluate our proposal, we set $ch = 50$ and we use the speedup as measure. Speedup is defined as the ratio between the sequential time and the parallel time. In Table 2 we show the results, we tagged as DBB (Distributed Brandes Betweenness), to our distributed algorithm based in the Brandes one; and DSPVB (Distributed the Shortest Paths Vertex Betweenness) is the modification based in the idea of the SPVB algorithm. As can be observed, the speedup of our DSPVB proposal always gets better behavior than the sequential algorithm and the speedup increases when we augment to number of processors in all cases. In other hands, we noted that when the graph, which represent the social network, is sparse is better to used the DSPVB algorithm, because it can achieve a speed up over 80x, as happened with Email-Euall dataset. It is important to highlight this result, because a special feature of social networks is their sparsity. However, a better performance depends of the network sparsity as well as the number of nodes eliminated on preprocessing step.

[1] https://snap.stanford.edu/data/.

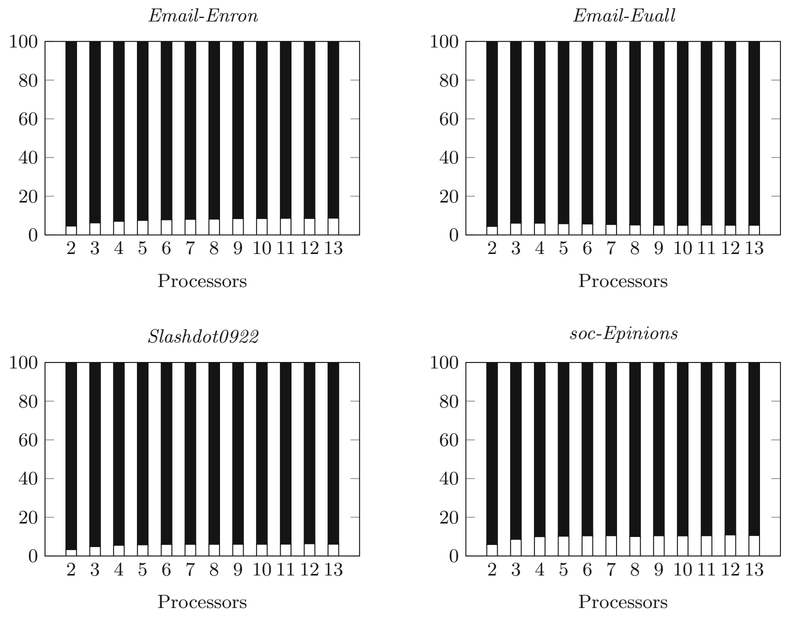

Fig. 1. Time per cent of communications (white bars) and processing (black bars)

We also study the behavior of our proposal with respect to time consuming when it is processing and when it is communicating. In each chunk of 50 sources nodes, from which we expanded the DAGs at the same time in each processor, we measured both times and we computed the average per chunk and the results are shown in the Fig. 1. To establish a comparison, we only show the result of our first proposal, because the other one has a similar behavior.

As we can be observed for all the datasets, the processing time is bigger than the communication time, because in all cases the average time that processors wait for send o receive messages never is greatest than 15 %. That proves that

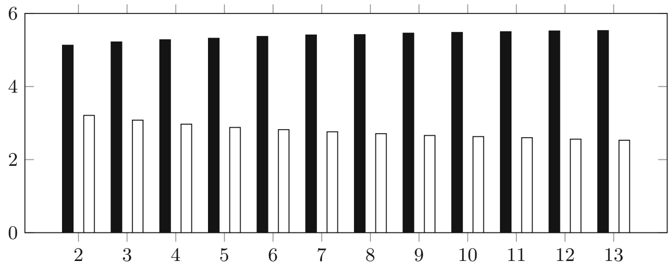

Fig. 2. Time log_{10} of Boost (black bars) and DBB (white bars).

our strategies for reducing the number of message actually work, also proves that we reach to overlap communications and processing by using two threads.

We conducted an experiment to compare our DBB algorithm with the one proposed in [4]. In the Fig. 2, we only show the results of the Email-Enron dataset set, because is the smallest one and there is high difference of time. We can notice that our algorithm is faster than the one implemented in the Boost Library.

5 Conclusions and Future Work

In this paper we presented a distributed and parallel algorithm to compute betweenness centrality. Also, we proposed a version which take advantage of the sparsity of social networks to remove iteratively vertexes of degree 1 and thus reduce the amount of computation. In the experimental results we showed that our proposal is more efficient than the sequential Brandes algorithm. In addition, we overlap communication and processing, thus reducing the delay in the messages interchange due to the use of two threads by processor. Moreover, it necessary to highlight that in our experimental results we observe that our DSPVB algorithm had better performance than the sequential one when we use 2 processors or more for all datasets. Also, in both algorithms, DBB and DSPVB, the speedup increases when we augment the number of processors, which means that our algorithms scale well.

Besides, we proposed to work in a strategy to reduce the memory consumption by not storing the successor arrays for each source node. Moreover, we believe that is possible to reduce the processing time employing a global array for each processor to indicate, during the computing the shortest paths step, the nodes that are going to be visited at next level, instead of traverse the distance arrays.

References

1. Aggarwal, C.C. (ed.): Social Network Data Analytics. Springer, Heidelberg (2011)
2. Baglioni, M., Geraci, F., Pellegrini, M., Lastres, E.: Fast exact computation of betweenness centrality in social networks. In: Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012), pp. 450–456. IEEE Computer Society (2012). http://dx.doi.org/10.1109/ASONAM.2012.79
3. Brandes, U.: A faster algorithm for betweenness centrality. J. Math. Sociol. **25**, 163–177 (2001)
4. Edmonds, N., Hoefler, T., Lumsdaine, A.: A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In: HiPC, pp. 1–10. IEEE Computer Society (2010)
5. Meyer, U., Sanders, P.: Δ-stepping: a parallel single source shortest path algorithm. In: Bilardi, G., Italiano, G.F., Pietracaprina, A., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 393–404. Springer, Heidelberg (1998). doi:10.1007/3-540-68530-8_33
6. Sariyüce, A.E., Saule, E., Kaya, K., Çatalyürek, Ü.V.: Shattering and compressing networks for betweenness centrality. In: SIAM Data Mining Conference (SDM). SIAM (2013)