BC-BSP: A BSP-Based System with Disk Cache for Large-Scale Graph Processing

Yubin Bao, Zhigang Wang, Qiushi Bai, Yu Gu, Ge Yu
School of Info. Sci and Eng.
Northeastern University
Shenyang, China
{baoyubin, yuge }@ise.neu.edu.cn

Hongxu Zhang
Software Division
Neusoft Corp.
Shenyang, China
kevinzhang@neusoft.com

Chao Deng, Leitao Guo
China Mobile Institute
China Mobile Corp.
Beijing, China
dengchao@chinamobile.com

***Abstract*—Many applications in real life can be modeled by Graph, and the data scale is very large in many fields. People have paid more attention to large-scale graph processing. A BSP-based system with disk cache for large-scale graph processing is proposed in this paper. The system has the ability to expand the 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 balance load, and to run clustering or classification algorithms on metric datasets. Some experiments are done to evaluate the scalability 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 the MapReduce-based applications when the volume of data can be put in the memory during the course of processing; on the contrary the latter is better than the former.**

Keywords- BSP; MapReduce; Large-Scale Graph Processing; Disk Cache; Big data

I. 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 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 is widely used to model real application. We can use graph to model the above-mentioned cases and then analyze them deeply. In spite of the theory and algorithms on graph have been researched in depth during the past decades, most of them aim at small-scale datasets. With the development of the information technology, the scale of all kinds of information keeps increasing rapidly, which leads to the scale of graphs becomes bigger and bigger. For examples, at present, more and more people begin use the Internet. With the help of Web2.0, the number of web pages has increased rapidly. According to the statistics of CNNIC, there are 60 billion web pages in China and the rate of increasing is 78.6%[1]. The situation may be even worse 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 the importance of web pages by related algorithms. The most famous one is *PageRank* algorithm. We can define a web page as a node in the 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, 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 large-scale graph has already beyond the ability of concentrated computing traditionally. In conclusion, it has become a new challenge to process large scale graph efficiently.

At present, *MapReduce* computing model based on *Hadoop* ecosystem can process large-scale graph data with the better fault-tolerance and scalability. While, most graph algorithms need to process graph data many times iteratively. We must start one or more jobs to complete one iterative step. As we known, the cost of the warm-up start of one *MapReduce* job is considerable. In order to solve this problem, Google developed a 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 is resident in memory without disk cache. Apparently, if the number of workers is limited, the scalability is also limited. *Hama* is an open source project of *Apache*[3]. It is also good at processing big data iteratively, especially at processing matrix. But *Hama* does not consider the disk cache too. *Giraph*[4] developed by Yahoo implements the BSP model on the *Hadoop* framework. Simply, an application on *Giraph* is a special *MapReduce* job without reduce task. It designs an inbuilt loop in the map task to simulate the super-steps of the BSP model.

In this paper, we design a system BC-BSP, different from above instances, which is good at processing large scale graph data. The features of BC-BSP are as follows: 1) BC-BSP implements the BSP model and considers the disk cache. So we can handle relative larger scale graph data if the available resources are limited. 2) It provides flexible configuration and scalability. User can choose or define the format of input and output. BC-BSP supports several data formats, such as distributed file system and key-value databases. User can define the special data format by related interface. BC-BSP also supports several strategies to partition the raw data. BC-BSP supplies hash partition, local partition and user-defined

978-0-7695-4908-8/12 $26.00 © 2012 IEEE
DOI 10.1109/OCS.2012.37

partition. 3) 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 among workers. Especially, the load balance among workers is more prior than the data locality. 4) Some experiments are done to compare and evaluate the performance and scalability between the applications based on BC-BSP platform and *MapReduce* model.

II. INTRODUCTION TO BSP MODEL

BSP (Bulk Synchronous Parallel) is a "bulk" synchronous model[5]. There is a concentrated 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 can be solved by a series of sequential super-steps. In each super-step, the tasks on the cluster workers are asynchronous parallel running, and they can send messages to other tasks for the requirements of the computing job. The next super-step can start until the computing of each worker ends, and messages that need to be sent to other tasks and to be received from other tasks are processed completely. It is called barrier synchronization.

III. OVERVIEW OF BC-BSP

The entire framework of the BC-BSP system consists of the inner core computing engine and the outer management tools. The core computing engine consists of the *Client*, the *BSPController*, the *Worker*, the *Task*, the *Global Synchronizer*, the *Message Communicator,* the *Fault-Tolerance Controller*, and API/CLI. The *management tools* consist of the cluster management, the automatic deployment and configuration tool, the performance management, and the fault management.

The *Client* splits the input data, adjusts the number of partitions, asks *BSPController* for the job ID, packs the job, and then submits the package to *BSPController*. After the job starts, it is also responsible for reporting the running status in time. *BSPController* manages the registration of the worker nodes in the computing cluster, the collecting of the heartbeats, the status information of the cluster, and acts as a control center of the fault-tolerance control. It also provides the interface for the status query. It is responsible for scheduling, initialization, running, and synchronization control of the jobs on the job level. The *Worker* manages the local tasks, local aggregation, and local synchronization control. The *Task* is the entity that runs the jobs, and is responsible for input and output the data, and processing the local data by invoking the *compute*() method provided by user. The *Global Synchronizer* manages the global synchronization among all the workers in each super-step. The global synchronization of the 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 user. 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 when the memory is not enough to hold them during the local computation of each super-step. *The Fault-Tolerance Controller* detects faults, backups the snapshots of graph data for fault-tolerance, and recovers failures. It uses the checkpoint mechanism for fault-tolerance. The management tools use the web interface to provide users a method of visual system management. The CLI/API provides users application program interfaces for local computation, sending or receiving messages. It also provides users the command-line interface for the startup and the shutdown of the system service, and submitting the jobs.

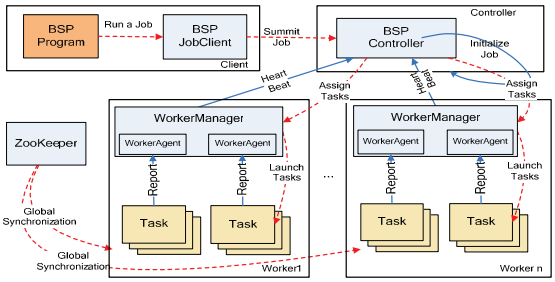

Figure 1. The structure of BC-BSP and the relationships among the components

Figure 1 shows the control mechanism of each component of the BC-BSP during its running. It shows the collaborative relationship among the *Client*, the *BSPController*, the *Workers*, the *Tasks* and the *Zookeeper* which is used for global synchronization. Users interact with the BC-BSP system by *Client*, such as submitting jobs and monitoring the job's running status. *BSPController* is the central nervous system of the entire BC-BSP system. The *WorkerManager* is the control center of the worker node, and manages the worker node, such as collecting the information of all the tasks of a job on the worker node and communicating with the *BSPController* and with other *Workers*. It sends heartbeat information to *BSPController* to report the status about the worker and the tasks on the worker. A job may have several tasks running on one *Worker*, and these tasks are managed by the *WorkerAgent* on this *worker*. The synchronization during the system running is controlled by the *Zookeeper*. One *worker* runs one *WorkerManager* process while it also has several jobs. Therefore, at the same time one *WorkerManager* may consist of several *WorkerAgent* objects.

IV. THE APIS

The system provides some APIs to user for extending BSP functions to fit their special needs, such as different graph processing or scientific computation. For example, using *Combiners* interface to specify a combiner function for merging messages, using *Aggregators* interface to perform aggregation function, using *Partitioner* interface to control the partitioning of the input data, using *VertexContextInterface* 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's value and edges' value, the incoming messages 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 functions, for example, sum, and count. Therefore, users can specify their own combiner to merge messages.

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

Partitionar: Before processing the graph, the graph data should be assigned to each task by a certain principle. The default *Partitioner* provided by the system is hash function based on MD5. The *getPartitionID*() method in *Partitioner* interface map a vertex ID into the corresponding partition ID. Users can override that method according to their own needs.

Input and Output: The Input interface aims to read the graph data from data source, e.g., *HDFS* or *HBase*. Therefore, the *RecordReader* and *InputFormat* interfaces are provided for defining the input format and users can implement them to specify the input format to meet their own 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.

V. IMPLEMENTATION OF THE SYSTEM

This section introduces some strategies and details of the BC-BSP system, such as the format of graph data, the implementation of *BSPController*, *WorkerManager*, and task and message passing with disk cache.

A. The Presentation of Graph

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 vertex value) and the information on outgoing edges. Meanwhile, it supports related methods to operate these objects.

B. BSPController Implementation

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

C. WorkerManager Implementation

A worker node is a computation unit. After each worker starting, a *WorkerManager* process is created on every node. It manages the tasks and maintains the synchronization among them. Then *WorkerManager*s should register to *BSPController* to join the BSP cluster. During their life time, they send heartbeats to *BSPController* to report their status, respectively. When a new task arrives at a worker, *WorkerManager* reads and unpacks job profiles from *HDFS* into the local file system, creates task process, and finally starts the task process. Then, *WorkerManager* creates a *WorkerAgent* object for managing the tasks belonging to the same job. In this case, two levels of global synchronization are needed. The low level one is for all tasks belonging to the same job on the same worker to synchronize, and then the high level one is to register to *ZooKeeper* by worker as a whole for synchronization. So, the two levels of synchronization decline the amount of client-ends keeping connection with *ZooKeeper* server and then decrease the *ZooKeeper* workload. Because *WorkerManager* manages the tasks belonging to the same job as a whole, it can do some local computing tasks, like the aggregation of local results computed by local tasks.

D. Task and Message Passing with Disk Cache

A task is a logic computing unit. Task scheduler in *BSPController* assigns tasks to worker nodes according to load balance and data locality and *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 passed to other tasks by the way of message passing. After the completion of data partitioning, a global synchronization is needed to wait for all tasks belonging to a job to complete 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, tasks may send heartbeat 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 in memory completely. But in fact, this assumption doesn't hold. There are two folder reasons. The first is that it is difficult for users to determine how many workers to be used and whether the graph data can be held in main memory for a given dataset. The second is that the system can handle relative large-scale data under the

limitation of the cluster scale. These cases also exist for the messages.

For the above reasons, BC-BSP system applies disk cache mechanism 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 storing 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 the real values of the three parameters can be given according to the real situation of the processed data.

In order to avoid the overflow of memory, the graph data and messages should be cache 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 cached applying hash bucket technique. In this case, the hash buckets for graph data objects and the ones for message data objects have one to one relationship. Therefore, the super-step computing of a task can process graph data objects in hash buckets 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 objects 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 previous super-step, processed in the current super-step, and kept in memory as far as possible. The second one is *IncomingQueue* for managing the messages which are received from other vertices in the current super-step, will be processed in the next super-step, and has the highest priority to be cached into disk. The third one is *OutgoingQueue* for managing the messages which are produced during the computing, are never cached into disk, will 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.

E. Data Partitioning Implementation

Each task calls data partitioning function to read the binding data splits from a specified data source, and then uses the data partitioning principles, such as hash partitioning, to allocate the graph data to a partition which is processed by a task. The system provides MD5 method 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 (namely *PartitionID*). Therefore, a map table between *PartitionID* and *Worker* should be established to record a partition on which worker. The system can know a vertex by its *vertexID* in which worker and which partition by looking at the map table.

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

VI. Applications

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

A. PageRank

PageRank algorithm needs to send the current rank value of the vertex to its adjacent vertices 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 implement a *combine*() method by overriding the *combine*() method of *Combiner* interface. The programs for *PageRank* algorithm on BC-BSP platform are omitted because the space limitation.

B. K-means on metric data

In this subsection, we describe the basic idea about *K*-means clustering on 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. We convert i^{th} data point (i.e. i^{th} line in data file) $<d_1, d_2, d_3, ..., d_n>$ into the following format: < *i:tagvalue* <tab> $1{:}d_1$ $2{:}d_2$ $\ldots n{:}d_n.>$. The stop criterion may be the error of a cluster center point between the two adjacent 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 two adjacent super-steps. The programs for k-means algorithm are omitted because the space limitation.

VII. Experiments

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

The hardware environments for the experiments are shared-nothing cluster linked by Gigabit Ethernet, in which each node has 2 hyperthreaded 2.00GHz Intel Xeon CPUs, 2GB memory, 73GB and 7200rpm hard disk. The experiments are done on Linux Redhat 5.6, and JDK1.6 for Linux. Two kinds of datasets, real world data and synthetic data, are used in the experiments. The features of the datasets are listed in Tab. 1.

TABLE I. THEFEATURES OF DATASETS FOR EXPERIMENTS

Data set	*Dataset Name*	*#Vertex*	*#Edges*	*Size of data file*
Real world data	Wiki. Talk network	2394385	5021410	45.4MB
	Berkeley Stanford web graph.	685,230	7,600,595	76.8MB
	Autonomous systems by Skitter	1,696,415	11095298	116MB
	Patent citation networks	3,774,768	16,518,948	203 MB
	Live Journal SN.	4,847,571	68,993,773	705 MB
Synth. data	Syth1	10,000	676,640	4.55MB
	Syth2	50,000	7,543,140	56.51MB
	Syth3	100,000	21,136,232	160.15MB
	Syth4	4,000,000	53,991,808	685.13MB

We use 10 nodes of the cluster to run our experiments, one acts as *Controller* and the other 9 notes are workers, and setup the JVM memory to 1 GB. The raw data are assigned to each node equally. Under *MapReduce* framework, the number of *Mappers* is determined by *Hadoop* according to the data block size, and the number of *Reducers* is setup as 9 (i.e. 9 nodes). Based on the above setting, the performance comparison between the *PageRank* algorithm based on BC-BSP and *MapReduce* on real world data is in Fig. 2, and on synthetic data in Fig. 3.

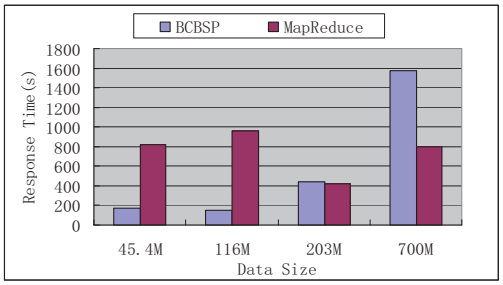

Figure 2. Comparison between BC-BSP and MapReduce on real dataset

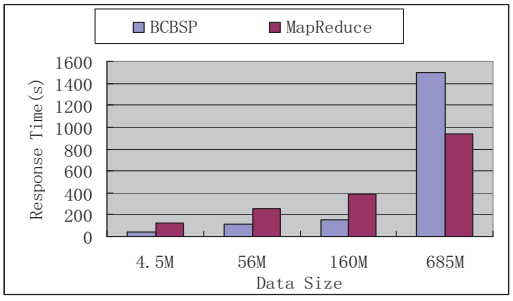

Figure 3. Comparison between BC-BSP and MapReduce on synthetic dataset

The experiment results show that the performance of *PageRank* algorithm on BC-BSP platform is better than that on *MapReduce* 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 are relative large, that is, 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 cache.

VIII. CONCLUSIONS AND DISCUSSION

This paper describes the system BC-BSP for large-scale graph processing based on BSP model. 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 to make each task have the approximately equal graph nodes to process. It implements disk cache to cache graph data and the messages to other tasks 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 the *MapReduce*-based ones 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 is 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 cache mechanism.

Although we have done many efforts to optimize the system, there are many aspects to optimize and improve the system. For instance, 1) we can optimize and improve the data structure for storage and presentation of graph data using template technique; 2) we should consider the locality and relevance of data at the data partitioning stage except considering the balance of each task..

ACKNOWLEDGE

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 Fundamental Research Funds for the Central Universities under Grant No. N100704001.

REFERENCES

[1] S. Brin, L. Page, “The anatomy of a large-scale hypertextual web search engine”, Computer Networks and ISDN Systems, 1998, 30, pp.1-7.

[2] G. Malewicz, M. H. Austern, A. J.C.Bik, et al. “Pregel: A System for Large-Scale Graph Processing”, SIGMOD, 2010, pp 135-145.

[3] Welcome to Hama Project, http://incubator.apache.org/ hama/, 2011-7-13

[4] A Ching, C Kunz. “Giraph: Large-scalegraph processing infrastructure on Hadoop”, Hadoop Summit 2011, 6/29/2011, https://github.com/aching/Giraph.

[5] L. G Valiant, “A Bridging Model for Parallel Computation”, Communications of the ACM, Aug., 1990, Vol. 33, No. 8, pp. 103-111.