View metadata, citation and similar papers at core.ac.uk brought to you by CORE
provided by Agora University Editing House: Journals

INT J COMPUT COMMUN, ISSN 1841-9836
Vol.7 (2012), No. 3 (September), pp. 432-449

Performing MapReduce on Data Centers with Hierarchical Structures

Z. Ding, D. Guo, X. Chen, X. Luo

Zeliu Ding, Deke Guo, Xueshan Luo
Key Lab of Information System Engineering,
School of Information Systems and Management,
National University of Defense Technology,
Changsha 410073, China
E-mail: zeliuding@nudt.edu.cn, guodeke@gmail.com
xsluo@nudt.edu.cn

Xi Chen
School of Computer Science, McGill University,
Montreal H3A 2A7, Canada
E-mail: chenxiwarm@gmail.com

Abstract:
Data centers are created as distributed information systems for massive data storage and processing. The structure of a data center determines the way that its inner servers, links and switches are interconnected. Several hierarchical structures have been proposed to improve the topological performance of data centers. By using recursively defined topologies, these novel structures can well support general applications and services with high scalability and reliability. However, these structures ignore the details of some specific applications running on data centers, such as MapReduce, a well-known distributed data processing application. The communication and control mechanisms for performing MapReduce on the traditional structure cannot be employed on the hierarchical structures.
In this paper, we propose a methodology for performing MapReduce on data centers with hierarchical structures. Our methodology is based on the distributed hash table (DHT), an efficient data retrieval approach on distributed systems. We utilize the advantages of DHT, including decentralization, fault tolerance and scalability, to address the main problems that face hierarchical data centers in supporting MapReduce. Comprehensive evaluation demonstrates the feasibility and excellent performance of our methodology.
Keywords: MapReduce; Data Center; distributed hash table (DHT).

1 Introduction

In the recent years, the data centers have emerged as distributed information systems for massive data storage and processing. A data center provides many online applications [8] [2] and infrastructure services [4] [13] through its large number of servers, which are interconnected via high-speed links and switches. These devices construct the networking infrastructure of a data center, named data center network [9]. The structure of a data center network determines the way these devices are organized. To design suitable structures and make them match well with the data storage and processing applications are fundamental challenges.

MapReduce, proposed by Google, is a well-known and widely used data processing mechanism on data centers [6]. MapReduce works by separating a complex computation into Map tasks and Reduce tasks, which are performed in parallel on hundreds or thousands of servers in a data center. MapReduce provides a good control and execution mode for distributed computing and cloud computing [18]. Users

Copyright © 2006-2012 by CCC Publications

without any experience of distributed programming can easily process terabytes of data on data centers with the help of MapReduce.

Nowadays, increasingly diverse applications and services call for an improvement of data centers in the topological performance, including scalability, reliability, etc. Especially, users' various requirements for processing large amount of data results in an exponentially increasing number of servers. The traditional structure, however, can hardly sustain the incremental expanding of data centers [10]. Several novel data center structures, such as DCell [9], FiConn [15], and BCube [11], have been proposed to optimize the topological performance of data centers. These structures are all recursively defined to construct data center networks, interconnecting the servers by a hierarchical way. We represent them as hierarchical structures. These structures mainly focus on the scalability and reliability for data centers. However, the details of some specific applications running on the data centers with these structures are ignored. For example, none of these hierarchical structures treat servers as masters and workers, although this requirement is the basis of many distributed data processing applications, especially MapReduce. The communication and control mechanisms for performing MapReduce on the traditional structure can hardly be operated on these hierarchical structures.

To solve this problem, this paper presents a methodology for performing MapReduce on the data centers with hierarchical structures, represented as hierarchical data centers for short. Our methodology is based on the distributed hash table (DHT) [20] [21], an efficient data retrieval approach on distributed systems. DHT works by assigning each server a hash table that records the range of keys handled by all adjacent servers. Responsibility for hashing data to keys is distributed among the servers. This approach possesses several advantages. First, DHT makes all servers freely communicate with each other without any central coordination. This provides a control mechanism for MapReduce on hierarchical data centers without designating the masters or workers. Second, DHT ensures that the whole system can tolerate any single node failure. Since the information of a server is held by its adjacent servers, a failed server can affect only its neighbors, which causes a minimal amount of disruption [22]. Finally, DHT can flexibly deal with a large amount of nodes joining or leaving the system. This matches well with the scalability of hierarchical data centers. These advantages bring a feasibility to perform MapReduce on hierarchical data centers.

In this paper, we address the main problems that face hierarchical data centers in supporting MapReduce. Our methodology utilizes the above advantages of DHT to execute the procedure of MapReduce on hierarchical data centers. Comprehensive evaluation shows that our methodology is effective and possesses excellent performance. The main contributions of this paper are as follows.

- First, we propose the schemes for designating master servers and worker servers, and storing data files on hierarchical data centers, so as to facilitate the execution of MapReduce.

- Second, we present a specific DHT architecture and a corresponding routing scheme for assigning Map and Reduce tasks and delivering intermediate data on hierarchical data centers. Comprehensive evaluation demonstrates that our scheme can evenly distribute the workload and well support throughput-hungry MapReduce applications.

- Third, we deal with server and switch failures by proposing suitable fault-tolerant approaches for performing MapReduce on hierarchical data centers. Experimental results prove that our methodology is a reasonable solution even considering node failures.

The remainder of this paper is organized as follows: Section 2 introduces the background, related work and our motivation. Section 3 proposes the schemes for executing the basic procedure of MapReduce on hierarchical data centers. Section 4 presents the DHT architecture and routing scheme. Section 5 looks into the fault-tolerant routing and issues for performing MapReduce on hierarchical data centers. Section 6 evaluates the performance of the proposed methodology. Section 7 concludes this paper.

2 Preliminaries

2.1 Background

With a simple and practical processing procedure, MapReduce provides a standard mechanism for distributed data processing. A basic MapReduce procedure consists of a Map phase and a Reduce phase [5]. Each phase includes multiple parallel Map or Reduce tasks, respectively. A MapReduce procedure can process terabytes of data through numbers of Map and Reduce tasks.

Map tasks are applied for data classification and preparing intermediate data for Reduce tasks. By means of predefined Map programs, Map tasks transform the input data into intermediate data, which are organized as key/value pairs, and then deliver those intermediate data with the same key to corresponding Reduce tasks. The keys represent the types of intermediate data. The values represent the content of intermediate data.

Reduce tasks are responsible for merging those intermediate data and producing output files. After retrieving intermediate data from Map tasks, Reduce tasks integrate the intermediate values associated with the same key by means of predefined Reduce programs, and therefore generate output values.

In a data center, MapReduce lets a master server control many worker servers in executing Map and Reduce tasks [7]. The master assigns each Map or Reduce task to a worker, and each Map task is assigned to the worker that stores the input data for the Map task. The workers executing Map and Reduce tasks are called Mappers and Reducers, respectively. In the Map phase, Mappers execute corresponding Map tasks simultaneously. When Mappers accomplish Map tasks, they store derived intermediate data on local disks, and then send the location information of intermediate data to the Master. In the Reduce phase, the master distributes the location information of intermediate data to Reducers. Then Reducers read corresponding intermediate data from Mappers and execute their respective Reduce tasks simultaneously. Fig.1 shows the basic process of a MapReduce procedure.

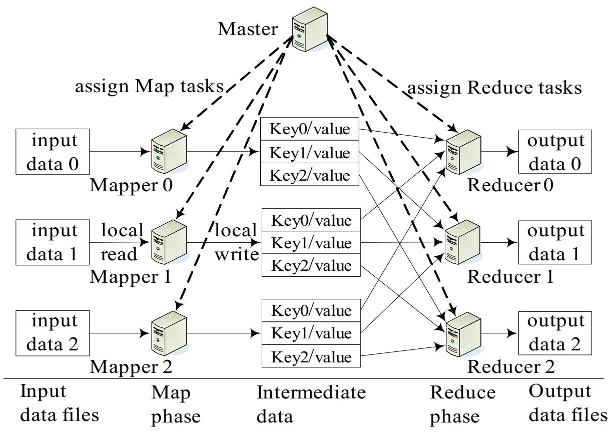


Figure 1: The basic process of a MapReduce procedure

2.2 Related Work

Many existing data centers adopt the traditional tree structure. Namely, all servers are located at leaf nodes. Aggregation switches and core switches are placed at inner nodes and root nodes, respectively. The servers are connected by the aggregation switches, which are linked by the core switches. Using these expensive and high-speed switches, the servers are fully interconnected. There is a path between each pair of servers without passing through any other server. The traditional tree structure is simple and easy to build, but it does not scale well. Expanding such a structure needs to add more inner nodes and root nodes, using more expensive and higher-speed switches. Actually, these aggregation switches and core switches easily lead to bottlenecks. A core switch failure can break down hundreds or even thousands of servers.

Fat-tree [1] is an improved structure of the traditional tree structure. Every inner node in a Fat-tree has more than one father node. This improvement increases the number of links between the aggregation switches and core switches. The network connectivity is increased, making Fat-tree a relatively reliable structure. However, like the traditional tree structure, it still does not scale well.

Hierarchical structures constructed recursively are believed to be scalable and reliable. To construct a hierarchical structure, a high level structure utilizes a lower level structure as a unit and connects many such units by means of a given recursive rule. As the level of a structure increases, more and more servers can be added into a hierarchical DCN without destroying the existing structure.

DCell [9], FiConn [15], and BCube [11] are three typical hierarchical structures. They use the same smallest recursive unit, in which a switch interconnects several servers. However, they are different in their recursive rules. DCell employs a complete graph as its recursive rule. There is a link between any two units of the same level. As a result, DCell possesses the advantage of the complete graph. In order to obtain high connectivity, each server in DCell should be equipped with multiple network ports. Although FiConn and DCell employ similar design principles for constructing high level compound graphs recursively, they have fundamental differences. Each server in FiConn is equipped with only two network ports, and the recursive units in FiConn are connected with only a half of the idle network ports in each unit. BCube employs the generalized hypercube as its recursive rule. The neighboring servers, which are connected to the same switch, differ in only one digit in their address arrays. Thus, BCube holds the advantages of the generalized hypercube, such as high connectivity and reliability.

2.3 Motivation

The above hierarchical structures efficiently solve the problems of scalability and reliability in different ways by using recursively defined topologies. Nevertheless, they ignore the particular approaches for performing MapReduce on the data centers with these structures. In a hierarchical structure, servers are not fully interconnected, and each server only connects with several adjacent servers. To support MapReduce, the hierarchical structure cannot utilize the approaches proposed by Google for performing MapReduce on the traditional tree structure [6].

First, hierarchical structures do not treat servers as masters and workers. In the procedure of Google's MapReduce, the servers controlling the execution of MapReduce procedures are called masters. The servers executing Map and Reduce tasks are called workers. In the traditional tree structure, servers are partitioned into masters and workers for MapReduce. A master can directly communicate with its workers without intermediate server. In a hierarchical structure, however, the communication among servers is multi-hop. Assume the servers are partitioned into masters and workers, there will be lots of control information transmitted through the servers shared by different paths. It is difficult to designate servers as masters or workers in hierarchical data centers.

Second, hierarchical structures can hardly support the data maintenance mechanism used by the traditional tree structure. In the traditional tree structure, each data file is divided into 64 megabytes blocks, and each block has several copies which are stored on different workers to keep data locality [3]. When a worker updates its data files, it sends related maintaining information to a distributed file system [12], which runs on some particular servers. Since the number of servers in a data center can be up to several thousands or even more, if the same method is used on a hierarchical data center whose servers are not fully connected, the amount of maintaining information transmitted on the data center will generate huge traffic load.

Third, the approaches for transmitting intermediate data on the traditional tree structure may be inefficient on hierarchical structures. In a traditional tree structure, after accomplishing Map tasks, Mappers store all intermediate data on local disks and send corresponding messages to the master. Then the master forwards the information about intermediate data to Reducers. After that, Reducers send massages to Mappers asking for intermediate data. Finally, Mappers distribute all intermediate data to Reducers concurrently. In a hierarchical data center with non-fully connected network, this process becomes much more complex. Since Mappers are not directly connected with Reducers, the intermediate data may be transmitted through several servers shared by different paths. Sometimes a Mapper can generate megabytes of intermediate data. When Mappers are executing Map tasks, communication channels might be idle. However, when the Map tasks are accomplished, some Mappers may have to wait to deliver intermediate data. Therefore, delivering all intermediate data concurrently through multi-hop reduces network resource utilization, takes too much bandwidth and may result in network congestion.

To perform MapReduce on hierarchical data centers, we would like to propose a methodology for addressing all the above problems with high fault-tolerance. Details about the proposed methodology are introduced in the rest of the paper.

3 MapReduce on Hierarchical Data Centers

In this section, we study the methodology for performing MapReduce on hierarchical data centers. Our methodology is mainly based on the distributed hash table (DHT), and can efficiently solve the above problems in terms of storing data files, assigning Map and Reduce tasks, and delivering intermediate data.

3.1 Roles of Servers

In a data center, servers control and execute MapReduce procedures. In our research, each server in a hierarchical data center can work as a master or a worker.

If a server receives a MapReduce request, it will be regarded as a master for the current MapReduce procedure. Different from a traditional data center, this master is only responsible for assigning Map and Reduce tasks to workers. It is not responsible for controlling the transmission of intermediate data.

If a server receives a Map or Reduce task, it will be regarded as a worker for the current MapReduce procedure. Moreover, the worker receiving a Map task is regarded as a Mapper, and the worker receiving a Reduce task is regarded as a Reducer. A worker executes received tasks according to the rule of FCFS (first come, first served). After accomplishing Map tasks, Mappers directly send derived intermediate data to Reducers without masters' control.

3.2 Scheme for Storing Data Files

We would like to design a scheme for storing data files, so that the traffic load due to transmission of the maintaining information can be reduced and the Map tasks can be easily assigned. According to the rule of DHT [20] [17], the scheme for storing data files on hierarchical data centers can be summarized as three steps. The first step is to define a suitable file key space for all servers, and assign a set of sequential file keys to each server. A file key refers to a fixed-length number or string used for denoting a data file block. The number of file keys assigned to a server depends on the disk capacity of the server. A server with more disk capacity can get more file keys. The second step is to build a file key table on each server, which records the range of file keys of every adjacent server. The third step is to define a suitable function to hash the name of each data file block to a file key, and store the data file block on a server which holds that file key.

Each server is responsible for maintaining its data file blocks, file keys and the file key table. If a server updates its data file blocks, there is no need to inform other servers. If a server updates its file keys, it sends maintaining information only to its adjacent servers to update their file key tables. Each server has only a finite number of adjacent servers. Consequently, storing data files according to the above scheme can reduce the amount of maintaining information transmitted on the whole data center network.

3.3 Scheme for Assigning Map and Reduce Tasks

In a traditional tree structure, a master directly connects with all workers. Therefore, a master can easily send a Map or Reduce task to a worker. In a hierarchical structure, however, a master usually sends a Map or Reduce task to a worker through a number of other servers.

(1) Assigning Map Tasks.

Based on the scheme for storing data files and the rule of DHT [20], we propose the scheme for assigning Map tasks on hierarchical data centers as follows. When a server receives a MapReduce request, it first determines the input data file block processed by each Map task. This server, namely the master, then hashes the name of the input data file block to a file key for the Map task. After that, the master chooses a server from all its adjacent servers to send the Map task, and the selected server has the range of file keys *closest* to the derived file key. This server will further choose another server from its neighbors to forward the Map task according to the same rule. This process is iteratively performed until a server, which receives the Map task, holds the corresponding file key. That implies the server stores the input data for the Map task. Here the *closest* is measured by a function, which is specified according to the definition of file keys.

(2) Assigning Reduce Tasks.

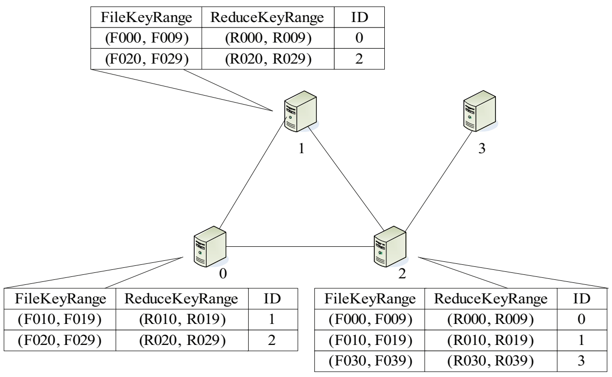
We propose the scheme for assigning Reduce tasks on hierarchical data centers as following steps. Similar to the scheme for storing data files, the first step is to define a suitable Reduce key space for all servers, and assign a set of sequential Reduce keys to each server. A Reduce key refers to a fixed-length number or string. The number of Reduce keys assigned to a server depends on the computing ability of the server. A server with faster computing capacity can get more Reduce keys. The second step is to build a Reduce key table on each server, which records the range of Reduce keys of every adjacent server. The third step is to define a suitable function used for hashing the keys of intermediate data to Reduce keys. Finally, when assigning Reduce tasks, the master hashes the key of intermediate data processed by each Reduce task to a Reduce key. The master chooses a server from all adjacent servers to send the Reduce task, and the selected server has the range of Reduce keys *closest* to the derived Reduce key. This process is iteratively performed until a server, which receives the Reduce task, holds the corresponding Reduce key.

Since the master can send Map and Reduce tasks immediately without searching data files node by node, in a hierarchical data center, assigning Map and Reduce tasks through DHT can reduce the execution time of MapReduce procedures. More than that, since there is no particular path from the master to a worker, the transmission of the Map or Reduce task cannot be interrupted by node failures. The communication becomes more reliable.

3.4 Scheme for Delivering Intermediate Data

Based on the scheme for assigning Reduce tasks, it is easy to delivering intermediate data. Our scheme for delivering intermediate data on hierarchical data centers are as follows. When a Mapper is executing a Map task, it hashes the key of a derived intermediate key/value pair to a Reduce key in the same way as that of assigning Reduce tasks. Then, according to its Reduce key table, it directly sends the intermediate key/value pair to an adjacent server, which has the range of Reduce keys *closest* to the Reduce key hashed from the intermediate key among all adjacent servers. In a similar way to the scheme for assigning Map and Reduce tasks, this intermediate key/value pair will be delivered node by node to a server which holds the corresponding Reduce key. According to the scheme for assigning Reduce tasks, this server has received the Reduce task that is responsible for processing the intermediate data.

Delivering intermediate data with above scheme can avoid the unnecessary controlling process performed by masters, which is complicated and can delay executing Reduce tasks in a non-fully connected data center network. Since each intermediate key/value pair is delivered immediately after being generated, this scheme can increase network resource utilization, and facilitate intermediate data transmission. Like the scheme for assigning Map tasks, delivering intermediate data in this way can avoid the impact of node failures, and therefore can increase the reliability of the transmission.



FileKeyRange	ReduceKeyRange	ID
(F000, F009)	(R000, R009)	0
(F020, F029)	(R020, R029)	2

FileKeyRange	ReduceKeyRange	ID
(F010, F019)	(R010, R019)	1
(F020, F029)	(R020, R029)	2

FileKeyRange	ReduceKeyRange	ID
(F000, F009)	(R000, R009)	0
(F010, F019)	(R010, R019)	1
(F030, F039)	(R030, R039)	3

Figure 2: An example of our DHT

4 DHT Architecture and Routing Scheme

This section introduces the specific DHT architecture and corresponding routing scheme for executing MapReduce on hierarchical data centers.

Since the servers in a hierarchical data center are homogeneous and work in the same manner, we assign the file or Reduce keys to the servers in the order of their identifiers. The keys held by different servers are logically arranged in a line or a circle, like Chord [20]. This dose not means that the longest path to retrieve a key is the whole line. Adjacent servers in a hierarchical data center can belong to different recursive units and do not have sequential identifiers. The range of keys held by adjacent servers may not be sequential. Only the adjacent servers in the same smallest recursive unit hold sequential identifiers and keys. Consequently, for a hierarchical data center, the retrieval efficiency depends on its physical structure instead of the logical form in which the keys are arranged.

For ease of maintenance, we integrate the aforementioned file key table and Reduce key table into one hash table. This table consists of three attributes, including the range of file keys, the range of Reduce keys and the identifier of the adjacent server that holds those file keys and Reduce keys. In a hierarchical data center, each server stores such a table for recording the range of file keys and Reduce keys of all adjacent servers. In our work, $Item_i$=($FileKeyRange$, $ReduceKeyRange$, ID) denotes a record of the hash table, where *FileKeyRange, ReduceKeyRange*, and ID represent the three attributes, respectively. When a server is added to or removed from the data center, only its adjacent servers need to renew the records of their hash tables, so all other severs will not be affected. Fig.2 shows an example of our DHT architecture for executing MapReduce on hierarchical data centers.

Based on the hash table stored on each server, it is easy to implement the routing scheme for assigning Map and Reduce tasks and sending intermediate data. When a server receives a Map or Reduce task or an intermediate key/value pair, it first determines whether the corresponding file key or Reduce key is in its charge. If the server holds that file key or Reduce key, it performs the corresponding task according to reference [6]. Otherwise, it forwards the Map or Reduce task or intermediate data to an adjacent server in the way indicated by Algorithm 1.

In Algorithm 1, $TransObject$ denotes a Map or Reduce task or an intermediate key/value pair, which needs to be delivered by any server in a hierarchical data center. $TransObject.key$ denotes the file key or Reduce key derived by corresponding hash functions. Function $d(,)$ is used for calculating the distance between $TransObject.key$ and the range of keys held by an adjacent server. Algorithm 1 sets a variable, denoted by $Server$, for recording the record item of the hash table on current server, which represents the next hop to send $TransObject$. It initializes $Server$ with the first record item, and then determines the type of $TransObject$. After that, Algorithm 1 iterates over the hash table to find a record whose range of file or Reduce keys is the closest to $TransObject.key$ according to function $d(,)$, and assigns the obtained record item to $Server$. Finally, it sends $TransObject$ to the adjacent server whose identifier is denoted by $Server.ID$. In this way, $TransObject$ will be delivered node by node to the server responsible for executing corresponding Map or Reduce task.

Algorithm 1 Deliver1 (object $TransObject$)

```
1: object $Server = Item_0$;
2: if $TransObject$ is a Map task then
3:   for $i = 1; i < I - 1; i++$ do
4:     if $d(TransObject.key, Item_i.FileKeyRange)$
       $< d(TransObject.key, Server.FileKeyRange)$ then
5:       $Server = Item_i$;
6:     end if
7:   end for
8: else
9:   for $i = 1; i < I - 1; i++$ do
10:    if $d(TransObject.key, Item_i.ReduceKeyRange)$
       $< d(TransObject.key, Server.ReduceKeyRange)$ then
11:      $Server = Item_i$;
12:    end if
13:  end for
14: end if
15: send $TransObject$ to $Server.ID$;
```

5 Fault Tolerance

A hierarchical data center consists of much more servers, switches and links than a traditional tree structure data center, so it has more tendency to machine or link failures. A link failure leads to the disconnection of the two machines interconnected through the link, and the two machines can be regarded as failed to each other. Hence we only focus on server and switch failures in this paper.

5.1 Fault-tolerant Routing Against Failures of Servers and Switches

A MapReduce procedure cannot utilize a failed server or switch to assign tasks or transmitting intermediate data. To address this problem, we propose the fault-tolerant routing for MapReduce in hierarchical data centers.

In a traditional tree structure data center, a switch failure can break down all the servers connecting to it. Since a hierarchical data center employs the redundant structure, a switch failure may not affect the servers connecting to it. However, these servers cannot communicate to each other directly. In this work, we treat a switch failure as several disconnected servers.

To ensure that our routing scheme can forward all tasks and intermediate data to available servers, we employ the following approach. Each server sends the number of tasks in its service queue as its state information to all adjacent servers periodically, and therefore each server knows the running state of all its adjacent servers. If a server cannot update its state information to its adjacent servers, it will be regarded as a failed server and its corresponding records in the hash tables of adjacent servers will be denoted as unavailable. Based on this failure notification mechanism among adjacent servers, we modify Algorithm 1 in order to achieve Algorithm 2 as the fault-tolerant routing scheme. Algorithm 2 assigns the first available record item to $Server$, and iterates over the hash table from that record to update $Server$ with an available record whose range of file or Reduce keys is closer to $TransObject.key$. In such a way, $Server.ID$ finally gives the identifier of the available next hop to deliver $TransObject$.

Algorithm 2 Deliver2 (object $TransObject$)

```
1:  object $Server = Null$; int $j = 0$;
2:  for $i = 0; i < I - 1; i++$ do
3:    if $Item_i.available == true$ then
4:      $Server = Item_i$;
5:      $j = i$;
6:      break;
7:    end if
8:  end for
9:  if $TransObject$ is a Map task then
10:   for $i = j; i < I - 1; i++$ do
11:     if $Item_i.available == true$ then
12:       if $d(TransObject.key, Item_i.FileKeyRange)$
          $< d(TransObject.key, Server.FileKeyRange)$ then
13:         $Server = Item_i$;
14:       end if
15:     end if
16:   end for
17: else
18:   for $i = j; i < I - 1; i++$ do
19:     if $Item_i.available == true$ then
20:       if $d(TransObject.key, Item_i.ReduceKeyRange)$
          $< d(TransObject.key, Server.ReduceKeyRange)$ then
21:         $Server = Item_i$;
22:       end if
23:     end if
24:   end for
25: end if
26: send $TransObject$ to $Server.ID$;
```

5.2 Fault-tolerant Approaches to Address Failures of Masters and Workers

In a hierarchical data center, a running MapReduce procedure can be interrupted by a server failure, no matter a master failure or a worker failure. To address this problem, we propose the following approaches.

1) Addressing the failure of a master server:

- As soon as a master receives a MapReduce request, it sends the request to an adjacent server as a replica. When assigning Map and Reduce tasks, the master concurrently sends a confirmation message for each task to that adjacent server. If that server cannot receive any confirmation message of a task within a threshold time, the master will be regarded as failed, and that server will take over the current MapReduce request and reassign the corresponding Map or Reduce task.

2) Addressing the failure of a worker server:

- If a worker server receives a Map task and its service queue has achieved a predefined threshold length, it will discard the Map task and send a message to the Master for reassigning the Map task to another server which stores a replica of the corresponding input data file.
- According to the aforementioned failure notification mechanism, each server keeps the running state of all adjacent servers. If a worker server receives a Reduce task and its service queue has achieved a predefined threshold length, it will forward the Reduce task to an available adjacent server. Moreover, it will forward all the intermediate data to that adjacent server.
- When a worker server accepts a Map or Reduce task, it sends corresponding information of the task to an adjacent server. As soon as the worker server accomplishes that task, it sends a confirmation massage of the task to that adjacent server. If that adjacent server cannot receive the confirmation massage within a threshold time, it will send the task information to the master for reassigning the task to another available server. In this case, the worker server will be regarded as a straggler [16].

6 Evaluation

Since BCube is a representative hierarchical structure [11], in this section we conduct a comprehensive evaluation based on BCube, to demonstrate that our method is feasible for executing MapReduce on hierarchical data centers.

In the following evaluation, we employ Equation 1 as the hash function for transferring a task name or an intermediate key into a file key or a Reduce key.

$$TransObject.key = f(TransObject) \text{ MOD } K \tag{1}$$

Function $f(TransObject)$ calculates the decimal ASCII number of $TransObject$. For example, suppose that $TransObject$ denotes a character string "abc", then $f(TransObject)$ equals 979899. K is a prime number less than the total number of file keys or Reduce keys. Here the value of $TransObject.key$ is an integer, hence the aforementioned function $d(,)$ can be defined for calculating the absolute value of the difference between $TransObject.key$ and the range of keys held by a server.

6.1 Load Balance

According to aforementioned schemes, Map tasks are assigned to the servers that hold corresponding data files, so the distribution of input data file blocks determines the load balance of Map tasks. Since researchers have studied how to allocate date to servers evenly with consistent hashing [14] [23], we assume that all input data file blocks are well distributed in a hierarchical data center. Therefore, we can also assume that Map tasks can be evenly assigned to different servers. Here we mainly study the load balance of Reduce tasks, which is much more uncertain than that of Map tasks.

We perform the simulation for evaluating load balance as follows. We simulate the structure of BCube and corresponding communication among its servers. Let N denote the number of servers in a level 0 BCube, and H denote the number of levels. The number of servers in BCube varies from 4 to 625 when N varies from 2 to 5 and H varies from 1 to 3. We assign a unique identifier to each server, and calculate the identifiers of its adjacent servers. In BCube, two servers that connect to the same switch can be regarded as adjacent servers. In reality, different MapReduce applications generate different kinds of Reduce tasks, whose arrival to a data center is a stochastic process. For ease of evaluation, in our simulation, we assume that there is only one Reduce task arriving to the BCube data center within a short period of time. The execution time of a Reduce task is random and is longer than the arrival period. We consider two cases about the execution time. The first case is that the execution time of a Reduce task varies randomly from 10 to 100 times the length of arrival period. The second case is that the execution time of a Reduce task varies randomly from 100 to 500 times the length of arrival period. According to Hadoop [24], we define that each server can execute two Reduce tasks simultaneously. When a server is busy in executing two Reduce tasks, it will reject the newly arrived Reduce task and foreword it to an adjacent server. If a Reduce task is rejected for three times, we regard that this Reduce task is dropped. We consider a workload with 2×10^3 Reduce tasks, which are assigned according to the schemes studied in Section 3.

Since a server can only execute limited number of Reduce tasks simultaneously, some Reduce tasks may get dropped in a data center with skewed load. We repeat the simulation 30 times for each number of servers to calculate the mean percentage of dropped Reduce tasks. Fig.3 plots the results in the two cases about the execution time. As shown in Fig.3, the percentage of dropped Reduce tasks decreases rapidly with the increase of the number of servers. In the first case, the percentage of dropped Reduce tasks decreases to 0 when the number of servers achieves 64. In the second case, the percentage of dropped Reduce tasks decreases to 0 when the number of servers achieves 256. When we further increase the number of arrival Reduce tasks to 10^4, the results of the two cases remain the same. This implies that running MapReduce on such a data center according to our approaches can hardly drop any tasks.

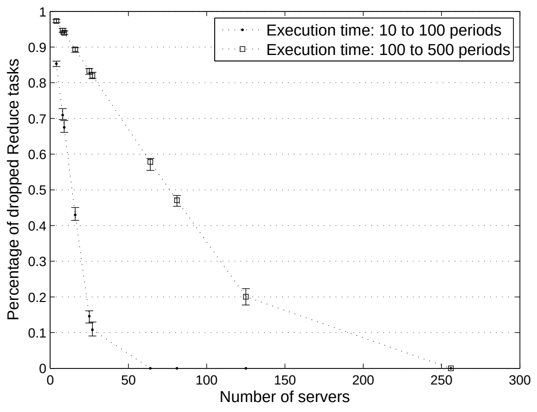


Figure 3: Variation of the percentage of dropped Reduce tasks along with the number of servers

To evaluate the workload of each server, based on the above simulation, we calculate the Reduce tasks executed by each server in the two cases. Fig.4 plots the variation of the mean percentage of Reduce tasks that per server executes along with the number of servers, which varies from 4 to 625. As shown in Fig.4, when the number of servers is small, the mean workload of a server in the first case is higher than that in the second case, as the execution time of the first case is much lower than that of the second case. In the first case, the mean workload of a server keeps on a relatively high level until the number of servers achieves 27. In the second case, the mean workload of a server dose not decrease until the number of servers achieves 125. The reason of these variations is, when the number of servers is not enough to sustain continually arrival Reduce tasks, all servers work at full capacity and many tasks are dropped. While the number of servers achieves 256, the mean workload of a server decreases to the same low level in both cases. Fig.5 shows the percentage of Reduce tasks executed by each server when there are 256 servers in all. We can derive from Fig.5 that the difference between the workload of any two servers is less than 1.6×10^{-3}, a vary small value. Therefore, the workload can be evenly distributed.

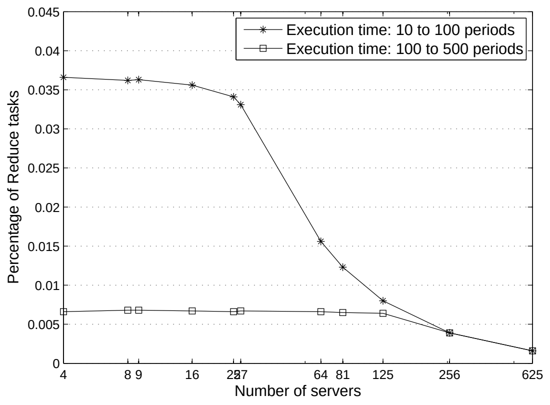


Figure 4: Variation of the mean percentage of Reduce tasks that per server executes along with the number of servers

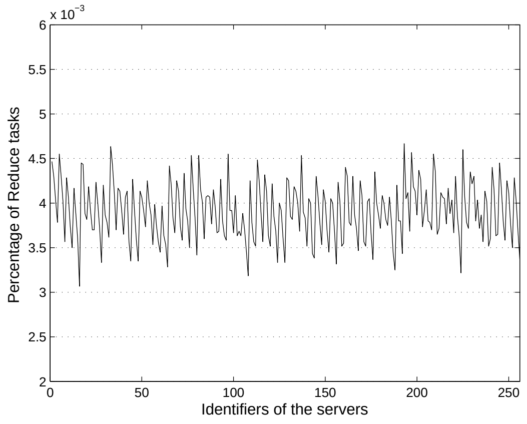


Figure 5: Percentage of Reduce tasks executed by each server when there are 256 servers

6.2 Data Forwarding Performance

A data center network mainly consists of servers, switches and links. In a hierarchical data center, servers are not only used for executing Map and Reduce tasks, but also used for forwarding intermediate data. However, since servers are not network devices, they have lower data forwarding capacity than switches and links. In this work, we assume that switches and links can provide sufficient bandwidth and only consider the servers.

Based on the former simulation, we perform the simulation for evaluating data forwarding performance, including data forwarding throughput and bandwidth between servers, as follows. We assume that our routing scheme held by each server supports receiving and sending only one data packet, namely a key/value pair, within an extremely short period of time. According to literature [11], in a BCube data center, a server forwards at 750Mb/s in all-to-all traffic pattern. MTU is usually set to be 1.5KB for a commodity computer. Thus, on this condition, we can easily calculate that the short period is 1.6×10^{-5} seconds. In such a period, a server with a packet to transmit chooses an adjacent server as the destination server, according to our routing scheme. If this destination server is available for receiving a packet, the former server will forward that packet successfully. Then any other packets to the same destination server should wait to be transmitted until the server is available in another period. We vary the number of servers that simultaneously generate packets to calculate the best possible data forwarding performance. This procedure is recursively performed in our simulation so as to evaluate data forwarding performance on a steady working condition.

When the number of servers varies from 4 to 625, we repeat the simulation 30 times for each number of servers to obtain the mean and range of the maximum data forwarding throughput, as shown in Fig.6. We find that the maximum data forwarding throughput does not closely track the increase of the total number of servers. The throughput of 27 servers is lower than that of 25 servers, and the throughput of 81 servers is lower than that of 64 servers. The reason for that is, throughput depends not only on the number of servers, but also on the number of hops for data forwarding. If the data are forwarded through more hops, the throughput will be lower. For a hierarchical data center, the maximum number of hops is determined by the number of levels, and more levels bring more hops. In our simulation, the BCube data centers with 27 servers and 81 servers have one level more than the BCube data centers with 25 servers and 64 servers, respectively. Given a fixed number of servers in a recursively hierarchical data center, it is crucial to make a tradeoff between the number of levels and the number of servers in the smallest recursive unit to achieve desired performance. As shown in Fig.6, when the number of servers is larger than 125, data forwarding throughput increases rapidly. Therefore, our approaches can well support throughput-hungry MapReduce applications on hierarchical data centers.

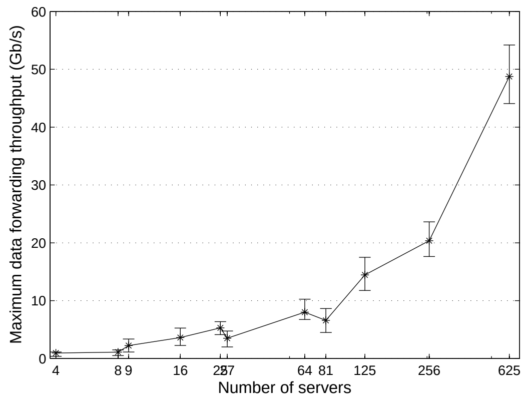


Figure 6: Variation of the maximum data forwarding throughput along with the number of servers

To evaluate the bandwidth between servers, we first keep the number of servers that simultaneously generate packets in a certain value, which ensures that data forwarding throughput remains at the maximum value. On this condition, we then calculate the mean of the maximum bandwidth that each server can achieve for sending data to another server through maximum number of hops. Fig.7 illustrates the result when there are 256 servers. We can derive that most values of the bandwidth are larger than 0.3Gb/s and less than 0.57Gb/s. This variance, which is less than 0.3Gb/s, is acceptable for MapReduce applications. In practice, different servers store different types of data, and internet service providers may store popular data on certain servers to save power [19]. Hence some of the Map or Reduce tasks in a MapReduce procedure process and generate more data than other tasks. Overall, the result of our simulation implies that the bandwidth can be evenly distributed according to our approaches.

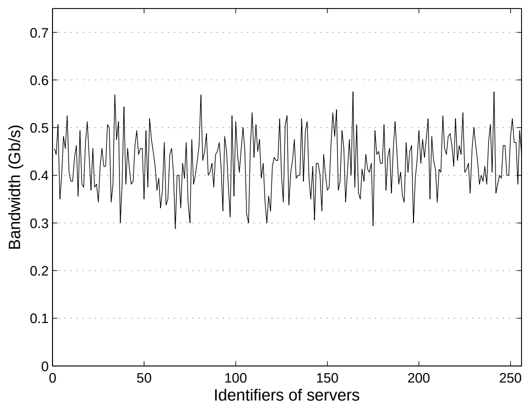


Figure 7: Mean of the maximum bandwidth that each server can achieve for sending data to another server through maximum number of hops, when there are 256 servers and data forwarding throughput remains at maximum value

6.3 Fault Tolerance

We evaluate the load balance and data forwarding performance under given failure rates of links and nodes to further validate the fault-tolerant capability of our methodology.

For a hierarchical data center, let P_1 and P_2 denote the expected probabilities that a server or a switch fails when they are executing a MapReduce procedure, respectively. Here we omit link failures which can be regarded as adjacent nodes failures. Then we can calculate the probability that a server keeps on working according to Theorem 1.

Theorem 1. *For a hierarchical data center, let P_3 denote the probability that a server keeps on working in a MapReduce procedure. Let I and J denote the number of servers and switches directly connecting with this server, respectively. P_3 is given by*

$$P_3 = (1 - P_1) \times (1 - \sum_{i=1}^{I}(\binom{I}{i} \times P_1^i)) \times (1 - \sum_{j=1}^{J}(\binom{J}{j} \times P_2^j)) \qquad (2)$$

Proof: The probability that this server does not fail equals $1-P_1$. The probability that i of the I servers fail is $\binom{I}{i}\times P_1^i$. Then the probability that these I servers keep on working equals $1-\sum_{i=1}^{I}(\binom{I}{i}\times P_1^i)$. Similarly, the probability that the J switches keep on working equals $1-\sum_{j=1}^{J}(\binom{J}{j}\times P_2^j)$. This server can keep on working only if all the I servers and J switches can keep on working and itself does not fail. Therefore, the probability that this server keeps on working equals $(1-P_1)\times(1-\sum_{i=1}^{I}(\binom{I}{i}\times P_1^i))\times(1-\sum_{j=1}^{J}(\binom{J}{j}\times P_2^j))$. Theorem 1 is proved.

Based on Theorem 1, we modify the aforementioned simulations to evaluate the fault-tolerant load balance and data forwarding performance. In the corresponding period, a server with a Reduce task (or a packet) for transmitting chooses an available adjacent server in working order as the destination server, according to the fault-tolerant routing. If there is a third server having a Reduce task (or a packet) to the same destination server in the same period, it has to wait until the destination server is available in anther period.

In BCube, each server can only directly connect to switches, so P_3 equals $(1-P_1)\times(1-\sum_{j=1}^{J}(\binom{J}{j}\times P_2^j))$. We obtain different values of P_3 by varying P_1 and P_2. Then we calculate the variation of the mean percentage of Reduce tasks that per server executes along with the total number of servers, when

P_3 equals 0.98, 0.90 and 0.80, respectively. Fig.8 illustrates the results in the aforementioned two cases. In the first case, many tasks are dropped when the number of servers is small, so there is no significant difference among the workload of each server for the three values of P_3. In the second case, since the workload of each server keeps very low, the difference is not notable when the number of servers is less than 125. While the number of servers are enough to sustain the continually arrival Reduce tasks, our method has good fault tolerance, so the difference is also small in both cases. Fig.9 plots the percentage of Reduce tasks executed by each server when there are 256 servers and P_3=0.90. The mean percentage of Reduce tasks that per server executes is only 0.5×10^{-3} lower than that shown in Fig.5. Moreover, the variance is less than 1.5×10^{-3}. Thus, the workload can be evenly distributed under high failure rates of links and nodes.

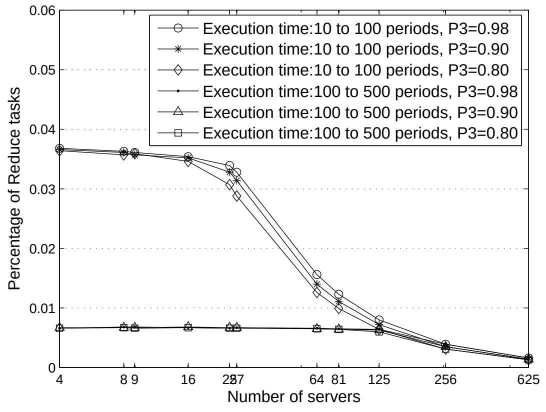


Figure 8: Variation of the mean percentage of Reduce tasks that per server executes along with the number of servers, when P_3=0.98, P_3=0.90 and P_3=0.80

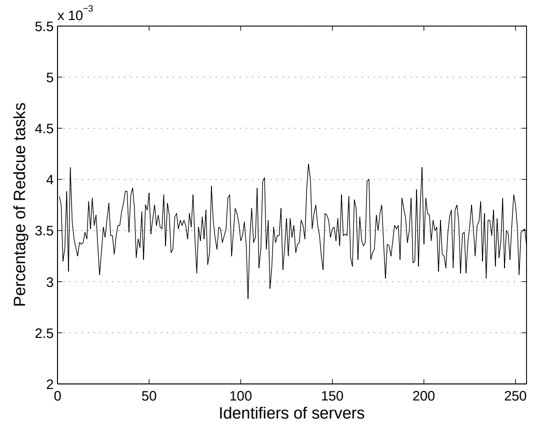


Figure 9: Percentage of Reduce tasks executed by each server, when there are 256 servers and P_3=0.90

Due to the failures of links and nodes, we calculate the maximum data forwarding throughput when P_3 equals 0.98, 0.90 and 0.80, respectively. For each value of P_3, we repeat the modified simulation 30 times to obtain the mean value. Fig.10 illustrates the result. When the number of servers is small, the throughput of the network is low, so there is no obvious difference among the throughput for the three values of P_3. When the number of servers is larger than 125, the throughput increases rapidly for all the three values of P_3. Thus, with enough servers, our method can provide satisfactory data forwarding throughput against failures of links and nodes. Considering link and node failures, we

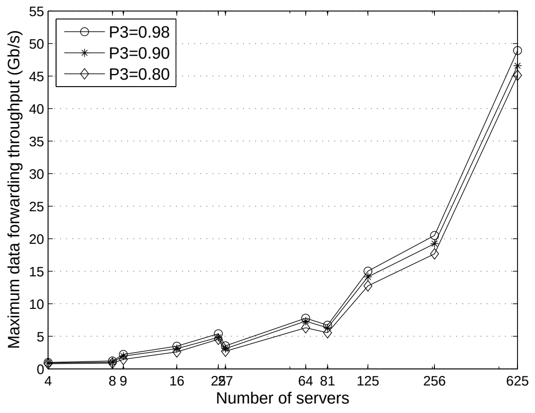


Figure 10: Variation of the maximum data forwarding throughput along with the number of servers, when P_3=0.98, P_3=0.90 and P_3=0.80

recalculate the mean of the maximum bandwidth that each server can achieve for sending data to another server through maximum number of hops. Fig.11 illustrates the result when there are 256 servers and P_3=0.90. Although the probability that a server cannot keep on working is 0.10, which is really high in practice, the bandwidth for every server is only 0.05Gb/s lower than that shown in Fig.7. Hence the bandwidth between servers is abundant against failures of links and nodes. Moreover, the range of variance, as shown in Fig.11, is less than 0.25Gb/s. Therefore, the bandwidth between servers can be evenly distributed under high failure rates of links and nodes.

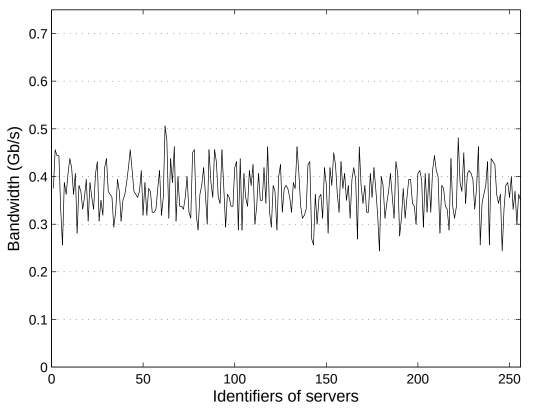


Figure 11: Mean of the maximum bandwidth that each server can achieve for sending data to another server through maximum number of hops, when there are 256 servers and P_3=0.90

7 Conclusion

Several hierarchical structures have been proposed to improve the topological properties of data centers. However, the communication and control mechanisms proposed by Google for performing MapReduce on the traditional structure can hardly be operated on these hierarchical structures. This paper presents a methodology for performing MapReduce on data centers with hierarchical structures. Comprehensive analysis and simulations show that our methodology can evenly distribute the workload and well support throughput-hungry MapReduce applications. It is also proved that our methodology is competent for MapReduce even under node failures. The mismatch problem between hierarchical data centers and Mapreduce is effectively solved in this paper.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive comments. Our work is supported in part by the NSF China under Grants No.60903206, No.60972166, No.61170284, No.71031007, No.71071160 and No.71171197, the China Postdoctoral Science Foundation under grant No.201104439, and the Preliminary Research Foundation of National University of Defense Technology under grant No.JC10-05-01.

Bibliography

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. *Proc. ACM SIGCOMM*, pp.63-74, Aug. 2008.

[2] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/current/hdfsdesign.pdf

[3] C. Bastoul and P. Feautrier. Improving Data Locality by Chunking. *Springer Lecture Notes in Computer Science*, vol.2622, pp.320-334, 2003.

[4] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E.Gruber. Bigtable: A Distributed Storage System for Structured Data. *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp.205-218, Nov. 2006.

[5] J. Cohen. Graph Twiddling in a MapReduce world. *Computing in Science and Engineering, IEEE Educational Activities Department*, vol.2, no.4, pp.29-41, 2009.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pp.137-150, Dec. 2004.

[7] J. Dean, and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *Communications of the ACM*, vol.53, no.1, pp.72-77, 2010.

[8] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *ACM SIGCOMM computer communication review*, vol.39, no.1, pp.68-73, Jan. 2009.

[9] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. *Proc. ACM SIGCOMM*, pp.75-86, Aug. 2008.

[10] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. *ACM SIGCOMM Computer Communication Review*, vol.39, no.4, pp.51-62, Aug. 2009.

[11] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. *Proc. ACM SIGCOMM*, pp.63-74, Aug. 2009.

[12] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google File System. *Proc. 19th ACM Symposium on Operating Systems Principles*, pp.29-43, Dec. 2003.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel programs from Sequential Building Blocks. *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pp.59-72, Jun. 2007.

[14] W. Jun. A Methodology for the Deployment of Consistent Hashing *Proc. 2nd IEEE International Conference on Future Networks*, Jan. 2010.

[15] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, and S. Lu. FiConn: Using Backup Port for Server Interconnection in Data Centers. *Proc. IEEE INFOCOM*, pp.2276-2285, Apr. 2009.

[16] J. Lin. The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce *Workshop on Large-Scale Distributed Systems for Information Retrieval*, Jul. 2009.

[17] J. Pang, P.B. Gibbons, M. Kaminsky, S. Seshan, and H. Yu. Defragmenting DHT-based Distributed File Systems *Proc. 27th IEEE International Conference on Distributed Computing Systems*, Jun. 2007.

[18] T. Redkar. Introducing Cloud Services. *Windows Azure Platform, Apress*, pp.1-51, 2009.

[19] L. Rao, X. Liu, L. Xie, and W. Liu. Minimizing Electricity Cost: Optimization of Distributed Internet Data Centers in a Multi-Electricity-Market Environment *Proc. IEEE INFOCOM*, Mar. 2010.

[20] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peertopeer Lookup Service for Internet Applications *Proc. ACM SIGCOMM*, pp.1-12, Aug. 2001.

[21] D. Talia and P. Trunfio. Enabling Dynamic Querying over Distributed Hash Tables. *Elsevier Journal of Parallel and Distributed Computing*, vol.70, no.12, pp.1254-1265, 2010.

[22] G. Urdaneta, G. Pierre and M.V. Steen. A Survey of DHT Security Techniques. *Journal of ACM Computing Surveys*, vol.43, no.2, pp.1-49, 2011.

[23] X. Wang and D. Loguinov. Load-balancing performance of consistent hashing: asymptotic analysis of random node join *IEEE/ACM Transactions on Networking*, vol.15, no.4, pp.892-905, 2007.

[24] http://hadoop.apache.org.