BC-BSP: A BSP-Based Parallel Iterative Processing System for Big Data on Cloud Architecture

Yubin Bao[1], Zhigang Wang[1],Yu Gu[1], Ge Yu[1], Fangling Leng[1], Hongxu Zhang[2], Bairen Chen[2], Chao Deng[3], and Leitao Guo[3]

[1] School of Information Science and Engineering, Northeastern University, Shenyang, China
{baoyubin,guyu,yuge,lengfangling}@ise.neu.edu.cn
[2] Software Division, Neusoft Corp., Shenyang, China
{kevinzhang,chenbr}@neusoft.com
[3] China Mobile Institute, China Mobile Corp., Beijing, China
{dengchao,guolt}@chinamobile.com

Abstract. Many applications in real life can produce and collect large amount of data and many of them can be modeled by Graph. The number of vertexes of a graph could be several hundreds of millions to billions and the number of edges could be ten or more times of the number of its vertexes. A BSP-based system for large-scale data (especially graph data) parallel and iterative processing is discussed in this paper. The system has the ability to flexible configuration and the extendibility for functions and strategies (such as adjusting the parameters according to the volume of data and supporting multiple aggregation functions at the same time), to process large-scale data, to tolerate faults, to balance load, and to run clustering or classification algorithms on metric datasets. Lots of experiments are done to evaluate the extendibility of the system implemented in the paper, and the comparison between BC-BSP-based applications and *MapReduce*-based ones are made. The experimental results show that BSP-based applications have higher efficiency than that of *MapReduce*-based applications when the volume of data can be put in the memory during the course of processing; on the contrary the latter are better than the former, and the performance of BC-BSP platform outperforms *Hama* and *Giraph*.

Keywords: BSP, MapReduce, Graph Processing, Disk Cache, Big data.

1 Introduction

Graph is an abstract data structure which has been researched deeply in the area of computer science. It is so common to express the real world using graph, such as the road network, the spread of disease, the reference among technological literature, the links among web pages, the relationship among all kinds of objects in social network and the biological information network. So graph model can be used widely to model many applications. In spite of the theory and algorithms on graph have been researched in depth during the past several decades, most of them focus on small-scale datasets. With the development of the information technology, the scale of all kinds of

B. Hong et al. (Eds.): DASFAA Workshops 2013, LNCS 7827, pp. 31–45, 2013.

information keeps increasing rapidly, which leads to the scale of graphs larger and larger. The number may be even high in social network. Such as *Facebook*, the largest scale social network has about 700 million users. For search engines, such as *Google* and *Baidu*, it is necessary to evaluate web pages importance by related algorithms. The most famous one is *PageRank* algorithm. We can define a web page as a node in a link graph and the link between two pages is regarded as an edge with direction. So the rank score of a web page can be computed according to the links among pages. Given that the graph is organized by adjacent list and one whole record needs 100 bytes to store, if we store 10 billion nodes and 60 billion edges, the whole storage space will be more than 1 TB. The situation is similar with other applications, such as social network. The cost of time and space during processing the big scale graph has already exceeded the ability of concentrated computing traditionally. In conclusion, it has become a new challenge to process large scale data, especially large scale graph efficiently.

At present, *MapReduce*[1] computing model based on *Hadoop* ecosystem can process large-scale graph data with better fault-tolerance and scalability. While, most graph algorithms need to process graph data many times iteratively. One or more jobs are needed to complete an iterative computing task. As we known, the cost of the warm-up start of a *MapReduce* job is considerable. In order to solve this problem, *Google* developed an system for large-scale graph processing based on BSP model, called *Pregel*[2]. *Pregel* can process graph data in parallel and implement the communication among workers by message passing. However, *Pregel* assumes that all data including the processed data and intermediate data (such as message data) is resident in memory during the processing. Apparently, if the number of workers and the main memory capacity of each worker machine are limited, the scalability is also limited. It is not an open source project. There are two open source projects based on BSP model, *Hama*[3] and *Giraph*[4]. *Hama* is also good at processing big data iteratively, especially for processing matrix. But it does not consider the disk as an assistant device to temporary store graph data or messages when main memory is overflowed too in its early version. *Giraph* developed by *Yahoo* implements the BSP model based on *Hadoop* framework. Simply speaking, an application on *Giraph* is a special *MapReduce* job without reduce stage. It designs an inbuilt loop in the map task to simulate the super-steps of BSP model.

In this paper, we design a system BC-BSP in the Big Cloud environment of China Mobile Corp. Therefore we call it BC-BSP (Big Cloud-BSP), which is good at iteratively processing large scale graph data and other structured data. The features of BC-BSP and our contributions are as follows. a) BC-BSP implements the BSP model and uses the disk as the swap space to store part raw data and some intermediate data during the iterative processing when they all can not be put in main memory. So we can handle relative larger scale graph data if the number of available workers and the total memory capacity are limited. b) It provides flexible configuration and scalability. Users can choose or define the format of input and output. BC-BSP supports many data format, such as distributed file system, database. If it is need, users can define the special data format by relative interface. BC-BSP also supports a lot of strategies to partition the raw data. BC-BSP supplies hash partition, local partition and user-defined partition. c) It

takes load-balance into consideration. BC-BSP schedules tasks to workers with the consideration of data locality and tries to keep the load-balance. Especially, the load balance among workers is more prior than the data locality. d) Some experiments are made to compare and evaluate the performance and scalability between the applications based on BC-BSP and *MapReduce*.

The rest parts of the paper are arranged as follows. Section 2 introduces BSP model, section 3 gives the overview of BC-BSP, section 4 describes the interfaces of BC-BSP, the implementation of BC-BSP is presented in section 5, section 6 presents two application examples, *PageRank*[5] and *K-means*, on BC-BSP, section 7 shows the experimental results and the analysis about them, and the last section draws the conclusions and discusses some points and the future work.

2 Introduction to BSP Model

BSP[6](Bulk Synchronous Parallel) is a "bulk" synchronous model. There is a master to coordinate the whole other workers, which are the nodes in the cluster for storing data and running program to process data. BSP model is a parallel computing model based on super-step. A BSP-based application consists of a series of super-steps (see in Fig. 1(a)). In each super-step, the tasks on the cluster workers are asynchronous parallel running, and they can send messages to other workers for satisfying the requirements of the computing job. The next super-step can start until the computing of each worker has ended and messages sending and receiving of each task has completed. It is called barrier synchronization (see in Fig. 1(b)).

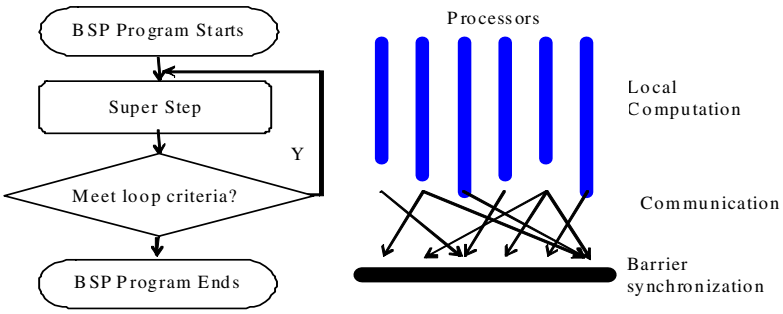

Fig. 1. BSP model. (a) The macro-running procedure of a BSP program, (b) The running procedure in a BSP super-step

3 Overview of BC-BSP

Figure 2 shows the framework of BC-BSP system which consists of the core computing engine and management tools. The core computing engine consists of the *Client*, the *BSPController*, the *Worker*, the *Task*, the *Global Synchronizer*, the *Message*

Communicator, API/CLI, and the *Fault Tolerance Controller*. The management tools consist of deployment and configuration tool, log management tool, performance management tool, and fault management tool.

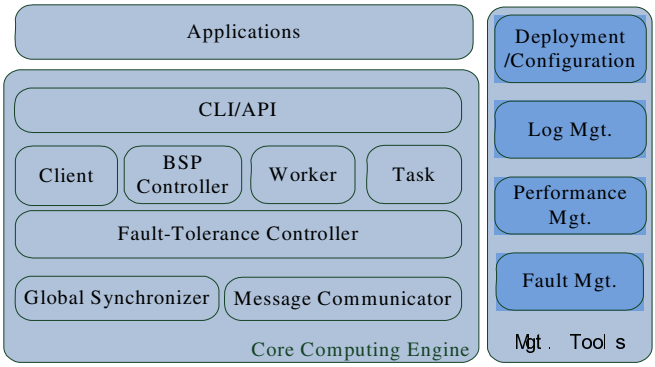

Fig. 2. The entire system framework of BC-BSP

The *Client* splits the input data according to the input path given by users, adjusts the number of partitions which is the processing unit of each task in a worker, asks the *BSPController* for the job ID, packs the job, and then submits the package to the *BSPController*. After the job begins, it is also responsible for reporting the running status in time. The *BSPController* manages the registration of the worker nodes in the computing cluster, the heartbeats from each worker, the status information of the cluster, and acts as a control centre of the fault-tolerance control. It also provides all the interfaces for the status query. It is responsible for scheduling, initialization, running monitor, and synchronization control of the jobs. The *Worker* manages the local jobs, local synchronization control, and local aggregation on a worker node. The *Task* is the entity that runs the jobs, and is responsible for inputting and outputting data and processing the local data. The *Global Synchronizer* manages the global synchronization among all the workers in each super-step by using *Zookeeper*, which is an open source middleware for a centralized service for providing distributed synchronization and et al. The global synchronization of a super-step is completed by the *BSPController*, the workers, and the tasks in cooperation. During the synchronization, the aggregation can be completed by invoking the aggregation function provided by users. The *Message Communicator* is responsible for sending and receiving messages, and for caching the messages received from other tasks to the local queue of received messages which can be saved into disks temporally when the main memory is not enough during the processing in every super-step. The *Fault-Tolerance Controller* detects faults, backups the snapshots for fault-tolerance, and recovers the system from failures. It uses the checkpoint mechanism for fault-tolerance. The *CLI/API* provides application program interfaces for local computation, sending or receiving messages, and etc. It also provides the commandline interface for the startup and shutdown of the system service, submitting the jobs, and manually specifying checkpoint and etc. The *Management Tools* uses the web interface or visual interface to provide users a method to manage the system.

The *Deployment/Configuration* tool provides users a visual interface for deploying system and configuring each node in the computing cluster. *Log (Fault) management* tool is used to check running logs (faults occurred during the running of a job) of the running jobs and the completed jobs. *Performance management* tool is used to monitor the system running status and the job running status.

Figure 3 shows the control mechanism of the BC-BSP's running. It shows the collaborative relationship among the *Client*, the *BSPController*, the *Workders*, the *Tasks* and the *Zookeeper*. Users interact with the BC-BSP system by *Client*, such as submitting jobs and monitoring the job running status. *BSPController* is the central nervous system of the entire BC-BSP system, and is responsible for controlling the whole cluster. The *WorkerManager* is the control center of a worker node, and manages the running and controlling of the worker node including collecting the information of all the tasks of a job on the worker node, communicating with the *BSPController* and with other workers. The *Task* is the work entity which performs the specific computing work. A job may have several tasks running on one worker, and these tasks are managed by the *WorkerAgentForJob* on this worker. The synchronization during the system running is controlled by the *Zookeeper*. One worker runs one *WorkerManager* process while there are several jobs running on it. Therefore, at the same time one *WorkerManager* may consist of several *WorkerAgentForJob* objects which manage all the tasks of a job on this worker.

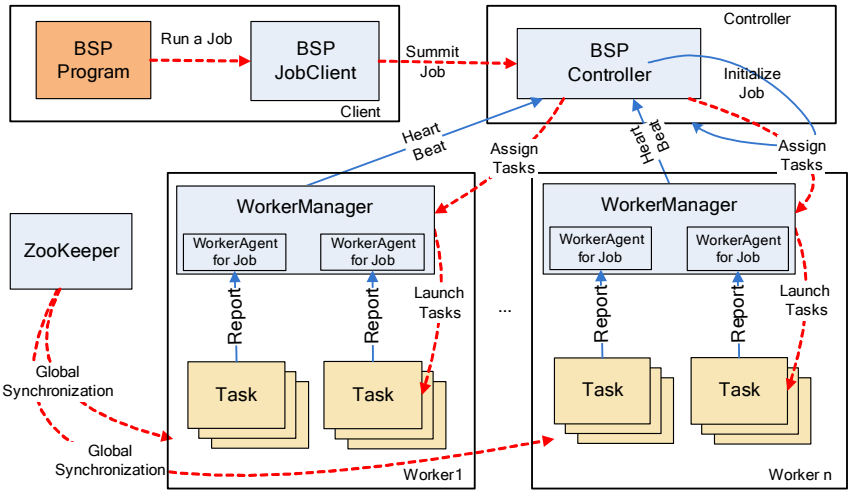

Fig. 3. The internal structure of BC-BSP

4 The Interfaces

BC-BSP system provides two types of interface, command-line interface and APIs (Application Program Interface). We mainly introduce the APIs.

When users write their application programs, they can invoke the APIs to extend BSP functions to meet their needs. For example, they can use *Combiners* interface to specify a combiner function for merging messages, use *Aggregators* interface to perform aggregation function, use *Partitioner* interface to control the partitioning of the input data, use *ContextInterface* interface to pass messages and execute local computation. The following is a brief introduction to these APIs.

VertexContextInterface: It is used to supply the context of a vertex for message passing and computing. At each super-step in a certain job, the system needs the related attributes of the vertex which is being processed, including vertex ID, value and current aggregation results, the update of vertex value and each edge value, the incoming and outgoing messages. So, these attributes and the operation methods on these attributes are encapsulated in this interface.

Combiners: During the graph processing, graph vertex is processed one by one. At each super-step, a vertex sends messages to its adjacent vertices and receives messages which are sent by other vertices at the same time. *Combiners* are used to merge messages at the sender side to reduce communication overhead. And different applications may require different combine function, for example, *sum*(), or *count*(). Therefore, users can specify their own combiner function to merge messages by extending this interface.

Aggregators: The graph processing needs aggregation in many cases, e.g., in order to examine whether the iteration should be stopped, *PageRank* application needs to aggregate the rank errors between the current super-step and last one. So, users can specify their own aggregators by implementing the *Aggregators* interface.

Partitioner: Before processing the data, the raw data should be assigned to each task by a certain principle. The default *Partitioner* provided by the system is hash function based on *hashCode*() method of Java. The *getPartitionID*() method in *Partitioner* interface maps a vertex ID into the corresponding partition ID. Users can override that method according to their own requirements.

Input and Output: The Input Interface is used to read the graph data from data source, e.g., *HDFS* or *HBase*. Therefore, *RecordReader* and *InputFormat* interface are provided for defining the input format like *Hadoop* and users can implement them to specify an input format to meet their needs. For example, the input format used to read data from *HBase* is implemented these interfaces. The output of the processed results should be output using the output format. Its definition is like to the input.

5 Implementation of BC-BSP

This section will introduce some implementation strategies and details of BC-BSP system. That includes the format of graph data, *BSPController* which is in the *Controller* node, *WorkerManager* which is the manager of a worker, *Task* which executes the *compute*() provided by user, passes messages, controls global synchronization and fault tolerance under the computing framework of BS-BSP system.

5.1 The Presentation of Graph

The system is mainly for processing large scale graph data, but it can also process structured data with the same data type elements. So, we mainly introduce the presentation structure of graph data. The graph is made up of vertex collection and edge collection, so there are *Vertex* class and *Edge* class to present the graph data. BC-BSP adopts the adjacent list to organize the graph data. In the *Vertex* class, there are some vertex attributes (such as vertex ID) and the information on outgoing edges. Meanwhile, it supports related methods to operate the member variables (see Fig. 4).

```
public class Vertex implements Writable {
    String vertexId = null; // vertex ID
    String vertexValue = null; // vertex value
    List<Edge> outEdgeList = new ArrayList<Edge>(); //store outgoing edge information
    public void addOutEdge(Edge outEdge) { }; // add an outgoing edge
    public List<Edge> getAllOutEdge() { }; // get all outgoing edges
    public void setAllOutEdge(List<Edge> aoutEdgeList) { }; // set all outgoing edges
    public boolean removeEdge(Edge edgeNode) { }; // remove an outgoing edge
    public boolean updateEdge(Edge edgeNode) { }; //update an edge value of the vertex
    public int hashCode() { }; // computing the hash code of a vertex
    public int getOutEdgeNumber() { }; // get # of outgoing edges from the vertex
    ……
   }
```

Fig. 4. The structure of graph vertex class

The above structure of graph vertex class is suitable for graph data. However, it also can describe structured data by converting them to fit the above structure. For example, we can treat a record of the raw structured data as the vertex value string. Therefore, each record of raw structured data as a graph vertex.

5.2 BSPController Implementation

The *Controller* node is the center of the whole BC-BSP cluster. From the hardware perspective, it is responsible for managing all the worker nodes; from the software point of view, it is responsible for monitoring the working status of the computing cluster, receiving heartbeat information from each worker and process it, controlling the global synchronization among workers for each job. When the cluster starts, the *Controller* node receives registration information from each node to form unified cluster resource information. During the course of normally working, *Controller* collects and updated the cluster resource information (such as the number of free task slots) by the heartbeat mechanism periodically. When a user requests to submit jobs, the *Controller* assigns a unique job ID to the job, and then generates a job control object and puts it into the job waiting queue. The job scheduler selects high priority jobs to run in accordance with the priority and FIFO strategy. Then the

task scheduler assigns tasks to workers in accordance with the principle of load balance and data locality. Because all the tasks for a job need to run at the same time, the task scheduler pushes down the tasks to each worker node, not like the scheduling way in *Hadoop* ecosystem, in which worker node applies a task to run when it has free task slots.

5.3 Worker Manager

A worker node is a unit of computation. After starting of BC-BSP, each worker node of the BC-BSP cluster will create a *WorkerManager* process which manages the tasks and maintains the synchronization among them. As soon as the startup of each *WorkerManager*, it should register itself to the *BSPController* to join the BC-BSP computing cluster. During their life time, they send heart-beat signals to the *BSPController* to report their status, respectively. When a new task arrives at a worker, *WorkerManager* reads and unpackes job profiles from *HDFS* into the local file system, and creates a *TaskinProgress* object and task process, and finally starts the task process. *WorkerManager* will build a *WorkerAgent* object for the tasks, which are running on the same node, to collect the status information of the task for a job. In this case, two levels of synchronizations are needed. The low level synchronization is for all tasks belong to the same job on the same worker node to synchronize, and then the high level is to register to *ZooKeeper* by worker as a whole for synchronization. So, the two synchronization levels decline the amount of client-ends keeping connection with *ZooKeeper* server and then decrease the workload of *ZooKeeper*. *WorkerManager* manages the tasks belong to the same job as a whole, so that it can manipulate some local computing, like the aggregation of local results computed by local tasks.

5.4 Task and Message Passing with Disk Assistant

A task is a logic computing unit. Task scheduler in *BSPController* assigns tasks to worker nodes according to load balance and data locality, and then *WorkerManager* on worker node creates task processes. After task starting, it first loads data processed by it, that is, it reads data from storage media according to the specified input format and then partitions the data into different partitions. During the course of partitioning, some graph vertex data need to be transferred to other tasks. After the completion of data partitioning, a global synchronization is needed to wait for all tasks belonging to a job to finish data partitioning. Then, tasks can go into BSP's super-steps to process graph data iteratively, that is, local computing, message passing and global synchronization. During the computing, task may send heart-beat information periodically to *WorkerAgent* object in *WorkerManager* process to report its current status.

Pregel system and its different implements all suppose that there are enough worker nodes and resources in the cluster to hold all graph data processed in a task and the related intermediate data (such as messages) during the course of each super-step in main memory completely. But actually, this assumption doesn't hold. There are two-folder reasons. The first folder is that it is difficult for a user to determine how many workers to be used and whether there are enough main memory to hold the graph data

and the messages for a given dataset. The second is that the system can handle relative large-scale data under the limitation of the cluster scale.

For the above reasons, BC-BSP system applies disk space to store some graph data and intermediate messages temporally in order to process relative large-scale data. BC-BSP divides the JVM heap space into three parts. They are the spaces used by temporary defined objects, the spaces for storing graph data objects, and the spaces for messages. The space percentages occupied by the three parts are α, β and γ, respectively. The sum of these parameters is equal to 1. So, user only needs to give any two ones, and their values can be given in configuration file according to the real situation of the processed data.

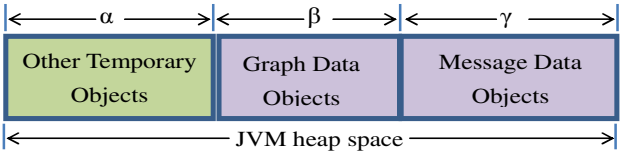

Fig. 5. The management model for JVM heap space

In order to avoid the overflow of memory, the data should be spilled into disk when the memory occupied by the data exceeds the given threshold. In our system, no matter graph data or message data both are swapped applying hash bucket techniques. In this case, the hash bucket for graph data objects and the one for message data objects have one to one relationship. Therefore, the super-step computing of a task can process graph data objects in hash bucket one by one, at the same time the message objects are all in a hash bucket related to the hash bucket holding graph data objects. Therefore, the system can quickly match the graph data object with the messages sent to it.

For message objects, each task maintains three queues. The one is *IncomedQueue* for managing the messages which are sent from the last super-step, processed in the current super-step, and is kept in memory as far as possible. The second one is *IncomingQueue* for managing the messages which are sent from other vertexes in the current super-step, will be processed in the next super-step, and has the highest priority to be spilled into disk. The third one is *OutgoingQueue* for managing the messages which are produced during the computing, is never spilled into disk. Messages in this queue can be combined by invoking the *combine*() method defined by user when the length of the queue exceeds a given threshold, and will be sent to other tasks. Practically, these message queues can be organized by hashmap.

5.5 Data Partitioning

Each task calls data partitioning function to read the binding data split from a specified data source, and uses the data partitioning strategies, such as hash partitioning, to allocate the graph data to a partition which is processed by a task. The system provides *hashCode*() method of Java as a default hash function, and also provides an interface for users to define their own hash functions to meet their special partitioning

requirements. The input of the hash function is the value of a vertex ID, which can be an integer or a string; the output of the hash function is the partition ID. Therefore, a map table between PartitionID and Worker can be established to record a partition on which worker node. We can use MD5 method, which is a widely used hash algorithm for getting digest information of the input string, to partition graph data in order to get balanced distribution. But in practice, we find that its time consuming is very large. There are two reasons. The first is the MD5 computing of an input string needs more time than that of *hashCode*() method, the second is that the MD5 value should be computed for each graph vertex frequently during each super-step.

Whether the size of each data partition is equalized approximately will make a direct impact on the system load balance and the performance. It is known that hash map is difficult to ensure the equilibrium of each partition. To this end, we use the division method of multi-hash buckets merge to achieve the load balance. The basic idea is that assuming that we need to get n partitions, first we divide input data into $k*n$ buckets ($k \geq 1$), then send the number of objects in each bucket to *BSPController*, who merges $k*n$ buckets into n buckets according to load-balance principles. The merge principles can make the data objects in each bucket as possible as balance; it can also consider the data locality.

6 Application Examples

We design and implement several applications using the *API*s supplied by the system, such as *PageRank*, *SSSP*(Single Source Shortest Path), and *K-means* on non-Graph structured data. But only the *PageRank* algorithm based on BSP is discussed on detail and K-means algorithm is discussed briefly because of the space limitation.

6.1 PageRank

Fig. 6 describes the implementation of *PageRank* algorithm based on BS-BSP platform. *PageRank* algorithm needs send the current rank value of the vertex to its adjacent vertexes which are linked by the current vertex by some rules (such as equal allocation) as the contribution value to the adjacent vertex. According to the *PageRank* algorithm, the messages send to the same vertex can be merged by the way of summary. Therefore, we design and implement a *combine*() method by overriding the *combine*() method of *Combiner* interface class (see in Fig. 7).

```
import com.chinamobile.bcbsp.*;
public class PageRankBSP extends BSP {
    …… // omitted some variable definition
    @Override
    public void compute(Iterator<BSPMessage> messages, BSPStaffContextInterface
                        context) throws Exception {
        /* Receive messages sent to this vertex */
        receivedMsgValue = 0.0;
```

```
        receivedMsgSum = 0.0;
        while (messages.hasNext()) {
              receivedMsgSum+=Double.parseDouble(new
                                  String(messages.next().getData())); }
        /* Process received messages and Update vertex value */
        if (context.getCurrentSuper - stepCounter() == 0) {
           sendMsgValue = Double.valueOf(context.getVertexValue())
                                  /context.getOutgoingEdgesNum();
        } else {
              /* According to the sum of error to judge the convergence */
             errorValue=(ErrorAggValue)context.getAggregateValue(ERROR_SUM);
             if (Double.parseDouble(errorValue.getValue())< ERROR_THRSHLD) {
                    context.voltToHalt();   // This vertex can halt
                    return; }
            /*Compute new vertex rank value and the contribution to adjacent vertex*/
            newVertexValue=CLICK_RP*FACTOR+receivedMsgSum*(1- ACTOR);
             sendMsgValue = newVertexValue / context.getOutgoingEdgesNum();
             context.updateVertexValue(String.valueOf(newVertexValue)); }
        /* Send new messages */
        outgoingEdges = context.getOutgoingEdges();
        while (outgoingEdges.hasNext()) {
                EdgeNode = outgoingEdges.next();
                msg = new BSPMessage(Integer.parseInt(EdgeNode.getVertexID()),
                                          Double.toString(sendMsgValue).getBytes());
                context.send(msg); }
        return; }
}
```

Fig. 6. PageRank algorithm based on BS-BSP platform

```
   public class SumCombiner extends Combiner {
          public BSPMessage combine(Iterator<BSPMessage> messages) {
                BSPMessage msg;
                double sum = 0.0;
                do {
                      msg = messages.next();
                      String tmpValue = new String(msg.getData());
                      sum = sum + Double.parseDouble(tmpValue);
                } while (messages.hasNext());
                String newData = Double.toString(sum);
               msg = new BSPMessage(msg.getDstPartition(), msg.getDstVertexID(),
                           newData.getBytes());
                return msg; }
   }
```

Fig. 7. The *combiner* class for *PageRank* algorithm on BC-BSP platform

6.2 K-Means on Metric Data

In this subsection, we describe the basic idea about k-means clustering on ordinary multi-dimensional metric dataset on BC-BSP platform. Because the data structure of BC-BSP is designed for processing graph data, the multi-dimensional metric data must be converted in order to fit the input requirement of BC-BSP, but it is easy to do. So, we convert i^{th} data point (i.e. i^{th} line in data file) $<d_1, d_2, d_3,\ldots, d_n>$ into the following format: *i:tagvalue* <*tab*> $1{:}d_1\ 2{:}d_2\ \ldots\ n{:}d_n$. Where, the first part <*i:tagvalue*> is regarded as a graph vertex, *i* is the line number of a record in the data file, and *tagvalue* can be used as the cluster tag that the data record belongs to and its initial value can be a random integer value. The second part <*tab*> stands for *tab* key, maybe 4 or 8 blank-spaces. The others are the outgoing-edge-list constructed from the each dimensional value of the data record separated by a blank-space, the part before semicolon, such as 1, 2, stands for the outgoing-edge vertex ID and isn't used in the computing procedure; the part after semicolon, such as d_n, stands for the n^{th} dimensional value d_n of the data point. The stop criterion may be the error of a cluster center point between the adjacent two super-steps is less than a given threshold. Therefore, user must design an *aggregator* to compute the error. But user need not write *Combiner* to combine messages since no message need be sent between each task in the adjacent two super-steps. The program for k-means algorithm on BC-BSP platform is omitted because the space limitation.

7 Experiments

Some experiments are done to evaluate the performance and extendibility of our system under different circumstances. Because Google's *Pregel* does not open source, we do not compare with it. We compare *PageRank* algorithm implemented on BC-BSP platform with the one based on MapReduce framework, and the ones on *Hama* and *Giraph*.

The hardware environments for the experiments are IBM shared-nothing cluster linked by Gigabit Ethernet, in which each node has 2 hyper-threaded 2.00 GHz Intel Xeon CPUs, 2GB memory, 73GB and 7200rpm hard disk. The experiments are done on Linux Redhat V5.6 and JDK1.6 for Linux.

7.1 Comparison between BC-BSP and MapReduce

Two kinds of datasets, real world data and synthetic data, are used in the experiments. The features of the datasets are listed in Table 1.

We use 9 nodes of the cluster to run our experiments, and setup the JVM memory to 1GB. The raw data are assigned to each node near-equally under BC-BSP platform. Under MapReduce framework, the number of *Mappers* is determined by *Hadoop* framework according to the data block size, and the number of *Reducers* is setup as 9 (i.e. 9 nodes). Based on the above environment and configuration, the performance comparison between the *PageRank* algorithm based on BC-BSP and MapReduce on real world data is in Fig. 8, and on synthetic data in Fig. 9.

The experiment results show that the performance of *PageRank* algorithm on BC-BSP platform is better than that on *MapReduce* framework when the data including graph data and the messages sent to other workers can be stored on the memory during the course of computing. While when the volume of data is relative large, where data including graph data and messages, the response time for *PageRank* algorithm on BC-BSP platform is large greatly than that on *MapReduce* because the former needs to use disk space as spilled space.

Table 1. The features of datasets for experiments

DatasetType	Dataset Name	#Vertex	#Edges	Data file size
Real world data	Wikipedia Talk network	2394385	5021410	45.4MB
	Autonomous system by Skitter	1,696,415	11095298	116MB
	Patent citation network	3,774,768	16,518,948	203 MB
	Live Journal social network	4,847,571	68,993,773	700 MB
Synthetic data	Syth1	10,000	676,640	4.5MB
	Syth2	50,000	7,543,140	56MB
	Syth3	100,000	21,136,232	160MB
	Syth4	4,000,000	53,991,808	685MB

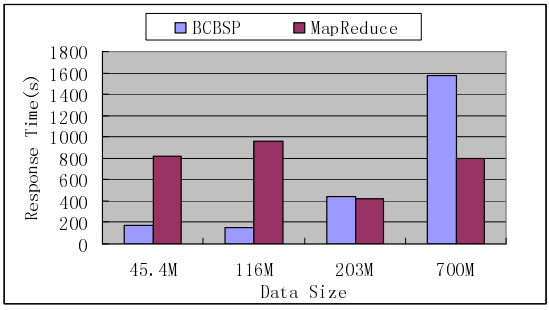

Fig. 8. Comparison between BCBSP and MapReduce on real dataset

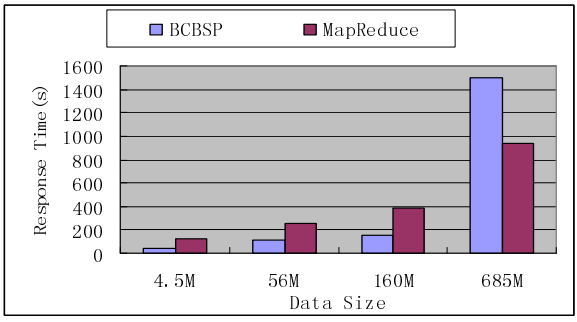

Fig. 9. Comparison between BCBSP and MapReduce on synthetic dataset

7.2 Comparison among BCBSP, Hama and Giraph

In order to test the processing ability among BC-BSP, Hama without disk help, and Giraph, we design an experiment to run *PageRank* algorithms on synthetic dataset. Because the difference of the processing capability of the three BSP-based system and the difference of their expressions on data, we generate the synthetic datasets with the same numbers of vertexes and edges, respectively. the experiments are executed on 1G JVM. The experimental results are shown in Fig. 10.

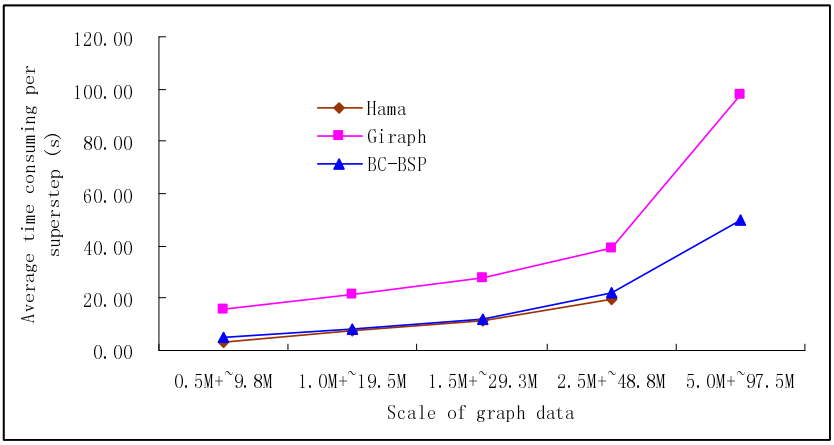

Fig. 10. Comparison among BCBSP, Hama and Giraph on another synthetic dataset, where 0.5M+~9.8M stands for the data set with 500000 vertexes and about 9800000 edges

By now, from Fig. 10, we can find that *Hama* is faster than BC-BSP a little more when the data scale is small, but when the data scale exceeds a certain value, such as 2.5M vertexes, the jobs on *Hama* platform can not execute because of memory overflow, the average response time for a super-step on BC-BSP is faster some degree than that on *Giraph* on each test data set. We can also find that applications run on BC-BSP are faster than that on *Giraph*.

8 Conclusions and Discussion

This paper describes the system BC-BSP for large-scale graph processing based on BSP model under Java environment. The system implements the main functions mentioned in *Pregel*, and adds some optimized strategies to improve and enhance the performance of the system. It implements the balanced partitioning strategy in data partitioning stage in order to make each task have the approximately equal graph nodes to process. It implements disk swap function temporally to store graph data and the messages when they can not be hold in main memory to make the system can handle large-scale graph under constraint of computing and storage resources. We do many experiments to evaluate the performance of the system. We can conclude that the BSP-based applications have

higher efficiency than that of *MapReduce*-based applications when the volume of data is relative not very large and it can be held in the memory during the course of processing; on the contrary the latter are better than the former. But our system can handle relative large-scale graph data when the computing and storage resources are limitation because it applies disk assistant mechanism.

Although we have done many efforts to optimize the system, there are many aspects to optimize and improve the system. For instance, a) we can optimize and improve the data structure for storage and presentation of graph data using template technique of Java to enhance the flexibility and to save the storage consumption; b) we should consider the locality and relevance of data at the data partitioning stage except considering the balance of each task.; c) we could enhance the capture and detection of every kinds of faults and handle each kind of faults using different policy; d) we could improve the programming skills to decrease memory consumption and increase the system performance. We can use object-pool technique to cache some kinds of object to decrease the CPU consumption on object constructions and the main memory occupation.

Acknowledgement. This work is partially supported by the Key National Natural Science Foundation of China under Grant No 61033007, the National Natural Science Foundation of China under Grant No. 61173028, and the joint Foundation of Ministry of Education of China and China Mobile under Grant No. MCM20125021.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of 6th USENIX Symp. on Operating Syst. Design and Impl., pp. 137–150 (2004)
2. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems 30(1-7) (1998)
3. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A System for Large-Scale Graph Processing. SIGMOD (2010)
4. Welcome to Hama Project, `http://incubator.apache.org/hama/`
5. Snoek, J.: Computing PageRank using MapReduce. Technical Report, Report No. CSC2544. University of Toronto, Toronto (2008)
6. Ching, A., Kunz, C.: Giraph: Large-scale graph processing infrastructure on Hadoop, Hadoop Summit (2011)