Future Generation Computer Systems 32 (2014) 324–337

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Parallel processing of large graphs

Tomasz Kajdanowicz *, Przemyslaw Kazienko, Wojciech Indyk

Wroclaw University of Technology, Poland

HIGHLIGHTS

- We compared three parallel computing techniques in terms of large graph processing.
- MapReduce, map-side join and Bulk Synchronous Parallel tested for two distinct problems.
- Iterative graph processing with the BSP implementation significantly outperforms MapReduce.
- Map-side join design pattern may improve the original MapReduce performance.

ARTICLE INFO

Article history:
Received 9 March 2013
Received in revised form
30 July 2013
Accepted 2 August 2013
Available online 28 August 2013

Keywords:
Large graph processing
Parallel processing
Big data
Cloud computing
Collective classification
Shortest path
Networked data
Bulk Synchronous Parallel
MapReduce

ABSTRACT

More and more large data collections are gathered worldwide in various IT systems. Many of them possess a networked nature and need to be processed and analysed as graph structures. Due to their size they very often require the usage of a parallel paradigm for efficient computation. Three parallel techniques have been compared in the paper: MapReduce, its map-side join extension and Bulk Synchronous Parallel (BSP). They are implemented for two different graph problems: calculation of single source shortest paths (SSSP) and collective classification of graph nodes by means of relational influence propagation (RIP). The methods and algorithms are applied to several network datasets differing in size and structural profile, originating from three domains: telecommunication, multimedia and microblog. The results revealed that iterative graph processing with the BSP implementation always and significantly, even up to 10 times outperforms MapReduce, especially for algorithms with many iterations and sparse communication. The extension of MapReduce based on map-side join is usually characterized by better efficiency compared to its origin, although not as much as BSP. Nevertheless, MapReduce still remains a good alternative for enormous networks, whose data structures do not fit in local memories.

1. Introduction

Many technical and scientific problems are related to data with the networked nature, which can be relatively simply represented by means of graph structures. Graphs provide a very flexible abstraction for describing relationships between discrete objects. Many practical problems in scientific computing, data analysis and other areas can be modelled in their essential form by graphs and solved with the appropriate graph algorithms.

In many environments graph structures are so big that they require specialized processing methods, especially parallel ones. This becomes particularly vital for data collections provided by users leaving their traces in various online or communication services, such as multimedia publishing portals or social networking sites, e.g. YouTube or Facebook. Additionally, these datasets reflect various user behaviour, so their graph representation may be complex with multiple relationships linking network nodes. This requires analytical methods dealing not only with simple graphs but also hypergraphs or multigraphs.

As graph problems grow larger in scale and more ambitious in their complexity, they easily outgrow the computation and memory capacities of single processors. Given the success of parallel computing in many areas of scientific computing, parallel processing appears to be necessary to overcome the resource limitations of single processors in graph computations. Parallel graph computation is, however, challenging [1] and before the advent of cloud computing and Hadoop, programmers had to use ill-suited distributed systems or design their own systems, which required additional effort to provide fault-tolerance and to address other problems related to parallel processing [2]. The rise of the MapReduce concept and Hadoop—its open source implementation—provided researchers with a powerful tool to process large data collections. Recently, Hadoop has become a de facto standard in

* Corresponding author. Tel.: +48 713203609.
E-mail address: tomasz.kajdanowicz@pwr.wroc.pl (T. Kajdanowicz).

0167-739X
http://dx.doi.org/10.1016/j.future.2013.08.007

academia and a significant solution for parallel processing in industry. It has been used in various areas, including some graph processing problems [3].

The MapReduce model is, however, badly suited for iterative and graph algorithms. There has been a lot of research in creating design patterns improving MapReduce performance for graphs like [4,5], or building systems that would aid iterative processing on MapReduce [6–10]. Google reacted to that with the development of Pregel [2]—an alternative system that implements the Bulk Synchronous Parallel (BSP) programming model [11] for graph processing.

The main difference in the processing of regular data structures (tables) and relational models (graphs) relies on different problem decomposition. Processing table structures is composed of handling of individual records (rows in the table). For the networked data, single processing of a graph vertex usually requires access to the neighbourhood of this vertex, which for most algorithms remains fixed for the whole processing time. This data may be either accessed at every algorithm iteration via a distributed file system (e.g. HDFS), as in the case of MapReduce, or preserved locally for the entire processing, the case for BSP.

Both different parallel processing methods, i.e. MapReduce and BSP, along with the map-side join MapReduce modification, have been implemented in the Hadoop environment—all three were used in the experiments presented in this paper. Each approach was independently applied to solve two distinct graph analytical problems: single source shortest path (SSSP) calculation and collective classification of network vertices with Relational Influence Propagation (RIP). The graph algorithms had an iterative nature, which enabled testing their various parallel implementations in the following steps. The iterative computation was carried out in cloud environments containing various numbers of machines to compare scalability of Bulk Synchronous Parallel and MapReduce. Additionally, all approaches were tested on several large graph data sets coming from various domains.

The initial version of the paper was presented at the ICDM 2012 conference [12].

The following Section 2 provides a short state-of-the art study on graph problem solutions by means of cloud computing. The main architectures for graph processing, including distributed memory and shared memory, are presented in Section 3. Two parallel processing models MapReduce and Bulk Synchronous Parallel (BSP) are sketched in Section 4. Some discussion on their similarities as well as potential improvements is provided in Section 5. Also in this section, an important and experimentally verified MapReduce modification based on map-side join design patterns is proposed for graph processing. Two iterative graph algorithms: single source shortest path computation and collective classification are described in more depth in Section 6. Experimental setup and cloud environment, including data set profiles can be found in Section 7. The results of experiments are presented in Section 8. Discussions on the results and solutions of some problems, which arose during research, are depicted in Section 9. The last section, Section 10 contains general conclusions and further directions for work.

2. Related work

The dynamic development of distributed and cloud computing has led to stable solutions for massive data processing. Nowadays, there is an intensified focus on new models useful for specific kinds of processing. On top of distributed storage systems many solutions dedicated for particular tasks are located, for example fast random access data, pipeline processing, graph computations, etc. [13].

There are several concepts for parallel processing in clusters. Two of them are widely used in offline batch analysis systems and merit special attention: MapReduce and the less popular Bulk Synchronous Parallel (BSP). The former is especially very popular and applied to many real solutions [13].

The general idea behind the Bulk Synchronous Parallel (BSP) method was first coined and studied in early 90s [11,14]. Recently, it was adapted by Google to graph processing in clouds in the Pregel system [2]. Pregel's idea of using BSP for graph processing in clouds inspired others to create similar systems, some of which are open source e.g. [15,16].

The overview of large-scale graph engines is presented in [17], which contains graph systems designed to achieve different goals—from offline analytics system to online low-latency systems.

An empirical comparison of different paradigms for large-scale graph processing is presented in [18]. However, the presented paradigms require a proprietary and/or prototypical platforms, while, in this paper, we focus on approaches which are available on Hadoop, a highly popular, open source platform, which can be run on a set of commodity hardware.

Pace et al. [19] provided a theoretical comparison of BSP and MapReduce models. In terms of graph processing, they noticed that the Breadth First Search algorithm (for the shortest path computation) cannot be efficiently implemented by means of the MapReduce model. In this paper, we go forward and focus more on an empirical comparison for the real world data sets, using available implementations as well as evaluation for an additional graph problem—collective classification. The general conclusions remain the same: BSP usually appears to be a better model for solving graph problems than MapReduce. The results included in this paper provide quantitative analyses supporting that statement.

3. Parallel architectures for graph processing

Regardless of the nature of a particular computational problem it can be parallelled and scaled well when the overall solution is balanced in terms of the problem solution, the algorithm expressing the solution, the software that implements the algorithm and hardware. The algorithms, software, and hardware that worked properly for standard parallel applications are not necessarily effective for large-scale graph problems. In general, graph problems have specific properties that make them difficult to fit in existing distributed computational solutions. Among others, the following characteristics of graph processing causes challenges in effective parallel processing [20]:

- *Computation driven by relational data*. The majority of graph algorithms are executed according to the structure of a graph, where computation for each next vertex is strictly dependent on the results calculated for all antecedents. It means that the algorithm relies on the graph structures rather than on explicitly stated sequential processing. This implies that the structure of the whole computation is not known at the beginning of execution and efficient partition is hardly possible.
- *Diverse and unbalanced data structures*. Usually graph data is highly unstructured or irregular, which do not give many options for parallel processing based on partitioning. Additionally, a skewed distribution of vertex degrees makes scalability difficult, limiting it to unbalanced computational loads.
- *High overload for data access in comparison to computation*. Algorithms often explore graphs rather than performing complex computations on their structure, e.g. the shortest path problem requires only single arithmetic operations in path cost calculation but requires the performance of many data queries. Runtime can be easy dominated by the wait for memory access, not by computational activities.

Due to the fact that commercially available computer appliances have varying capabilities there can be distinguished several processing architectures suitable for distinct hardware. Depending on the amount of available storage and memory for computation the data might be processed in a different manner, reducing or increasing the latency. There can be distinguished distributed memory architectures and shared-memory architectures.

3.1. Distributed memory architecture

The most widespread class of parallel computing is distributed memory architecture. This architecture comprises a set of machines (a set of processors and storage/memory) connected by a high speed network. It is possible that the environment can be composed of quite common machines and this makes the architecture inexpensive. According to reported results the architecture is effective on many scientific problems but is able to handle only trivially parallelled algorithms.

The general idea behind the distributed memory architecture is to implement the message passing technique. In this method the data is divided into the memories of different machines. The distribution of the data is controlled centrally and this means it has to be additionally decided which processor performs which tasks. Usually data from memory is processed by a local processor. The distributed memory architecture has a big disadvantage: all tasks to be performed have to be explicitly formulated at the beginning of computation. However it means that the user can almost completely control the computation. Due to the fact that the data is exchanged between processors by a specially designed message passing protocol, the user has to specify it. This makes the architecture very flexible but full control of communication and data partitioning can influence errors. Such problems can be overwhelmed by the usage of standards such as the MPI protocol [21]. As long as the architecture enables full customization of implementation, smart users can plan such a system realization that achieves high performance.

One of the best known message passing styles in the distributed computing is Bulk Synchronous style [11]. Even though it is quite mature it has been re-discovered again as a powerful implementation [2]. In general the processors in this technique switch from working independently on local data to performing collective communication operations. This implies the collective exchange of data unable to be accomplished on demand, but only at the pauses between computational steps (synchronization). It can cause problems with load balancing and actually makes it difficult to exploit parallelism in an application. The Bulk Synchronous technique is described in more detail in Section 4.2, where the programming models are considered.

There exists an improvement for the message passing technique which is still able to utilize distributed memory—partitioning of global address space. Introducing an additional layer of abstraction in the implementation that provides operations on remote memory locations with simple manipulation mechanisms facilitates writing programs with complex data access patterns and, therefore, asynchronous communication. An example of partitioned global address space implementation is UPC [22].

3.2. Shared-memory architecture

There are two possible realizations of shared-memory architecture in computing: hardware based or software based. Both of these approaches are required to provide support for global memory accessible for all processors. As mentioned in the previous paragraph, the UPC implementation is actually an example of software providing the illusion of globally addressable memory but still working on distinct machines. But the support for a global address space can also be provided in hardware.

One of the common realizations of shared-memory architecture are symmetric processors which can utilize global memory universally. The architecture assumes that, thanks to proper hardware support, any processor can access addresses in global memory. This feature allows processors to retrieve the data directly and relatively in a very quick manner. Additionally, solutions of highly unstructured problems, such as graph processing, may benefit from that characteristic and achieve a higher performance than environments based on a distributed memory architecture.

The shared-memory approach is not ideal and thread synchronization and scheduling reveal another performance challenge. For instance, if several threads are trying to access the same region of memory, the system must apply some protocol to ensure correct program execution. Obviously, some threads may be blocked for a period of time.

Another noticeable problem, that has to be considered while implementing the architecture is the fact that the best efficiency is obtained when processors are kept occupied with a large number of threads. Fortunately, many graph algorithms can be written in a multi-thread fashion. However this may imply increases of memory access.

Finally, the architecture requires processors that are not custom and more expensive than the ones used in distributed memory architecture. Moreover, the preprocessors have a much slower clock than mainstream ones. Even though the architecture is quite interesting, flexible and more effective, it might not be the most attractive for graph processing.

4. Open source parallel programming models

The main purpose of this paper are comparative studies of different practical approaches to parallel graph processing using open source platforms. In particular, the most popular MapReduce (MR), see Section 4.1, and less common Bulk Synchronous Parallel (BSP), see Section 4.2, are considered. Additionally, an extended version of MapReduce, namely, map-side join modification of MapReduce (MR2), see Section 5.3, together with MR and BSP have been experimentally examined, see Sections 7 and 8. The overall comparison of these three approaches is described in Table 1. In general, MapReduce-based solutions (MR and MR2) require graph data to be re-allocated at each algorithm iteration, whereas BSP enables them to be assigned only once, at the beginning. Additionally, MR and MR2 utilize the Hadoop Distributed File System (HDFS), which facilitates processing of huge graphs. BSP, in turn, stores its data in the local memory.

4.1. MapReduce

MapReduce is a parallel programming model especially dedicated for complex and distributed computations, which has been derived from the functional paradigm [13]. In general, MapReduce processing is composed of two consecutive stages, which for most problems are repeated iteratively: the map and the reduce phase. The former processes the data on hosts in parallel, whereas the latter aggregates the results. At each iteration independently, the whole data is split into chunks, which, in turn, are used as the input for mappers. Each chunk may be processed by only one mapper. Once the data is processed by mappers, they can emit $\langle key, value \rangle$ pairs to the reduce phase. Before the reduce phase the pairs are sorted and collected according to the *key* values, therefore each reducer gets the list of values related to a given *key*. The consolidated output of the reduce phase is saved into the distributed file system, see Fig. 1.

The MapReduce model is already an mature concept and although it has not been originally designed to process graphs, a set of design patterns for graph processing have been developed [4,5,23]. These good practices show how to express iterative algorithms in MapReduce, but they do not overcome the widely recognizable inefficiencies of MapReduce model in networked data processing.

In the case of iterative graph processing algorithms, the graph structure and other static data, which do not change through computation, must be transferred over the network of computational nodes from mappers to reducers at each single iteration. It causes a great network overhead and appears to be the greatest pitfall of graphs processing by means of MapReduce. The stateless, two-phased computational nature of MapReduce does not allow vertices to reside on the same host for all iterations. It means that after every map-reduce iteration, the entire data must be written to the

Table 1
Profile of various parallel graph processing approaches: MapReduce (MR), its extension based on map-side join design patterns (MR2) and Bulk Synchronous Parallel (BSP).

Feature	MapReduce (MR)	Enchanced MapReduce (MR2)	BSP
General unit of processing	An iteration composed of two Map-Reduce jobs, Fig. 3	An iteration composed of one join and one Map-Reduce phases, Fig. 4	Superstep, Fig. 5
Data processed	Graph structure (neighbourhoods) & vertex labels	Graph structure (neighbourhoods) & vertex labels	Graph structure (neighbourhoods) & vertex labels
Graph vertex allocation	At every iteration, before map	At every iteration, before join	Once, at the beginning
Work allocation among machines	Flexible, repeated at every iteration	Flexible, repeated at every iteration	Fixed at the beginning
Set of vertices processed on single machine	Change before each map	Change before each map	Fixed at the beginning
Data about vertex and its neighbours	Transferred before each map	Transferred before each join	Transferred once before entire processing
Location of graph data (neighbourhoods & labels)	HDFS	HDFS	Local memory
Messages with updated data related to a given vertex (self-update)	Transferred before each map	Transferred before each join	Data stored locally; no message

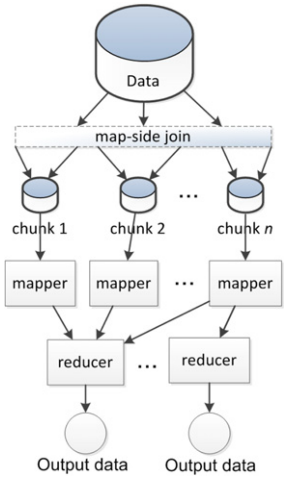


Fig. 1. Data flow in the MapReduce programming model.

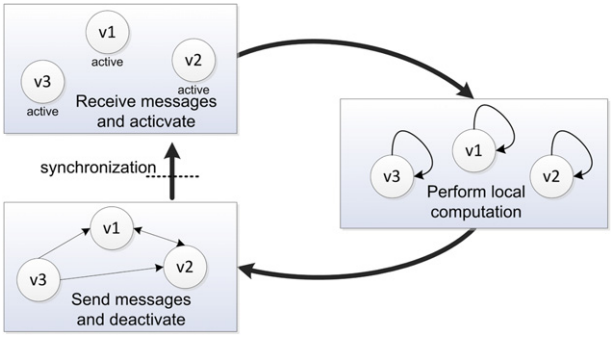


Fig. 2. Data flow in the Bulk Synchronous Parallel programming model.

global memory in order to be consumed in the next iteration. Since the distributed file system serves as the global memory in the cloud environment, the whole computation is very disk-intensive. Additionally, the map and reduce workers reside on different physical machines and for that reason, the data is constantly transferred over the network, which is the scarce resource in the cloud environment.

4.2. BSP—Bulk Synchronous Parallel

To address the problem of MapReduce inefficiency for iterative graph processing, Google has created another a distributed, fault-tolerant system called Pregel [2], which is based on the Bulk Synchronous Parallel (BSP) processing model [11]. Although Pregel is a proprietary system, it has inspired the creation of several open-source systems which implement the BSP model, such as Apache Hama [16] or Apache Giraph [15].

The computation process in BSP comprises of a series of supersteps (equivalent to MapReduce iterations), see Figs. 2 and 5. In every superstep, a user defined function is executed in parallel on every item from the dataset acting as an agent. Pregel and Pregel-inspired graph analytics systems are vertex-centric: a single agent computation has a graph representation in BSP. It consists of graph vertex identifiers, their current values or states, as well as lists of vertices' outgoing edges. Before any computation, all graph vertices are partitioned and loaded into the local memories of machines (hosts). They stay there throughout all computations, so that the whole processing is carried out using the local hosts' memories. Graph processing in BSP is organized by means of messages sent between machines hosting individual graph vertices. At every superstep, each host receives from other hosts the messages related to vertices preserved by this host and executes a user defined computation function. This function performs local processing on local vertices and sends messages to some or all vertices' neighbours in the graph. Once the local computation is finished for a given vertex, processing deactivates itself and the host waits for all other vertices to finish. The barrier of the synchronization mechanism allows the next superstep to begin when processing for all vertices is completed in the current superstep. Afterwards, only the vertices that have received a message are activated.

Since the graph's structure (especially edges and values assigned) does not need to be sent over the network at every superstep (iteration), BSP may be a very efficient graph computation model. Only specific messages necessary for the algorithm execution are exchanged between hosts. There is no network overhead related to graph structure passing as in the MapReduce model. Moreover, storing the whole graph structure in the local memories of workers allows in-memory computation. The need for disk write-reads as well as object serialization between iterations is eliminated. However, it is possible only if the graph structure fits in the memories of all workers. Otherwise, spilling-to-disk techniques must be implemented. Although such techniques are claimed to be implemented in Pregel [2], to our best knowledge, they are not yet supported in its open source followers. This might be a drawback of choosing the BSP model for graph processing, as will be presented later on.

5. Similarities between MapReduce and BSP

Both BSP and MapReduce models differ in many ways but simultaneously they have many common features, which expose them to bottlenecks in graph processing as well as raise the possibilities for similar improvements. Some practical enhancements that can be applied for both models are presented in Sections 5.1

and 5.2. Finally, the new *map-side join* design pattern for MapReduce graph processing is proposed in Section 5.3. It eliminates passing the graph structure between map and reduce phases, bringing the two models much closer.

5.1. Partitioning of the graph vertex set

By default, in both MapReduce and BSP models, graph vertices are partitioned for the purpose of computation and they are assigned to hosts based on a given, fixed hash function. This enables the workers-hosts to serve a similar number of vertices. However, it would be beneficial for both models, if graph topologically-close vertices would be processed on the same physical machine. It would increase local computing and decrease network transfer. For that reason, the effects of graph partitioning were studied for both the MapReduce model [24,4,9] and for the BSP model [25,26]. Commonly, two types of partitioning may be distinguished: (i) range and (ii) cluster partitioning. The former splits vertices using the a priori knowledge about them, e.g. users from the same geographical region, web sites in the same language, etc. On the other hand, the cluster partitioning strives to extract groups of vertices closely interlinked in the graph. Both of these approaches have two major drawbacks, for which the partitioning aspect was abandoned from the research presented in this paper. The range partitioning requires some a priori knowledge about nodes, which is unavailable, if the source data is anonymized—the case of the data sets used in experiments, see Section 7. The graph vertex clustering, in turn, is a complex and resource-consuming task itself, especially for large graph structures.

5.2. Aggregation of messages

Distribution of a workload related to processing of a single vertex is impossible to optimize in both models. Overall, the entire computation for a given vertex is always executed on a single machine. Since the real world complex network data sets satisfy the power-law distribution for in-degree values, a few vertices may require much more processing than most of the others. It makes practical load balancing difficult to achieve both for the MapReduce and SSP model. Additionally, the total computation duration depends on the processing time of the mostly loaded computational node and is valid for both models, even with perfect balancing and scalability. This problem can be partially addressed by the introduction of so called combiners, also to be used both in MapReduce and BSP. Combiners can put together messages destined to any vertex originating from the same machine. They are executed after the map phase in the MapReduce model and after the computation phase, before messages are sent, in BSP. Combiners can distribute the workload connected with high-degree vertices and limit the number of messages transferred over the computer network. Unfortunately, there is one significant limitation for this solution—operations on the data must be cumulative and associative, and this condition is not met for all of the graph algorithms. The effect of using combiners for MapReduce is extensively studied in [4], while no equivalent studies are known for graph processing in BSP.

5.3. Graph processing using map-side join design patterns in MapReduce

The need for reshuffling the graph structure between map and Reduce phases is the main disadvantage of graph processing by means of MapReduce. To solve this problem, the Schimmy design pattern was proposed by Lin et al. [4]. With Schimmy, mappers emit only messages that are being exchanged between vertices. The result of this message processing is merged with the graph structure in the reduce phase and written to the disk in order to be utilized in the following MapReduce iteration. To make the merge possible, the key-value pairs representing the graph structure must be partitioned into as many files as the number of reducers, and they remain split, similarly as in the MapReduce shuffle phase. Afterwards, every reducer locates and reads (possibly remotely) a file with the range of keys suitable for it.

Table 2
Profile of MapReduce-based solutions to data processing.

Feature	MapReduce with map-side join (MR2)	Schimmy	iMapReduce
Join	Map-side	Reduce-side	Map-side
Shuffling	Occurs	Occurs	Avoided for static data
Synchronization	Standard	Standard	Asynchronous

Instead of a reduce-side join (which actually happens in Schimmy), a better design pattern would be to perform a map-side join. In most graph algorithms, the only time in the MapReduce graph processing when vertices must be aware of the graph structure is in the map phase, when they need to know to which vertices send messages. The reduce phase usually computes new values for vertices and does not require knowledge about the vertex neighbourhoods.

In the map-side join design pattern, the input is partitioned into two groups of files containing either the graph structure or the current state/values of vertices. The input files must be split in the same manner as in the case of Schimmy into a number of files equal to the number of reducers and with the same partitioning functions that route messages from mappers to reducers. Two types of files are merged in the map phase, i.e. the mapper reads record by record from two files. However, unlike in Schimmy, there is no need to go beyond the standard API. Hadoop offers a special input format that allows the input to consist of more than one file. Mappers emit only the messages exchanged between vertices, likewise Schimmy. Reducers receive vertex messages, perform a computation and emit the result. Output created by reducers will be merged in the next iteration with the graph structure in the map phase. The drawback of this approach is that since mappers will have to read input files, one of them usually will require a remote read. However, the Schimmy also requires remote reads in the reduce phase, but with the map-side join approach, the graph structure is never written to a disk. It appears to be the greatest advantage over Schimmy. The proposed idea of MapReduce with map-side join, hereinafter referred to as MapReduce 2 (MR2), tends to be noticeably more efficient then the typical MapReduce concept.

Specificity of the tested algorithms (see Section 6) enabled the authors to use a more efficient approach than the Schimmy solution. It should be noticed that Schimmy allows one to efficiently use of information about the neighbourhood of the node in the reduce phase. However, in the tested algorithms, such information is needed in the map phase only and the reduce phase contains recalculation of nodes' labels, so that Schimmy appeared to be improper for the graph problems analysed in the paper.

The idea of joining both static and dynamic graph data in the map phase has already been proposed in the iMapReduce system [8]. However, it has not been described in detail and for that reason it can be hardly used directly for comparison. Additionally, the iMapReduce approach eliminates the shuffling of the static data and executes map tasks asynchronously. Since the main goal of this paper was to compare BSP to MapReduce, which are both synchronous, the authors decided not to include the iMapReduce implementation in the experiments.

The general profile of various MapReduce-based concepts to data processing is presented in Table 2.

6. Selected graph algorithms implementing parallel processing

To evaluate the efficiency and other features of different parallel graph processing models, two separate problems were selected.

Table 3
Profile of graph problems used in parallel computation.

Feature	Single Source Shortest Paths (SSSP)	Relational Influence Propagation (RIP)
Graph type (edges)	Unweighted	Weighted
Graph structure processed independently in parallel within a single iteration	Individual vertices with their closest neighbours	Individual vertices with their closest neighbours
Vertex activation (label update and further broadcasting)	Only if the message from the neighbour contains smaller path length	For all vertices

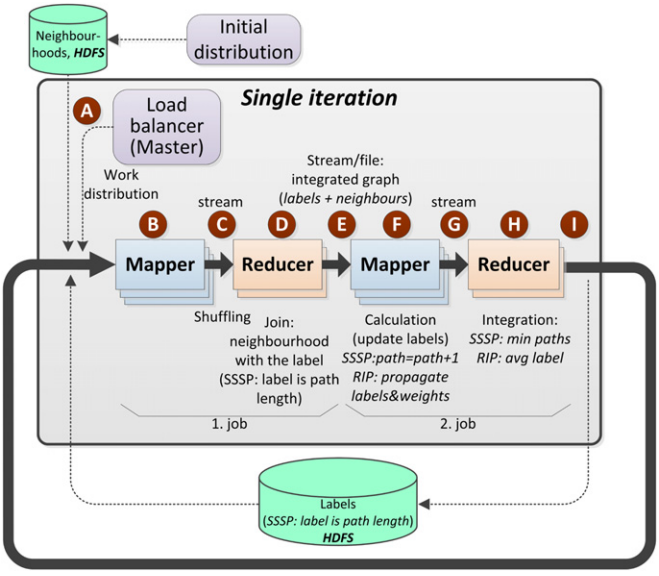


Fig. 3. MapReduce model for processing of graph problems.

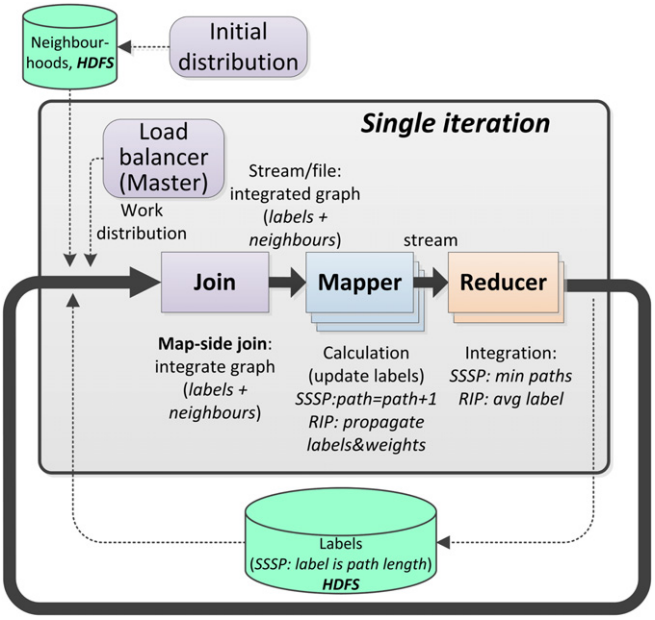


Fig. 4. MapReduce model with map-side join design patterns (MR2) in graph data processing.

The former is calculation of single source shortest paths (SSSP), see Section 6.1. The latter is a collective classification algorithm based on Relational Influence Propagation (RIP) used for collective classification of graph vertices, see Section 6.2. Both problems, i.e. finding single source shortest paths as well as the collective classification belong to the wide group of emerging challenges in graph processing. They represent typical and fairly complex issues in the computation of networks and can be the basis for solving further extension problems, for instance graph colouring, topological sorting, graph searching, minimum spanning trees or all-pairs shortest paths. It means that the analysis of the efficiency of SSSP and RIP solutions could also be common for other similar graph algorithms.

The general profile of both graph problems are presented in Table 3. Their three different implementations MapReduce, MapReduce with map-side join extension and Bulk Synchronous Parallel are depicted in Figs. 3–5, respectively.

6.1. Single source shortest path

A typical graph problem that was widely studied in graph analysis is the single source shortest paths (SSSP) calculation. For unweighed graphs, this problem can be interpreted as the computation of the minimal number of edges that form a path from an arbitrary chosen vertex to every other vertex in a graph. Implementations of the algorithm have been studied and proposed both for MapReduce [5] and for BSP [2].

At the beginning, all vertices are initiated with the maximum possible value and the source vertex with value 0. In the first iteration, the source vertex sends an updated shortest path value equal to one to its immediate neighbours. In the next iteration, these neighbours may propagate the path lengths incremented by 1 to their neighbours, and so on. Every vertex (neighbour) may receive a message with the shortest path smaller than its currently stored value. In such a case, the vertex becomes active and broadcasts the updated shortest path (increased by 1) to all its neighbours. However, if the received path length is greater than the current value for a given vertex, it does not send any message. In this way, at every iteration (MapReduce job/BSP superstep) a frontier of vertices with the newly discovered shortest paths is extended by one hop. The entire algorithm can be simply translated from the MapReduce to BSP model and vice versa. The MapReduce processing is expressed in two functions and the map functions emit the vertex itself to preserve the graph structure.

6.2. Collective classification of graph vertices based on Relational Influence Propagation

Overall, the term *collective classification* refers to the specific classification task of a set of interlinked objects. The common representation used in such problems are graphs, where each object is represented by a vertex and relationships between objects are graph edges. In collective classification, each vertex can be classified according to the generalized knowledge derived from correlations between the labels and attributes of vertices or correlations between the labels and observed attributes supplemented by labels of neighbouring vertices. Unlike standard classification, the collective approach utilizes information about relationships between vertices. For example, it is very likely that a given web page x is related to sport (label *sport*), if page x is linked by many other web pages about sport.

Collective classification can be accomplished by two distinct inference approaches: within-network classification or across-network classification. Whereas the former tries to recover partially unknown labels in the network based on information of known ones, the latter applies the recovered label distribution from one network to another.

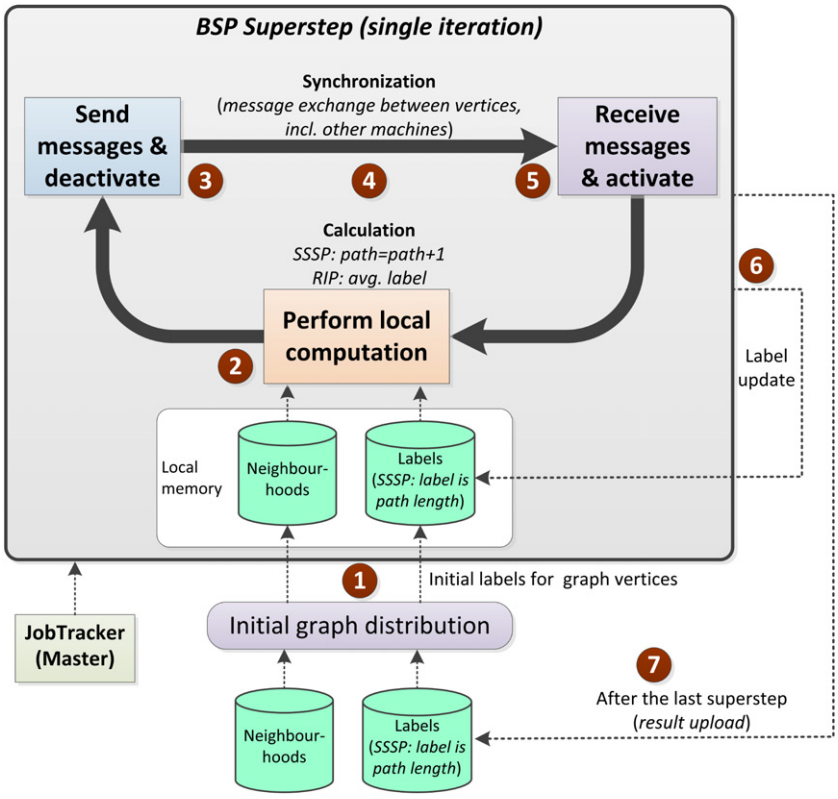


Fig. 5. Bulk Synchronous Parallel model (BSP) in graph processing.

Algorithm 1 MapReduce approach to Single Source Shortest Path

```
1: function MAP(vertexId n, vertex V)
2:    if V.active==TRUE then
3:       for outgoingEdge E ∈ V.adjacencyList do
4:          emit(E.neighborId, V.currentDistance+1)
5:       end for
6:       V.active=FALSE
7:    end if
8:    emit(vertexId n, vertex V)
9: end function

1: function REDUCE(vertexId n, messages[m1, m2, ...])
2:    integer minimalDistanceFoundInMessages = Inf
3:    originalVertex V = null
4:    for m ∈ messages do
5:       if m isTypeOf Vertex then
6:          V ← m
7:       else
8:          if m < minimalDistanceFoundInMessages then
9:             minimalDistanceFoundInMessages = m
10:         end if
11:      end if
12:   end for
13:   if V is null OR V.minimalDistance >
   minimalDistanceFoundInMessages then
14:      V.minimalDistance=minimalDistanceFoundInMessages
15:      V.active=TRUE
16:   else
17:      V.active=FALSE
18:   end if
19:   emit(vertexId n, Vertex V)
20: end function
```

The within-network collective classification algorithm based on *Relational Influence Propagation* (RIP) has been examined in this paper. The general idea of this approach is based on the iterative propagation of the known labels assigned to some known vertices to the non-labelled vertices in the same network. The method was originally derived from the enhanced hypertext categorization [27].

The RIP algorithms for MapReduce and BSP have already been considered in [28,29]. The pseudo-codes of implementations for

Algorithm 2 Bulk Synchronous Parallel approach to Single Source Shortest Path

```
1: function COMPUTE(vertex V, messages[m1, m2, ...])
   minimalDistanceFoundInMessages = Inf
2:    for m in messages do
3:       if m < minimalDistanceFoundInMessages then
4:          minimalDistanceFoundInMessages = m
5:       end if
6:    end for
7:    if V.minimalDistance>minimalDistanceFoundInMessages then
8:       V.minimalDistance=minimalDistanceFoundInMessages
9:       for E ∈ V.adjacencyList do
10:         sendMsg(E.neighborId, V.minimalDistance+1)
11:      end for
12:   end if
13:   voteToHalt()
14: end function
```

MapReduce and BSP are presented in Algorithms 1 and 2, respectively, in order to demonstrate the differences between these two programming models. The implementation in the MapReduce paradigm consists of two separate functions: *map*() and *reduce*(). Each *map*() reads and processes vertex by vertex. For every vertex with a known label, it propagates its label to its all immediate neighbours (lines 2–3 in Algorithm 1). Since MapReduce is a stateless model, it must also send its own representation to itself (line 4). Messages transmitted by the map functions are consumed by the reduce functions, which are usually executed on different machines. When the reduce function receives all the messages, it must distinguish messages from two types: a message sent by itself, which contains the vertex adjacency list (lines 4–5), and messages sent by its direct neighbours, having their labels already assigned. Next, it aggregates all received external labels along weights of graph edges linking the vertex to its neighbours (lines 7–8). Finally, the updated likelihood expressed by means of the weighted mean of neighbourhood likelihood is computed for a given vertex (line 11). As a result, the vertex obtains its own new label. At the end,

the vertex propagates its own representation, containing its updated label. The output will serve as the input for the map function in the next iteration (MapReduce job).

Unlike MapReduce, in the BSP implementation, the whole processing for a vertex is expressed with only one function *compute*(), which is executed on a single machine, see Algorithm 2. Function *compute*() is triggered once in each superstep (iteration) separately for every vertex that has received a message sent in the previous superstep. The logic of the RIP algorithm is expressed similarly as in MapReduce: vertices receive and aggregate messages from their neighbours (lines 2–5), calculate the new likelihood (line 6) and send their updated likelihood to all their neighbours. The main difference between the MapReduce model is that the vertices do not have to propagate themselves. This is guaranteed by the stateful nature of processing in BSP—all the computation for one vertex is always executed on the same machine, and the vertex state including information about its neighbours (in particular weights of edges) can be stored in-memory or another local storage.

Algorithm 3 MapReduce approach to Relational Influence Propagation

```
1: function MAP(vertexId n, vertex V)
2:    for outgoingEdge E ∈ V.adjacencyList do
3:       emit(E.neighborId, <V.label, E.edgeWeight>)
4:    end for
5:    emit(vertexId n, vertex V)
6: end function

1: function REDUCE(vertexId n, messages[m1, m2, ...])
2:    originalVertex V = null
3:    for m ∈ messages do
4:       if m.isVertex() then
5:          V ← m
6:       else
7:          sumLabels += m.label*m.weight
8:          sumWeights += m.weight
9:       end if
10:   end for
11:   V.label ← sumLabels/sumWeights
12:   emit(vertexId n, Vertex V)
13: end function
```

Algorithm 4 Bulk Synchronous Parallel approach to Relational Influence Propagation

```
1: function COMPUTE(vertex V, messages[m1, m2, ...])
2:    for m in messages do
3:       sumLabels += m.label*m.weight
4:       sumWeights += m.weight
5:    end for
6:    V.label ← sumLabels/sumWeights
7:    for E ∈ V.adjacencyList do
8:       sendMsg(E.neighborId, [V.label, E.edgeWeight])
9:    end for
10:   voteToHalt()
11: end function
```

7. Experimental environment

The main goal of experiments was to validate and compare the open source parallel programming models: (1) MapReduce (MR), (2) MapReduce with map-side join (MR2) and (3) Bulk Synchronous Parallel (BSP), see Section 4. These three approaches: MR, MR2 and BSP were evaluated in terms of computational complexity for distinct settings of a distributed environment. The efficiency measures were recorded for clusters with various numbers of computational nodes (from 10 to 85 machines). The analysis was performed for distinct real world datasets and for two graph analysis problems: SSSP and RIP, see Section 6. Since all implementations required the same number of algorithm iterations (equal to 10) to reach satisfactory stability of results, the mean execution time of a single iteration was used as the evaluation measure in order to compare all approaches.

Table 4
Description of datasets used in experiments.

	tele_small	tele	youtube	twitter
Domain	telecom	telecom	multim.	microblog
No. of nodes	5,098,639	13,914,680	16,416,516	43,718,466
No. of edges	21,285,803	67,184,654	66,068,329	688,352,467
Avg deg.	4.17	4.83	4.02	15.75
Max indegree	40,126	294,690	4104	1,228,086
Directed (y/n)	y	y	y	y
No. of classes	2	2	15	–

7.1. Environment setup

The experiments were carried out using a cluster environment provided by the Wroclaw Networking and Supercomputing Center. The distributed system consisted of 85 identical machines with 4 CPUs, 16 GB RAM and 750 GB of storage each. Computers were connected through 1 Gb/s Ethernet. The experiments were deployed on Hadoop (version 0.20.205.0) and Apache Giraph (version 0.2). Apache Giraph [15] is an immature, open-source implementation of the BSP model for graph processing in the cloud environment. All programs for Giraph were written as a Hadoop jobs because Giraph launches BSP workers within mappers and then uses Hadoop RPC to enable workers communicate with each other.

7.2. Datasets

To compare the MapReduce and BSP approaches four separate datasets were used: *tele*, *tele_small*, *youtube* and *twitter*. The *tele* dataset is a network of telecommunication clients built over three months history of phone calls from one of the leading European telecommunication company. The raw dataset used to construct the network consisted of about 500 000 000 phone calls and more than 16 million unique users. Extracted network was built using the activities of clients performing calls in the two most popular from 38 tariffs. Another dataset, *tele_small*, was composed based on the next two most common tariffs. In both datasets users were marked with the empirical probability of the tariffs they used, namely, the sum of outcoming phone call durations in a particular tariff was divided by summarized duration of all outcoming calls.

The YouTube dataset [30] was crawled using the YouTubeAPI in 2008. A subset of all attributes was used to create a weighted graph structure and labels: *video_id, age, category, related_IDs*. Using *related_IDs* the connection strengths were calculated as a fraction of the number of related adjacent videos, i.e. if there were 20 related videos each of them was linked by an edge with the weight of 0.05.

The Twitter dataset was the largest dataset utilized in the experiments. It contained a list of two users' identifiers, implying a link from the first to the second user (the first user follows the second one). As there were no available labels for a user, to enable RIP validation a binary class was assigned randomly to each of the network vertices using the uniform distribution. Some other details about the datasets are presented in Table 4.

8. Results of experiments

8.1. Time of computation for various graph problems

First of all, MR, MR2 and BSP approaches were examined against the type of graph processing problems using a fixed number of computational nodes (60 machines). The Relational Influence Propagation (RIP) as well as Single Source Shortest Paths (SSSP) problems were executed in order to test the execution time. The results for three parallel approaches separately for the RIP and SSSP problems are presented in Figs. 6 and 7, respectively. In general, BSP outperformed the regular MapReduce (MR) by 70%–90% (3–10

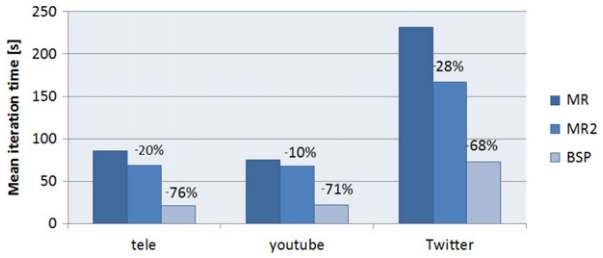


Fig. 6. Performance of Relational Influence Propagation problem on 60 machines (with the percentage drop of mean iteration time in comparison to MR).

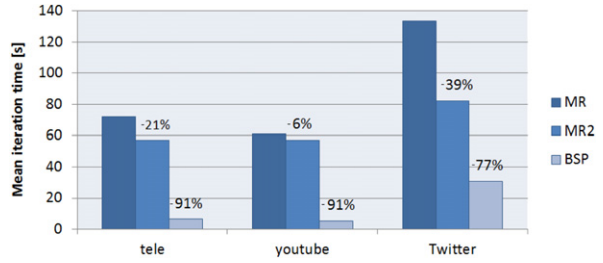


Fig. 7. Performance of Single Source Shortest Paths on 60 machines (with the percentage drop of mean iteration time in comparison to MR).

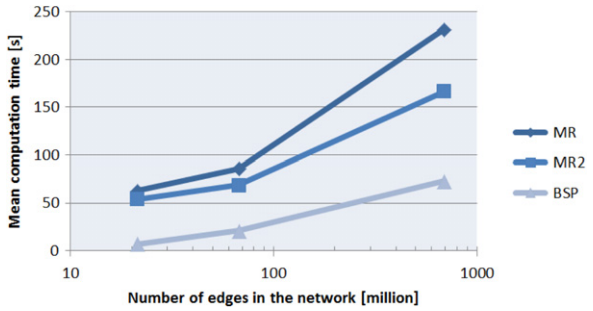


Fig. 8. Scalability with respect to the size of graph for the Relational Influence Propagation algorithm with a fixed number of machines (60).

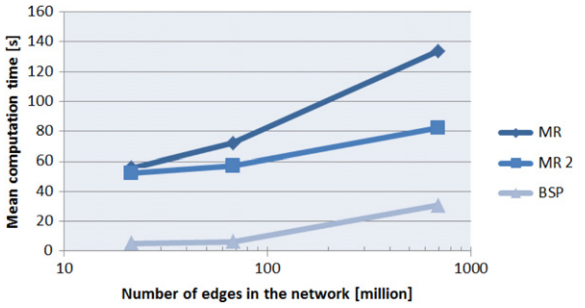


Fig. 9. Scalability with respect to the size of graph for the Single Source Shortest Path algorithm with a fixed number of machines (60).

times better) and the improved MapReduce (MR2) by 50%–90% (2–10 times better), and more for the SSSP problem than for RIP. Note also that the efficiency gain obtained by MR2 is slightly higher for the SSSP problem than for RIP.

8.2. Scalability in relation to the graph size

The next research focused on the question, how the processing time of MR, MR2 and BSP solutions depends on the sizes of graphs for the fixed configuration (60 machines). Again, the experiment was performed on three datasets with significantly different numbers of nodes (*tele_small*, *tele* and *twitter*). According to Figs. 8 and 9 all three approaches shown their nearly linear scalability for both graph problems. Once again, MR2 and BSP outperformed the MR implementation, especially for the largest graph, even though the gain is quite stable. Similar characteristics were observed in relation to the increasing number of edges.

8.3. Horizontal scalability in relation to the number of parallel machines

The influence of the computer cluster (cloud) size on efficiency (processing time) was also examined. Three parallel models (MR, MR2 and BSP) were compared against each other for different numbers of machines, starting from 5 to 85. The results are depicted in Fig. 10(a–h).

For the smaller datasets like *tele_small*, *tele*, *youtube* all three solutions, MR, MR2 and BSP, converge to a limit of the best mean computation time of iteration with about 30 computational nodes (hosts) in the cloud. Adding more machines eliminates the benefit of additional computing power due to the overhead related to network communication. The mean execution time of a single iteration can be decreased by 2 to 4 times by increasing the number of computational nodes from 5 to 30. This benefit is relatively higher for the RIP problem than for SSSP, and in most cases it is higher for BSP than for MapReduce approaches. The 17 times increase of the number of machines (from 5 to 85) speeds up the processing by 14 times (2900–205 s) for MR, the RIP problem and *twitter* (Fig. 10(d)) and even 16 times for MR2 (2400–146 s). Due to the in-memory calculation limit of the BSP approach, it was able to execute the *twitter* dataset on an environment starting only from 50 machines (Fig. 10(d)). Adding more machines improves the performance only slightly.

On the other hand, the mean time of one iteration depends only to a moderate degree on the number of computational nodes; it refers to computing for the SSSP problem (Fig. 10(e–h)). The biggest observed improvement for MR and MR2 was only 2.7 and 4.6 times faster, respectively.

Similarly to the RIP algorithm for the *twitter* dataset, the BSP implementation of SSSP was not able to be executed until 30 or more machines were in the cluster (Fig. 10(h)). Once more it was observed, that in case of BSP approach, increasing the number of computational nodes reveals only a slight improvement of the processing time.

The worst scalability can be observed for the *tele_small* dataset, less than 2 times for MR and MR2, while increasing the number of hosts from 5 to 10, and about 3 times for BSP, but not until 20–30 nodes (Fig. 10a,e).

For some cases, the extension of the computer cluster over 30 machines even decreases efficiency—average processing time slightly rises (Fig. 10(a,e,f,g)), but it is valid only for MapReduce solutions (MR and MR2). In these cases, the efficiency stabilizes at the certain level for BSP.

8.4. Scalability in relation to the number of algorithm iterations

The total processing time for RIP and SSSP algorithms is shown in Figs. 11 and 12. The values presented additionally aggregate the initiation time attributed to each iteration for BSP and MapReduce models (MR and MR2). In all approaches, regardless of the calculated problem, the processing time increases linearly with the number of iterations. The coefficient of determination (R^2) between real values and their linear equivalence for MR and MR2 is

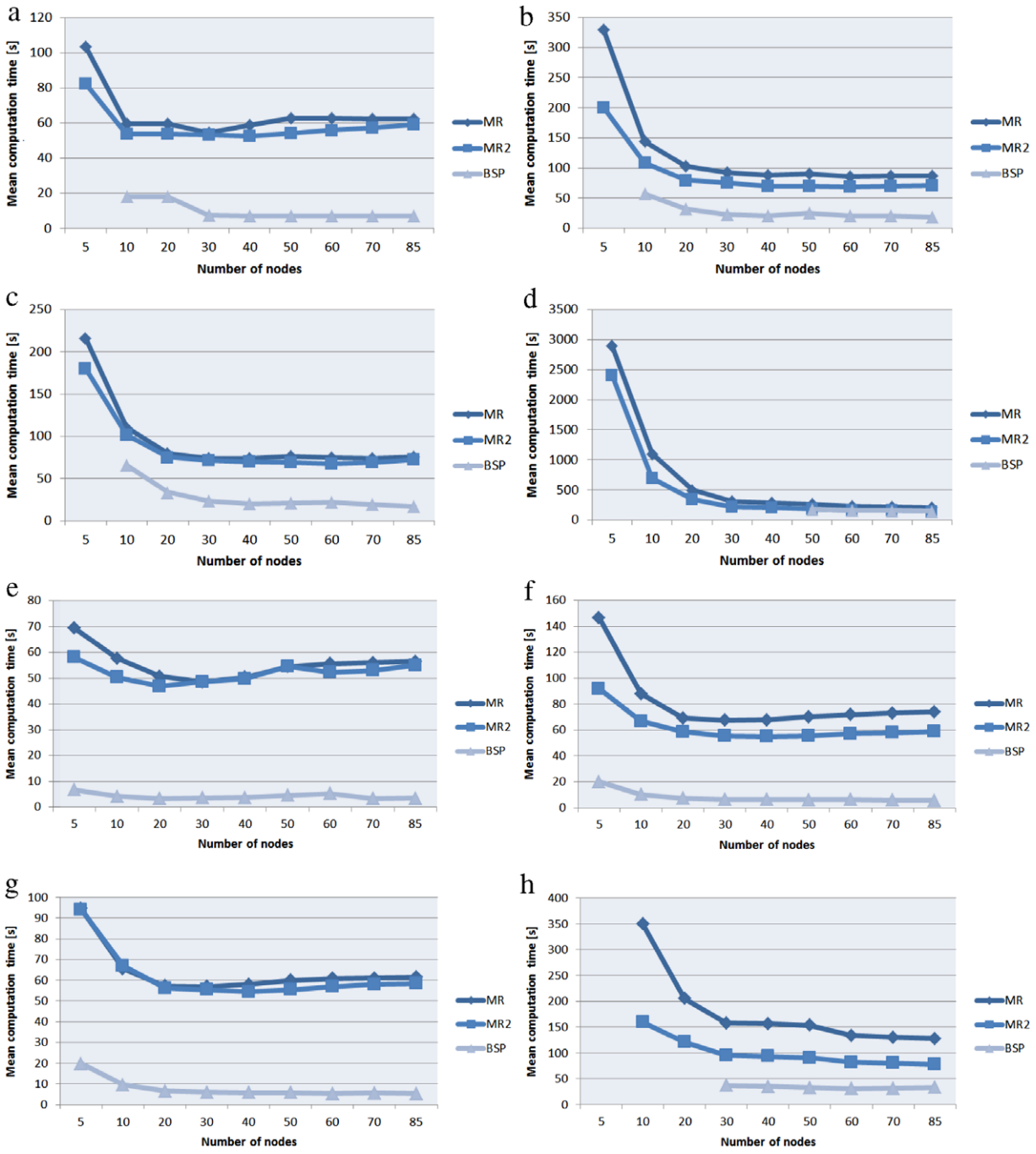


Fig. 10. Scalability with respect to the number of hosts (nodes) in the computer cluster for the Relational Influence Propagation (RIP) algorithm applied to *tele_small* (a), *tele* (b), *youtube* (c), *twitter* (d) datasets as well as for the Single Source Shortest Path (SSSP) algorithm executed on *tele_small* (e), *tele* (f), *youtube* (g), *twitter* (h) datasets.

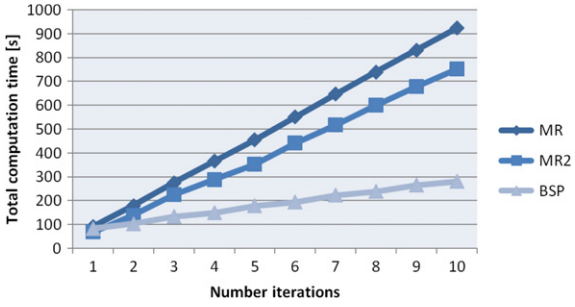


Fig. 11. Total processing time with respect to the number of iterations in the Relational Influence Propagation algorithm run on 40 machines (*tele* dataset).

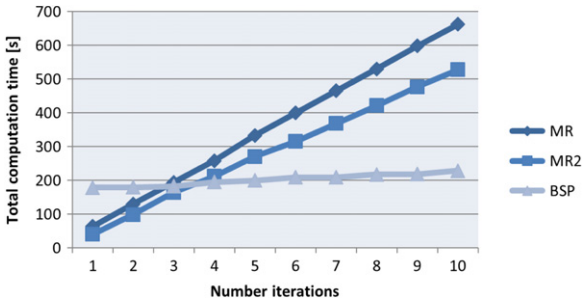


Fig. 12. Total processing time with respect to the number of iterations in the Single Source Shortest Path algorithm executed on 40 machines (*tele* dataset).

at the level of 0.9998–0.9999 and for BSP is $R^2 = 0.9747$–0.9976. A slightly lower correlation between real values and the linear approximation for BSP compared to MR/MR2 was caused by the slightly longer duration of the first iteration (superstep) of BSP, which contained the graph data distribution among parallel machines.

Additionally, MapReduce approach iterations are significantly slower and their total processing time rises more rapidly.

It can be also noticed that the time required to complete the first iteration is approximately three times longer for BSP than for the MR/MR2 implementation for the SSSP problem. It is caused by the initialization procedure required for loading the graph structure into the distributed memory before the first superstep of BSP. It is valid for the SSSP problem; however, for RIP, the first iteration lasts the same time for both approaches. It means that the time consumed for loading the structure of the graph may vary on many factors and it is strongly implementation dependent.

8.5. Resource usage over time

The detailed charts with the consumption of the main resources include data about: CPU usage of processing machines (cluster), CPU usage for the master host, network traffic (communication between hosts), memory occupation for processing hosts and separately for the master host, see Figs. 13 and 14. They were monitored for the MapReduce approach and BSP separately for RIP (Fig. 13) and SSSP graph algorithms (Fig. 14). Obviously, BSP processing lasted shorter than MR regardless of the graph problem, see X scales for both approaches.

The iterative nature of the algorithm is clearly visible in almost all the charts, especially for CPU cluster usage and memory usage. In the case of MapReduce, the highest consumption of CPU is observable for both mapper processing (B and F steps), lower for shuffling (especially steps C and lower for steps G), even lower for the reduce phase (steps D and H) and processors are least used for storing results (steps E and I). Similarly, for BSP and the RIP problem, the lowest CPU demand is at the synchronization stage between supersteps (steps 6). In the BSP + SSSP, the synchronization is very short and actually invisible in the chart (Fig. 14, top-right).

It can be noticed that one of the crucial problems in graph processing are diverse distributions of the sizes of node's neighbourhoods. For that reason, some computing hosts have to wait until all graph nodes are processed, see orange spaces in CPU usage, especially before synchronization (stages E, I and 6).

In general, BSP transfers much less data than MR, please note the different Y axis scales. The data exchange for BSP is much greater for the RIP problem than for SSSP, except for graph loading stage at the beginning. For RIP + BSP, data exchange is lower at the beginning, when there are only a few graph nodes with labels assigned at the first iterations. After the 2–3 first iterations, all nodes have their labels and the algorithm only re-assigns them. It means that more data needs to be transferred.

For the BSP + SSSP, after initial graph structure loading, the CPU usage and network traffic grows for every iteration since the shortest paths grow to their maximum. Then, some of them are already completed: CPU usage drops and network transfer keeps rising. Also, longer paths need to be stored leading to greater usage of memory.

BSP solutions (both for RIP and SSSP) utilize much more memory space, which comes from its general concept.

9. Discussion

In general, the results presented in Section 8 revealed that the MapReduce approach (MR) is outperformed by Bulk Synchronous Parallel (BSP) as well as by application of the map-side join design pattern (MR2). Nevertheless, the efficiency gain differs for distinct computational problems. It is exposed that usage of MR2 or BSP instead of raw MR provides greater performance improvement for Single Source Shortest Paths (SSSP) calculation than for Relational Influence Propagation (RIP) computation. This can be explained by the time required for communication between computational nodes (hosts). In RIP, the communication was very dense, as 70% of all graph vertices used as a source for propagation were propagating their labels to almost all other vertices. In the SSSP problem, in turn, the number of propagating vertices was changeable and depended on the iteration, but in general, was much smaller than in RIP.

The general concept of the BSP model facilitates in-memory computation for the entire graph. This is the main reason why the BSP implementation outperforms MapReduce for all considered graph-based problems. In case of MapReduce, the messages exchanged between vertices as well as intermediate data transferred in-between iterations must be written to the disk. However, in-memory processing requires that the whole graph structure and all intermediate data produced during processing (e.g. messages exchanged between vertices) must fit in the memory. Otherwise, a spilling mechanism should be provided in order to manage the data in the memory and external storage. Unfortunately, to the best of our knowledge, none of the recent open-source BSP systems provides any equivalent mechanism. Hence, for very large networks, MapReduce remains the only alternative. As was shown in Section 8.3, the risk of not fitting in memory is higher for more communication intensive problems such as Relational Influence Propagation.

Overall, in-memory processing allows graph's vertices to remain in the same physical location throughout the computation, see Fig. 5. On the other hand, additional overhead is required at the beginning of computation in order to partition the network and load it into the memory of workers. The overhead may be compensated after a few iterations (as was discussed in Section 8.4, see Fig. 12). Notwithstanding, the BSP model may be a very good solution for non-iterative graph algorithms.

The necessity of data exchange and access to distributed file systems (HDFS) grows with a higher number of parallel machines. It especially refers to MapReduce approaches, both original (MR) and improved (MR2), since the graph vertex may be re-allocated at every iteration of MapReduce-based algorithms. As a result, adding new machines does not need to decrease the average processing of a single algorithm iteration. For the datasets and graph problems analysed in the experiments, the reasonable upper limit of computational nodes is about 20–30, see Section 8 and Fig. 10.

10. Conclusions and further work

Three main approaches to parallel processing of graph data, namely: (i) MapReduce (MR), along with (ii) its extension based on map-side joins (MR2) as well as (iii) the Bulk Synchronous Parallel (BSP) were examined in the paper. Two graph problems that can be solved by means of iterative algorithms were implemented and tested separately for the above three approaches: calculation of the lengths of single source shortest paths (SSSP) as well as Relational Influence Propagation (RIP) used for collective classification of graph nodes.

The experimental studies on four large graph datasets with different profiles revealed that the Bulk Synchronous Parallel approach outperforms other solutions (MR and MR2) for all datasets and all tested iterative graph problems. The BSP model, despite its relatively young implementation, worked even up to one order of magnitude faster than the MapReduce-based approaches. The superiority of BSP was greater for telecom data rather than for twitter data (compare Fig. 10(a,e) with Fig. 10(d,h)).

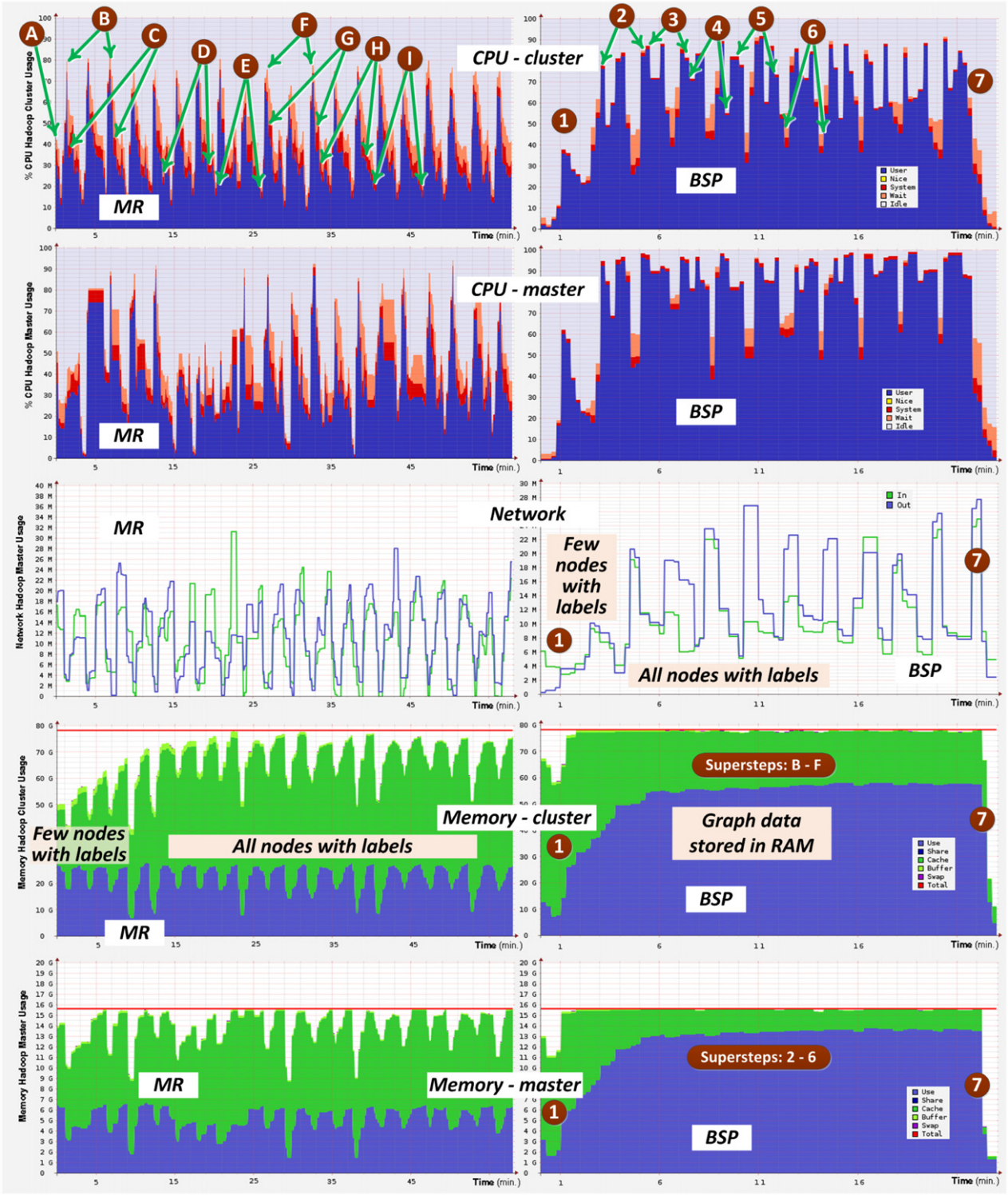


Fig. 13. Resource usage for the Relational Influence Propagation (RIP) problem; MR step letters and BSP step numbers correspond to stages in Figs. 3 and 5, respectively; results for the *tele* dataset.

Simultaneously, it was discovered that the map-side join design pattern (MR2) may improve the original MapReduce performance up to two times. It is caused by replacement of one Map-Reduce job by single join at every iteration (compare, Figs. 3 and 4).

Parallel processing of graph data also has some limitations. The gain in increasing the number of parallel hosts is visible only up to certain quantity. Based on experiments from Section 8, see Fig. 10, we can state that about 20–30 hosts is the upper limit. The increase above this threshold does not result in faster processing. It is visible especially for MapReduce solutions (MR and MR2). This phenomena was caused by the necessity of more extensive data exchange in the case of a larger number of parallel machines.

Nevertheless, MapReduce can still remain the only mature enough alternative for parallel processing of graph algorithms on huge datasets. It results from the main BSP limitation: the very high

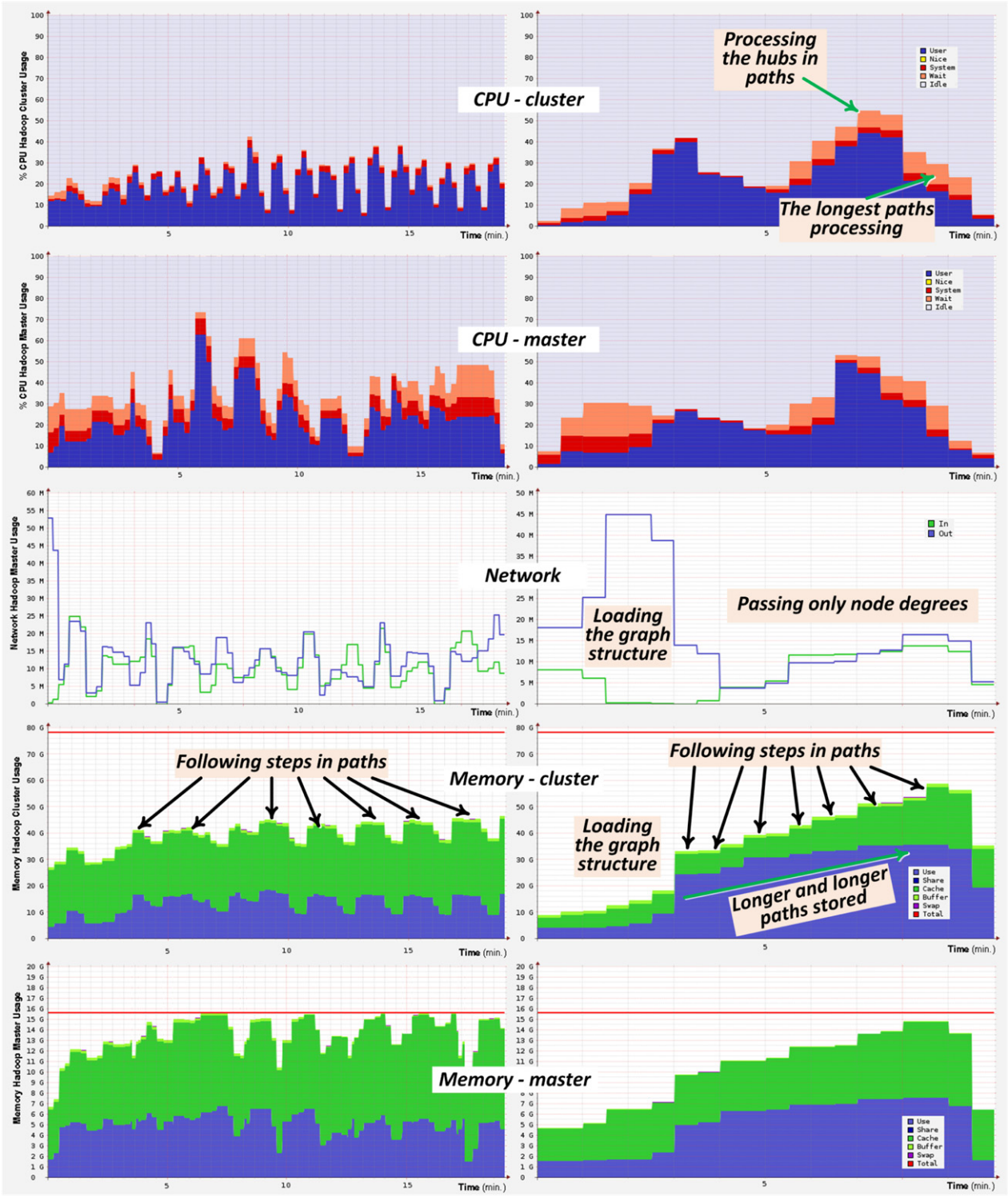


Fig. 14. Resource usage for the SSSP problem and *tele* dataset.

memory requirements—all the data used for local processing in BSP must fit in the local memory.

However, it should be noticed that recently some implementations of the BSP processing model are also becoming more and more mature, e.g. Apache Giraph has just been featured with disk storage support for large (larger than RAM memory) graphs. Also a new framework for MapReduce in-memory computations has just been developed (Spark project at Berkeley University). The comparison of both in-memory frameworks for BSP (Giraph) and MapReduce (Spark) could be a very interesting direction for future work.

Acknowledgements

This work was partially supported by The Polish National Center of Science research project 2010–13 and 2011–14, Institute of Informatics of Wroclaw University of Technology, The European Commission under the 7th Framework Programme, Coordination

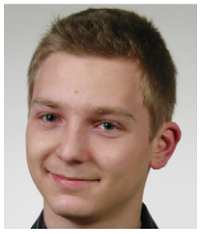
and Support Action, Grant Agreement Number 316097, ENGINE—European research centre of Network intelliGence for INnovation Enhancement http://engine.pwr.wroc.pl/. Calculations have been carried out in Wroclaw Centre for Networking and Supercomputing (http://www.wcss.wroc.pl), grant No 177.

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, Challenges in parallel graph processing, Parallel Process. Lett. 17 (1) (2007) 5–20.
[2] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 International Conference on Management of Data, SIGMOD'10, ACM, New York, NY, USA, 2010, pp. 135–146.
[3] S. Yang, B. Wang, H. Zhao, B. Wu, Efficient dense structure mining using MapReduce, in: ICDM Workshops'09, IEEE Computer Society, 2009, pp. 332–337.
[4] J. Lin, M. Schatz, Design patterns for efficient graph algorithms in MapReduce, in: Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG'10, ACM, New York, NY, USA, 2010, pp. 78–85.
[5] J. Lin, C. Dyer, Data-Intensive Text Processing with MapReduce, in: Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers, 2010.
[6] Y. Bu, B. Howe, M. Balazinska, M. Ernst, Haloop: efficient iterative data processing on large clusters, Proc. VLDB Endowment 3 (1–2) (2010) 285–296.
[7] U. Kang, C. Tsourakakis, 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, IEEE Computer Society, Washington, DC, USA, 2009, pp. 229–238.
[8] Y. Zhang, Q. Gao, L. Gao, C. Wang, iMapReduce: a distributed computing framework for iterative computation, J. Grid Comput. 10 (1) (2012) 47–68.
[9] R. Chen, X. Weng, B. He, M. Yang, Large graph processing in the cloud, in: Proceedings of the 2010 International Conference on Management of Data, SIGMOD'10, ACM, New York, NY, USA, 2010, pp. 1123–1126.
[10] E. Elnikety, T. Elsayed, H. Ramadan, iHadoop: asynchronous iterations for MapReduce, in: CloudCom'11, 2011, pp. 81–90.
[11] L. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[12] T. Kajdanowicz, W. Indyk, P. Kazienko, J. Kukul, Comparison of the efficiency of MapReduce and bulk synchronous parallel approaches to large network processing, in: Proceedings of the ICDM 2012 Workshops, The Second IEEE ICDM Workshop on Data Mining in Networks, ICDM'2012, IEEE Computer Society, New York, NY, USA, 2012, pp. 218–225.
[13] T. White, Hadoop: The Definitive Guide, O'Reilly, 2010.
[14] D. Krizanc, A. Saarimaki, Bulk synchronous parallel: practical experience with a model for parallel computing, in: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques 1996, 1996, pp. 208–217.
[15] Apache Giraph, http://incubator.apache.org/giraph/ (accessed: 31.05.2012).
[16] S. Seo, E. Yoon, J. Kim, S. Jin, J. Kim, S. Maeng, Hama: an efficient matrix computation with the MapReduce framework, in: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 721–726.
[17] B. Shao, H. Wang, Y. Xiao, Managing and mining large graphs: systems and implementations, in: Proceedings of the 2012 International Conference on Management of Data, SIGMOD'12, ACM, New York, NY, USA, 2012, pp. 589–592.
[18] M. Najork, D. Fetterly, A. Halverson, K. Kenthapadi, S. Gollapudi, Of hammers and nails: an empirical comparison of three paradigms for processing large graphs, in: Proceedings of the Fifth ACM International Conference on Web Search and Data Mining, WSDM'12, ACM, New York, NY, USA, 2012, pp. 103–112.
[19] M.F. Pace, BSP vs MapReduce, CoRR arXiv:abs/1203.2081.
[20] A. Lumsdaine, G. Douglas, B. Hendrickson, J. Berry, Challenges in parallel graph processing, Parallel Process. Lett. 17 (2007) 5–20.
[21] MPI: a message passing interface, Proc. of Supercomputing'93 Message Passing Interface Forum, IEEE Computer Society Press, 1993, pp. 878–883.
[22] T.A. El-Ghazawi, W.W. Carlson, J.M. Draper, UPC language specification, 1.2 edition, http://www.gwu.edu/~upc/downloads/upc_specs_1.2.pdf.
[23] J. Cohen, Graph twiddling in a MapReduce world, J. Comput. Sci. Eng. 11 (2009) 29–41.
[24] J. Huang, D. Abadi, K. Ren, Scalable SPARQL querying of large RDF graphs, Proc. VLDB Endowment 4 (21) (2011).
[25] S. Salihoglu, J. Widom, Gps: a graph processing system, Technical Report, Stanford University. URL: http://ilpubs.stanford.edu:8090/1039/.
[26] P. Kalnis, Z. Khayyat, K. Awara, H. Jamjoom, Mizan: optimizing graph mining in large parallel systems. URL: http://hdl.handle.net/10754/217609.
[27] S. Chakrabarti, B. Dom, P. Indyk, Enhanced hypertext categorization using hyperlinks, SIGMOD Rec. 27 (2) (1998) 307–318.
[28] W. Indyk, T. Kajdanowicz, P. Kazienko, Relational large scale multi-label classification method for video categorization, Multimedia Tools Appl. 65 (1) (2013) 63–74.
[29] P. Haller, H. Miller, Parallelizing machine learning-functionally: a framework and abstractions for parallel graph processing, in: 2nd Annual Scala Workshop, 2011.
[30] X. Cheng, C. Dale, J. Liu, Statistics and social network of YouTube videos, in: 16th International Workshop on Quality of Service, IWQoS 2008, 2008, pp. 229–238.

Tomasz Kajdanowicz received his M.Sc. and Ph.D. degrees in computer science with honours, both from Wroclaw University of Technology, Poland, in 2008 and 2012, respectively. Recently, he serves as an assistant professor of Wroclaw University of Technology at the Institute of Informatics, Poland. He was also an Intelligent Systems Solution Consultant at Hewlett Packard, Poland. He was a co-chair and a member of organizing committee of workshops MMAML'10, MMAML'11, MMAML'12, MMAML'13. He regularly serves as a member of international programme committees and the reviewer for prestige international journals and scientific conferences. He has authored over 50 research articles in a variety of areas related to ensemble classification, collective classification, machine learning, social network analysis, collaborative systems, data mining and recommender systems. He also initialized and led 5 research projects in cooperation with commercial companies, including large international corporations.

Przemyslaw Kazienko received his M.Sc. and Ph.D. degrees in computer science with honours, both from Wroclaw University of Technology, Poland, in 1991 and 2000, respectively. He obtained his habilitation degree from Silesian University of Technology, Poland, in 2009. Recently, he serves as a professor of Wroclaw University of Technology at the Institute of Informatics, Poland. He was also a Research Fellow at Intelligent Systems Research Centre, British Telecom, UK in 2008. For several years, he has held the position of deputy director for development at the Institute of Applied Informatics. He was a co-chair of the international conference WSKS'11 and workshops RAAWS'05, RAAWS'06, MMAML'10, MMAML'11, MMAML'12, SNAA'11, SNAA'12, CSNA'12 and a guest editor of New Generation Computing, International Journal of Applied Mathematics and Computer Science, Journal of Universal Computer Science and International Journal of Computer Science & Applications. He regularly serves as a member of international programme committees and the reviewer for prestige international journals and scientific conferences. He is a member of the Editorial Board of International Journal of Knowledge Society Research, International Journal of Human Capital and Information Technology Professionals, as well as Social Informatics. He has authored over 130 scholarly and research articles in a variety of areas related to multiple model classification, collective classification, social network analysis and application, knowledge management, collaborative systems, data mining, recommender systems, information retrieval, data security, and XML. He also initialized and led over 25 projects chiefly in cooperation with commercial companies, including large international corporations.



Wojciech Indyk is a student at Wroclaw University of Technology, Poland. He has been engaged in multiple research projects, including a research project on multi-label classification for debt portfolio valuation. His research interest include multi-label classification, relational machine learning and network analysis. He has co-authored four scientific papers and participated in the 7th International Summer School on Pattern Recognition (ISSPR), Plymouth, UK, in 2011. Recently, he serves as the president of the "Data and exploration" student's scientific group at Wroclaw University of Technology.