Bisimulation Reduction of Big Graphs on MapReduce

Yongming Luo[1], Yannick de Lange[1], George H. L. Fletcher[1], Paul De Bra[1], Jan Hidders[2], and Yuqing Wu[3]

[1] Eindhoven University of Technology, The Netherlands
{y.luo@,y.m.d.lange@student.,g.h.l.fletcher@,P.M.E.d.Bra@}tue.nl
[2] Delft University of Technology, The Netherlands
a.j.h.hidders@tudelft.nl
[3] Indiana University, Bloomington, USA
yuqwu@cs.indiana.edu

Abstract. Computing the bisimulation partition of a graph is a fundamental problem which plays a key role in a wide range of basic applications. Intuitively, two nodes in a graph are bisimilar if they share basic structural properties such as labeling and neighborhood topology. In data management, reducing a graph under bisimulation equivalence is a crucial step, e.g., for indexing the graph for efficient query processing. Often, graphs of interest in the real world are massive; examples include social networks and linked open data. For analytics on such graphs, it is becoming increasingly infeasible to rely on in-memory or even I/O-efficient solutions. Hence, a trend in Big Data analytics is the use of distributed computing frameworks such as MapReduce. While there are both internal and external memory solutions for efficiently computing bisimulation, there is, to our knowledge, no effective MapReduce-based solution for bisimulation. Motivated by these observations we propose in this paper the first efficient MapReduce-based algorithm for computing the bisimulation partition of massive graphs. We also detail several optimizations for handling the data skew which often arises in real-world graphs. The results of an extensive empirical study are presented which demonstrate the effectiveness and scalability of our solution.

1 Introduction

Recently, graph analytics has drawn increasing attention from the data management, semantic web, and many other research communities. Graphs of interest, such as social networks, the web graph, and linked open data, are typically on the order of billions of nodes and edges. In such cases, single-machine in-memory solutions for reasoning over graphs are often infeasible. Naturally, research has turned to external memory and distributed solutions for graph reasoning. External memory algorithms often suffice, but their performance typically scales (almost) linearly with graph size (usually the number of graph edges), which is then limited by the throughput of the I/O devices attached to the system. In this respect, distributed and parallel algorithms become attractive. Ideally, a

well-designed distributed algorithm would scale (roughly) linearly with the size of the computing resources it has, making use of the parallelism of the infrastructure. Though there are many alternatives, recently the MapReduce platform [8] has become a *de-facto* parallel processing platform for reasoning over Big Data such as real-world graphs, with wide adoption in both industry and research.

Among fundamental graph problems, the bisimulation partition problem plays a key role in a surprising range of basic applications [24]. Informally, the bisimulation partition of a graph is an assignment of each node n of the graph to a unique block consisting of all nodes having the same structural properties as n (e.g., node label and neighborhood topology). In data management, variations of bisimulation play a fundamental role in constructing structural indexes for XML and RDF databases [21, 23], and many other applications for general graph data such as compression [4, 11], query processing [16], and data analytics [10]. Being well studied for decades, many main-memory efficient algorithms have been developed for bisimulation partitioning (e.g., [9, 22]). The state-of-the-art I/O efficient algorithm takes just under one day to compute a standard "localized" variant of bisimulation on a graph with 1.4 billion edges on commodity hardware [20]. This cost can be a potential bottleneck since bisimulation partitioning is typically one step in a larger workflow (e.g., preparing the graph for indexing and query processing).

Contributions. Motivated by these observations, we have studied the effective use of the MapReduce framework for accelerating the computation of bisimulation partitions of massive graphs. In this paper we present, to our knowledge, the first efficient MapReduce-based algorithm for localized bisimulation partitioning. We further present strategies for dealing with various types of skew which occur in real-world graphs. We discuss the results of extensive experiments which show that our approach is effective and scalable, being up to an order of magnitude faster than the state of the art. As a prototypical graph problem, we hope that sharing our experiences with graph bisimulation will stimulate further progress in the community on MapReduce-based solutions for reasoning and analytics over massive graphs.

Related work. While there has been work on distributed computation of bisimulation partitions, existing approaches (e.g., [3]) are not developed for the MapReduce platform, and hence are not directly applicable to our problem. Research has been conducted to investigate using the MapReduce framework to solve graph problems [6, 18] right after the framework was proposed. A major issue here is dealing with data skew in graphs. Indeed, skew is ubiquitous in real-world graphs [5]. During our investigation, we also experienced various types of skew from graph bisimulation, as we discuss below. There has been much progress done from the system side to tackle this problem. The main approach in this literature is to devise mechanisms to estimate costs of MapReduce systems (e.g., [13]) and modify the system to mitigate the skew effects, both statically [12, 15, 17] and dynamically [17, 25], so that the modification is transparent to the users. However, it is still possible to gain much efficiency by dealing with

skew from the algorithm design perspective [19], as we do in the novel work we present in this paper.

Paper organization. In the next section we give a brief description of localized bisimulation and MapReduce. We then describe in Section 3 our base algorithm for computing localized bisimulation partitions using MapReduce. Next, Section 4 presents optimizations of the base algorithm, to deal with the common problem of skewed data. Section 5 presents the results of our empirical study. We then conclude in Section 6 with a discussion of future directions for research.

2 Preliminaries

2.1 Localized bisimulation partitions

Our data model is that of finite directed node- and edge-labeled graphs $\langle N, E, \lambda_N, \lambda_E \rangle$, where N is a finite set of nodes, $E \subseteq N \times N$ is a set of edges, λ_N is a function from N to a set of node labels $\mathcal{L}_N$, and λ_E is a function from E to a set of edge labels $\mathcal{L}_E$.

The localized bisimulation partition of graph $G = \langle N, E, \lambda_N, \lambda_E \rangle$ is based on the *k-bisimilar* equivalence relation.

Definition 1. *Let $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph and $k \geq 0$. Nodes $u, v \in N$ are called* k-bisimilar *(denoted as $u \approx^k v$), iff the following holds:*

1. $\lambda_N(u) = \lambda_N(v)$,
2. *if $k > 0$, then for any edge $(u, u') \in E$, there exists an edge $(v, v') \in E$, such that $u' \approx^{k-1} v'$ and $\lambda_E(u, u') = \lambda_E(v, v')$, and*
3. *if $k > 0$, then for any edge $(v, v') \in E$, there exists an edge $(u, u') \in E$, such that $v' \approx^{k-1} u'$ and $\lambda_E(v, v') = \lambda_E(u, u')$.*

Given the k-bisimulation relation on a graph G, we can assign a unique partition identifier (e.g., an integer) to each set of k-bisimilar nodes in G. For node $u \in N$ and relation $\approx^k$, we write $pId_k(u)$ to denote u's k-partition identifier, and we call pId_k a *k-partition identifier function.*

Definition 2. *Let $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph, $k \geq 0$, and $\{pId_0, \ldots, pId_k\}$ be a set of i-partition identifier functions for G, for $0 \leq i \leq k$. The* k bisimulation signature *of node $u \in N$ is the pair $sig_k(u) = (pId_0(u), L)$ where:*

$$L = \begin{cases} \emptyset & \text{if } k = 0, \\ \{(\lambda_E(u, u'), pId_{k-1}(u')) \mid (u, u') \in E\} & \text{if } k > 0. \end{cases}$$

We then have the following fact.

Proposition 1 ([20]). *$pId_k(u) = pId_k(v)$ iff $sig_k(u) = sig_k(v), k \geq 0$.*

Since there is a one-to-one mapping between a node's signature and its partition identifier, we can construct $sig_k(u)$ ($\forall u \in N, k \geq 0$), assign $pId_k(u)$ according to $sig_k(u)$, and then use $pId_k(u)$ to construct $sig_{k+1}(u)$, and so on. We call

each such construct-assign computing cycle an *iteration*. This signature-based approach is robust, with effective application in non-MapReduce environments (e.g., [3, 20]). We refer for a more detailed discussion of localized bisimulation to Luo et al. [20].

Example. Consider the graph in Figure 1. In iteration 0, nodes are partitioned into blocks *P1* and *P2* (indicated by different colors), based on the node label *A* and *B* (Def. 1). Then in iteration 1, from Def. 2, we have $sig_1(1) = (1, \{(w, P1), (l, P2)\})$ and $sig_1(2) = (1, \{(w, P1), (l, P2)\})$, which indicates that $pId_1(1) = pId_1(2)$ (Prop. 1). If we further compute 2-bisimulation, we see that $sig_2(1) \neq sig_2(2)$, and we conclude that nodes 1 and 2 are not 2-bisimilar, and block *P1* will split.

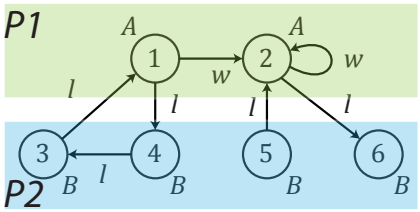


Fig. 1. Example graph

The partition blocks and their relations (i.e., a "structural index") can be seen as an abstraction of the real graph, to be used, for example, to filter unnecessary graph matching during query processing [11, 23]. A larger k leads to a more refined partition, which results in a larger structural index. So there is a trade-off between k and the space we have for holding the structural index. In practice, though, we see that a small k value (e.g., $k \leq 5$) is already sufficient for query processing. In our analysis below, we compute the k-bisimulation result up to $k = 10$, which is enough to show all the behaviors of interest for structural indexes.

2.2 MapReduce framework

The MapReduce programming model [8] is designed to process large datasets in parallel. A MapReduce *job* takes a set of key/value pairs as input and outputs another set of key/value pairs. A MapReduce *program* consists of a series of MapReduce jobs, where each MapReduce job implements a *map* and a *reduce* function ("[ ]" means a list of elements):

$$\begin{array}{lll} map & (k_1, v_1) & \rightarrow [(k_2, v_2)] \\ reduce & (k_2, [v_2]) & \rightarrow [(k_3, v_3)]. \end{array}$$

The *map* function takes key/value pair (k_1, v_1) as the input, emits a list of key/value pairs (k_2, v_2). In the *reduce* function, all values with the same key are grouped together as a list of values v_2 and are processed to emit another list of key/value pairs (k_3, v_3). Users define the *map* and *reduce* functions, letting the framework take care of all other aspects of the computation (synchronization, I/O, fault tolerance, etc.).

The open source Hadoop implementation of the MapReduce framework is considered to have production quality and is widely used in industry and research [14]. Hadoop is often used together with the Hadoop Distributed File System (HDFS), which is designed to provide high-throughput access to application data. Besides *map* and *reduce* functions, in Hadoop a user can also write a

custom *partition* function, which is applied after the *map* function to specify to which reducer each key/value pair should go. In our work we use Hadoop for validating our solutions, making use of the *partition* function as well.

3 Bisimulation partitioning with MapReduce

For a graph $G = \langle N, E, \lambda_N, \lambda_E \rangle$, we arbitrarily assign unique (integer) identifiers to each node of N. Our algorithm for computing the k-bisimulation partition of G has two input tables: a node table (denoted as N_t) and an edge table (denoted as E_t). Both tables are plain files of sequential records of nodes and edges of G, resp., stored on HDFS. The schema of table N_t is as follows:

nId	node identifier
pId_{0_nId}	0 *bisimulation* partition identifier for the given *nId*
pId_{new_nId}	*bisimulation* partition identifier for the given *nId* from the current computation iteration

The schema of table E_t is as follows:

sId	source node identifier
tId	target node identifier
eLabel	edge label
pId_{old_tId}	*bisimulation* partition identifier for the given *tId* from the last computation iteration

By combining the idea of Proposition 1 and the MapReduce programming model, we can sketch an algorithm for k-bisimulation using MapReduce, with the following workflow: for each iteration i ($0 \leq i \leq k$), we first construct the signatures for all nodes, then assign partition identifiers for all unique signatures, and pass the information to the next iteration. In our algorithm, each iteration then consists of three MapReduce tasks:

1. task `Signature` performs a merge join of N_t and E_t, and create signatures;
2. task `Identifier` distributes signatures to reducers and assigns partition identifiers; and,
3. task `RePartition` sorts N_t to prepare it for the next iteration.

Note that a preprocessing step is executed to guarantee the correctness of the map-side join in task `Signature`. We next explain each task in details.

3.1 Task `Signature`

Task `Signature` (Algorithm 1) first performs a sort merge join of N_t and E_t, filling in the pId_{old_tId} column of E_t with pId_{new_nId} of N_t. This is achieved in the map function using a map-side join [18]. Then records are emitted grouping by *nId* in N_t and *sId* in E_t. In the reduce function, all information to construct a signature resides in [*value*]. So the major part of the function is to iterate through [*value*] and construct the signature according to Definition 2. After doing so, the node identifier along with its pId_{0_nId} value and signature are emitted to the next task. Note that the key/value pair input of `SignatureMapper` indicates

the fragments of N_t and E_t that need to be joined. The fragments need to be preprocessed before the algorithm runs.

Algorithm 1 task `Signature`

```
1: procedure SignatureMapper(key, value)
2:     perform map-side equi-join of N_t and E_t on nId and tId, fill in pId_{old_tId} with
       pId_{new_nId}, put all rows of N_t and E_t into records
3:     for row in records do
4:         if row is from E_t then
5:             emit (sId, the rest of the row)
6:         else if row is from N_t then
7:             emit (nId, the rest of the row)
```

```
1: procedure SignatureReducer(key, [value])                    ▷ key is nId or sId
2:     pairset ← {}
3:     for value in [value] do
4:         if (key,value) is from N_t then
5:             pId_{0_nId} ← value.pId_{0_nId}                 ▷ record pId_{0_nId}
6:         else if (key,value) is from E_t then
7:             pairset ← pairset ∪ {(eLabel, pId_{old_tId})}
8:     sort elements in pairset lexicographically, first on eLabel then on pId_{old_tId}
9:     signature ← (pId_{0_nId}, pairset)
10:    emit (key, (pId_{0_nId}, signature))
```

3.2 Task `Identifier`

On a single machine, in order to assign distinct values for signatures, we only need a dictionary-like data structure. In a distributed environment, on the other hand, some extra work has to be done. The `Identifier` task (Algorithm 2) is designed for this purpose. The map function distributes nodes of the same signature to the same reducer, so that in the reduce function, each reducer only needs to check locally whether the given signature is assigned an identifier or not; if not, then a new identifier is generated and assigned to the signature. To assign identifiers without collisions across reducers, we specify a non-overlapping identifier range each reducer can use beforehand. For instance, reducer i can generate identifiers in the range of $[i \times |N|, (i+1) \times |N|)$.

Algorithm 2 task `Identifier`

```
1: procedure IdentifierMapper(nId, (pId_{0_nId}, signature))
2:     emit (signature, (nId,pId_{0_nId}))
```

```
1: procedure IdentifierReducer(signature, [(nId, pId_{0_nId})])
2:     pId_{new_nId} ← get unique identifier for signature
3:     for (nId, pId_{0_nId}) in [(nId, pId_{0_nId})] do
4:         emit (nId, (pId_{0_nId}, pId_{new_nId}))
```

3.3 Task `RePartition`

The output of task `Identifier` is N_t filled with partition identifiers from the current iteration, but consists of file fragments partitioned by signature. In order to perform a map-side join with E_t in task `Signature` in the next iteration,

N_t needs to be sorted and partitioned on *nId*, which is the main job of task `RePartition` (Algorithm 3). This task makes use of the MapReduce framework to do the sorting and grouping.

Algorithm 3 task `RePartition`

```
1: procedure RePartitionMapper(nId, (pId_0_nId, pId_new_nId))
2:     emit (nId, (pId_0_nId, pId_new_nId))                    ▷ do nothing
1: procedure RePartitionReducer(nId, [(pId_0_nId, pId_new_nId)])
2:     for (pId_0_nId, pId_new_nId) in [(pId_0_nId, pId_new_nId)] do
3:         emit (nId, (pId_0_nId, pId_new_nId))
```

3.4 Running example

We illustrate our algorithm on the example graph of Figure 1. In Figures 2(a), 2(b) and 2(c), we show the input and output of the map and reduce phases of tasks `Signature`, `Identifier` and `RePartition`, respectively, for the first iteration ($i = 0$), with two reducers (bounded with gray boxes) in use.

4 Strategies for dealing with skew in graphs

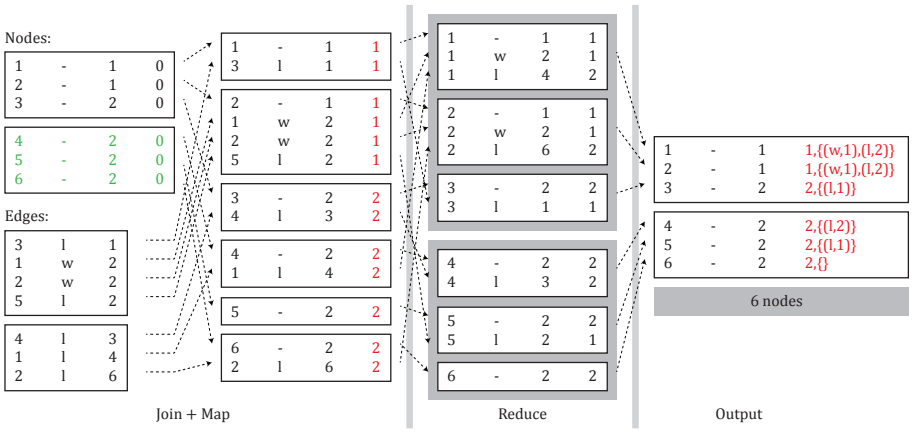
4.1 Data and skew

In our investigations, we used a variety of graph datasets to validate the results of our algorithm, namely: `Twitter` (41.65M, 1468.37M), `Jamendo` (0.48M, 1.05M), `LinkedMDB` (2.33M, 6.15M), `DBLP` (23M, 50.2M), `WikiLinks` (5.71M, 130.16M), `DBPedia` (38.62M, 115.3M), `Power` (8.39M, 200M), and `Random` (10M, 200M); the numbers in the parenthesis indicates the node count and edge count of the graph, resp.. Among these, `Jamendo`, `LinkedMDB`, `DBLP`, `WikiLinks`, `DBPedia`, and `Twitter` are real-world graphs described further in Luo et al. [20]. `Random` and `Power` are synthetic graphs generated using GTgraph [2], where `Random` is generated by adding edges between nodes randomly, and `Power` is generated following the power-law degree distribution and small-world property.

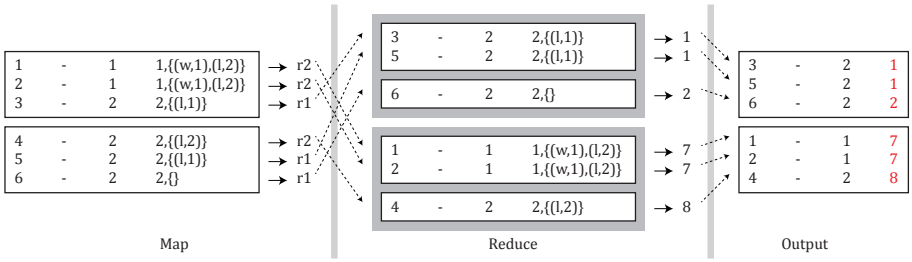
During investigation of our base algorithm (further details below in Section 5), we witnessed a highly skewed workload among mappers and reducers. Figure 4(a) illustrates this on the various datasets, showing the running time for different reducers for the three tasks in the algorithm. Each spike is a time measurement for one reducer. The running time in each task is sorted in a descending order. We see that indeed some reducers carry a significantly disproportionate workload. This skew slows down the whole process since the task is only complete when the slowest worker finishes.

From this empirical observation, we trace back the behavior to the data. Indeed, the partition result is skewed in many ways. For example, in Figure 3, we show the cumulative distribution of partition block size, i.e., number of nodes assigned to the block, to the number of partition blocks having the given size, for the real-world graphs described above. We see that for all of the datasets, block size shows a power-law distribution property.

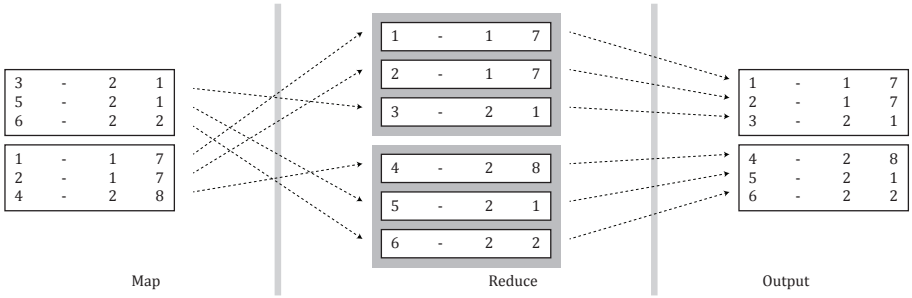
This indicates the need to rethink the design of our base algorithm. Recall that at the end of the map function of Algorithm 2, nodes are distributed to



(a) Task `Signature`



(b) Task `Identifier`



(c) Task `RePartition`

Fig. 2. Input and output of the three tasks of our algorithm for the example graph of Figure 1 (iteration $i = 0$).

reducers based on their signatures. Since some of the signatures are associated with many nodes (as we see in Figure 3), the workload is inevitably unbalanced. This explains the reducers' behavior in Figure 4(a) as well. In the following, we propose several strategies to handle such circumstances.

4.2 Strategy 1: Introduce another task `Merge`

Recall from Section 3.2 that nodes with the same signature must be distributed to the same reducer, for otherwise the assignment of partition block identifiers

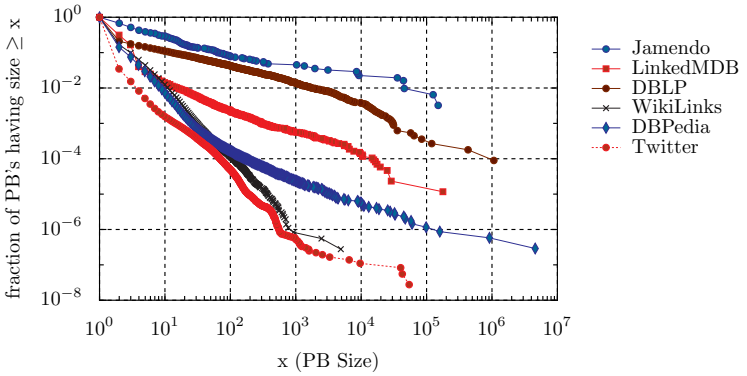


Fig. 3. Cumulative distribution of partition block size (PB Size) to partition blocks having the given size for real-world graphs

cannot be guaranteed to be correct. This could be relaxed, however, if we introduce another task, which we call `Merge`. Suppose that when we output the partition result (without the guarantee that nodes with the same signature go to the same reducer), for each reducer, we also output a mapping of signatures to partition identifiers. Then in task `Merge`, we could merge the partition IDs based on the signature value, and output a local_pid to global_pid mapping. Then in the task `RePartition`, another map-side join is introduced to replace the local_pid with the global_pid. Because the signature itself is not correlated with the partition block size, the skew on partition block size should be eliminated. We discuss the implementation details in the following.

In N_t assume we have an additional field holding the partition size of the node from the previous iteration, named $pSize_{old}$, and let $MR_{Load} = \frac{|N|}{number\ of\ reducers}$. We define rand(x) to return a random integer from 0 to x, and % as the modulo operator.

Algorithm 4 modified partition function for task `Identifier`

```
1: procedure IDENTIFIER_GETPARTITION                    ▷ for each key/value pair
2:     if pSize_old > threshold then
3:         n = pSize_old / MR_Load                      ▷ numbers of reducers we need
4:         return (signature.hashcode()+rand(n)) % number of reducers
5:     else
6:         return signature.hashcode() % number of reducers
```

We first change the partition function for `Identifier` (Algorithm 4). In this case, for nodes whose associated $pSize_{old}$ are above the threshold, we do not guarantee that they end up in the same reducer, but make sure that they are distributed to at most n reducers. Then we come to the reduce phase for `Identifier` (Algorithm 5). Here we also output the local partition size (named $pSize_{new}$) for each node.

Then the task `Merge` (Algorithm 6) will create the mapping between the locally assigned pId_{new_nId} and *global_pid*.

Algorithm 5 modified reduce function for task `Identifier`

```
1: procedure IdentifierReducer(signature, [(nId,pId_0_nId,pSize_old)])
2:     pId_new_nId ← get unique identifier for signature
3:     pSize_new ← size of [(nId,pId_0_nId, pSize_old)]
4:     for (nId, pId_0_nId, pSize_old) in [(nId, pId_0_nId, pSize_old)] do
5:         emit (nId, (pId_0_nId, pId_new_nId))
6:         emit (signature, (pId_new_nId, pSize_new))
```

Algorithm 6 task `Merge`

```
1: procedure MergeMapper(signature, (pId_new_nId, pSize_new))
2:     emit (signature, (pId_new_nId, pSize_new))                ▷ do nothing
1: procedure MergeReducer(signature, [(pId_new_nId, pSize_new)])
2:     global_pid_count ← 0
3:     global_pid ← get unique identifier for signature
4:     for (pId_new_nId, pSize_new) in [(pId_new_nId, pSize_new)] do
5:         global_pid_count ← global_pid_count + pSize_new
6:         emit (global_pid, (pId_new_nId))          ▷ the local - global mapping
7:     emit (global_pid, global_pid_count)
```

Finally at the beginning of task `RePartition`, the partition identifiers are unified by a merge join of the local_pid - global_pid mapping table and N_t. We achieve this by distributing the local_pid - global_pid table to all mappers before the task begins, with the help of Hadoop's distributed cache. Also, global_pid_count is updated in N_t.

While this is a general solution for dealing with skew, the extra `Merge` task introduces additional overhead. In the case of heavy skew, some signatures will produce large map files and performing merging might become a bottleneck. This indicates the need for a more specialized solution to deal with heavy skew.

4.3 Strategy 2: Top-K signature-partition identifier mapping

One observation of Figure 3 is that only a few partition blocks are heavily skewed. To handle these outliers, at the end of task `Signature`, besides emitting N_t, we can also emit, for each reducer, an aggregation count of signature appearances. Then a merge is performed among all the counts, to identify the most frequent K signatures and fix signature-partition identifier mappings for these popular partition blocks. This mapping is small enough to be distributed to all cluster nodes as a global variable by Hadoop, so that when dealing with these signatures, processing time becomes constant. As a result, in task `Identifier`, nodes with such signatures can be distributed randomly across reducers.

There are certain drawbacks to this method. First, the output top-K frequent signatures are aggregated from local top-K frequent values (with respect to each reducer), but globally we only use these values as an estimation of the real counts. Second, the step where signature counts have to be output and merged becomes a bottleneck of the whole workflow. Last but not least, users have to specify K before processing, which may be either too high or too low.

However, in the case of extreme skew on the partition block sizes, for most of the real world datasets, there are only a few partition blocks which delay the whole process, even for very large datasets. So when we adopt this strategy, we can choose a very small K value and still achieve good results, without introducing another MapReduce task. This is validated in Section 5.1.

5 Empirical analysis

Our experiments are executed on the Hadoop cluster at SURFsara in Amsterdam, The Netherlands.[1] This cluster consists of 72 Nodes (66 DataNodes & TaskTrackers and 6 HeadNodes), with each node equipped with 4 x 2TB HDD, 8 core CPU 2.0 GHz and 64GB RAM. In total, there are 528 map and 528 reduce processes, and 460 TB HDFS space. The cluster is running Cloudera CDH3 distribution with Apache Hadoop 0.20.205. All algorithms are written in Java 1.6. The datasets we use are described in Section 4.1. A more detailed description of the empirical study can be found in de Lange [7].

5.1 Impact on workload skew of the Top-K strategy

Figure 4(b) shows the cluster workload after we create a identifier mapping for the top-2 signature-partitions from Section 4.3. We see that, when compared with Figure 4(a), the skew in running time per reducer is eliminated by the strategy. This means that workload is better distributed, thus lowering the running time per iteration and, in turn, the whole algorithm.

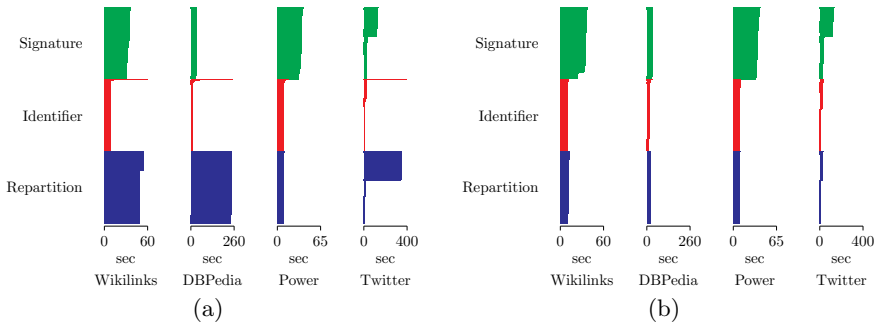


Fig. 4. Workload skewness for base algorithm (a) and for top-2 signature strategy (b)

5.2 Overall performance comparison

In Figure 5, we present an overall performance comparison for computing 10-bisimulation on each graph with: our base algorithm (Base), the merge (Merge) and top-K (Top-K Signature) skew strategies, and the state of the art single-machine external memory algorithm (I/O Efficient) [20]. For the Top-K Signature strategy, we set $K = 2$ which, from our observation in Section 5.1, gives excellent skew reduction with low additional cost. For the Merge optimization we used a threshold of $1 \times MR_{load}$ such that each partition block larger than

[1] `https://www.surfsara.nl`

the optimal reducer block size is distributed among additional reducers. Furthermore, for each dataset, we choose $3 \times \lceil \frac{edge\ table\ size}{64\ MB} \rceil$ number of reducers, which has been tested to lead to the minimum elapsed time. We empirically observed that increasing the number of reducers beyond this does not improve performance. Indeed, adding further maps and reducers at this point negatively impacts the running time due to framework overhead. Each experiment was repeated five times, and the data point reported is the average of the middle three measurements.

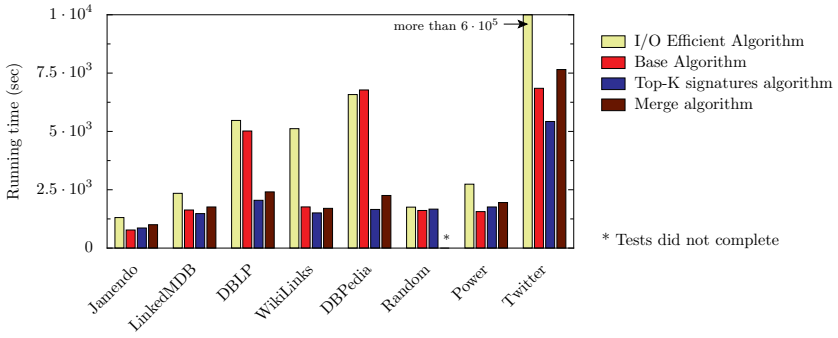


Fig. 5. Overall performance comparison

We immediately observe from Figure 5 that for all datasets, among the Base algorithm and its two optimizations, there are always at least two solutions which perform better than the I/O efficient solution. For small datasets such as Jamendo and LinkedMDB, this is obvious, since in these cases only 1 or 2 reducers are used, so that the algorithm turns into a single-machine in-memory algorithm. When the size of the datasets increases, the value of MapReduce becomes more visible, with up to an order of magnitude improvement in the running time for the MapReduce-based solutions. We also observe that the skew amelioration strategies give excellent overall performance on these big graphs, with 2 or 3 times of improvement over our Base algorithm in the case of the highly skewed graphs such as DBLP and DBPedia. Finally, we observe that, relative to the top-K strategy, the merge skew-strategy is mostly placed at a disadvantage due to its inherent overhead costs.

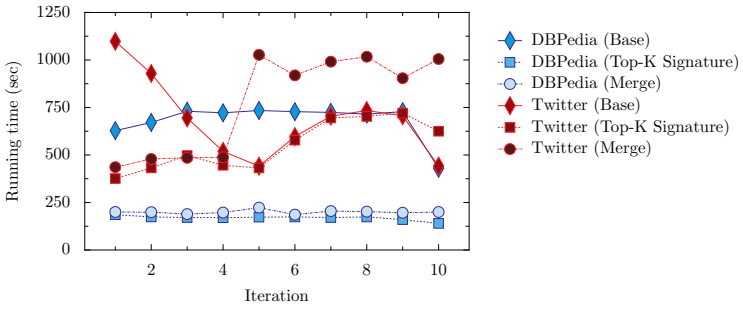


Fig. 6. Performance comparison per iteration for Twitter and DBPedia

To further study the different behaviors among the MapReduce-based solutions, we plot the running time per iteration of the three solutions for DBPedia and Twitter in Figure 6. We see that for the Twitter dataset, in the first four iterations the skew is severe for the Base algorithm, while the two optimization strategies handle it well. After iteration 5, the overhead of the Merge strategy becomes non-negligible, which is due to the bigger mapping files the `Identifier` produces. For the DBPedia dataset, on the other hand, the two strategies perform consistently better than the Base algorithm.

Over all, based on our experiments, we note that our algorithm's performance is stable, i.e., essentially constant in running time as the number of maps and reducers is scaled with the input graph size.

6 Concluding remarks

In this paper, we have presented, to our knowledge, the first general-purpose algorithm for effectively computing localized bisimulation partitions of big graphs using the MapReduce programming model. We witnessed a skewed workload during algorithm execution, and proposed two strategies to eliminate such skew from an algorithm design perspective. An extensive empirical study confirmed that our algorithm not only efficiently produces the bisimulation partition result, but also scales well with the MapReduce infrastructure, with an order of magnitude performance improvement over the state of the art on real-world graphs.

We close by indicating several interesting avenues for further investigation. First, there are additional basic sources of skew which may impact performance of our algorithm, such as skew on signature sizes and skew on the structure of the bisimulation partition itself. Therefore, further optimizations should be investigated to handle these additional forms of skew. Second, in Section 5.2 we see that all three proposed solutions have their best performance for some dataset, therefore it would be interesting to study the cost model of the MapReduce framework and combine the information (e.g., statistics for data, cluster status) within our algorithm to facilitate more intelligent selection of strategies to use at runtime [1]. Last but not least, our algorithmic solutions for ameliorating skew effects may find successful applications in related research problems (e.g., distributed graph query processing, graph generation, and graph indexing).

Acknowledgments. The research of YW is supported by Research Foundation Flanders (FWO) during her sabbatical visit to Hasselt University, Belgium. The research of YL, GF, JH and PD is supported by the Netherlands Organisation for Scientific Research (NWO). We thank SURFsara for their support of this investigation and Boudewijn van Dongen for his insightful comments. We thank the reviewers for their helpful suggestions.

References

1. F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a Map-Reduce computation. *CoRR*, abs/1206.4377, 2012.

2. D. A. Bader and K. Madduri. GTgraph: A suite of synthetic graph generators. `http://www.cse.psu.edu/~madduri/software/GTgraph/index.html`.
3. S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *Int J Softw Tools Technol Transfer*, 7:74–86, 2005.
4. P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *Proc. VLDB*, pages 141–152, Berlin, Germany, 2003.
5. A. Clauset, C. Shalizi, and M. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.
6. J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science Engineering*, 11(4):29–41, 2009.
7. Y. de Lange. MapReduce based algorithms for localized bisimulation. Master's thesis, Eindhoven University of Technology, 2013.
8. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
9. A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comp. Sci.*, 311(1-3):221–256, 2004.
10. W. Fan. Graph pattern matching revised for social network analysis. In *Proc. ICDT*, pages 8–21, Berlin, Germany, 2012.
11. W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *Proc. SIGMOD*, pages 157–168, Scottsdale, AZ, USA, 2012.
12. B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Handling data skew in MapReduce. In *Proc. CLOSER*, pages 574–583, 2011.
13. B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in MapReduce based on scalable cardinality estimates. In *Proc. ICDE*, pages 522–533, 2012.
14. Hadoop. `http://hadoop.apache.org/`, 2012.
15. S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. LEEN: Locality/fairness-aware key partitioning for MapReduce in the cloud. In *CloudCom*, pages 17–24, 2010.
16. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, San Jose, 2002.
17. Y. Kwon, K. Ren, M. Balazinska, and B. Howe. Managing Skew in Hadoop. *IEEE Data Eng. Bull.*, 36(1):24–33, 2013.
18. J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.
19. J. Lin and M. Schatz. Design patterns for efficient graph algorithms in MapReduce. In *Proc. MLG*, pages 78–85, Washington, D.C., 2010.
20. Y. Luo, G. H. L. Fletcher, J. Hidders, Y. Wu, and P. De Bra. I/O-efficient algorithms for localized bisimulation partition construction and maintenance on massive graphs. *CoRR*, abs/1210.0748, 2012.
21. T. Milo and D. Suciu. Index structures for path expressions. In *Proc. ICDT*, pages 277–295, Jerusalem, Israel, 1999.
22. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16:973, 1987.
23. F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *ESWC*, pages 406–421, Heraklion, Greece, 2012.
24. D. Sangiorgi and J. Rutten. *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.
25. R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce using situation-aware mappers. In *Proc. EDBT*, pages 420–431, 2012.