CC-MR - Finding Connected Components in Huge Graphs with MapReduce

Thomas Seidl, Brigitte Boden, and Sergej Fries

Data Management and Data Exploration Group
RWTH Aachen University, Germany
{seidl, boden, fries}@cs.rwth-aachen.de

Abstract. The detection of connected components in graphs is a well-known problem arising in a large number of applications including data mining, analysis of social networks, image analysis and a lot of other related problems. In spite of the existing very efficient serial algorithms, this problem remains a subject of research due to increasing data amounts produced by modern information systems which cannot be handled by single workstations. Only highly parallelized approaches on multi-core-servers or computer clusters are able to deal with these large-scale data sets. In this work we present a solution for this problem for distributed memory architectures, and provide an implementation for the well-known MapReduce framework developed by Google. Our algorithm CC-MR significantly outperforms the existing approaches for the MapReduce framework in terms of the number of necessary iterations, communication costs and execution runtime, as we show in our experimental evaluation on synthetic and real-world data. Furthermore, we present a technique for accelerating our implementation for datasets with very heterogeneous component sizes as they often appear in real data sets.

1 Introduction

Web and social graphs, chemical compounds, protein and co-author networks, XML databases - graph structures are a very natural way for representing complex data and therefore appear almost everywhere in data processing. Knowledge extraction from these data often relies (at least as a preprocessing step) on the problem of finding connected components within these graphs. The horizon of applications is very broad and ranges from analysis of coherent cliques in social networks, density based clustering, image segmentation, where in some way connected parts of the image have to be retrieved, data base queries and many more. Thus, it is not surprising that this problem has a long research history, and different efficient algorithms were developed for its solution. Nevertheless, modern information systems produce more and more increasing data sets whose processing is not manageable on single workstations any more. Social networks like Facebook process networks with more then 750 million users[1] where each

[1] http://www.facebook.com/press/info.php?statistics, state Sep. 2011

node is connected to 130 other nodes on average. The analysis of such enormous data volumes requires highly scalable parallelized algorithms. In this paper, we present a highly scalable algorithm for MapReduce [4], which is a programming model for the development of parallel algorithms developed by Google Inc. in 2004. Since then it experienced a fast spread out and nowadays its open-source implementation Hadoop[2] is used in companies like Yahoo! Inc. or Facebook Inc..

In MapReduce, the data is given as a list of records that are represented as (key, value) pairs. Basically, a MapReduce program consists of two phases: The first phase is the "Map" phase, in which the records are arbitrarily distributed to different computing nodes (called "mappers") and each record is processed separately, independent of the other data items. In the second phase, called "Reduce" phase, records having the same key are grouped together and processed in the same computing node ("reducer"). Thus, the reducers combine information of different records having the same key and aggregate the intermediate results of the mappers. By using this framework, programmers may concentrate on the data flow which is implemented by map jobs and reduce jobs. They do not have to take care of low-level parallelization and synchronization tasks as in classic parallel programming.

On top of this new programming model, Hadoop and other implementations of the MapReduce framework show a lot of non-functional advantages: First, they are scalable to clusters of many computing nodes, which are easily expanded by new nodes. Moreover, they show a high fault-tolerance: If one of the computing nodes fails during the execution of the program, the work of the other nodes is not affected or discarded, instead just the records that were currently processed on the failing node have to be processed again by another node.

In this paper, we propose an algorithm for finding connected components which is based on the MapReduce programming model and is implemented using Hadoop. Thus, our approach can make use of the aforementioned advantages of Hadoop such as high scalability and fault-tolerance.

The main contributions of our paper are:

- We present the parallelized algorithm CC-MR for the efficient detection of connected components in a graph using the MapReduce framework.
- We evaluate the performance of our algorithm compared to state-of-the-art approaches using synthetic and real-world datasets.
- We develop a technique to improve the load balancing of CC-MR for graphs with heterogeneous component sizes.

2 Fundamentals

In this section we give a short formal problem definition for the finding of connected components in Section 2.1. In Section 2.2 we introduce the MapReduce framework, which is used for the implementation of our algorithms.

[2] `http://hadoop.apache.org/`

2.1 Connected components

Let $G = (V, E)$ be an undirected graph without self loops, with V being a set of vertices and $E = \{(v, u), (u, v)\}, u, v \in V$ a set of edges. Intuitively, a connected component in G is a maximal subgraph $S = (V^S, E^S)$ in which for any two vertices $v, u \in V^S$ there exists an undirected path in G with v as start and u as end vertex. The term "maximal subgraph" means that for any additional vertex $w \in V \setminus V^S$ there is no path from any $v \in V^S$ to w.

In this work we present a solution for finding all connected components inside the graph G. The algorithm can as well be applied to directed graphs, in this case the result is the set of all weak connected components in the graph.

2.2 MapReduce

MapReduce is a programming model for processing web-scale datasets presented by Deam and Ghemawat at Google [4]. The main idea of this model is to divide data processing into two steps: map and reduce. The map phase is responsible for processing given (key,value) pairs stored on the distributed file system and generating intermediate (key,value) pairs. In the reduce phase, intermediate pairs with the same key are collected, processed at once, and the results are stored back to the distributed file system.

In the MapReduce model, communication between different computing nodes only takes place during a single communication phase, when the intermediate pairs from the map nodes are transferred to the reduce nodes. Apart from this, no further communication takes place. Neither do the individual mappers nor the individual reducers communicate with each other. This loose coupling of the computational nodes enables the framework to perform the calculations in a highly distributed and fault-tolerant way. Since all computational nodes process the data independently from each other, the only limitation for the number of parallel reducer-jobs is the number of unique intermediate key values. Additionally, since the single jobs do not depend on the results of other jobs, the failure of hardware can be easily managed by restarting the same job on another computational node. This high fault-tolerance and the loose coupling of computational nodes suits perfectly for usage of this model on commodity hardware like personal PCs connected to a cluster over a network. However, this limited communication also poses a challenge for the development of algorithms, which have to be designed such that the data in different mappers/reducers can be processed completely independently. Especially, the results computed by different reducers can not be combined in a MapReduce job. Thus, many problems cannot be solved using a single MapReduce job, but have to be solved by a chain of MapReduce jobs, such that the result records of one iteration can be re-distributed to the reducers and thus combined in the next iteration.

3 Related Work

The detection of connected components in a graph is a fundamental and well-known problem. In the past, different approaches for finding connected compo-

nents were introduced. The diversity of the proposed techniques ranges from simple linear time techniques using breadth-first search / depth-first search to efficient logarithmic algorithms. Though, due to the fast growing data sizes (just think of social network graphs of Facebook or Google+), even these efficient algorithms cannot deal with such big graphs. Thus, approaches to parallelize the detection of this components have already been developed for several decades: Hirschberg et al. [6] present an algorithm which uses n^2 processors (where $n = |V|$ denotes the number of vertices in the graph) and having a time complexity of $O(\log^2 n)$. Chin et al. [2] present a modified version of this algorithms which achieves the same time bound with only $n \left\lceil \frac{n}{\log^2 n} \right\rceil$ processors. A parallel algorithm with a time bound of $O(\log n)$ is presented by Shiloach and Vishkin [11]. Greiner [5] presents an overview of several parallel algorithms for connected components.

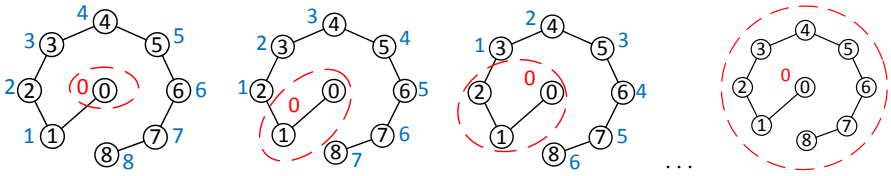
All of the aforementioned approaches assume that all computing processors have access to a shared memory and, thus, can access the same data. In contrast, the MapReduce model relies on distributing the data as well as the computation between the computing nodes and, thus, reduce the required communication between the computing nodes. There also exist some approaches that are based on a "distributed memory" model, i.e they consider the cost of communication between the computing nodes, e.g., the approach proposed in [1] which is an extension of the algorithm of [8]. In their distributed memory model, every computing node is able to access the memory of other computing nodes, which, however, leads to certain communication costs. In contrast, the MapReduce model only allows for special communication flows. E.g., communication between different reducers in a MapReduce job is not possible. Thus, for computing connected components using MapReduce, special types of algorithms are necessary.

Recently, a few approaches for the detection of connected components using the MapReduce model were proposed. Wu et al. [12] present an algorithm for detecting connected components based on Label Propagation. PEGASUS [7] is a graph mining system based on MapReduce and also contains an algorithm for the detection of connected components. In this system, graph mining operations are represented as repeated matrix-vector multiplications. In [9] the problem is solved by finding a minimum spanning tree of the graph. For that, edges which certainly do not belong to any MST are iteratively removed until the subgraphs are small enough to be processed by a single machine. Two further algorithms were proposed in [10]. These aims at minimizing the number of iterations and communication per step. The authors provide probable bounds which are logarithmic in the largest component size but claim that in practice the number of iterations for one of the algorithms is at most $2 \log d$ (d=diameter of the graph).

In [3] another connected components algorithm based on MapReduce is presented. As this algorithm is the most similar one to our approach, it will be introduced in the following. In this algorithm, nodes are assigned to so-called zones, where each zone is identified by the vertex with the smallest ID contained in this zone. Initially, each node defines an own zone. The zones are then merged iteratively until finally each zone corresponds to a connected component of the

graph: In each iteration, each edge is tested whether it connects nodes from different zones. Subsequently the algorithm finds for each zone z the zone z_{min} with the smallest ID that is connected to z and adds all vertices of z to z_{min}.

A drawback of this algorithm is that for each iteration, three MapReduce jobs have to be executed and in each iteration all edges of the original graph have to be processed.



(a) Initial graph (b) First Iter. (c) Second Iter. (d) Last Iter.

Fig. 1. Example for the algorithm from [3]

In Fig. 1 we show the processing of this algorithm for a simple example graph consisting of just one component. The numbers inside the vertices are the IDs of the vertices, the numbers beside the vertices denote the number of the zone a vertex is currently assigned to. The vertices that are already assigned to the "final" zone 0 are encircled. Initially, each vertex is assigned to its own zone. In the first iteration, the algorithm determines the edges that connect vertices from different zones, i.e. the zones 0 and 1. Then, i.e. for zone 1 the algorithm detects that the smallest zone connected to it is zone 0, i.e. the vertex 1 is now assigned to zone 0. Similarly, the vertex 2 is assigned to zone 1 etc.. In the second iteration, the same processing is done, i.e. vertex 2 is added to zone 1, vertex 3 (former zone 2) is added to zone 1 etc. Overall, the algorithm needs 8 iterations to detect the component. This example shows another drawback of the algorithm: Although in the first iteration e.g. the connection between zone 1 and zone 0 and the connection between zone 1 and zone 2 are detected, this information is not used in the second iteration. Using this information, we could e.g. directly add the vertex 3 from zone 2 to zone 0, as we know they are connected via zone 1 and thus have to belong to the same connected component. By neglecting these information, the algorithm needs a large number of iterations.

The basic idea of our new algorithm is to use this kind of information from previous iterations by adding additional edges ("shortcuts") in the graph such that fewer iterations are needed to find the final components. In our experimental section we compare our approach to the approaches from [3] and [7].

4 Algorithm

In this section we present our CC-MR-algorithm for detecting components in large-scale graphs using the MapRecuce framework. In subsection 4.1 we describe

our solution. For better understanding, we show in section 4.2 the processing of CC-MR on the example from section 3. Section 4.3 provides a formal proof of the correctness of CC-MR.

The basic idea of our algorithm is to iteratively alter growing local parts of the graph until each connected component is presented by a star-like subgraph, where all nodes are connected to the node having the smallest ID. For that, in each iteration, we add and delete edges such that vertices with larger IDs are assigned to the reachable vertex with smallest ID. Applying an intelligent strategy to use the information from previous iterations, our algorithm needs significantly less iterations than existing approaches.

4.1 CC-MR algorithm

Basically there are two states for a (sub)component S: either it is already maximal or there are still further subcomponents which S can be merged with. The main question of every algorithm for finding connected components is, therefore, how to efficiently recognize those two states and how to react on them. I.e., if a component is already maximal, no further step should be performed, and in the second case the merging or some other equivalent action should be done with as little effort as possible. When dealing with parallel algorithms the question of balanced distribution of the calculations arises. Considering a distributed memory programming model like MapReduce additionally complicates the problem since an efficient information flow between independent computational nodes has to be established.

We propose a solution to handle the aforementioned states locally for every graph vertex in such a way that after at most linearly many iterations (experiments often show a logarithmic behavior for big graphs) in terms of the diameter of the largest component the solution is found. Pushing down the problem to the single vertices of the graph enables a very scalable processing in the MapReduce framework. Additionally, by using techniques for prevention of duplicated data, which often appears in distributed memory models, CC-MR-algorithm significantly outperforms the state-of-the art approaches as e.g. [3].

Let $G = (V, E)$ be an undirected graph where V is a set of vertices with IDs from $\mathbb{Z}$ and $E = \{(v_{source}, v_{dest}) \in V^2\}$ is a set of edges. The algorithm's basic idea is simple: independently check for each vertex v and its adjacent vertices $adj(v)$ whether v has the smallest ID or not. If yes (*locallyMaxState*), assign all $u \in adj(v)$ to v and stop the processing of v, since the component of v is already locally maximal. Otherwise (*mergeState*), there is a vertex $u \in adj(v)$ with $u < v$; then connect v and $adj(v)$ to u. This corresponds to assigning (merging) the component of v to the component of u. By iteratively performing these steps each component is finally transformed to a star-like subgraph where the vertex having the smallest ID is the center. The overall algorithm stops as far as the *mergeState* situation does not occur any more.

In the following, we present an efficient implementation based on a simple concept of *forward* and *backward edges*. We call an edge $v \to u$ a forward edge, if $v < u$, and a backward edge, if $v > u$, where the comparison of vertices means

the comparison of their IDs. Both types of edges are represented by a tuple (v, u) which we represent by $(key, value)$ pairs in the MapReduce framework. The semantic of the forward edge (v, u) is that vertex u belongs to the component of the vertex v. The backward edge can be regarded as a "bridge" between the component of vertex u and the component of a vertex w which has a connection to v as shown in Fig. 2. The backward edge between vertices v and u enables the reducer of v to connect u and w in a single iteration of the algorithm.

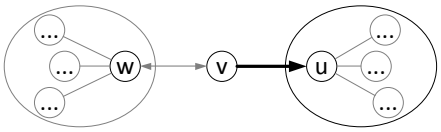


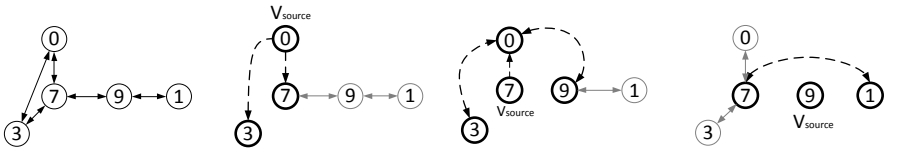
Fig. 2. The backward edge (v, u) connects components of w and u in the reducer of v.

These concepts will become clearer from the explanation of the algorithm.

Listing 1.1. Reducer implementation.

```
1 newIterationNeeded = false // global variable
2 void reduce(int v_source, Iterator<int> values)
3   isLocMaxState = false
4   v_first = values.next();       // take first element
5   if ( v_source < V_first )
6     isLocMaxState = true
7     emit(v_source, v_first)
8   v_dest_old = v_first
9   while ( values.hasNext() )
10    v_dest = values.next()
11    if ( v_dest == v_dest_old ) continue // remove duplicates
12    if ( isLocMaxState )        // locMaxCase
13      emit( v_source, v_dest ) // only fwd. edge
14    else  // cases stdMergeCase, optimizedMergeCase
15      emit( v_first, v_dest )  // fwd. edge and
16      emit( v_dest, v_first )  // backwd. edge
17      newIterationNeeded = true
18    v_dest_old = v_dest
19  // stdMergeCase
20  if ( v_source < v_dest && !isLocMaxState )
21    emit( v_source, v_first ) // backwd. edge
```

As described earlier, a MapReduce job consists of a map and a reduce phase. In our case, the mapper is a so-called identity mapper which simply passes all read data to the reducer without performing any changes. The pseudo-code for the reduce phase is given in listing 1.1. The emitted edges (v_{source}, v_{dest}) are automatically grouped by their v_{source} values and then sorted in ascendent order of their v_{dest}-values. Technically we use the secondary sort method of Hadoop



(a) Original graph (b) *locMaxCase* (c) *stdMergeCase* (d) *optimizedMergeCase*

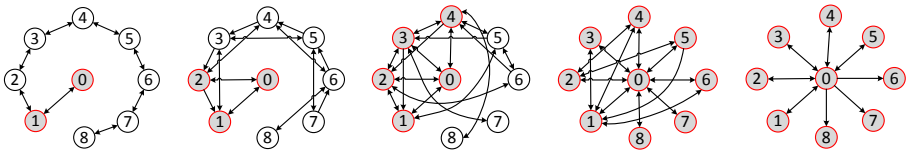
Fig. 3. Three cases of the algorithm. v_{source} strings marks the node under consideration in the considered reducer. Bold highlighted nodes are adjacent nodes of v_{source}.

to establish the desired sorting. The main part of the algorithm is located in the reducer (listing 1.1), where both aforementioned cases are handled. Tuples having the same key arrive as a data stream and are processed one after another in the 'while' loop. After the elimination of duplicate entries in the line 11, three cases are distinguished:

- *locMaxCase*: lines 5–7 and 12–13
- *optimizedMergeCase*: lines 15–18
- *stdMergeCase*: lines 15–18 and 20 – 21

locMaxCase corresponds to the *locallyMaxState*, i.e., it deals with the situation when a local maximal component with root v_{source} is already found and therefore all adjacent nodes $v_{dest} \in adj(v_{source})$ have to be assigned to v_{source}. This assignment is performed by emitting forward edges $v_{source} \rightarrow v_{dest}$ in the lines 7 and 13. Fig. 3(b) depicts the processing of the case *locMaxCase* by showing the changes of the original graph structure from Fig. 3(a). Nodes marked by v_{source} are the nodes which are considered in single reducer with all its adjacent nodes, which for their part are highlighted by bold circles. The dimmed circles show the remaining vertices of the graph which are not regarded during the computation of the node v_{source}. Dashed arrows represent the newly created edges inside the reducer. In this example the reducer of the node 0 emits therefore two edges $0 \rightarrow 3$ and $0 \rightarrow 7$. Cases *stdMergeCase* and *optimizedMergeCase* on their part deal with the merge state (*mergeState*), where *optimizedMergeCase* is a special case of *stdMergeCase* which reduces duplicate edges, as will be shown later. Both cases arise, if the condition $v_{source} > v_{first}$ holds, which means that at least one of the adjacent nodes $v_{dest} \in adj(v_{source})$ has a smaller ID than v_{source}. Due to the fact that the vertices are sorted in order of their IDs, v_{first} has the smallest value. Since the main aim of the algorithm is to assign all vertices with larger IDs to the vertex with smallest ID, this implies that all vertices in $adj(v_{source})$ except for v_{first} itself are assigned to v_{first}, i.e., for each of this vertices a forward edge $v_{first} \rightarrow v_{dest}$ (line 15) is emitted. Please note that this is not the case for the edge $v_{first} \rightarrow v_{source}$, since this edge will be emitted in the reducer of the vertex v_{first}. Therefore, in the example of *stdMergeCase* in Fig. 3(c) the edges $0 \rightarrow 3$ and $0 \rightarrow 9$ are emitted.

In addition to the forward edges, the algorithm emits backward edges $v_{dest} \rightarrow v_{first}$ (line 16), i.e., edges $3 \rightarrow 0$ and $9 \rightarrow 0$ in the example. These edges form



(a) Initial graph (b) Iteration 1 (c) Iteration 2 (d) Iteration 3 (e) Iteration 4

Fig. 4. Example for CC-MR

"bridges" or "shortcuts" between components and are needed due to the fact that v_{dest} (nodes 3, 9) could be connected to some other vertex w with even smaller ID than 0 such that at some point of time node 0 could have to be connected to w. If there were no backward edge $v_{dest} \rightarrow v_{first}$ then there would not be any reducer which would be able to merge 0 and w.

Because of the same arguments, the backward edge $v_{source} \rightarrow v_{first}$ should be actually emitted too. This indeed happens in the case when v_{source} is smaller than one of its adjacent vertices (lines 20 and 21). If v_{source} has the biggest ID among all its adjacent nodes (*optimizedMergeCase*), then this edge can be omitted due to the fact that all adjacent nodes of v_{source} are already reassigned to the vertex with smallest ID and therefore v_{source} will never deal as a bridge node between two components. In Fig. 3(d) case *optimizedMergeCase* is depicted.

The identity mapper and the reducer from Listing 1.1 form one iteration of the CC-MR-algorithm. These jobs have to be iterated as long as there are subcomponents which can be merged. In order to recognize this case, we have to check whether in the last iteration a backward edge was emitted. If this is the case then there are still subcomponents which could be merged and a new iteration has to be started. Otherwise, all components are maximal and the algorithm can stop. The information whether backward edges were created or not is indicated by the global variable *newIterationNeeded*, which can be implemented as a global counter in Hadoop. Setting the value of this variable to e.g. value 1 indicates the boolean value 'true' and value 0 indicates 'false'. This variable is set to true if either case *stdMergeCase* or case *optimizedMergeCase* holds (line 17).

4.2 Example for the processing of CC-MR

In Fig. 4 we show the processing of CC-MR using the same example that was used in section 3 for the algorithm from [3]. For each iteration, we show the edges that the algorithm emits in this iteration. The vertices that are already connected to the vertex with the smallest ID 0 are marked in the graph for each iteration. In table 1, the output of the single iterations is shown as lists of edges. For each iteration, the all edges that are emitted in this iteration are shown, sorted by their key vertices. Some edges occur repeatedly in the same iteration. This is due to the fact that the same edge can be generated by different reducers.

In the initial graph, for each edge $(u, v) \in E$ both directions, i. e. (u, v) and (v, u) are given. In the first iteration, the reducers mostly insert "two-hop"

iter.	0	1	2	3	4	5	6	7	8
0	0-1	1-0	2-1	3-2	4-3	5-4	6-5	7-6	8-7
		1-2	2-3	3-4	4-5	5-6	6-7	7-8	
1	0-1	1-0	2-0	3-1	4-2	5-3	6-4	7-5	8-6
	0-2	1-3	2-1	3-2	4-3	5-4	6-5	7-6	
			2-4	3-5	4-6	5-7	6-8		
2	0-1 (2x)	1-0	2-0	3-0	4-0	5-1	6-2	7-3	8-4
	0-2	1-0	2-1	3-1	4-2	5-3	6-4		
	0-3	1-2	2-3	3-2	4-3	5-4	6-5		
	0-4	1-5	2-6	3-4	4-5	5-6			
				3-7	4-8				
3	0-1 (3x)	1-0	2-0	3-0	4-0	5-0	6-0	7-0	8-0
	0-2 (4x)	1-0	2-0	3-0	4-0	5-0	6-1		
	0-3 (3x)	1-0	2-0	3-0	4-1	5-1			
	0-4 (2x)	1-3	2-0	3-1	4-2	5-2			
	0-5 (2x)	1-4	2-4						
	0-6	1-6	2-5						
	0-7								
	0-8								

iter.	0	1	2	3	4	5	6	7	8
4	0-1 (5x)	1-0	2-0	3-0	4-0	5-0	6-0		
	0-2 (3x)	1-0	2-0		4-0				
	0-3 (2x)	1-0	2-0						
	0-4 (3x)	1-0							
	0-5 (2x)	1-0							
	0-6 (2x)								
	0-7								
	0-8								
5	0-1								
	0-2								
	0-3								
	0-4								
	0-5								
	0-6								
	0-7								
	0-8								

Table 1. Edges generated by CC-MR for the example graph

edges, e.g. the reducer for the vertex 3 connects the vertices 2 and 4. In the second iteration, for example, the reducer for the vertex 2 inserts an edge between the vertices 0 and 4 and the reducer of the vertex 6 inserts an edge between 4 and 8. Thus, vertices are already connected to each other that had a shortest path of 4 in the initial graph. In the third iteration, finally all vertices have direct connections to the node 0. However, to obtain a star graph for this compoment and thus to detect that the final component as already been found, the algorithm still has to delete all edges (v, w) with $v, w \neq 0$. This is done in iteration 4. Though the resulting graph is already the desired star graph, some vertices (i.e. vertex 3) still have backward edges, which are finally removed in a fifth iteration (not depicted here). Overall, our algorithm needs 5 iterations for this example. In comparison, the algorithm from [3] needed 8 iterations in the same example.

4.3 Proof of correctness

In this section we present a proof for the correctness of CC-MR, i.e. we show that CC-MR correctly detects the connected components of the graph. As presented in the previous section, the idea of our algorithm is to add and delete edges in the graph such that the resulting graph consists of one star graph per component where the center of each star graph is the vertex with the smallest ID from the corresponding component. Thus, in each iteration we have a different set of edges in the graph. Let E_i denote the set of edges that exist after iteration i.

To prove that the resulting graphs of the algorithm really correspond to the connected components, we prove two different steps:

1. An edge (v_1, v_2) is emitted in iteration $i \Rightarrow$ There has already been a path between v_1 and v_2 in iteration $i - 1$.
   (We never add edges between vertices that were not in the same component before.)

2. There exists a path between v_1 and v_2 in iteration $i-1 \Rightarrow$ there exists a path between them in iteration i.
   (We do not disconnect components that existed before).

Steps 1 and 2 together show that although the algorithm adds and removes edges in the graph, the (weak) connected components do never change during the algorithm. Please note that as our input graph is undirected, the connectedness of the components does not depend on the directions of the edges, even though CC-MR sometimes only adds one direction of an edge for optimization reasons. Thus, for the paths we construct in our proof the directions of the edges are neglected. In the following we present the proofs for the single steps:

1. In CC-MR, edges are only added in the reducers. Thus, to add an edge $(v_1, v_2) \in E_i$, the vertices v_1 and v_2 have to occur together in the reducer of some vertex v_{source}. Therefore, for each $v_j, j \in \{1,2\} : v_j = v_{source}$ or $(v_{source}, v_j) \in E_{i-1}$. Thus, there existed a path between v_1 and v_2 in the iteration $i-1$.
2. It suffices to show: There exists an *edge* $(v_1, v_2) \in E_{i-1} \Rightarrow$ there exists a path between them in iteration i (Because a path between some vertices u and w in E_{i-1} can be reconstructed in E_i by replacing each edge $(v_1, v_2) \in E_{i-1}$ on the path by its corresponding path in E_i):
   Case 1: $v_1 < v_2$ (i.e. (v_1, v_2) is a forward edge):
   (v_1, v_2) is processed in the reducer of $v_{source} = v_1$. Now we can look at the three different cases that can occur in the reducer:
   - *locMaxCase*: (v_1, v_2) is emitted again.
   - *stdMergeCase*: We emit edges (v_{first}, v_2), (v_2, v_{first}) and (v_1, v_{first}), thus there still exists a path between v_1 and v_2.
   - *optimizedMergeCase*: Not possible because $v_1 < v_2$.

   Case 2: $v_1 > v_2$ (i.e. (v_1, v_2) is a backward edge):
   For this case, we can show that in some iteration $i_x \leq i-1$ also the corresponding forward edge (v_2, v_1) was emitted:
   For the backward edge $(v_1, v_2) \in E_{i-1}$, there are two possible scenarios where (v_1, v_2) can have been emitted:
   - $(v_1, v_2) \in E_{i-1}$ can have been emitted by the reducer of v_1 in the case *stdMergeCase*. In this case, the edge has already existed in the previous iteration , i.e. $(v_1, v_2) \in E_{i-2}$, else the vertex v_2 would not be processed in the reducer of v_1.
   - $(v_1, v_2) \in E_{i-1}$ can have been emitted by the reducer of a vertex v_{source} with $v_{source} \neq v_1$ and $v_{source} \neq v_2$ in the case *stdMergeCase* or *optimizedMergeCase*. In this case, also the forward edge (v_2, v_1) has been emitted.
   - $i-1 = 0$, i.e. the edge (v_1, v_2) already existed in the initial graph. Then, by definition of the original (undirected) graph, also the forward edge (v_2, v_1) existed.

   Thus, we know that for each backward edge (v_1, v_2), the corresponding forward edge (v_2, v_1) exists or has existed in an earlier iteration. In case 1 it was already shown that the path between v_2 and v_1 is preserved in the following iterations. □

4.4 Dealing with large components

In CC-MR a single reducer processes a complete component, which can result in high workloads of a single reducer for very large components. Now we briefly present a solution which distributes the calculations for large components over multiple reducers and that way balances the workload. Consider the example in Fig. 5, in which vertex 7 is a center of a component with too many elements. Assume that we have this information from a previous iteration and we want to distribute the computations on three reducers. For that, in the map-phase

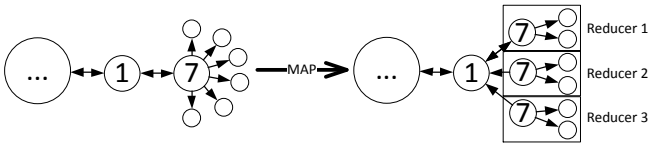


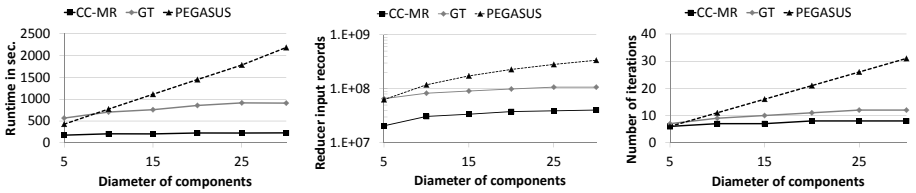
Fig. 5. Example for workload balancing for big components.

each forward edge $(7, x)$ is augmented by a hash value to $((7, hash(x)), x)$ which is then used in the partitioner in order to distribute edges $(7, \cdot)$ to different reducers. In the example a small circle represents such a vertex x, which is then sent to one of the three reducers. For backward edges (in the example $(7, 1)$), a set of edges $\{((7, hash(i)), 1) | i = 1, 2, 3\}$ is produced, which guarantees that each vertex 7 in each of the three reducers has a backward edge to 1 and can reassign its neighbors to 1 in the reduce phase. All other edges whose source vertices do not have too many neighbors are not augmented by a hash value and therefore are processed as in the original algorithm.
The reduce-phase remains as in the original algorithm, with the difference that for each vertex the number of neighbors in E_i is determined and for vertices with too many neighbors the value is stored in the distributed cache for the next iteration. This simple strategy produces almost no additional overhead but achieves a very good balancing of the workload as will be shown in the experiments.

5 Experiments

In this section we present the experimental evaluation of the CC-MR approach, comparing it to the approach from [3] (denoted as 'GT' here) and to the approach from the Pegasus system [7]. All experiments were performed on a cluster running Hadoop 0.20.2 and consisting of 14 nodes with 8 cores each that are connected via a 1 Gbit network. Each of the nodes has 16 Gb RAM. For each experiment the number of distance computations and the runtime is measured[3].

[3] The source code of the CC-MR-algorithm and the used datasets can be found at: `http://dme.rwth-aachen.de/en/ccmr`



(a) Component diameter vs. runtime (b) Component diameter vs. number of input records (c) Component diameter vs. number of iterations

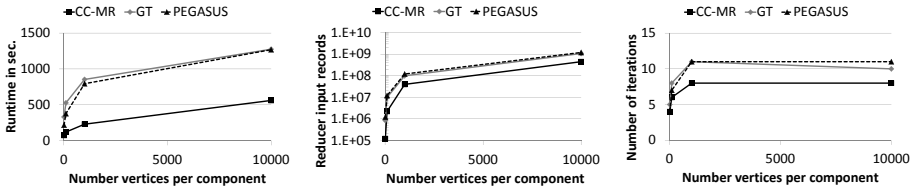
Fig. 6. Performance for varying component diameters

5.1 Scalability on synthetic data

In this section we evaluate CC-MR with respect to different properties of the input graphs. Therefore, we use synthetic datasets such that in each experiment, one property of the generated graphs changes while the others remain stable.

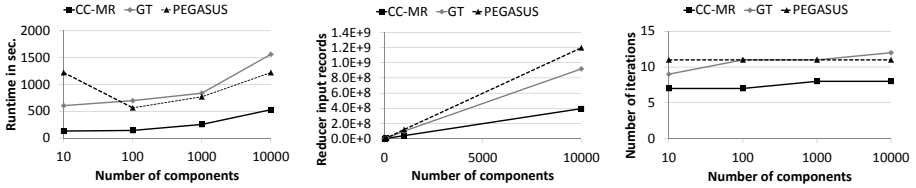
Performance for varying component diameters In this experiment we vary the diameter of the connected components in our synthetic datasets. Each dataset consists of one million vertices and is divided into 1000 connected components with 1000 vertices each. The diameter of the generated components is varied from 10 to 30 in this experiment.
In Fig. 6(a), the runtime of the algorithms is depicted. For all algorithms the runtime increases for higher diameters. However, the runtime of CC-MR is always significantly lower (at least a factor of two) than that of GT and Pegasus and scales better for increasing diameters. In Fig. 6(c) we show the number of iterations needed by the algorithms to find the connected components. As expected, for higher diameters the number of iterations increases for all algorithms. For CC-MR, the number of iterations is always lower than for GT and Pegasus. In Fig. 6(b), we depict for each algorithm the number of input records (summed over all iterations) that are processed, i.e. the number of records that are communicated between the iterations. For all algorithms, this number increases with increasing diameter. The number of records for CC-MR is significantly lower (at least by a factor of two) than that of GT and Pegasus, due to the fact that they perform more iterations than CC-MR and GT needs to perform 3 MapReduce jobs per iteration.

Performance for varying component sizes In this experiment we vary the number of vertices per component in the generated graphs and examine its influence on the runtime and the number generated records in reducers. The created datasets consist of 1000 components with 15, 150, 1500 and 15000 edges and 10, 100, 1000, 10000 vertices, respectively. The figures 7(a), 7(b) and 7(c) depict the results for the runtime (in sec.), the number of records, and the iteration number respectively. As expected, all of these values increase with growing size of the components. The runtime of the CC-MR algorithm remains smaller (up to a factor of 2) then the runtimes of GT and Pegasus for each component size.



(a) Component size vs. runtime (b) Component size vs. number of input records (c) Component size vs. number of iterations

Fig. 7. Performance for varying component sizes



(a) Number of components vs. runtime (b) Number of components vs. number of input records (c) Number of components vs. number of iterations

Fig. 8. Performance for varying numbers of components

Performance for varying numbers of components This experiment shows the dependency between the number of components in a graph, the runtime, the number of input records and the number of iterations. The synthetic datasets consist of 10, 100, 1000, 10000 components each with 1000 vertices and 1500 edges. The figures 8(a), 8(b) and 8(c) depict the runtime, the processed number of input records and the number of iterations. Similar to previous results, CC-MR has a much lower runtime and produces in the worst case (10 components) 15% and in the best case (10000 components) over 55% less input records compared to the other approaches. Furthermore, CC-MR outperforms both competitors in terms of the number of performed iterations.

5.2 Real-world data

We use three real-world datasets to evaluate CC-MR: a web graph (Web-google) and two collaboration networks (IMDB, DBLP). The Web-google dataset can be found at `snap.stanford.edu/data/web-Google.html` and consists of 875713 nodes, 5105039 edges and 2746 connected components. In this web graph, the vertices represent web pages and the edges represent hyperlinks between them. In the IMDB dataset (176540 nodes; 19992184 edges; 16 comps.), actors are represented by vertices, edges between actors indicate that the actors worked together in some movie. The data was extracted from the IMDB movie database (`imdb.com`). In the DBLP dataset (generated using the data from `dblp.org`, 553797 nodes; 1677487 edges; 24725 comps.) each vertex corresponds to an author, while each edge represents a collaboration between two authors.

	CC-MR			GT			PEGASUS		
	web-google	imdb	dblp	web-google	imdb	dblp	web-google	imdb	dblp
Runtime (sec.)	535	1055	472	3567	3033	4385	4847	2834	3693
#iters	8	6	7	10	6	12	16	6	15
#input rec. ($\cdot 10^6$)	29	179	27	102	564	210	292	299	108

Table 2. Results of CC-MR, GT and PEGASUS.

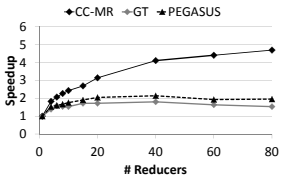


Fig. 9. Speedup for varying number of reduce nodes

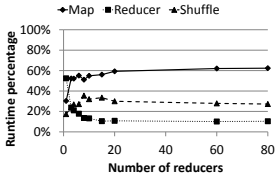


Fig. 10. CC-MR-Runtime distribution among map, shuffle and reduce phases.

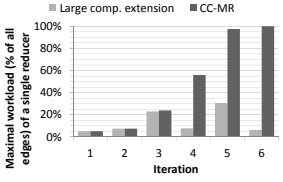


Fig. 11. Maximal reducer workload for processing large components.

The results of the comparison are given in Table 2. CC-MR clearly outperforms GT and PEGASUS in terms of the number of iterations as well as in the runtime and the communication (i.e the number of input records).

Figure 9 depicts the scalability results of the evaluated algorithms. The CC-MR-algorithm shows a speedup more than twice as high compared with competing approaches. As Figure 10 shows, the reduce time of our approach decreases very fast with growing number of reducers. The moderate overall speedup of 4.7 with 80 reducers puts down to the fact that the map phase does not depend on the number of used reducers and therefore the speedup of calculations is limited by I/O speed. The significant communication reduction of our approach is therefore a very big advantage in comparison to the competing approaches.

5.3 Load balancing for large components

Fig. 11 shows the load balancing properties of our large component extension (cf. Section 4.4) on the IMDB dataset using 20 reducers and threshold for the maximal size of a component set to 1% of the number of edges in an iteration. This dataset contains 16 components, 15 small ones and one, C_{max}, containing more than 99% of the vertices. In the first three iterations both algorithms perform similarly well and distribute the workload almost evenly among all reducers. In iteration 4, however, the reducer R_{max} of the CC-MR responsible for C_{max} already processes about 50% of all edges remaining in the iteration, as more and more vertices are assigned to C_{max}. This trend goes on until in the last iteration almost all vertices of the graph are processed by R_{max}. In contrast, our extended algorithm is able to balance the workload after each iteration when a disbalance occurs, as it is the case in the iterations 3 and 5.

6 Conclusion

In this paper, we propose the parallel algorithm CC-MR for the detection of the connected components of a graph. The algorithm is built on top of the MapReduce programming model. CC-MR effectively manipulates the graph structure to reduce the number of needed iterations and thus to find the connected components more quickly. Furthermore, we propose an extension to CC-MR to deal with heterogeneous component sizes. Apart from the description of the algorithm, we also provide a proof for the correctness of CC-MR. The performance of CC-MR is evaluated on synthetic and real-world data in the experimental section. The experiments show that CC-MR constantly outperforms the state-of-the-art approaches.

References

1. L. Bus and P. Tvrdík. A parallel algorithm for connected components on distributed memory machines. In *PVM/MPI*, pages 280–287, 2001.
2. F. Y. L. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, 1982.
3. J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.
4. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
5. J. Greiner. A comparison of parallel algorithms for connected components. In *SPAA*, pages 16–25, 1994.
6. D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, 1979.
7. U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.
8. A. Krishnamurthy, S. Lumetta, D. Culler, and K. Yelick. Connected components on distributed memory machines. *DIMACS implementation challenge*, 30:1, 1997.
9. S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, pages 85–94, 2011.
10. V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components on map-reduce in logarithmic rounds. *Computing Research Repository (CoRR)*, abs/1203.5387, 2012.
11. Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.
12. B. Wu and Y. Du. Cloud-based connected component algorithm. In *Artificial Intelligence and Computational Intelligence (AICI)*, volume 3, pages 122 –126, 2010.