2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining

Distributed Community Detection in Web-Scale Networks

Michael Ovelgönne
UMIACS
University of Maryland
College Park, MD 20740
mov@umiacs.umd.edu

***Abstract*—Partitioning large networks into smaller sub-networks (communities) is an important tool to analyze the structure of complex linked systems. In recent years, many in-memory community detection algorithms have been proposed for graphs with millions of edges. Analyzing massive graphs with billions of edges is impossible for existing algorithms. In this contribution, we show how to find community partitions of networks with billions of edges. Our approach is based on an ensemble learning scheme for community detection that provides a way to identify high quality partitions from an ensemble of partitions with lower quality. We present a pre-processing procedure for community detection algorithms that significantly decreases the problem size. After reducing the problem size, traditional non-distributed community detection algorithms can be applied. We implemented a weak but highly scalable label propagation algorithm on top of the distributed-computing framework Apache Hadoop. The evaluation of our implementation on a 50-node Hadoop cluster and with evaluation datasets up to 3.3 billion edges shows very good results with respect to clustering quality as well as scalability. For a smaller 260 million edge network, we show that our preprocessing can improve the results of the popular Louvain modularity clustering algorithm.**

***Keywords*-Graph Clustering, Community Detection, Distributed Algorithms, MapReduce**

I. Introduction

Decomposing networks into cohesive subgroups, mostly denoted as graph clustering or community detection, became an important network analysis method. Community detection is used by many scientific disciplines (e.g. data mining, web science and complex systems science) and in many different ways (as an analytical tool to analyze and describe network topologies as well as a component of data mining and information retrieval systems). Unsurprisingly, a lot of work on community detection has been conducted recently.

So far the development of scalable community detection algorithms focused on networks with a few millions edges. However, many networks of interest have hundreds of millions or even billions of vertices and billions of edges. For example, the online social network Facebook reports as of March 2013 to have more than 1.1 billion active users.[1] The size gap between networks processable with current algorithms and many networks of interest is at least one order of magnitude.

Almost all algorithmic innovation in the area of community detection with respect to scalability was on runtime scalability. The limits of processable network sizes were extended by developing algorithms that run in near-linear time. So far, algorithms proposed for community detection in networks are almost exclusively non-distributed, in-memory and single-threaded algorithms. While very fast and scalable algorithms have been developed [16], [20], [2], [14], these algorithms require to store the complete graph in main memory. Processing graphs with billions of edges would require hardware with at least several hundred GB of RAM.

To address these shortcomings, we present in this paper a distributed core groups detection algorithm that scales to networks with billions of edges. We denote as core groups rather small cohesive groups of vertices that belong to the same community. These core groups could also be regarded as sub-communities or communities at a higher level of granularity. We see the core groups detection as a pre-processing method for community detection. By building core groups, we are able to decrease the problem size for community detection. The core groups induce a graph that is significantly smaller than the original one so that the induced graph can be processed with existing community detection algorithms.

The main contributions of this paper are:

- We present a distributed ensemble learning algorithm for graph clustering that scales to graphs with billions of edges (Sec. III).
- We show the importance of creating a diverse ensemble of base partitions and an implementation that is able to create such an ensemble (Sec. IV).
- We present extensive experiments on real-world datasets (Sec. V).

To avoid confusion, in the following we use the term cluster only for a computer cluster. A machine as a part of a computer cluster will be denoted as a node. The representation of an object in a graph is called a vertex and groups of vertices will be denoted as communities.

II. Community Detection and Ensemble Learning

Community detection is the detection of 'natural' groups of vertices in networks. Unfortunately, there is no general agreement on what a good partition of a graph is. In the last decade many new community detection algorithms have been proposed that address shortcomings the authors perceived in

[1] http://newsroom.fb.com/content/default.aspx?NewsAreaId=22

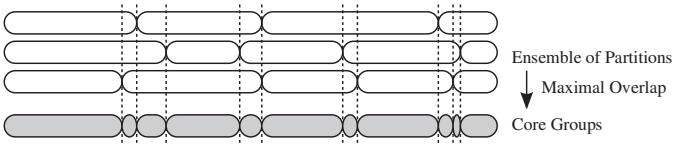

Fig. 1. In a core groups partition (bottom row) those vertices are in one group that are in the same community in all partitions of the ensemble (3 top rows).

the algorithms of others. The only agreement in the community detection community is that a partition of the vertices of a graph into 'natural' or 'cohesive' groups should group vertices in a way that there is a high density of edges within a group and a low density of edges between groups. The most popular algorithms to create such partitions are based on the optimization of the quality measure modularity [13]. Many other algorithms with implicit (e.g. [16]) or explicit (e.g. [17]) objective function exist. A comprehensive discussion of community detection techniques provides [5].

Algorithms for community detection follow a variety of approaches. Many community detection algorithms use the multi-level approach of [9], e.g. [2]. Another class of algorithms uses spectral techniques [8]. Explicit quality functions can be optimized with various optimization heuristics, e.g. simulated annealing [12].

Instead of regarding community detection as an optimization problem, it could be seen as a learning problem as well. An algorithm based on the concept of ensemble learning has been proposed in [14]. Ensemble learning is a general learning concept whose key component is to create good classifiers by learning several (weak) classifiers and combining them. The simplest ensemble learning approach is to learn several classifiers from the complete training data with so-called base algorithms and combine them by majority voting to a new and better classifier.

Ensemble learning is interesting for us because it allows to derive strong classifiers from an ensemble of weak classifiers. Schapire could show that strong classifiers can be derived from weak classifiers that are only slightly better than random choice [18]. The general outline of an ensemble learning clustering algorithm for community detection is roughly as follows [14]: (1) Create a set of partitions with some (weak) learning algorithm, (2) Identify the maximal overlap of the partitions as depicted in Figure 1, (3) Continue the search from the maximal overlap with the algorithm used in (1) or any other appropriate algorithm.

The creation of maximal overlaps is a key element of ensemble learning community detection. The overlap can be regarded as the consensus of several partitions. To create an ensemble of pairwise distinct partitions a non-deterministic base algorithm must be used or every partition in the set must be computed by a different base algorithm. Let $c_P(v)$ denote the community that vertex v belongs to in partition P. We create from a set S of partitions $\{P_1, \ldots, P_k\}$ of the vertex set V of G a core group partition $\hat{P}$ of V so that

$$\forall v, w \in V : (\bigwedge_{i=1}^{k} c_{P_i}(v) = c_{P_i}(w)) \Leftrightarrow c_{\hat{P}}(v) = c_{\hat{P}}(w).$$

A partition created by calculating the maximal overlap is denoted as a core groups partition and not a community partition because such a partition has different properties. A core groups partition usually includes many singletons and other small vertex sets. These small vertex sets will probably lie – depending on the actual definition of a community – on the border of a community or in-between communities. A particular community detection algorithm might classify those vertex sets to one of the bordering communities, to a community of their own or as outliers. The term core groups is intended to make clear that these groups are intermediate results that may form larger structures depending on the selected community detection criterion.

Given these preliminaries we can discuss why the concept of ensemble learning is interesting for community detection in huge graphs. Porting algorithms designed for non-distributed environments to distributed environments is usually a complicated task and many non-distributed algorithms cannot be ported efficiently. Ensemble learning provides a solution for this problem. We know that the quality of the base algorithm is not that important in an ensemble learning scheme. So we can select the base algorithm because of its execution performance in a distributed environment without having to worry too much about the algorithms community detection quality.

These considerations are the basis of our solution for scaling community detection to billion-edge graphs. We use a highly scalable (but maybe weak) learning strategy in an ensemble learning scheme and on an Apache Hadoop [6] cluster. Hadoop is an open-source implementation of the MapReduce framework [4] and provides a distributed computing environment with a high performance and a convenient programming interface.

As the base algorithm for community detection we select the Label Propagation (LP) algorithm of Raghavan et al. [16] because it is ideally suited to be used in a MapReduce environment. The LP algorithm works as follow: It starts with assigning every vertex a unique label. Then the labels are propagated. In iterative sweeps of the set of vertices, the label of a vertex is updated to the label which is most prevalent among its neighbors. If ties between several labels occur, they are broken randomly. The algorithm stops when every vertex has a label that at least half of its neighbors have. The final labels of the vertices represent the partition of the graph. Vertices with the same label are in one group and vertices with different labels are in different groups.

Propagating labels has been successfully used for several other MapReduce implementations of graph algorithms like shortest paths and connected components [11]. Formulating other graph clustering algorithms in the MapReduce paradigm have not been successful so far (see Section VI). Direct translations of non-distributed algorithms to MapReduce seem to conflict with the concept of this paradigm.

III. Algorithm

Our pre-processing algorithm for community detection has to compute a core groups partition and the graph induced by this partition. To compute core groups we need to compute several clusterings with the LP algorithm and determine the maximal overlap. So the outline of our algorithm has to be as follow:

1) Compute set S of k partitions of the input graph G
2) Compute maximal overlap O of S
3) Compute graph G' induced by partition O of G
4) Return O and G'

A. Hadoop MapReduce

Our solution is based on the distributed computing framework Hadoop. The two main components of Hadoop are the Hadoop Distributed File System (HDFS) and the MapReduce engine. In short, the objective of HDFS is to store huge files and make them quickly accessible. To do so, HDFS chops files into several chunks and each chunk is stored on several (by default 3) nodes of the Hadoop cluster. The MapReduce engine consists of a JobTracker and one TaskTracker per node. The JobTracker splits up the workload and assigns each part to one of the TaskTrackers - if possible a TaskTracker that runs on a node that also stores the required data chunk.

A Hadoop job consists of three phases. First, records/rows from an input file are read and transformed into (new) key-value pairs (map phase, see (1)). For one input key-value pair (k, v), the map function can create an arbitrary number of output key-value pairs $[(l_1, w_1), \ldots, (l_r, w_r)]$. These key-value pairs are written to the map output. Second, the key-value pairs created for all input data are sorted by their key (sort phase). Third, a key l and all associated values $[u_1, \ldots, u_s]$ are passed to a reduce function call (on one node) to create a set of output key-value pairs $[(m_1, x_1), \ldots, (m_t, x_t)]$ (reduce phase, see (2)). The output keys do not need to match the input key and they do not have to be identical.

$$\begin{aligned} &\text{map: } K_1 \times V_1 \to (K_2 \times V_2)^*, \\ &\qquad (k, v) \mapsto [(l_1, w_1), \ldots, (l_r, w_r)] \end{aligned} \tag{1}$$

$$\begin{aligned} &\text{reduce: } K_2 \times V_2^* \to (K_3 \times V_3)^*, \\ &\qquad (l, [u_1, \ldots, u_s]) \mapsto [(m_1, x_1), \ldots, (m_t, x_t)] \end{aligned} \tag{2}$$

B. Distributed Core Group Detection

Every iteration of the LP algorithm, i.e. every sweep over the set of vertices and updating each vertex label, will be executed by one Hadoop job. To identify core groups, we need to calculate an ensemble of partitions. We could create the partitions for the ensemble in sequence but this would be highly inefficient. For each partition we would need to run several map-reduce jobs which brings a large overhead. So instead of calculating all partitions of the ensemble sequentially, we compute all partitions in parallel. Computing several partitions in parallel means, we do not initialize every vertex with an unique label and propagate it through the network. Instead, we assign each vertex i a vector $l_i = (l_{i1}, \ldots, l_{ir})$ of r labels and initially set $l_{ix} = i$ for every label instance x. Let a label be an element from the label set L. Then, all r labels of vertex i are updated in parallel but independently:

$$l_{ix} = \underset{l \in L}{\operatorname{argmax}} \left(\sum_{j \in V, (i,j) \in E} \delta(l_{jx}, l) \right) \tag{3}$$

where δ is the Kronecker symbol, i.e. $\delta(x, y) = 1$ for $x = y$ and otherwise 0. In case of ties, they are broken arbitrarily. Breaking ties randomly leads to different community partitions for each label instance although all r labels of a label vector are initialized with the same value.

Using Hadoop, it is best to bundle the propagation of labels in this way, as we only increase the bit size of the value (label vector instead of single label) of the key-value-pairs. This lowers the total amount of data to be transferred compared to the sequential computation of several partitions. Furthermore, larger data blocks can be transferred more efficiently than several smaller ones and sorting key-value pairs has to be done only once and not r times.

The rough outline of the algorithm with p label propagation steps is given in Algorithm 1. One map and one reduce procedure always belong together to one Hadoop job. That means the first job consists of `ReadMap` and `PropagateReduce`, the second job consists of `PropagateMap` and `PropagateReduce`, and so on. Next, we will go into detail of the jobs. The jobs performed can be separated into two groups by the task they serve: 1. jobs to propagate labels to identify core groups (lines 1-5) and 2. jobs to output the core groups and the graph induced by the core groups (lines 5-9). The job that includes `CoreGroupsExportReduce` serves both tasks.

Note, that the algorithm operates on the edges only. Therefore, no explicit community assignments for isolated vertices (i.e. vertices that are not an endpoint of any edge) are generated. This should not pose a problem, as we know that a vertex is in a community of its own when no explicit community assignment has been generated.

1) Initial Read and First Label Propagation: We start with the map function `ReadMap` and assume the input graph is provided as an edgelist, i.e. each row of the input file contains the IDs of the two endpoints of an edge. For each edge (i, j) we operate with tuples $(i, (l_i, j))$, where the second element of the tuple is a tuple itself and consists of the label vector of vertex i and its neighbor j. Initially, we set the label of a vertex to the vertex id, i.e. $l_{ix} = i$ for every label instance x. In the `ReadMap` procedure (line 1), we directly transform the tuple $(i, (l_i, j))$ to $(j, (l_i, i))$ and pass this new tuple to the reduce phase. The semantic interpretation of the new tuple $(j, (l_i, i))$ is that vertex j got passed the label vector l_i from its neighbor i.

In the reduce phase of the first MapReduce job (line 3), for each vertex a new label is determined. Before the function `PropagateReduce` is called, all key-value pairs outputted by `ReadMap` get sorted. Then, all values which belong to a key-value pair with the same key are passed to one reducer call so

Algorithm 1: Hadoop Core Group Identification

input : edgelist file $input\_file$, # of propagation steps p, # of label instances k
output: vertex-core group mapping file cg_file, edgelist file el_file

```
1 passed_labels ← ReadMap (input_file, k)
2 for i ← 1 to p − 1 do
3 |   new_labels ← PropagateReduce (passed_labels)
4 |   passed_labels ← PropagateMap (new_labels)
5 final_labels ← CoreGroupsExportReduce (new_labels, cg_file)
6 final_labels ← EdgeListPreparationMap (final_labels)
7 core_group_links ← EdgeListPreparationReduce (final_labels)
8 core_group_links ← EdgeListExportMap (core_group_links)
9 EdgeListExportReduce (core_group_links, el_file)
```

that PropagateReduce gets as input a key and all associated value in the form $(j, \{(l_{i_1}, i_1), \ldots, (l_{i_{d_j}}, i_{d_j})\})$ where d_j is the degree of vertex j. PropagateReduce determines the new label vector $\hat{l}_j$ of vertex j according to Equation (3) and writes for each neighbor i_x a tuple $(j, (\hat{l}_j, i_x))$. Altogether, the operation performed is $(j, \{(l_{i_1}, i_1), \ldots, (l_{i_{d_j}}, i_{d_j})\}) \rightarrow \{(j, (\hat{l}_j, i_1)), \ldots, (j, (\hat{l}_j, i_{d_j}))\}$, where d_i denotes the degree of vertex i, $i_1, \ldots, i_{d_j}$ denote the d_j neighbors of j. The label vector l_i denotes the labels propagated from vertex i to vertex j.

2) Consecutive Label Propagation: The consecutive label propagation jobs are in principle identically to the first label propagation step. The only difference are the input data. While the first label propagation job consisted of the procedure ReadMap (line 1) and PropagateReduce (line 3), all consecutive label propagation jobs consists of the pair PropagateMap (line 4) and PropagateReduce (line 3). While ReadMap reads the edges from the input file and passes an initial label from vertices to their neighbors, the procedure PropagateMap reads the output of the preceding PropagateReduce and passes the labels that have been determined there.

As the total runtime of our algorithm depends on the number of jobs we have to start, we want to run as few as possible. That means, we have to limit the label iteration steps to a minimum. Raghavan et al. [16] noted for asynchronous label updates that 95% of the vertices are classified to their final cluster after only 5 steps irrespective of the number of vertices in the graph. However, Leung et al. [10] showed that synchronous label updates converge much slower. The number of iterations is in Algorithm 1 the parameter p. We will analyze the best setting for this parameter in Section V.

3) Last Label Propagation and Export of Core Group Mapping: The last label propagation iteration is conducted by a MapReduce job consisting of the procedures PropagateMap (line 4) and CoreGroupsExportReduce (line 5).

CoreGroupsExportReduce determines the new vertex labels based on the labels passed by PropagateMap in the same way as the PropagateReduce procedure does. But instead of feeding the resulting labels into a next propagation iteration, the CoreGroupsExportReduce function calculates the core group of every vertex.

After we finished propagating labels, each vertex is assigned a final vector of labels. If two vertices have the same label for each of the k label instances, the vertices belong to the same core group. To identify those vertices that are in the same cluster in every partition we use the following approach. We concatenate all labels in the label vector. Then, we calculate a hash value from this combined label representation. The hash value is the core group id of the vertex in the core groups partition. In the following we denote the core group id of vertex i as l_i^*. To avoid hash collisions, the length of the hash value has to be selected appropriately. With a 128 bit hash value we do not have to worry about hash collisions even for networks with billions of vertices.

4) Edgelist Export of the Induced Graph: CoreGroupsExportReduce writes the core group assignment of every vertex to file. Beside that, we need to store in a second file the edgelist of the graph induced by the core groups partition. That means, we need to translate the connections between vertices to connections between core groups. Furthermore, we need to determine the multitude of connections between a pair of core groups to assign weights to the edges of the induced graph.

The edgelist export is a complex task that requires several steps. The first challenge is the transformation of the vertex-to-vertex links to the appropriate core-group-to-core-group links. Remember, that the input data for CoreGroupsExportReduce is the tuple $(j, \{(l_{i_1}, i_1), \ldots, (l_{i_{d_j}}, i_{d_j})\})$. We already updated for vertex j the label vector $\hat{l}_j$ and used that information to determine the core group id l_j^*. Now, we need to create a record (l_i^*, l_j^*) for every edge (i, j). We have the l_j^* at hand, but the core group ids of the neighbors of vertex j are missing. Therefore, we create for every neighbor i_x of vertex j a record of the following type $(i_x, (NULL, j, l_j^*))$. Additionally, we create the record $(j, (l_j^*, j, l_j^*))$. With the information of the additional record, we can replace the NULL values in EdgeListPreparationReduce (line 7). Again, the key-value pairs outputted by CoreGroupsExportReduce are sorted by their keys. This way, the values of the key-value pairs $(i, (NULL, j_x, l_{j_x}^*))$ are combined with the value of the key value-pair $(i, (l_i^*, i, l_i^*))$ in a set of values passed to one reducer call. Now, we can copy the missing core group id from the record $(i, (l_i^*, i, l_i^*))$ to the record $(i, (NULL, j_x, l_{j_x}^*))$. The vertex ids are no longer of any use and the final output created by EdgeListPreparationReduce are the tuples $(l_i^*, l_{j_x}^*)$.

After having transformed the vertex-to-vertex links to core-group-to-core-group links, we just need to count the multitude

of links between a pair of core groups and export this information to the edgelist file. The final job consisting of `EdgeListExportMap` (line 8) and `EdgeListExportReduce` (line 9) takes care of this task. We omit a detailed description of this simple counting problem because it is discussed in every Hadoop tutorial.

IV. Creating Diversity in Base Partitions

The concept of ensemble-learning is based on the combination of the information contained in an ensemble of base solutions. For the purpose of creating core groups homogeneous partitions in the ensemble are not desired. The ensemble of solutions we generate should cover very different but 'good' partitions. The maximal overlaps that we regard as core groups should reflect on which decisions all those diverse partitions agree. Tuning the label propagation algorithm to create this diversity seems to be crucial for creating good core groups.

Raghavan et al. [16] discussed for their original label propagation algorithm synchronous and asynchronous label update strategies. In a synchronous update a vertex gets in iteration t the label most of its neighbors had after iteration $t-1$. In an asynchronous update, the vertices are processed in random order. The current labels of neighboring vertices are used to determine the new label – whether the labels of the neighbors have already been updated in the current iteration or not. Leung [10] observed that synchronous label updates lead to more homogeneous results than asynchronous label updates. In a distributed environment an asynchronous label update strategy would require heavy communication between the nodes of the computer cluster. The information of asynchronous label updates would need to be communicated between the processing nodes during an iteration of label updates. Synchronous label update require this exchange only once at the end of an iteration.

To achieve greater diversity of the identified communities we need to use a different strategy than asynchronous updates. To direct the label propagation convergence process in different directions in each parallel label propagation run, we give a different randomly chosen set of vertices a head start. With a probability b instead of initializing l_{ij} with i we set l_{ij} to j. The interpretation here is that vertex i got the label j from vertex j in the preceding step and now passes this label back.

In Figure 2 we show the result of experiments with the simple and advanced label initialization strategies. First we analyzed the number of core groups in a core groups partition (two upper charts). We see that the number of core groups identified by the simple approach is always significantly lower than the number of core groups identified with the advanced implementation. A lower number of core groups means the base partitions are more similar. The higher number of core groups created with the advanced initialization strategy shows that our algorithm is indeed able to find community structures with a higher diversity than the simple strategy. To check whether the higher diversity of the base partitions results in core groups partitions of higher quality, we start the community detection algorithm of [2] from these partitions and compare the quality of the identified communities. The two lower charts in Figure 2 show the quality results of our experiments. The performance as a pre-processing method for modularity optimization is higher for the advanced implementation.

These results have been generated with b set to $1/3$. Experiments with different values of b showed that the diversity of the partitions increases with increasing values of b. However, the modularity scores increased only initially and started to decline for $b > 1/3$.

In conclusion we see that the diversity of the base partitions is important to find high quality core groups and that our advanced initialization strategy is able to create this diversity. The evaluation results we show in Section V are based on this advanced label initialization strategy with $b = 1/3$.

TABLE I
Test datasets used for algorithm evaluation.

dataset	uk-2002	uk-2007-05
# of vertices	18,520,486	105,896,555
# of edges	261,787,258	3,301,876,564
Avg. degree	16.1	35.3
edgelist file size	8.3 GB	110 GB

V. Evaluation

We evaluated our implementation on a cluster with 50 task tracker nodes. Each machine is equipped with 4 Quad-Core Intel Xeon CPU running at 2.26 GHz and 36 GB RAM. The evaluation environment is shared with other users. To get meaningful runtime measurements, we tried not to have interferences with jobs of other users. A few times other users started jobs while we conducted runtime measurements. In those cases, we stopped our tests and repeated them when the Hadoop cluster was idle again.

We enabled the standard data compression of Hadoop and stored all intermediate results in a binary format (Hadoop's *SequenceFileOutputFormat*). We went without any advanced optimization techniques such as those proposed by Lin and Schatz [11] for efficient graph algorithms in MapReduce.

As test datasets we used the two web graphs *uk-2002* and *uk-2007-05* crawled by Boldi et al. [3]. These datasets are publicly available at http://law.dsi.unimi.it/datasets.php. The graphs have been symmetrized (if a pair of vertices is connected by only one directed edge we added the reciprocal edge) and loops have been removed. Basic information of these graphs are summarized in Table I. For both datasets, we ran our algorithm with 5 and 10 label propagation iterations and between 8 and 40 label instances.

First we analyzed how much the core groups compress the problem size. The size of the graph induced by the core group partition determines whether one pre-processing phase is enough to process the result with traditional non-distributed algorithms on a single machine. Figure 3 shows that the dataset *uk-2002* gets contracted to roughly 1-2.5 million core groups and that the dataset *uk-2007-05* gets contracted to roughly 2-8 million core groups. So, the contraction factors are roughly in

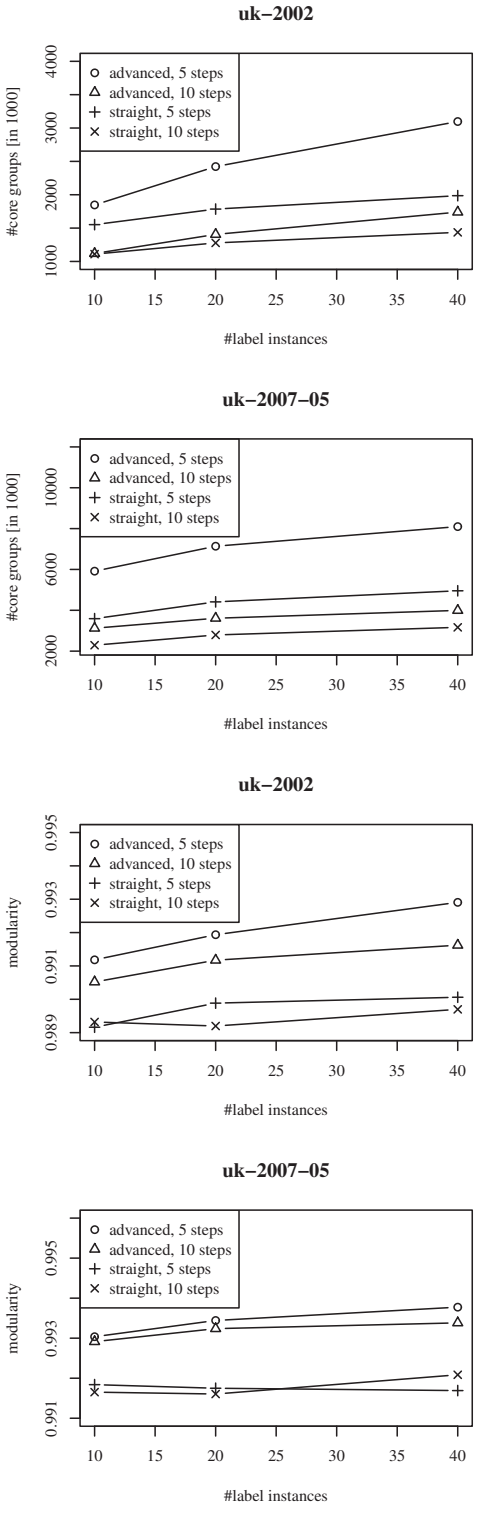

Fig. 2. Comparison of the algorithm implementations with advanced and straight initial label passing procedures with respect to the number of generated core groups (upper two figures) and modularity of the communities found starting from the core groups (lower two figures). The modularity values are the averages of 10 independent runs of the algorithm from [2].

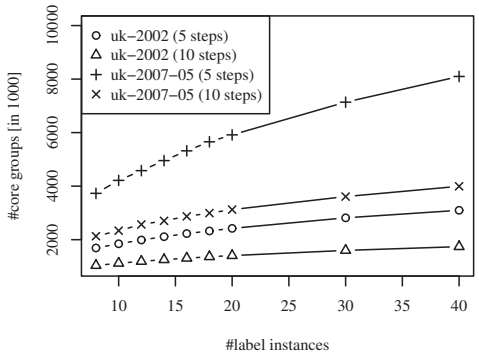

Fig. 3. Number of core groups depending on the number of label instances and the number of label propagation steps.

the ranges between 7–18 and 13–53, respectively. A phase of core groups creation significantly compresses the graph's size and the larger graph is compressed much stronger than the smaller one. From the original work on the LP algorithm [16] we know that with an increasing number of label propagation steps the number of distinct labels (and so clusters) decreases until a stable label assignment is reached. Labels propagate within clusters at a high granularity first and, finally, labels propagate within clusters at a lower granularity. So we can see from Figure 3 the unsurprising result that the number of core groups is lower for 10 label propagation steps than for only 5 steps.

Next, we analyzed the scaling behavior of our implementation. The results are summarized in Figure 4. We see, that the runtime increases about linear to the number of label instances we propagate. The slow increase in runtime is especially important for the label-propagation step, because this step is repeatedly executed. The runtime for the edgelist export step does not increase with the number of label instances. The edgelist is exported after we calculated the maximal overlap of the partitions given by the label instances.

To assess the suitability of the core groups approach as a pre-processing step for community detection, we identified communities in the graphs induced by the core groups with help of the popular Louvain community detection algorithm based on modularity optimization [2]. Figure 5 shows the average modularity of 10 test runs on the different induced graphs of our two evaluation datasets. All communities identified starting from the core groups partitions achieve very high modularity values. For the smaller dataset *uk-2002* we were able to start the modularity optimization algorithm also from scratch (i.e. without our pre-processing method). For this dataset, the pre-processing improved the modularity of the identified communities. Search heuristics make errors on their path through the search space. The ensemble-based approach is able to revert some faulty decisions of its base algorithm. So, the pre-processing creates a partition with less errors than the final clustering method is able to create at the same level of graph contraction. This indicates that the presented core groups identification method is a suitable pre-processing procedure for community detection. Note however, that the

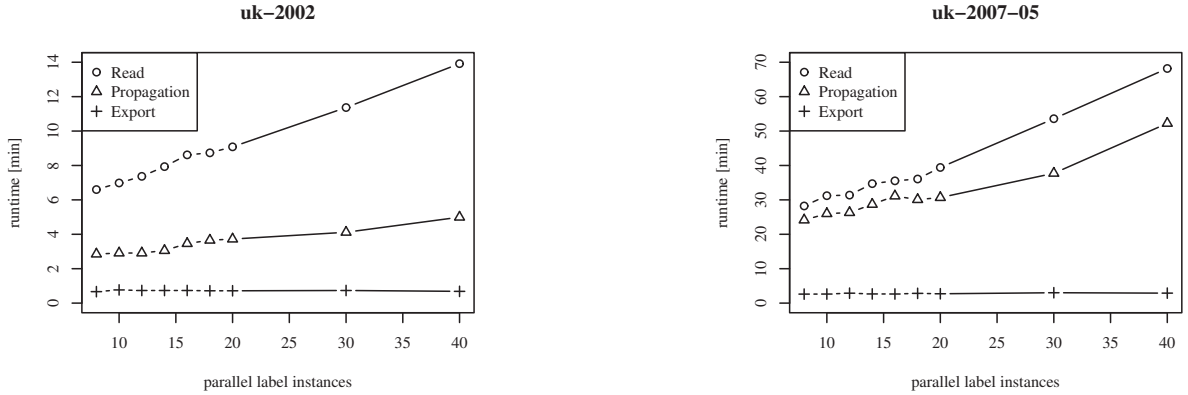

Fig. 4. Hadoop job runtime depending on the number of label instances. The runtimes are depicted separately for the steps Initial Read and first label propagation (Read, Alg. 1 lines 1+3), Consecutive Label Propagation (Propagation, Alg. 1 lines 4+3) and Edgelist Export (Export, Alg. 1 lines 8+9).

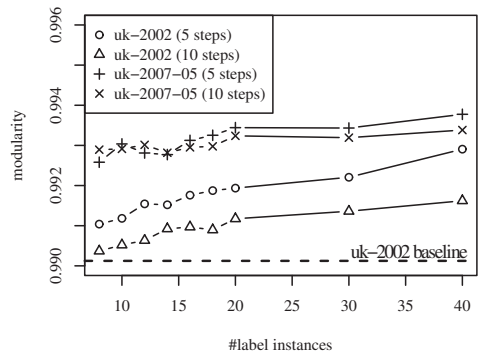

Fig. 5. Average modularity of 10 runs of the algorithm [2] of the graphs induced by core groups identified for settings of the number of label instances and the number of label propagation steps. For the smaller dataset *uk-2002* the average modularity identified without pre-processing is given as a baseline. For the larger dataset *uk-2007-05* starting the modularity optimization algorithm without pre-processing was not feasible because of memory limitations.

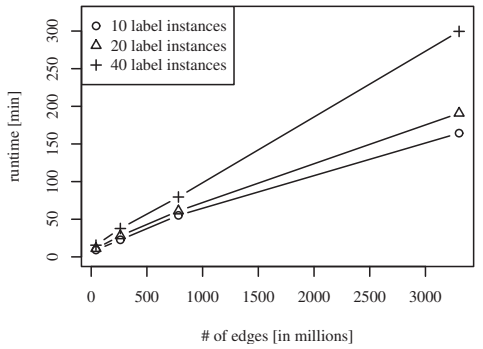

Fig. 6. Runtime of core group generation depending number of edges of graph for three different parameter values for the number of label instances. Additional to the two datasets used throughout the evaluation section we used a symmetrized version of the dataset *LiveJournal* [1] with ∼43 million edges and the dataset *uk-2005* [3] with ∼0.9 billion edges.

actual suitability for a particular community detection problem may differ. If the (implicit) definition of a good community is significantly different from that of the modularity concept, our pre-processing may perform less well.

The scaling of the algorithm in the number of edges can be seen from Figure 6. The runtime scales about linear in the number of edges. Linear scaling is an important property when we want to process datasets that are even larger than the datasets used in this evaluation. This results indicate that we can process graphs with about 20 billion edges in 1 day on the 50 node cluster (depending on the configuration).

VI. Discussion and Related Work

In this paper, we presented a distributed pre-processing algorithm for community detection to reduce the problem size of billion edge graphs. Pre-processing approaches to reduce the dimensionality of community detection problems have been suggested before [15], [19], but they have a weak clustering performance compared to state-of-the art methods and require to store the complete graph in memory. So, they are no solution to the problem of handling graphs at a size where the memory capacity of single machine does not suffice.

Zhang et al. [22] proposed a community detection implementation following the bulk synchronous parallel model. The communities are the connected components of a graph after iterations of adding and removing edges based on a propinquity measure. The key here is to define a suitable but fast computable propinquity measure. While in our approach the edges are distributed among the nodes, Zhang et al. distribute the vertices. For each vertex a virtual process is started on one of the nodes. This process retrieves messages from other virtual processes, sends messages to them and carries out local computations. Zhang et al. evaluated their implementation with graphs of up to about 118 million edges and clusters consisting of up to 1000 machines. Runtime and scalability in the number of nodes are good. However, the usability of this approach is limited by the specific community detection criterion and a custom distributed-computing environment.

Several authors worked on implementing graph analysis on top of the Apache Hadoop platform. Kang et al. [7] developed a Peta-Scale Graph Mining System (PEGASUS). This system is based on formulating different graph analysis operations

(PageRank, Random Walk with Restart, diameter estimation, connected components) as matrix-vector multiplications. Kang et al. tested their system with a web graph with roughly 6.7 billion edges and it showed about linear scaling in the number of edges. Their intended extension of PEGASUS by an eigensolver could be the basis of spectral community detection algorithms. The Singular Value Decomposition implementations of the Hadoop data mining library Mahout could be a starting point for spectral community detection, too.

Yang et al. [21] presented a Hadoop implementation of a community detection method that is based on maximal cliques. They identify maximal cliques, merge overlapping cliques and construct a graph representing connections between merged cliques that exceed some threshold. The connected components of this graph are supposed to be communities. The scalability of this approach suffers from the runtime complexity of the clique identification step. Yang et al. report a complexity of $O(3^{n/3}/m)$ with n the number of vertices in the graph and m the number of nodes in the computer cluster. The largest dataset they analyzed had only about 27 million edges. For this dataset however, the algorithm run in less than 10 minutes on a 32 node cluster.

Our core groups detection approach identifies nucleuses of communities and coarsens the original graph to the graph induced by the core groups partition. An arbitrary community detection algorithm can be used to identify communities of the coarsened graph. We demonstrated the quality of this preprocessing procedure by the competitive optimization performance for the quality function modularity. Our pre-processing algorithm improves the quality of the popular Louvain clustering algorithm. Our experiments also show the importance of creating a diverse ensemble of partitions. Only then the maximal overlaps of the ensemble result in high quality core groups.

We showed how several community partitions can be computed in parallel and how the information of the ensemble of partitions can be combined in a distributed-computing environment. The parallel computation of partitions is of major importance for the low runtime of the algorithm. We could process a graph as large as ~ 3.3 billion edges on our relatively small Hadoop cluster with 50 nodes in just a few hours. As Hadoop scales very well in the number of nodes, even larger graphs could be processed in just a few hours on larger Hadoop installations.

Acknowledgment

This work was partially funded by DARPA under contract No. W911NF11C0215.

References

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, 2006.

[2] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[3] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, pages 587–596, 2011.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, 2008.

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

[6] Apache Hadoop. Open source implementation of MapReduce, http://hadoop.apache.org/.

[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238. IEEE Computer Society, 2009.

[8] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM*, 51(3):497–515, 2004.

[9] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[10] I. X. Y. Leung, P. Hui, P. Liò, and J. Crowcroft. Towards real-time community detection in large networks. *Physical Review E*, 79:066107, Jun 2009.

[11] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85. ACM, 2010.

[12] A. Medus, G. A. na, and C. Dorso. Detection of community structures in networks via global optimization. *Physica A: Statistical Mechanics and its Applications*, 358(2–4):593–604, 2005.

[13] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004.

[14] M. Ovelgönne and A. Geyer-Schulz. Cluster cores and modularity maximization. In *ICDMW '10. IEEE International Conference on Data Mining Workshops*, pages 1204–1213, 2010.

[15] J. M. Pujol, J. Béjar, and J. Delgado. Clustering algorithm for determining community structure in large networks. *Physical Review E*, 74(1):016107, 2006.

[16] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76:036106, Sep 2007.

[17] M. Rosvall and C. T. Bergstrom. An information-theoretic framework for resolving community structure in complex networks. *Proceedings of the National Academy of Sciences*, 104(18):7327–7331, 2007.

[18] R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–227, 1990.

[19] Y. Wang, H. Song, W. Wang, and M. An. A microscopic view on community detection in complex networks. In *PIKM '08: Proceeding of the 2nd PhD workshop on Information and knowledge management*, pages 57–64, New York, NY, USA, 2008. ACM.

[20] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. SCAN: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 824–833, New York, NY, USA, 2007. ACM.

[21] S. Yang, B. Wang, H. Zhao, and B. Wu. Efficient dense structure mining using mapreduce. In *ICDMW '09. IEEE International Conference on Data Mining Workshops*, pages 332 –337, 2009.

[22] Y. Zhang, J. Wang, Y. Wang, and L. Zhou. Parallel community detection on large networks with propinquity dynamics. In *Proceedings of the 15th ACM SIGKDD Intern. Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 997–1006, 2009.