Big Data Research 10 (2017) 44–52

Contents lists available at ScienceDirect

Big Data Research

www.elsevier.com/locate/bdr

Community Detection Algorithm for Big Social Networks Using Hybrid Architecture

Rahil Sharma *,[1], Suely Oliveira [1]

University of Iowa, Dept. of Computer Science, Iowa city, IA-52246, USA

ARTICLE INFO

Article history:
Received 9 December 2016
Received in revised form 28 July 2017
Accepted 7 October 2017
Available online 26 October 2017

Keywords:
Community detection
Parallel distributed algorithms
Big data
Social networks

ABSTRACT

One of the most relevant and widely studied structural properties of networks is their community structure. Detecting communities is of great importance in social networks where systems are often represented as graphs. With the advent of web-based social networks like Twitter, Facebook and LinkedIn. community detection became even more difficult due to the massive network size, which can reach up to hundreds of millions of vertices and edges. This large graph structured data cannot be processed without using distributed algorithms due to memory constraints of one machine and also the need to achieve high performance. In this paper, we present a novel hybrid (shared + distributed memory) parallel algorithm to efficiently detect high quality communities in massive social networks. For our simulations, we use synthetic graphs ranging from 100K to 16M vertices to show the scalability and quality performance of our algorithm. We also use two massive real world networks: (a) section of Twitter-2010 network having $\approx 41M$ vertices and $\approx 1.4B$ edges (b) UK-2007 (.uk web domain) having $\approx 105M$ vertices and $\approx 3.3B$ edges. Simulation results on MPI setup with 8 compute nodes having 16 cores each show that, upto $\approx 6X$ speedup is achieved for synthetic graphs in detecting communities without compromising the quality of the results.

1. Introduction

One of the most relevant and widely studied structural properties of networks is their community structure. A community in a network is a set of nodes that are densely connected with each other and sparsely connected to the other nodes in the network. Community detection in a network extracts the structural properties of the network [1] and the various interactions in the network [2]. Detecting communities in social networks is of great importance because social networks consists of patterns which can be viewed as independent components, with each component having distinct features and can be detected based on network structure. Some major applications of community detection in social networks are a follows: (i) help to target users for marketing purposes, (ii) provide recommendations to users to connect with other users, join communities or forums, (iii) assist in market basket analysis to help group products likely to be sold together, (iv) help to generate user targeted advertisements.

The increasing size of social networks like Facebook, Twitter, LinkedIn, etc. has made community detection more difficult, with data size which can reach up to billions of vertices and edges. For example, Facebook has $\approx 1.1B$ users and LinkedIn has $\approx 500M$ users. As a result the ability to process this large graph-structured data in memory of a single machine is infeasible due to time and memory constraints. Most of the research in community detection has been focused on shared memory based algorithms on SMP machines and a thorough review of the same is presented in [3]. Where as some fast scalable community detection algorithms [4], [5], [6] which have been developed can only tackle network sizes which can be stored in the RAM of one machine. All of these algorithms adopt sequential, parallel shared-memory and non-distributed architectures. Processing networks with hundreds of millions of vertices and billions of edges require several hundred gigabytes of RAM. To address this challenge, parallel distributed community detection algorithms are necessary. *To avoid any confusion, we use the term cluster only for computer cluster; a part of the computer cluster will be denoted as machine or node, the objects in a network will be denoted as vertex and groups of vertices will be denoted as communities.*

In this paper, we modify and extend our multi-level multi-core (MCML), shared-memory based community detection algorithm [5] also explained in Section 3, to distributed memory parallel frame-

* Corresponding author.
E-mail addresses: rahil-sharma@uiowa.edu (R. Sharma), suely-oliveira@uiowa.edu (S. Oliveira).
[1] Equal contributor.

https://doi.org/10.1016/j.bdr.2017.10.003
2214-5796/

work using Message Passing Interface (MPI). This hybrid (shared + distributed memory) algorithm can process massive social networks to extract high quality communities efficiently. The main challenges we encountered were (1) the initial partitioning of the network and assigning each of these parts to different nodes in the parallel computers in such a way that, when community detection algorithm is applied on each individual node, it should not incur high communication overhead, (2) each node in the parallel computers should intelligently reduce the size of the network partition assigned to it such that, after merging, the entire network should fit in memory of one machine and quality of the communities detected is not compromised.

In this work, we integrate an existing network partitioning algorithm in our hybrid algorithm's flow so that, it will partition the original network into chunks to be distributed across the network of parallel machines, incurring minimum communication overhead between them. In order to minimize the probability of distributing vertices belonging to the same community across different machines, we use a network partitioning algorithm which tries to minimize the inter-partition edges [7]. After network partitioning and distribution, we intelligently reduce the size of every network partition on each machine in such a way that, when merging all the partitions back in the master node, the entire network can fit into the memory of a single master node to which we apply our MCML algorithm to extract high quality communities. All our simulations are done using MPI and OpenMP implementation on the HPC Neon cluster at the University of Iowa. The main contribution of this paper are as follows:

1. We develop a hybrid (shared + distributed memory) community detection algorithm, a modification and extension of our shared memory based MCML algorithm [5], which utilizes multiple cores of multiple machines and scales to hundreds of millions of vertices and edges without compromising quality of the detected communities.
2. We showcase our algorithms' efficiency by using synthetic graphs ranging from 100K up to 16M vertices and also on real world networks like (a) section of Twitter-2010 network having $\approx 41M$ vertices and $\approx 1.4B$ edges (b) UK-2007-05 (.uk web domain) having $\approx 1.2B$ vertices and $\approx 3.2B$ edges.

The structure of the remaining paper is as follows: In Section 2, we present an overview of related work in graph partitioning, community detection and parallel community detection. In Section 3, we describe the proposed core MCML [5] and our hybrid algorithm for large scale community detection. Following this, in Section 4, we discuss and present our experimental environment, datasets used and results, followed by our conclusion in Section 5.

2. Related work

Network partitioning: It aims to divide the network into k-parts in such a way that edge cuts are minimized and each partition roughly has same number of vertices. Most of the network partitioning problems are *NP-Hard* [3]. One group of techniques in graph partitioning relies on optimizing an objective function which is defined as a ratio of number of intra-partition edges to number of inter-partition edges. Another group of partitioning techniques uses multi-level partitioner [7], [8] whose implementation is in METIS and PMETIS library respectively. There exists other partitioning algorithms which scales better than METIS [9], [10] but incur very high communication overhead leading to large runtimes. We plan to utilize parallel METIS to perform our initial graph partitioning, due to its low communication overhead, ease of use and wide availability. The parallel implementation was done using GNU C++ and MPI.

Community Detection: This is an interesting problem in the domain of *graph partitioning*. Interest in community detection problem started with the new *partitioning* approach by [1], [11]; where the edges in the network with the maximum betweenness are removed iteratively, thus splitting the network hierarchically into communities. Similar algorithms were proposed later on, where attributes like 'local quantity' i.e. number of loops of a fixed length containing the given edge [12] and a complex notion of 'information centrality' [13], are used to decide removal of edges. *Hierarchical clustering* is another major technique used for community detection, where based on the similarity between the nodes, an agglomerative technique iteratively groups vertices into communities. There are different existing methods to choose the communities to be merged at each iteration. Algorithms described in [14] and [15] start with all the nodes as individual communities and iteratively merge them to optimize the 'modularity' function. Many other algorithms in the literature of community detection, like ones proposed by [16] and [17] rely heavily on *modularity maximization*. *Label propagation* is another well known technique used for community detection, which finds communities by iteratively spreading labels across the network. Raghavan et al. [6] proposed an algorithm, where each node picks the label in its 1-neighborhood that has the maximum frequency. These labels are permitted to spread synchronously and asynchronously across the network until near stability is attained in the network. This method has some limitations, where large communities dominate the smaller ones in the network, this phenomenon is called 'epidemic spread'. This limitation is tackled in [18]. Liu et al. [19] used affinity propagation, which is a similar approach to label propagation, for finding communities/clusters in images. Some community detection algorithms use *random walks* as a tool. The idea is that, due to the higher density of internal edges, the probability of a random walk staying inside the community is greater than going outside. This approach is used in Walktrap [20] and Infomap [21] algorithms. A thorough review on community detection algorithms for networks is given in [3]. A study presenting evolution and management of interest-based communities formed by humans is shown in [22]. Another interesting application of community detection is shown in [23], where due to the emergence of smart grids which enable bidirectional energy, finding economically motivated Prosumers-Community Groups (PCG) is important.

Parallel Community Detection: Community detection algorithms is a well studied research area, but achieving strong scalability along with detecting high quality communities is an open problem. Most of the past research on community detection has focused on single threaded algorithms. There is a rich and vast literature of such algorithms and the ones based on modularity maximization being the most prominent amongst them [11]. The *Louvain* method which is based on modularity maximization [4] is the most widely used community detection algorithm which can scale to networks with millions of vertices. However, the quality of results obtained deteriorates as the size of the network increases [24]. It is observed that modularity maximization based algorithms are unable to detect small and well-defined communities in large networks [25] [26]. One of the recent parallel algorithms developed to detect disjoint community structures based on maximizing weighted network partitioning is given in [20]. A scalable community detection algorithm, which partitions the network by maximizing the Weighted Community Clustering (WCC), is proposed in [27] which uses community detection metric based on triangle analysis [28]. Some other works which focused on developing parallel implementation for existing community detection heuristics is given in [29]. Recently, [30] proposed a scalable parallel algorithm for community detection, based on label propagation, which is optimized for GPGPU architectures. This algorithm just works on local information which drives the high scalability of this algorithm.

Recent works mentioned above on exploitation of parallelism for community detection has the form of multi-core algorithms for SMP machines i.e. shared memory architecture. In [31], a parallel multi-core Louvain algorithm is proposed which exhibits the above mentioned pitfalls of deteriorating quality of the communities with increasing size of the network. In [32] a parallel version of Infomap is presented which relaxes the concurrency assumption of the original method [21], achieving parallel efficiency of 70%. More literature on shared memory based parallel community detection algorithms is mentioned in Subsection 3.2.

There is minimal literature on distributed algorithms for community detection. In [33], a distributed memory parallel algorithm extending the Louvain method is proposed. Here, the most costly iteration of the algorithm is made embarrassingly parallel without any noticeable loss in final modularity. This approach was validated using an MPI implementation on a High Performance Computing (HPC) cluster. However, the original pitfalls of Louvain's method mentioned above and in [24] prevail. *Hadoop Map-Reduce Model* can be used speed up algorithms which can break down into embarrassingly parallel tasks. In [34], the proposed algorithm is a distributed memory parallel version of the Girvan–Newman algorithm [11]. This version adopts the Map-Reduce framework where it breaks down the algorithm into four embarrassingly parallel tasks: (1) calculating all-pair shortest paths in the network, (2) calculating the edge betweenness for every pair of nodes in the network, (3) k-edges (where k is the number of edges and is user settable parameter) are selected based on edge betweenness and removed, (4) network update for next iteration. The performance results showed that elapsed time decreased almost linearly with the number of reducers. Most simple community detection algorithms which can be broken down into the embarrassingly parallel independent tasks, do not yield high quality communities on real world networks.

We propose to extend our MCML shared memory parallel algorithm [5], to distributed memory parallel framework using the MPI implementation on University of Iowa's Neon HPC cluster, to detect communities in massive networks with high accuracy and attain scalability.

3. Algorithm

In Subsections 3.1 and 3.2 of this section, we shall describe our core shared memory based MCML community detection algorithm which also appears in [5]. This algorithm is used as a subroutine in our hybrid community detection algorithm which is proposed in Subsection 3.3. Our hybrid algorithm utilizes multiple cores of multiple machines and scales to hundreds of millions of vertices and edges without compromising quality of the detected communities.

3.1. Core MCML

The MCML algorithm involves a preprocessing stage, where each edge is assigned a strength based on the topology of the graph. Then based on the strength requirement of the communities, weak edges are removed and coarser graph instances are recursively created by identifying and removing communities, using the node with highest centrality each time. In our algorithm the node with the highest centrality is the node with highest degree. We recursively apply this step until every node is assigned to a community.

3.1.1. Preprocessing: edge strength assignment

The MCML algorithm finds communities in a graph $G(V, E)$ where V represents the nodes/vertices and E represents the edges between the nodes, by assigning strength to the edges initially. It is desirable to assign an edge strength value that most accurately represents the topological structure of the graph in the MCML algorithm. Since we do not have any prior knowledge of the community structure, we assign a strength value to each edge based on the significance of that edge to the other nodes in the graph, and to the nodes at the end points of that edge. For each edge $e(i, j)$ (where i and j are nodes) in the fine graph G, the topological edge strength value $\alpha(i, j)$ assigned to it is the ratio of number of triangles that edge $e(i, j)$ participates in to the total number of triangles containing node i.

If the strength value of an edge $e(i, j)$ is greater than other edges in the 1-neighborhood of i then, node i and node j are more likely to be in the same community. Whereas on the contrary, if edge $e(i, j)$ has lower strength value than most other edges in the 1-neighborhood of i, then node i and node j are less likely to be in the same community. Mathematically,

$$\alpha(i, j) = \frac{t_{(i,j)}}{\sum_{(i,k)}(t_{(i,k)})} \quad ; \quad k \in N_i \tag{1}$$

where N_i is the 1-neighborhood of i, and $t_{(i,j)}$ is the total number of triangles whose sides contain edge (i, j).

The MCML algorithm also works well with weighted graphs, where the edges are assigned weights w_{input} as an input. To get the total weight of an edge, we simply have to take product of the topological edge strength value, with its input weight.

$$\alpha_{total}(i, j) = \alpha(i, j) \times w_{input}(i, j) \tag{2}$$

After this we normalize the edge strength for all the edges, such that they range in between 0 and 1.

3.1.2. Remove weak edges

The procedure of removing the weak edges from the fine graph G is based on the required strength β of the communities. Edges with $\alpha(i, j) < \beta$ are removed. After deleting these edges we might label some nodes as *non-community nodes* i.e., the degree of these nodes is zero. For higher values of β we get a higher number of non-community nodes and more stronger, smaller and significant communities are extracted. Whereas for lower values of β we get smaller number of non-community nodes, and higher number of nodes are assigned a community.

3.1.3. Multilevel coarsening

Let G_i $(i \geq 1)$, be the graph obtained by removing weak edges (i.e., $\alpha(j, k) < \beta$) from G. We apply the following coarsening step recursively to extract meaningful community structure.

Multilevel coarsening We select a node v from G_i, having highest centrality and label it i. Now we distribute this label i assigned to v, to all the neighbors of v, denoted by $N(v)$. We continue distributing labels to the neighbors of all the nodes with label i. We do this until no more neighbors are left to send the label, or the maximum required size of the community is reached. Then we obtain a coarse graph G_{i+1} by removing all the nodes with label i from G_i, along with their associated edges. We continue this process recursively until all the nodes are assigned a community. The communities having number of nodes less than the *minimum number* required in a community, labels all the nodes in that community as non-community nodes. This idea of recursively deleting communities is along the same lines as the one used in [35]. The general schema for the algorithm is shown in Fig. 1.

3.2. MCML—shared memory parallel implementation

Parallel shared-memory based, multi-core implementation, for each stage of our MCML algorithm is described in this section.

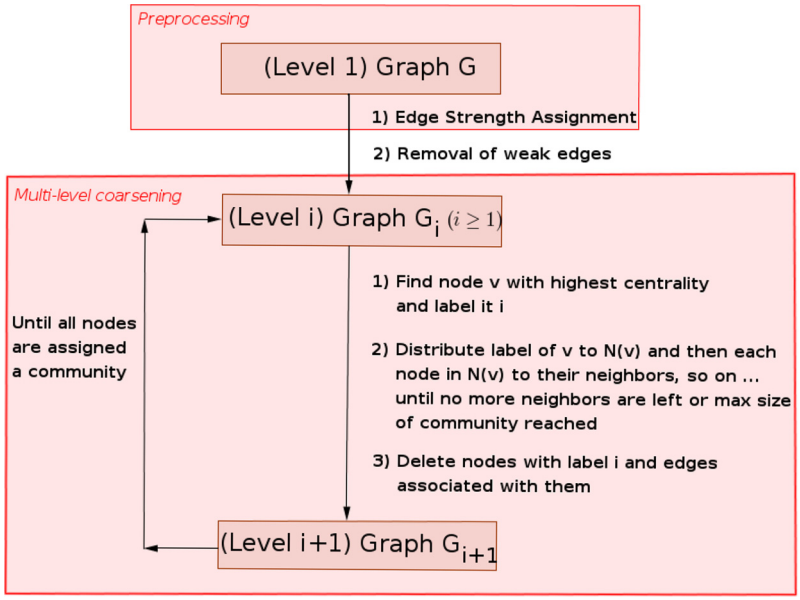

Fig. 1. MCML algorithm's general schema.

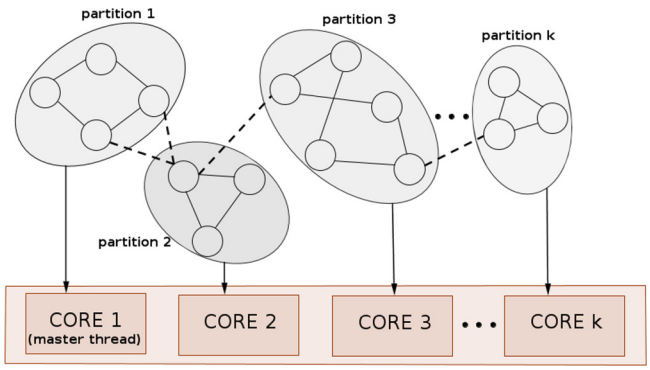

Fig. 2. Parallel preprocessing.

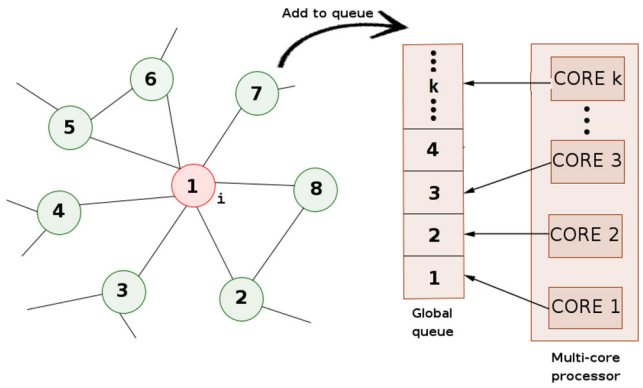

Fig. 3. Parallel multilevel coarsening. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.2.1. Parallel preprocessing

The first step of our algorithm, is to designate a master thread, which divides the graph roughly into k equal parts, where k is the number of cores or threads available, to distribute it across multiple cores. We perform a k partition on the input graph. We can use an existing k way graph partitioning library like KaHIP, METIS, PMETIS, etc. to divide the graph into k parts. The master thread then assigns each of these k parts to k threads individually (including itself), as shown in Fig. 2. Then each thread computes the edge strengths in the part of the graph assigned to them, using Equation (1). The inter-partition edges, which are the dashed edges in Fig. 2, are excluded in this computation. Once all the threads have completed their edge strength assignment computations, the master thread merges the k parts of the graph back together and computes the edge strengths of the previously excluded dashed edges. We parallelize the weak edge removal step in the same fashion. Here we do not have to worry about the inter-partition edges (dashed edges in Fig. 2) because, if they have strength less than the threshold they will be removed, else will be restored, by both the threads they are assigned to.

3.2.2. Parallel multilevel coarsening

In this step of the algorithm, multiple cores are utilized to coarsen the graph. We start by designating a master thread, which first finds the node with highest degree centrality in the graph and labels it i ($i \geq 1$). We then create a global queue, such that all the k cores point to the rear-end of this queue, as shown in Fig. 3. The highest centrality node is then pushed onto this global queue. The master thread is then assigned to this node, based on our construction of the queue. Then the master thread distributes label i to the 1-neighborhood of this node and also add the new nodes it discovers in the 1-neighborhood to the queue. Similarly in the consequent rounds, the threads are assigned nodes from the queue as shown in Fig. 3 and each thread distributes label i to a 1-neighborhood (nodes that have not yet received the label i) of the node assigned to it, along with adding the newly discovered nodes to the queue. So at a given time, there are k nodes that are assigned to k cores in a cyclic fashion, which can simultaneously propagate its labels. In Fig. 3, initially the master thread i.e. core 1 finds node 1 (red node) which is the highest centrality node and adds it to the queue after assigning label i to it. Node 1 is then assigned to core 1, which distributes label i of node 1 to the nodes in its 1-neighborhood (green nodes) which have not yet been labeled. Along with label distribution, it also add these nodes to the queue. Nodes $2, 3, 4, 5, 6, 7$, and 8 are added to the queue. In the second round, nodes $2, 3, 4$, and 5 are assigned to cores $1, 2, 3$, and 4 respectively in a cyclic fashion from the rear-end of the queue. Each of the cores then follow the same steps that was followed by core 1 in the initial round.

We place appropriate *barriers* and *write locks* to the queue in order avoid *race conditions* between threads. This process contin-

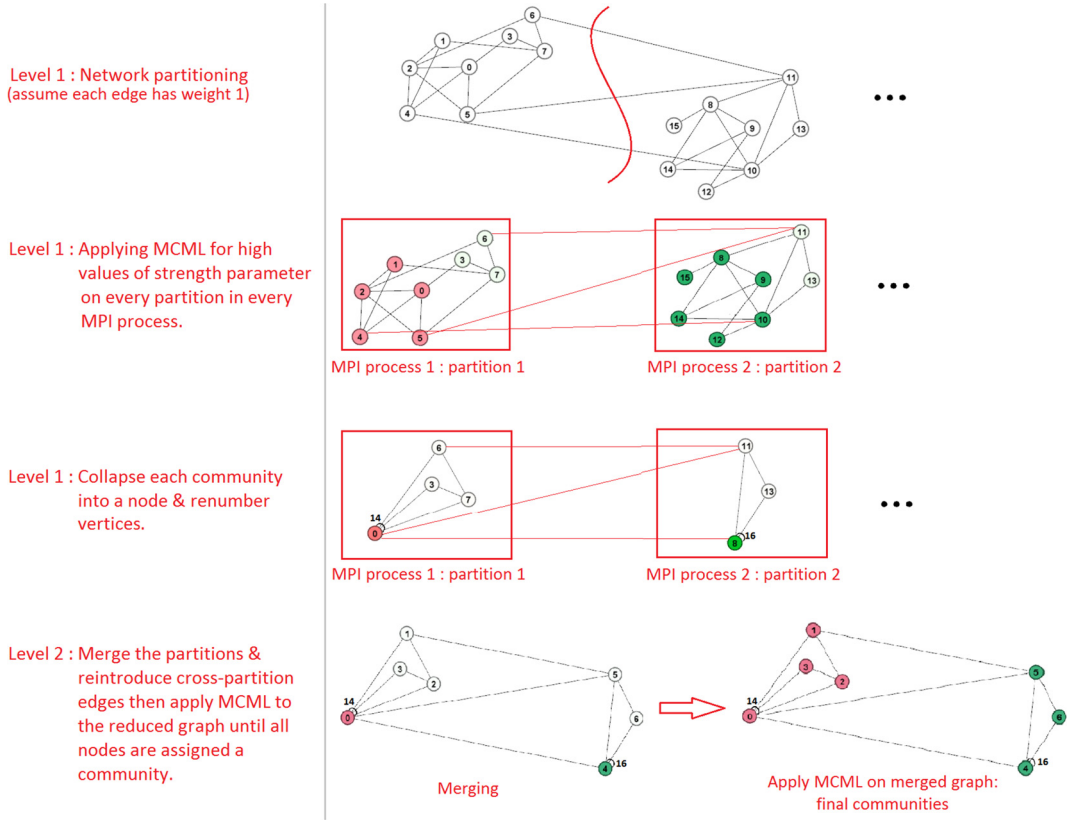

Fig. 4. Example: hybrid algorithm.

ues in cyclic fashion, until the queue is empty (disjoint component found) or maximum desired size of the community is reached. After finding a community, the k threads remove the community with label i from the graph (using a trivial *parallel for loop*) and the same process is iteratively applied on the remaining graph with label $i+1$. We continue this until the algorithm terminates, i.e. all nodes are assigned to a community. Note that we did not perform a graph partition in this stage to avoid nodes of the same community to be assigned to multiple threads.

3.3. Hybrid algorithm

We take advantage of the initial network partitioning when designing parallel distributed community detection algorithms, in order to speed up the processing time by minimizing the communication between processors. This reduces the possibility of vertices in the same community to spread across multiple partitions. We modify our parallel shared memory MCML algorithm presented in Sections 3.1, 3.2 to enable it to adapt the distributed MPI framework for processing massive networks. Our hybrid algorithm follows a multilevel algorithmic framework which includes the following steps (also explained with an example in Fig. 4):

1. **Level 1—network partitioning:** Using network partitioning we aim to split the original network k-ways such that, the number of edge cuts between partitions are minimized and there is a balance in the number of vertices in each partition. Network partitioning is a $\mathcal{NP}$-*Hard* problem [3]. Most of the existing network partitioning techniques use network processing tools such as Apache Giraph [36] which is based on hashing or vertex ordering i.e. random graph partitioning. We use parallel METIS partitioning algorithm (PMETIS) [8] due to its low communication overhead, ease of use and wide availability. We use the k-way partitioning library which divides the network based on minimum edge cuts. The parallel implementation was developed using GNU C++ and MPI.
2. **Level 1—apply MCML on partitions:** Each MPI processor is assigned a partition based on the process identifier, and MCML is applied locally on each processor. Interprocessor communications is allowed using message passing interface across cross-partition edges. MCML is applied for higher value of the strength parameter $\beta \geq 0.8$, where $0 \leq \beta \leq 1$, such that all nodes are not assigned to a community, but nodes with stronger affiliations are put in the same community. After finding these strong communities in each of those partitions we collapse each community into a single node such that, all the intra cluster edges will be represented as a self-loop on that node. Note that we do not collapse nodes residing in the partition on a different machine. This is the most crucial step where we expect to reduce the size of each partition considerably.
3. **Level 1—renumber vertices and merge partitions:** Next step is to merge all the partitions together at the master MPI process. Since all the individual vertices have local numbering, we are required to renumber all the vertices across all partitions in a continuous fashion. We use the following method to renumber vertices before the merging step. Using the *all gather* operation in MPI, each process collects the total number of vertices every other process has. Each MPI process p_i, now has a list of total number of vertices in every other partition $\{N_0, .., N_{i-1}, N_{i+1}, .., N_{\mathcal{P}-1}\}$ where, $\mathcal{P}$ is the total number of MPI processes. It then renumbers its vertices in a way that the ones associated to its partitions start from n_{start_i} which is based on the values of all processes p_j with $j < i$ as follows:

$$n_{start_i} = \sum_{0}^{i-1} N_j$$

Table 1
Random graph datasets.

Vertices	Edges
100,000	422,015
500,000	1,652,471
1,000,000	3,559,759
2,000,000	6,995,154
4,000,000	14,598,778
8,000,000	32,115,764
16,000,000	63,221,980

Once the renumbering is performed, each MPI process sends its partitions to the master MPI process where the merging takes place.

4. **Level 2—MCML:** The merged network represents the level 2 of the original network where the size of the network is reduced significantly. We then apply MCML algorithm again on this level until all the vertices are assigned a community. This step can be performed on a single machine i.e., master MPI process, since the size of the graph is reduced significantly and can completely loaded in to the memory.
Robustness: Our algorithm is robust i.e. can handle failures without affecting the quality of the results. We maintain a global bit associated with each partition and it is set to true only when the partition is completed processing at any compute node. If at the merge state, any of these bits are false we use the unfolded hierarchy from the input file and assign that partition back to the compute queue while other partitions are waiting to be merged. (There are warning messages sent out in this event so that the user will know the reason for the delay.)

4. Experimentation

4.1. Environment

The performance of our hybrid algorithm is evaluated by executing series of experiments on the High Performance Neon Cluster at University of Iowa. We use 8 heterogeneous standard machines each having 64 GB RAM, 16 Xeon Phi cores and 2.6 GHz processor. All the experiments were executed as a single batch command comprising of at most 8 compute machines having 16 cores each. Each experiment is executed 3 times and the average of the results from these runs are reported to preserve accuracy and consistency.

4.2. Datasets

We generate random graphs for our empirical studies to have control over the graph sizes and study the scalability of our algorithm over different graph sizes. We generate these random undirected and unweighted graphs using the same benchmarking package used by Fortunato in [3]. Graph generation using this package also offers fine control over the average and maximum degree distribution, etc. Many similar studies use this package for their empirical studies [33]. The properties of the 7 graphs we generated are shown in the Table 1.

For our empirical studies we also use massive portions of two real world social networks described in Table 2:

(a) **Twitter-2010:** Twitter is a website, owned and operated by Twitter Inc., which offers a social networking and microblogging service, enabling its users to send and read messages called tweets. Tweets are text-based posts of up to 140 characters displayed on the user's profile page. This is a crawl done in [37]. Every node represents a user and there is an edge from node x to node y if x is a follower of y, i.e. edges follow the direction of tweet transmission. This dataset is publicly available from http://law.di.unimi.it/datasets.ph.

Table 2
Real world social network datasets.

Datasets	(a) Twitter-2010	(b) UK-2007-05
No. of vertices	41, 652, 230	105, 896, 555
No. of edges	1, 468, 365, 182	3, 301, 876, 564
Edgelist file size	52.3 GB	110 GB
Average degree	35.253	35.7

(b) **UK-2007-05:** This web based graph is crawled by Boldi et al. [38]. The web-graphs of the 12 snapshots from each of the 12 months of .uk domain have been merged into a single graph. Each node represents a URL and there is an edge between URL x and URL y if the web page of URL x contains URL y. This dataset is publicly available from http://law.di.unimi.it/datasets.php.

4.3. Evaluation

Our empirical studies focuses mainly on analyzing the scalability of our hybrid algorithm and the quality of the results obtained. We scale the problem by increasing the graph size and the number of processor cores. We run the performance analysis which includes the steps 2, 3 and 4 of the hybrid algorithm described in Section 3.3.

In Fig. 5, we see the variation in the total runtime while scaling up the number of processor cores for different graph sizes. We can observe that our algorithm exhibits high scalability for all possible permutations of graph size and processor cores. The graph with $16M$ vertices could only be tested when 16 or more processor cores are used, due to memory constraints of one machine. In Fig. 6, we see that our algorithm achieves $\approx 6X$ speedups for synthetic graphs upto $8M$ vertices. We also see that speedups flatten and start declining for most of the graphs after scaling them past 64 processor cores. This is mainly due to Amdahl's law and increase in communication overhead.

In Fig. 7, we observe that the gap between runtime of parallel implementation with varying processor cores increases as the graph size increases. This shows the high scalability of our hybrid algorithm for large graphs. But we see a decline in this runtime improvement as we scale up to 64–128 processor cores. This is due to the similar reason explained above.

Our hybrid algorithms main goal is to achieve high scalability along with maintaining accuracy of the communities detected. In Fig. 8, we observe the percentage error using the difference in the modularity between sequential run and parallel run of our hybrid algorithm. This error is calculated using the formula:

$$\%Error = abs\left(\frac{mod_{par} - mod_{seq}}{mod_{seq}}\right) \times 100 \tag{3}$$

where mod_{seq} and mod_{par} represents the final modularity obtained by sequential run and parallel run of our hybrid algorithm respectively. We observe that the error percentage decreases as the size of graph increases.

This is an expected phenomenon, since PMETIS partitions the graph into multiple subgraphs by minimizing the number of cross partitioning edges between them i.e. minimum size edge cuts. For small graphs, PMETIS is constrained because of the number of partitions and hence will have to partition the graph with higher cross-partition edges. This leads to higher probability of partitioning the communities across multiple subgraphs. Whereas, for larger graphs this probability decreases since PMETIS is not constrained as much by the number of partitions and can effectively reduce the number of cross-partition edges between subgraphs.

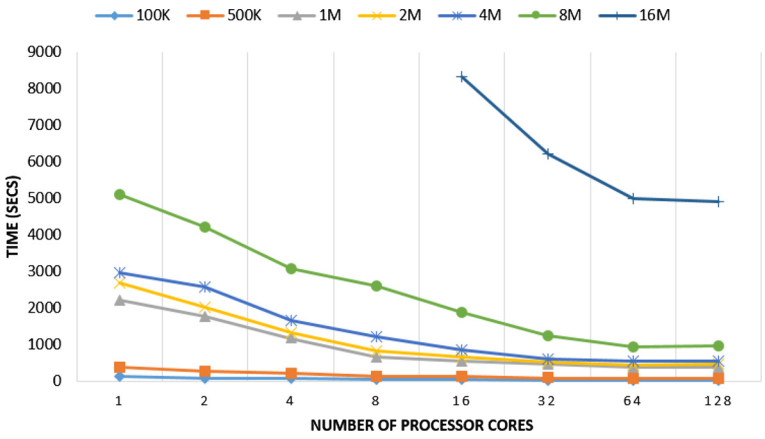

Fig. 5. Run-time while scaling up the number of processor cores over varying graph sizes.

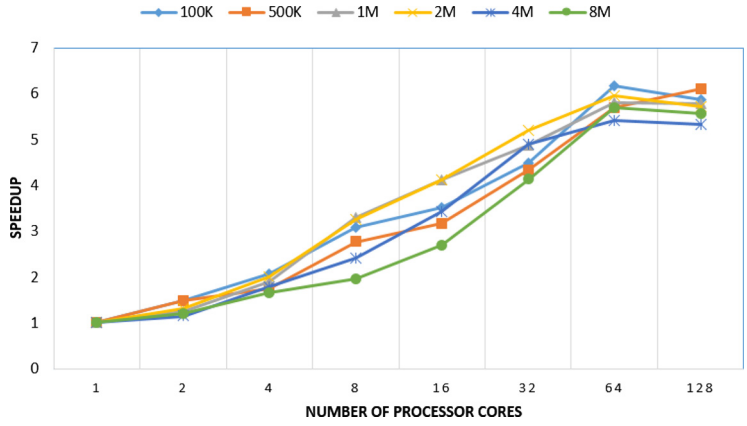

Fig. 6. Speedups compared to sequential hybrid algorithm while scaling up the number of processor cores over varying graph sizes.

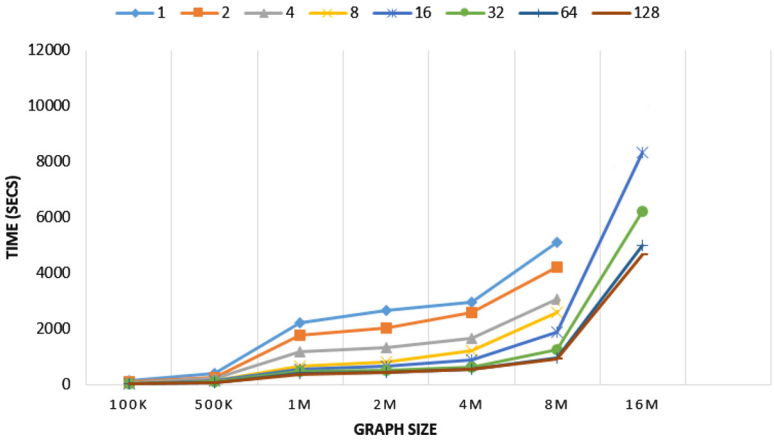

Fig. 7. Run-time while scaling up the graph sizes over varying processor cores.

This phenomenon is also observed in similar studies, like the one in [33]. It is important to note that our method does not ignore the cross partition edges completely, since labels are allowed to transfer in the form of messages across cross-partition edges. But we do not collapse the nodes on the boundary i.e. associated with these cross-partitioning edges, which is done in step 2 of our hybrid algorithm.

We also test our hybrid algorithm on real world networks described in Subsection 4.2. In Fig. 9, we see the variation in the total runtime while scaling up the number of processor cores for different graph sizes. The 16 processor core run of our hybrid algorithm acts as the *"base run"* for our real world data sets due to memory constraints of a single machine. We can observe that our algorithm scales better than base run for both the datasets, as the number of processor core increases. In Fig. 10, we see that our algorithm achieves $\approx 1.8X$ speedup for 128 processor cores, compared to the base run. In Fig. 11, we analyze the quality of the results obtained in terms of modularity of the communities detected. We observe the percentage error using the difference in the modularity between base run and higher processor cores hybrid algorithm. This error is calculated using the formula mentioned in Equation (3). It is evident that we maintain good quality of the results along with achieving high scalability for large real world social networks, scaling up to hundreds of millions of vertices and billions of edges. We

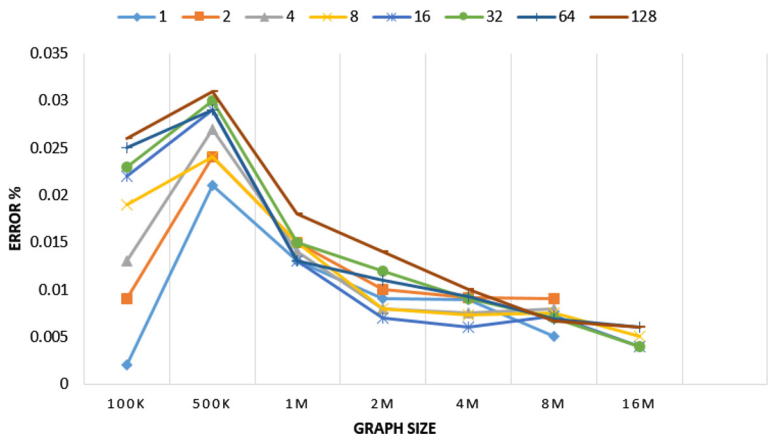

Fig. 8. Change in error percentage of final modularity compared to that achieved by sequential execution of hybrid algorithm.

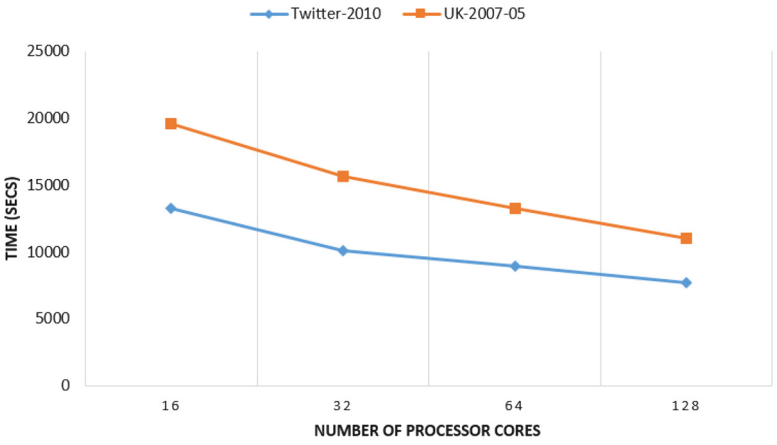

Fig. 9. Run-time while scaling up the number of processor cores.

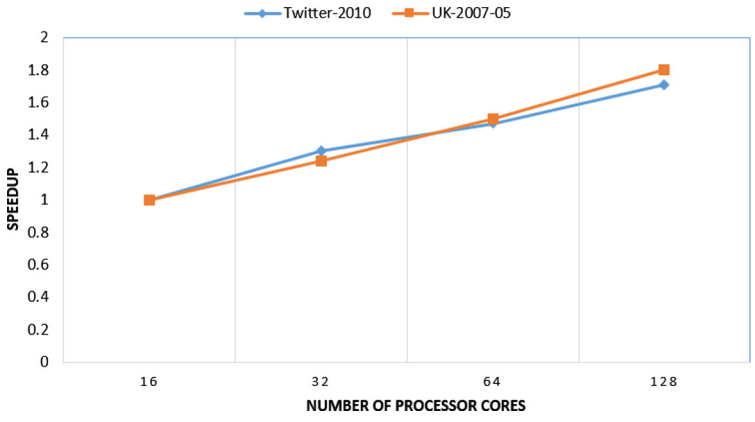

Fig. 10. Speedups compared to base run of hybrid algorithm while scaling up the number of processor cores up to 128.

observe that the quality of the communities extracted from real world networks are not as good as the ones we extract from synthetic graphs. We believe the primary reason for this is, synthetic graphs were generated with fine control over average and maximum degree distribution which made it possible for PMETIS to get good initial partitioning. On the other hand, we did not have any control over the degree distribution for real world networks.

5. Conclusion

Detecting communities in large networks, while achieving a good balance between scalability and quality of the results is one of the important open problems, especially due to the massive growth of social networks. Our work combines our existing MCML algorithm [5] and uses it as a subroutine in our hybrid community detection algorithm presented in this paper. We also combine existing graph partitioning technique i.e. PMETIS which minimizes cross-partition edges, as a preprocessing step to our algorithm. Our simulation results on a MPI setup with 8 compute nodes having 16 cores each shows that, upto $\approx 6X$ speedup is achieved for synthetic graphs upto $8M$ vertices in detecting communities without compromising the quality of results. We also show that, our hybrid algorithm can scale for large section of real world social networks like Twitter-2010 and UK-2007-05 having $\approx 41M$ and $\approx 105M$ vertices respectively and maintain good quality of the results when compared to other existing similar works.

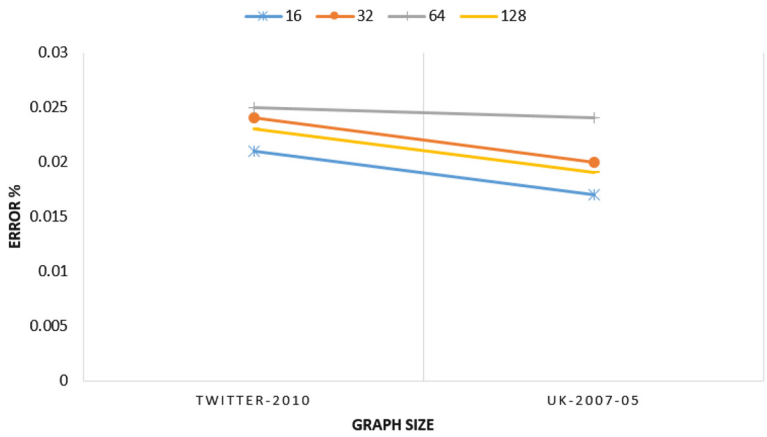

Fig. 11. Change in error percentage of final modularity compared to that achieved by base run of hybrid algorithm while scaling up the number of processor cores up to 128.

Competing interests

The authors declare that they have no competing interests.

Author's contributions

Both the authors have equal contribution towards all aspects of this research paper.

References

[1] M. Girvan, M.E. Newman, Community structure in social and biological networks, Proc. Natl. Acad. Sci. 99 (12) (2002) 7821–7826.
[2] A.-L. Barabasi, Z.N. Oltvai, Network biology: understanding the cell's functional organization, Nat. Rev. Genet. 5 (2) (2004) 101–113.
[3] S. Fortunato, Community detection in graphs, Phys. Rep. 486 (3) (2010) 75–174.
[4] V.D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, J. Stat. Mech. Theory Exp. 2008 (10) (2008) 10008.
[5] S. Oliveira, R. Sharma, High quality multi-core multi-level community detection algorithm, Int. J. Comput. Sci. Eng. 15 (3/4) (2017) 311–321.
[6] U.N. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks, Phys. Rev. E 76 (3) (2007) 036106.
[7] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.
[8] G. Karypis, V. Kumar, Parallel multilevel graph partitioning, in: The 10th International Parallel Processing Symposium, 1996, Proceedings of IPPS'96, IEEE, 1996, pp. 314–319.
[9] S. Kirmani, P. Raghavan, Scalable parallel graph partitioning, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 51.
[10] H. Meyerhenke, P. Sanders, C. Schulz, Parallel graph partitioning for complex networks, in: 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2015, pp. 1055–1064.
[11] M.E. Newman, M. Girvan, Finding and evaluating community structure in networks, Phys. Rev. E 69 (2) (2004) 026113.
[12] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, D. Parisi, Defining and identifying communities in networks, Proc. Natl. Acad. Sci. USA 101 (9) (2004) 2658–2663.
[13] S. Fortunato, V. Latora, M. Marchiori, Method to find community structures based on information centrality, Phys. Rev. E 70 (5) (2004) 056104.
[14] M.E. Newman, Fast algorithm for detecting community structure in networks, Phys. Rev. E 69 (6) (2004) 066133.
[15] V.d.F Vieira, C.R. Xavier, N.F. Ebecken, A.G. Evsukoff, Modularity based hierarchical community detection in networks, in: Computational Science and Its Applications–ICCSA 2014, Springer, 2014, pp. 146–160.
[16] P. De Meo, E. Ferrara, G. Fiumara, A. Provetti, Generalized Louvain method for community detection in large networks, in: 2011 11th International Conference On Intelligent Systems Design and Applications (ISDA), IEEE, 2011, pp. 88–93.
[17] T. Hashimoto, B. Chakraborty, Y. Shirota, Social media analysis—determining the number of topic clusters from buzz marketing site, Int. J. Comput. Sci. Eng. 7 (1) (2012) 65–72.
[18] S. Oliveira, R. Sharma, Identification and prediction of functional protein modules using a bi-level community detection algorithm, Int. J. Bioinform. Res. Appl. 12 (2) (2016) 129–148.
[19] J.-C. Liu, Y.-C. Liang, S.-W. Lin, Selection of canonical images of travel attractions using image clustering and aesthetics analysis, Int. J. Comput. Sci. Eng. 8 (4) (2013) 324–335.
[20] P. Pons, M. Latapy, Computing communities in large networks using random walks, in: Computer and Information Sciences—ISCIS 2005, Springer, 2005, pp. 284–293.
[21] M. Rosvall, C.T. Bergstrom, Maps of random walks on complex networks reveal community structure, Proc. Natl. Acad. Sci. 105 (4) (2008) 1118–1123.
[22] M. Pandey, V.K. Pathak, B.D. Chaudhary, A framework for interest-based community evolution and sharing of latent knowledge, Int. J. Grid Util. Comput. 3 (2–3) (2012) 200–213.
[23] A.D. Rathnayaka, V.M. Potdar, T.S. Dillon, S. Kuruppu, Formation of virtual community groups to manage prosumers in smart grids, Int. J. Grid Util. Comput. 6 (1) (2014) 47–56.
[24] A. Lancichinetti, S. Fortunato, Community detection algorithms: a comparative analysis, Phys. Rev. E 80 (5) (2009) 056117.
[25] R.R. Nadakuditi, M.E. Newman, Graph spectra and the detectability of community structure in networks, Phys. Rev. Lett. 108 (18) (2012) 188701.
[26] F. Radicchi, A paradox in community detection, Europhys. Lett. 106 (3) (2014) 38001.
[27] A. Prat-Pérez, D. Dominguez-Sal, J.-L. Larriba-Pey, High quality, scalable and parallel community detection for large real graphs, in: Proceedings of the 23rd International Conference on World Wide Web, ACM, 2014, pp. 225–236.
[28] A. Prat-Pérez, D. Dominguez-Sal, J.M. Brunat, J.-L. Larriba-Pey, Shaping communities out of triangles, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, ACM, 2012, pp. 1677–1681.
[29] I. Rytsareva, T. Chapman, A. Kalyanaraman, Parallel algorithms for clustering biological graphs on distributed and shared memory architectures, Int. J. High. Perform. Comput. Networking 7 (4) (2014) 241–257.
[30] J. Soman, A. Narang, Fast community detection algorithm with GPUs and multicore architectures, in: 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE, 2011, pp. 568–579.
[31] H. Lu, M. Halappanavar, A. Kalyanaraman, Parallel heuristics for scalable community detection, Parallel Comput. 47 (2015) 19–37.
[32] S.-H. Bae, D. Halperin, J. West, M. Rosvall, B. Howe, Scalable flow-based community detection for large-scale network analysis, in: 2013 IEEE 13th International Conference On Data Mining Workshops (ICDMW), IEEE, 2013, pp. 303–310.
[33] C. Wickramaarachchi, M. Frincu, P. Small, V.K. Prasanna, Fast parallel algorithm for unfolding of communities in large graphs, in: 2014 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2014, pp. 1–6.
[34] S. Moon, J.-G. Lee, M. Kang, Scalable community detection from networks by computing edge betweenness on mapreduce, in: 2014 International Conference On Big Data and Smart Computing (BIGCOMP), IEEE, 2014, pp. 145–148.
[35] P. Jancura, D. Mavroeidis, E. Marchiori, Deen: a simple and fast algorithm for network community detection, in: Computational Intelligence Methods for Bioinformatics and Biostatistics, Springer, 2012, pp. 150–163.
[36] C. Avery, Giraph: large-scale graph processing infrastructure on Hadoop, in: Proceedings of the Hadoop Summit, Santa Clara, 2011.
[37] H. Kwak, C. Lee, H. Park, S. Moon, What is twitter, a social network or a news media?, in: Proceedings of the 19th International Conference on World Wide Web, ACM, 2010, pp. 591–600.
[38] P. Boldi, B. Codenotti, M. Santini, S. Vigna, Ubicrawler: a scalable fully distributed web crawler, Softw. Pract. Exp. 34 (8) (2004) 711–726.