CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **24**:1271–1295
Published online 6 July 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1791

A MapReduce-supported network structure for data centers

Zeliu Ding [1,2,*,†], Deke Guo [1], Xue Liu [2], Xueshan Luo [1] and Guihai Chen [3]

[1] *School of Information Systems and Management, National University of Defense Technology, Changsha 410073, China*
[2] *School of Computer Science, McGill University, Montreal H3A 2A7, Canada*
[3] *Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China*

SUMMARY

Several novel data center network structures have been proposed to improve the topological properties of data centers. A common characteristic of these structures is that they are designed for supporting general applications and services. Consequently, these structures do not match well with the specific requirements of some dedicated applications. In this paper, we propose a hyper-fat-tree network (HFN): a novel data center structure for MapReduce, a well-known distributed data processing application. HFN possesses the advanced characteristics of BCube as well as fat-tree structures and naturally supports MapReduce. We then address several challenging issues that face HFN in supporting MapReduce. Mathematical analysis and comprehensive evaluation show that HFN possesses excellent properties and is indeed a viable structure for MapReduce in practice. Copyright © 2011 John Wiley & Sons, Ltd.

Received 3 November 2010; Revised 6 May 2011; Accepted 9 May 2011

KEY WORDS: data center network (DCN); MapReduce; structure

1. INTRODUCTION

Data centers have emerged as the distributed storage and computing infrastructures for many online applications and infrastructure services, for example cloud services [1, 2]. A data center can store and process massive data through its inner servers and then provide a variety of agile and effective services to users. A fundamental challenge facing data centers is to design a data center network (DCN) for efficiently interconnecting a large number of servers via high-speed links and switches [3].

In reality, a data center should be equipped with specific mechanisms to manage and process massive data effectively and efficiently, such as Google File System [4], Hadoop Distributed File System [5], Bigtable [6], Dryad [7], and other mechanisms. MapReduce [8] is one of the most important data processing mechanisms. It provides a good control and execution model for distributed computing and cloud computing and has been widely used for processing terabytes of data by Google, Yahoo, Amazon, and some other Internet service providers.

In recent years, emerging diverse services call for an improvement of the topological performances of a DCN, including scalability, reliability, etc. In order to meet these requirements, several novel DCN structures have been proposed, such as fat tree [9], DCell [3], FiConn [10], and BCube [11]. These structures focus on optimizing some fundamental topological properties. However, they usually do not pay much attention to the requirements of distributed data processing mechanisms running on them. None of these novel structures partition all servers into master servers and worker

*Correspondence to: Zeliu Ding, National 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

servers, although this requirement is the basis of many distributed data processing mechanisms, especially MapReduce. In the meantime, many novel procedures and programs for MapReduce have been proposed [12–14]. These modifications add some pretreatment or assistant steps to basic MapReduce, so as to facilitate the map and reduce operations and produce more accurate results. However, they ignore the inner relationship among the elements that execute these operations on a data center. Actually, the procedure of a complex MapReduce requires the support of dedicated data center structures to meet users' ever increasing new service requirements.

To solve this problem, we present a hyper-fat-tree network (HFN), a novel data center structure for MapReduce applications, in this paper. In order to be scalable, HFN is recursively defined in a similar way as BCube. So that a high-level HFN can be generated, a lower-level HFN acts as a unit, and many such units are interconnected by means of a hypercube [15]. The major difference from BCube is that the lowest-level HFN is a fat-tree-like redundant structure, which provides reliable services for MapReduce. The interconnection relationships among switches, master servers, and worker servers in the lowest-level HFN are determined according to the procedure of MapReduce. Consequently, HFN is well suited to the data processing mechanism of MapReduce. HFN inherits the advantages of BCube as well as fat tree and hence has a high connectivity and a low diameter. For ease of explanation, in the remainder of this paper, the interconnection rule of a recursive structure is named *recursive rule*, and the lowest-level structure is called the smallest *recursive unit*.

So that a complex MapReduce service request can be supported, the number of worker servers may be up to 1000 or even more. It is hard for one master server to control so many worker servers simultaneously. Using the control mechanism proposed in this paper, many master servers execute a complex MapReduce program together, and hence each master server only controls a limited number of worker servers. This paper provides a distributed data operating and transmitting method for executing MapReduce on a data center. The mismatch problem between existing DCN structures and MapReduce is effectively solved in this paper. The main contributions of this paper are as follows:

- First, we propose a MapReduce-supported DCN structure, named HFN. HFN partitions all servers into master servers and worker servers, so as to fully exploit their different benefits for executing MapReduce. Extensive evaluation and analysis results show that HFN is a reliable and scalable topology with high connectivity, low diameter, and low construction cost.
- Second, we propose an effective method for executing MapReduce on HFN. We present the routing schemes for assigning MapReduce jobs and data transmission and the approaches for performing map and reduce tasks on HFN. We then show how the HFN structure intrinsically supports a particular MapReduce application.
- Third, we concentrate on dealing with failed servers and switches by proposing suitable fault-tolerant approaches for executing MapReduce on HFN. The case study and experimental results prove that HFN is a reasonable solution for MapReduce even when node failures are considered.

The remainder of this paper is organized as follows. Section 2 introduces the background, related work, and our motivation. Section 3 proposes the physical structure and a construction method for HFN. Section 4 describes the routing and executing schemes for MapReduce on HFN. Section 5 gives the fault-tolerant routing and executing approaches for MapReduce on HFN. Section 6 evaluates several topological properties of HFN. Section 7 studies the execution time of MapReduce on HFN. Section 8 evaluates the data forwarding performance of HFN. Section 9 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 complex MapReduce procedure processes terabytes of data through a sequence of jobs, and each job consists of one map phase and one reduce phase [16]. Each phase includes multiple parallel map or reduce tasks.

DOI: 10.1002/cpe

In a data center, MapReduce lets a master server control many worker servers in executing map and reduce tasks. In the map phase, the master server assigns each map task to a worker server. If a map task needs particular data stored at a worker server, the master server will send the map task to that work server. Then those worker servers execute corresponding map tasks simultaneously. In the reduce phase, the master server assigns each reduce task to a worker server. Then those worker servers also execute their respective reduce tasks simultaneously.

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.

Reduce tasks are responsible for merging those intermediate data. After retrieving the intermediate data from map tasks, reduce tasks merge the intermediate values with the same key by means of predefined reduce programs and then generate the output values.

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. The traditional tree structure is simple and easy to build but does not scale well. Actually, these core switches easily lead to bottlenecks. A failed core switch can break down hundreds or even thousands of servers.

Fat tree [9] 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 the core switches. Thus, the network connectivity becomes high, making fat tree a relatively reliable structure. However, like the traditional tree structure, it still does not scale well.

Literature [17] proposes a novel network for data centers, which supports energy proportional communication. This network interconnects servers by means of a flattened butterfly topology, which demonstrates power efficiency for DCN, so as to ensure that the amount of energy consumed is proportional to the traffic intensity in DCN.

Hierarchical structures constructed recursively are believed to be scalable. For a hierarchical structure to be constructed, a high-level structure utilizes a lower-level structure as a cluster and connects many such clusters by means of a given recursive rule [18]. As the level of a structure increases, more and more servers can be added into a hierarchical DCN without destroying the existing structure.

DCell [3], FiConn [10], 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. For a high connectivity to be obtained, each server in DCell should be equipped with multiple network ports. Although FiConn and DCell have similar design principles for the construction of high-level compound graphs recursively, they possess 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. Note that each server in BCube should also be equipped with multiple network ports.

2.3. Motivation and challenges

The aforementioned structures are dedicated to improving the topological properties of a DCN in different aspects. Unfortunately, these structures usually ignore the special requirements of distributed data management or processing mechanisms of applications, for example MapReduce.

DOI: 10.1002/cpe

First, existing DCN structures do not treat servers as master servers and worker servers. They simply assume all servers possess the same function and hence interconnect all the servers in the same way. However, in Google's MapReduce program, servers are classified into masters and workers according to different functions. Worker servers are used for performing concrete tasks. Master servers are used for assigning tasks to worker servers and controlling their executions and communications. For the distinguished benefits of the two roles to be realized, servers of different roles should be interconnected in dedicated ways.

Second, the topologies of existing DCN structures are not suitable for executing MapReduce for the following reason. During the running of MapReduce, a master server needs to control multiple worker servers flexibly. A master server collects the execution information of worker servers that it controls and assigns different tasks to different work servers simultaneously. If a worker server is unable to accomplish its task within a certain time frame, the task is reassigned to another worker server at once. If a master server goes out of order, another candidate master server must take over the current MapReduce. Those properties require that the DCN structure must be fault tolerant and that one master server should connect with as many work servers as possible.

Third, the topological properties of existing DCN structures are not sufficiently suitable for the data management mechanism of MapReduce. Their diameters increase rapidly with the expanding of their scales. For the sake of maintaining data locality [19], one server stores only certain types of data. If a server needs the data stored on another server to execute a task, a data transmission between the two servers is necessary. For a complex MapReduce, there can be numerous data transmissions. This requires that the DCN structure possesses a low diameter for shortening the transmission length between any pair of servers, thus improving the performance of MapReduce.

We can derive from the above analysis that there should be a novel DCN structure that supports MapReduce operations and corresponding data management. However, there are several challenges to be faced in achieving this goal. First, the novel DCN structure should possess good topological properties, such as scalability and reliability. Second, this structure should be suited for running multijob MapReduce programs, including assigning jobs and transferring their results. Finally, it should be fault tolerant when executing MapReduce in practice. We address these challenging issues in detail in the rest of this paper.

3. HFN STRUCTURE

In this section, we propose the physical structure and the construction methodology of HFN. The physical structure of HFN is presented in terms of the smallest recursive unit and the recursive rule.

3.1. Physical structure

HFN is recursively defined in order to be scalable and to meet the design goals of a DCN. The servers and the switches in HFN are interconnected according to the basic procedure of MapReduce. We first present the smallest recursive unit of HFN and then discuss how to generate a higher-level HFN recursively.

(1) Smallest recursive unit

The smallest recursive unit, denoted as $HFN_0(N, M)$, is the basic building block of the whole network topology. Here, N denotes the number of master servers in an HFN_0. It also denotes the number of switches in an HFN_0. M denotes the number of worker servers each switch connects.

An HFN_0 connects N master servers and N switches by means of a bipartite graph [20]. These N master servers and N switches can be regarded as the two-vertex sets of the bipartite graph. Each master server connects to two or three switches. None of the master servers connect to each other directly. Neither do the switches. The structure of the smallest recursive unit looks like a fat tree. The inner nodes at the higher level of a fat tree are replaced by the master servers. The leaf nodes of a fat tree are replaced by the worker servers.

We construct the smallest recursive unit as a fat-tree-like topology to better support MapReduce. There are three advantages for designing the smallest recursive unit in such a way. First,

DOI: 10.1002/cpe

compared with the current smallest recursive units, this structure facilitates treating servers as masters and workers in large-scale data centers. Second, it makes each master server connect with more switches and therefore lets a master server directly control as many worker servers as possible. In our smallest recursive unit, the number of worker servers one hop away from a master server can be up to 144, when 48-port Ethernet switches are employed. For data forwarding hops to be reduced, the value of N should not be large, in general $N \leq 8$. This implies that a master server can control at most 384 worker servers in our smallest recursive unit, larger than that in the current smallest recursive units. Third, fat-tree topology brings high reliability. In such a network, a failed worker is unable to affect any other servers. Moreover, because each worker server connects to multiple master servers, a failed master server will not tear down the worker servers.

(2) Recursive rule

The recursive rule determines the interconnection mode of the recursive units. Let $HFN_i(K, N, M)$ denote a level i recursive unit ($i \geq 1$), which employs $HFN_0(N, M)$ as the smallest recursive unit. K denotes the number of switches at level i in an HFN_i. K is equal to the total number of master servers in an HFN_{i-1}, which denotes the level $i-1$ recursive unit.

HFN uses the generalized hypercube [15] as its recursive rule just like BCube [11]. A number of low-level hypercubes make up a high-level hypercube through interconnecting the nodes that are at the same position in different lower-dimensional hypercubes. In this paper, an HFN_i is constructed by N HFN_{i-1}s, which are interconnected through the K switches at level i.

We employ BCube as our recursive rule because BCube inherits the following advantages of the generalized hypercube. First, this structure provides multiple parallel short paths for each pair of servers. That brings good performance in terms of fault tolerance and load balance. Second, BCube accelerates one-to-several and one-to-all traffic by constructing edge-disjoint complete graphs and multiple edge-disjoint spanning trees [11]. Hence, BCube possesses better topological properties and data forwarding performance than other existing structures and provides a faster and more reliable data exchange for MapReduce.

3.2. Construction method

The number of HFN_{i-1}s in an HFN_i is N. It is easy to derive that an HFN_i consists of N^i HFN_0s. There are N master servers in an HFN_0. Thus, the total number of master servers in an HFN_i is N^{i+1}. We can derive from the recursive rule of HFN that $K = N^i$, and the level i recursive unit can simply be denoted as $HFN_i(N, M)$.

Let $MServer_{i,j}$ denote the jth master server in an HFN_i, where $0 \leq j < N^{i+1}-1$. Consequently, for any master server in an HFN_i, its unique ID can be given by $MServer_{i,j}$. Let $Switch_{i,k}$ denote the kth switch at level i in an HFN_i. Here, $0 \leq k < N^i$ for $i > 0$, and $0 \leq k < N$ for $i = 0$.

To generate a high-level HFN, we have to construct the smallest unit and then expand the smallest unit to any higher-level HFN recursively. We propose the construction method for an HFN_0 and an HFN_i as follows.

To construct an $HFN_0(N, M)$, we label the N switches and the N master servers both from [0,0] to [0,$N-1$]. According to the physical structure of an HFN_0, we connect M worker servers to each switch. Then, we connect $MServer_{0,0}$ and $MServer_{0,1}$ to $Switch_{0,0}$. We connect $MServer_{0,n-1}$, $MServer_{0,n}$, and $MServer_{0,n+1}$ to $Switch_{0,n}$, where $1 \leq n \leq N-2$. Finally, we connect $MServer_{0,N-2}$ and $MServer_{0,N-1}$ to $Switch_{0,N-1}$.

To construct an $HFN_i(N, M)$, we label the N^i switches at level i from [i,0] to [i,N^i-1]. For each of the N HFN_{i-1}s that compose the HFN_i, we label its N^i master servers from [$i-1$,0] to [$i-1$,N^i-1]. According to the recursive rule of HFN, for any k ($0 \leq k < N^i$), we connect every $MSever_{i-1,k}$ in each of the N HFN_{i-1}s to $Switch_{i,k}$ of the HFN_i.

Figures 1 and 2 illustrate the structures of $HFN_1(3, 4)$ and $HFN_i(N, M)$, respectively. We can derive from the construction of HFN that, just like BCube, each master server should be equipped

DOI: 10.1002/cpe

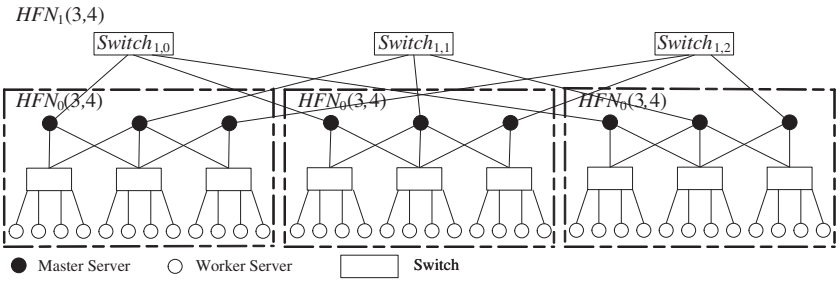

Figure 1. $HFN_1(3,4)$; HFN, hyper-fat-tree network.

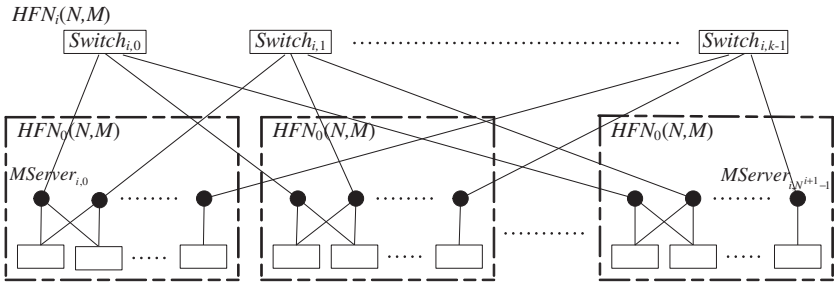

Figure 2. $HFN_i(N,M)$; HFN, hyper-fat-tree network.

with multiple network ports. Fortunately, each of the worker servers, which number in total far more than the master servers, only needs to be equipped with one network port.

4. MAPREDUCE ON HFN

As a famous distributed and cloud computing model, MapReduce has become one of the most important applications running on data centers. In this section, we will discuss how to execute MapReduce on HFN.

4.1. Roles of servers for MapReduce

In a DCN, the servers control and execute the procedure of MapReduce. For us to differentiate the control function from the execution function, this paper divides the servers into master servers and worker servers. The running process of MapReduce on HFN can be regarded as the controlling process of master servers on worker servers. In the smallest recursive unit, every master server controls the worker servers, which connect to the same switches as it does. According to the structure of the smallest recursive unit, each worker server can be controlled by at most three master servers. The roles of master servers and worker servers are described as follows.

Master servers are used for controlling the whole procedure of MapReduce and are responsible for receiving users' MapReduce requests. They record the location information of various data stored on worker servers. The jobs of a complex MapReduce request received by a master server, needing a variety of data, are further assigned to multiple master servers. Each master server that receives a job commands the worker servers under its control to execute the job in terms of predefined programs. Finally, the master server that runs the last job receives the results from worker servers and sends them to the initial master server that received the MapReduce request. If necessary, this master server will start a new round of MapReduce until it meets the request.

Worker servers are used for executing the concrete map and reduce tasks. A worker server stores certain types of data on its local disk. There are two kinds of states for a worker server, including running state and idle state. No matter which state a worker server is in, it can receive map and reduce tasks from any one of its master servers, which connect to the same switches with it.

DOI: 10.1002/cpe

All the map or reduce tasks are executed on the basis of first come, first served. When a worker server accomplishes a map or reduce task, it sends the number of tasks to be performed to its master servers. If the number equals 0, it means that the worker server is in idle state. Otherwise, the worker server is in running state.

4.2. Routing and assigning schemes of MapReduce on HFN

There are four kinds of routing schemes for MapReduce on HFN. The first one is the routing scheme between a master server and its worker servers, used for assigning map and reduce tasks. The second one is the routing scheme between two worker servers that are controlled by the same master server, used for transmitting intermediate data. The third one is the routing scheme among master servers, used for assigning jobs. The fourth one is the routing scheme between two worker servers that belong to different smallest recursive units, used for transmitting the necessary data that are not stored on local disks. Because there are only one or two hops in the first and second routings, which can be addressed only in the smallest recursive unit, this paper mainly focuses on the third and fourth routing schemes.

Let I denote the total number of levels of HFN. From the construction of HFN, we can derive the following theorems for the routing schemes of MapReduce on HFN.

Theorem 1
For any master server $MServer_{I,j}$ in an HFN_I, let d denote the sequence of an HFN_i, which contains $MServer_{I,j}$, in the HFN_I. The value of d is given by

$$d = j/N^{i+1}. \tag{1}$$

Here, j/N^{i+1} is the integer of N^{i+1} dividing j.

Proof
The number of master servers in one HFN_i is N^{i+1}. The sequence of $MServer_{I,j}$ in the HFN_I is j. Hence, in the HFN_I, j/N^{i+1} is the sequence of the HFN_i that $MServer_{I,j}$ belongs to. Theorem 1 is proven. □

Theorem 2
If two master servers, denoted as $MServer_{I,x}$ and $MServer_{I,j}$, connect to the same switch at level $i+1$, while separately belonging to a pair of adjacent HFN_is in an HFN_{i+1}, then we can derive that

$$|x - j| = N^{i+1}. \tag{2}$$

Here, $|x - j|$ denotes the absolute value of x minus j.

Proof
Suppose $Swich_{i+1,k}$ is the switch at level $i+1$, which connects $MServer_{I,x}$ as well as $MServer_{I,j}$. According to the recursive rule, in this pair of adjacent HFN_is, $MServer_{I,x}$ and $MServer_{I,j}$ are the only two master servers that connect to $Swich_{i+1,k}$. The number of switches at level $i+1$ in an HFN_{i+1} is N^{i+1}. Other master servers between $MServer_{I,x}$ and $MServer_{I,j}$ separately connect to other $N^{i+1}-1$ switches. Therefore, the number of master servers between $MServer_{I,x}$ and $MServer_{I,j}$ is $N^{i+1}-1$. This implies that the absolute value of x minus j is N^{i+1}. Theorem 2 is proven. □

Suppose $MServer_{I,x}$ and $MServer_{I,j}$ connect to the same switch at level $i+1$ and separately belong to any two HFN_is in an HFN_{i+1}. Let the sequences of these two HFN_is in the HFN_{i+1} be d_1 and d_2. We can derive from Theorem 2 that

$$|x - j| = |d_1 - d_2| \times N^{i+1}. \tag{3}$$

DOI: 10.1002/cpe

(1) Master-to-master routing for assigning jobs

The routing scheme among master servers depends on the job-assigning scheme of MapReduce services. For our research purposes, so that the load of master servers can be reduced, network traffic lowered, and data locality kept, the jobs of a complex MapReduce procedure are assigned to different master servers. Namely, a master server that receives a multijob MapReduce request sends each job to the *nearest* master server, which controls the worker servers containing the necessary data for the job. In this case, *nearest* means the smallest number of hops between the two master servers. Inspired by the characteristics of distributed file systems [4, 5], we suppose that all input data are stored on the worker servers of a data center.

On the basis of the aforementioned theorems, we propose Algorithms 1 and 2 to implement the master-to-master routing scheme for assigning MapReduce jobs on HFN. Here, we suppose that $MServer_{I,j}$ receives a MapReduce service request, which needs to be assigned to L ($L > 1$) master servers. *FindedServers* is an object list, which records in order those master servers that receive jobs. *Path*, which keeps a routing path from $MServer_{I,j}$ to its master server, is an attribute of each master server in the object list.

Algorithm 1 demonstrates the scheme of assigning jobs. In Algorithm 1, Job_l ($0 \leq l < L$) denotes the jobs that need to be assigned to a master server. Algorithm 1 first finds the master server, denoted as $MServer_{I,y}$, which controls the worker servers with the data required by Job_l. It then finds a routing path from $MServer_{I,j}$ to $MServer_{I,y}$ by invoking Algorithm 2

Algorithm 1 : AssignJobs (int j, int L)

```
1: objectList FindedServers;
2: for   l = 0; l < L; l++
3:   if   the worker servers of MServers_{I,y} hold the data for Job_l;
4:     MServers_{I,y}.Path = {MServer_{I,j}, };
5:     MServers_{I,y}.Path =CreatRouting1(I − 1, j, y);
6:     assign Job_l to MServer_{I,y};
7:     add MServer_{I,y} to FindedServers;
```

Algorithm 2 : CreateRouting1 (int f, int j, int y)

```
1: int g = 0; int x = 0;
2: for   i = f; i ≥ 0; i−−
3:   if   i > 0
4:     if   j/N^{i+1} ≠ y/N^{i+1}
5:       int h = (y − j)/N^{i+1}; // Here, h is the integer of N^{i+1} dividing y − j.
6:       x = j + h × N^{i+1};
7:       add MServer_{I,x} to MServer_{I,y}.Path;
8:       if   x == y
9:         return MServer_{I,y}.Path;
10:      g = i; break;
11:  if   i = 0
12:    if   j − y > 2
13:      for   x = j − 2; x > y; x− =2
14:        add MServer_{I,x} to MServer_{I,y}.Path;
15:    if   y − j > 2
16:      for   x = j + 2; x < y; x+ =2
17:        add MServer_{I,x} to MServer_{I,y}.Path;
18:    add MServer_{I,y} to MServer_{I,y}.Path;
19:    return MServer_{I,y}.Path;
20: CreateRouting1 (g,x,y);
```


DOI: 10.1002/cpe

and assigns the routing path to the *Path* attribute of $MServer_{I,y}$. Finally, it assigns Job_l to $MServer_{I,y}$ and adds $MServer_{I,y}$ to the object list *FindedServers*.

Algorithm 2 is used for identifying a master-to-master routing path for assigning jobs. From level I to level 0, Algorithm 2 recursively records each node in the routing path from $MServer_{I,j}$ to $MServer_{I,y}$. For level I, Algorithm 2 labels $MServer_{I,j}$ and $MServer_{I,y}$ as the source and destination nodes of the routing path, respectively. Then it determines according to Theorem 1 if $MServer_{I,j}$ and $MServer_{I,y}$ connect to the same switch at level I through Theorem 1. If not, according to Theorem 2 and Equation 3, Algorithm 2 records the master server, denoted as $MServer_{I,x}$, which connects to the same switch at level I with $MServer_{I,j}$ and belongs to the same HFN_{I-1} with $MSever_{I,y}$. Algorithm 2 then labels $MServer_{I,x}$ as the new source node, also denoted as $MServer_{I,j}$, and performs the above process again for level $I-1$. This process is recursively performed until $MServer_{I,y}$ is labeled as the new source node or $MServer_{I,j}$ and $MServer_{I,y}$ belong to the same HFN_0. For the latter event, if the number of hops from $MServer_{I,j}$ to $MServer_{I,y}$ is larger than one, Algorithm 2 further records in order minimal master servers in the routing path from $MServer_{I,j}$ to $MServer_{I,y}$. Otherwise, it just records $MServer_{I,y}$ as the last node and returns the whole routing path.

(2) Worker-to-worker routing

When a worker server is executing a map or reduce task, it may need the data stored at another worker server. On the basis of the master-to-master routing scheme, we propose Algorithm 3 as the worker-to-worker routing scheme in HFN.

Algorithm 3 adds two worker servers at the beginning and the end of the routing path between their master servers. We can obtain the routing path from Algorithm 2. Here, $WServer_{j,m1}$ denotes any worker server controlled by $MServer_{I,j}$, and $WServer_{y,m2}$ denotes any worker server controlled by $MServer_{I,y}$. We assume that $WServer_{j,m1}$ and $WServer_{y,m2}$ are not controlled by the same master server.

4.3. Map and reduce on HFN

On the basis of the routing and assigning schemes described above, the jobs of a complex MapReduce are assigned to multiple master servers. These master servers will control a number of worker servers to execute the received jobs. The execution of each job involves the map and reduce operations.

(1) Map on HFN

Suppose that $MServer_{I,y}$ receives a job. The number of map tasks is determined by the number of data chunks that job needs to process. The default mapping approach, which consists of three steps, is one map task for one data chunk. In the first step, $MServer_{I,y}$ chooses some idle or not busy worker servers, named map worker servers, and assigns a map task to each of them. In the second step, map worker servers divide the corresponding input data into intermediate key/value pairs by means of predefined map programs and store the intermediate data on local hard disks. In the third step, map worker servers feed back the types of keys of intermediate data to $MServer_{I,y}$ and then send the number of waiting tasks in their local queues, namely their state information, to the corresponding master servers.

Algorithm 3 : CreateRouting2 (int j, int y, int $m1$, int $m2$)

1: $WServer_{j,m1}.Path = \{WServer_{j,m1}, MServer_{I,j},\}$;
2: $WServer_{j,m1}.Path$ =CreatRouting1$(I-1, j, y)$;
3: add $WServer_{y,m2}$ to $WServer_{j,m1}.Path$;
4: return $WServer_{j,m1}.Path$;

DOI: 10.1002/cpe

(2) Reduce on HFN

The number of reduce tasks is determined by the types of keys of intermediate data. One reduce task can process one or several types of key/value pairs. But one type of key/value pairs is usually processed by only one reduce task. The default reducing approach consists of four steps. In the first step, $MServer_{I,y}$ chooses some idle or not busy worker servers, named reduce worker servers, and assigns a reduce task to each of them. In the second step, according to the types of keys of their received reduce tasks, reduce worker servers fetch the intermediate data from the corresponding map worker servers. In the third step, reduce worker servers merge the same type of key/value pairs by means of predefined reduce programs to generate output values. In the fourth step, reduce worker servers feed back the output values to $MServer_{I,y}$. They also send their state information to the corresponding master servers.

The output data of some jobs might be the input data of other jobs. When $MServer_{I,y}$ has finished its job, it sends the result directly to the master server that receives the next job, namely the next object in the object list *FindedServers*, which is derived from Algorithm 1. The routing scheme between $MServer_{I,y}$ and that master server can be obtained by means of Algorithm 2. The master server that executes the final job forwards its result to $MServer_{I,j}$ through the routing path recorded in its *Path* attribute.

5. FAULT TOLERANCE OF MAPREDUCE ON HFN

Node and link failures are very common during the running of a MapReduce on a DCN. Because a link failure can be regarded as the failure of adjacent nodes, this paper focuses on server and switch failures. The proposed fault-tolerant approaches for MapReduce on HFN not only deal with failures of servers and switches but also address the traffic congestion problem.

5.1. Fault-tolerant routing against failures of master servers and high-level switches

If a master server fails, all the routing paths that passes the master server become unusable. If a switch out of the smallest recursive unit goes wrong, all the master servers that interconnect to the switch become disconnected. In order to avoid the negative impacts of failed servers and switches, we propose a fault-tolerant routing scheme for executing MapReduce on HFN.

For the master-to-master routing for assigning jobs and the worker-to-worker routing, where a master server or a high-level switch goes wrong, we can replace the failed or disconnected master server with an adjacent master server in the same HFN_0. Therefore, we modify Algorithm 2 in order to achieve Algorithm 4. We set *Broken* as a Boolean variable, indicating whether a master server is failed or cannot be connected because of a failed high-level switch.

On the basis of Algorithm 2, before recording $MServer_{I,x}$, Algorithm 4 judges whether $MServer_{I,x}$ is failed or unconnected. If $MServer_{I,x}$ is not in the same HFN_0 with $MServer_{I,y}$ and is failed or unconnected, Algorithm 4 records an adjacent master server of the current source node $MServer_{I,j}$. Then it labels this master server as the new source node, which is also denoted as $MServer_{I,j}$. If $MServer_{I,x}$ belongs to the same HFN_0 with $MServer_{I,y}$ and is failed or unconnected, Algorithm 4 records the master server adjacent to $MServer_{I,x}$ as a node in the whole routing path.

5.2. Fault-tolerant approaches to address failed servers

In the smallest unit of HFN, each worker server connects to only one switch, whereas each switch connects to two or three master servers. Because of the redundant structure of the smallest unit of HFN, when a master server or worker server goes wrong, the other nodes in the same smallest unit might still be usable. The running MapReduce tasks, however, will be influenced by the failures of servers. To address this problem, we propose the following approaches:

(1) Addressing the failure of a master server:
   - As soon as a master server receives a MapReduce request, it sends the request to an adjacent master server as a replica. If the former master server fails to execute the request, the latter one will take over the current MapReduce.

DOI: 10.1002/cpe

Algorithm 4 : CreateRouting3 (int f, int j, int y)

```
1: int g = 0; int x = 0;
2: for  i = f; i ≥ 0; i−−
3:   if  i > 0
4:     if  j/N^{i+1} ≠ y/N^{i+1}
5:       int h = (y − j)/N^{i+1};
6:       x = j + h × N^{i+1};
7:       if  MServer_{I,x}.Broken == true
8:         if  (j + 1)%N == 0
9:           x = j − 1; // Here, (j + 1)%N is the remainder of N dividing j + 1. If this value
     is equal to zero, MServer_{I,j} is the last master server in its HFN_0.
10:        else  x = j + 1;
11:      add MServer_{I,x} to MServer_{I,y}.Path;
12:      g = i; break;
13:  if  i = 0
14:    if  j − y > 2
15:      for  x = j − 2; x > y; x− =2
16:        if  MServer_{I,x}.Broken == true
17:          x = x + 1;
18:        add MServer_{I,x} to MServer_{I,y}.Path;
19:    if  y − j > 2
20:      for  x = j + 2; x < y; x+ =2
21:        if  MServer_{I,x}.Broken == true
22:          x = x − 1;
23:        add MServer_{I,x} to MServer_{I,y}.Path;
24:    add MServer_{I,y} to MServer_{I,y}.Path;
25:    return MServer_{I,y}.Path;
26: CreateRouting3 (g,x,y);
```

- Assume a master server $MServer_{I,x1}$ receives a MapReduce job of the running MapReduce. Then $MServer_{I,x1}$ sends a replica to an adjacent master server denoted as $MServer_{I,x2}$, in the same HFN_0. When $MServer_{I,x1}$ is unable to send the result to a master server that executes the next MapReduce job within a predefined threshold time, $MServer_{I,x2}$ will be required to carry out and return the output.

(2) Addressing the failure of a worker server:

- The master server sets a threshold for the executing time of each map and reduce task. If a worker server does not respond with its state information within the threshold time, it will be regarded as a failed server. The map or reduce task allocated to the failed server will be reassigned to another available worker server.

5.3. Fault-tolerant approaches to address failed switches

Each switch in a data center connects a number of servers. A failed switch may affect all the servers that connect to it.

Fortunately, in HFN, when a switch out of the smallest recursive units goes wrong, all servers connecting to it might be usable. The running MapReduce tasks that use this switch, however, will be influenced. To address this problem, we propose the fault-tolerant approaches for running tasks in such condition as follows:

(1) Approach of assigning the MapReduce jobs influenced by a failed switch:

- If a master server receives a multijob MapReduce request but is unable to assign most of the jobs to other master servers because of a connection to a failed switch, then the MapReduce

DOI: 10.1002/cpe

request will be executed by an adjacent master server, which stores the corresponding replica.

(2) Approach of transmitting result messages of the jobs influenced by a failed switch:
   - A master server cannot utilize the failed switch to transmit any result message. If a master server is unable to receive an acknowledgment within a threshold time after sending out a result message, an adjacent master server will send out that result message.

   If a switch in a smallest recursive unit fails, all worker servers connected to it will be unusable. In that case, the approach proposed in Section 5.2 can be employed to improve the fault-tolerant capability.

6. TOPOLOGICAL PROPERTIES OF HFN

In this section, we analyze the topological properties of HFN, including network size, network diameter, and bisection width. We then compare HFN with BCube and FiConn, which are two representative DCN structures defined recursively.

6.1. Network size

The size of a DCN depends on the amount of servers it is able to accommodate.

Theorem 3
The total number of servers in an $HFN_i(N, M)$ is

$$(M + 1) \times N^{i+1}. \tag{4}$$

Proof
In an HFN_i, the number of master severs is N^{i+1}. There are $M \times N$ worker servers in each HFN_0 and N^i HFN_0s in an HFN_i. Therefore, the total number of worker servers in an HFN_i is $M \times N^{i+1}$. Thus, the total number of servers in an HFN_i is $(M + 1) \times N^{i+1}$, which is much more than that of a level-i BCube. Theorem 3 is proven. □

Figure 3 shows the number of servers that FiConn, BCube, and HFN can accommodate, when $N = 4$ and $M = 8$. We can derive that as i increases, the size of HFN is several times larger than that of FiConn and BCube.

The number of switches reflects the cost to construct a DCN.

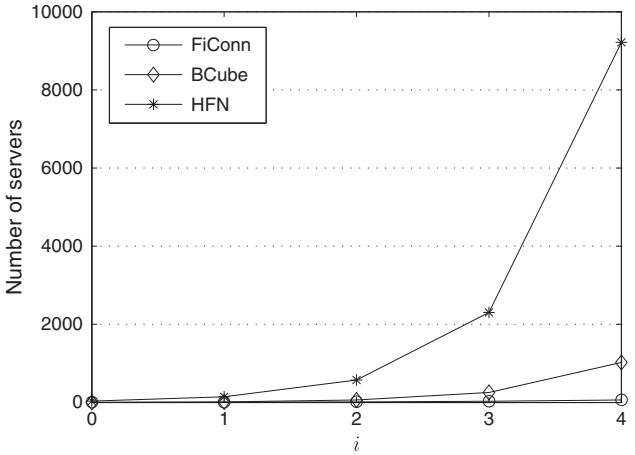

Figure 3. The network sizes of FiConn, Bcube, and hyper-fat-tree network (HFN) in the case that $N = 4$ and $M = 8$.

DOI: 10.1002/cpe

Theorem 4
The number of switches in an $HFN_i(N, M)$ is

$$(i + N) \times N^i. \tag{5}$$

Proof
There are N^i HFN_0s in an HFN_i and N switches in each HFN_0, so the number of switches in the smallest recursive units equals $N \times N^i$. There are N^i switches at each level except level 0, so the number of switches out of the smallest recursive units equals $i \times N^i$. Thus, the total number of switches in an HFN_i is $(i + N) \times N^i$. Theorem 4 is proven. □

Figure 4 plots the percentage of switches in FiConn, BCube, and HFN, when $N = 4$ and $M = 8$. The percentage of switches in BCube increases rapidly with levels. When the value of i is not large, the percentage of switches in HFN is lower than that of the other two structures. This implies that HFN needs much less switches than the other two structures so that the same number of servers can be interconnected. Hence, the cost for constructing HFN is lower than that of other structures.

6.2. Network diameter

The diameter of a DCN is the maximum number of hops in the shortest path between any pair of servers. A shorter diameter results in a faster data exchange.

Theorem 5
The diameter of an $HFN_i(N, M)$ is given by

$$i + 1 + \lfloor \frac{N}{2} \rfloor. \tag{6}$$

Here, $\lfloor N/2 \rfloor$ is the rounding of 2 dividing N.

Proof
The longest routing path between any two HFN_0s takes one hop in each level of an HFN_i. Therefore, the maximum number of hops between any two master servers is equal to i, when the two master servers belong to different HFN_0s. In an HFN_0, the minimum and maximum hops between a master server and a worker server are equal to 1 and $\lfloor N/2 \rfloor$, respectively. So the length of the maximum path between any two work servers is $i + 1 + \lfloor N/2 \rfloor$, which also denotes the diameter of an HFN_i. Theorem 5 is proven. □

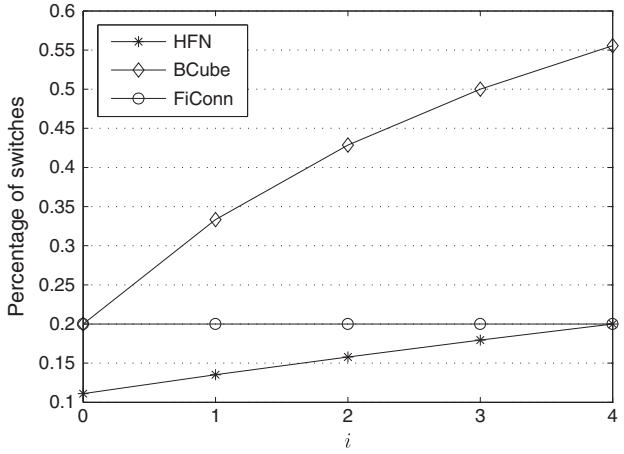

Figure 4. The percentage of switches in FiConn, Bcube, and hyper-fat-tree network (HFN) when $N = 4$ and $M = 8$.

DOI: 10.1002/cpe

Figure 5 illustrates the comparison result among the diameters of FiConn, BCube, and HFN in the case that $N = 4$. It shows that the diameters of HFN and BCube increase smoothly with the increase of i, and the difference between the diameters of HFN and BCube is relatively small when N is not large.

6.3. Bisection width

The bisection width of a DCN is the minimum number of links that can be removed to break the DCN into two approximately equal-sized disconnected networks. A large bisection width implies high network capacity and a more resilient structure against failures.

Theorem 6
When N is an even integer, the bisection width of an $HFN_i(N, M)$ is given by

$$\begin{cases} \frac{N^{i+1}}{2} & i > 0, \\ 2 & i = 0 \end{cases}. \tag{7}$$

Proof
According to the structure of HFN, one HFN_i is constructed by N HFN_{i-1}s with N^i switches at level i. Each of the N^i switches has N links connecting to the N HFN_{i-1}s. When $i > 0$ and N is an even integer, one HFN_i can be divided into two equal parts by removing $N/2$ links from each of the N^i switches. Each part consists of $N/2$ HFN_{i-1}s. Hence, the bisection width of an HFN_i is $(N/2) \times N^i$, namely $N^{i+1}/2$. From the structure of an HFN_0, we can derive that, for the bisection of an HFN_0, two links need to be removed in the case that N is an even integer. Theorem 6 is proven. □

Theorem 7
When $i > 0$ and N is an odd integer, the lower bound on the bisection width of an $HFN_i(N, M)$ is given by

$$2 + \frac{N-1}{2} \times \sum_{g=1}^{i} N^g. \tag{8}$$

Proof
When $i > 0$ and N is an odd integer, so that one HFN_i can be bisected, each of the N^i switches at level i needs at least $(N-1)/2$ links to be removed, and one of the N HFN_{i-1}s needs to

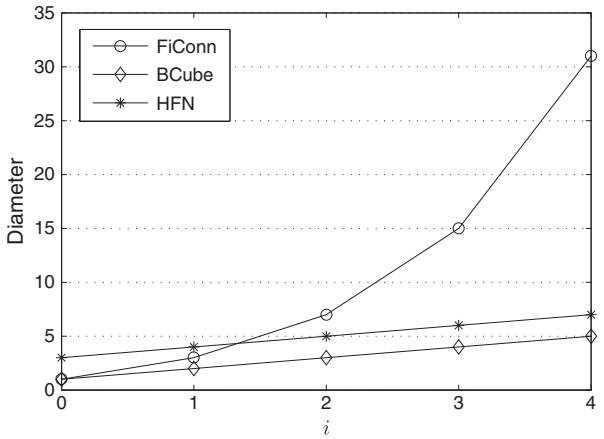

Figure 5. The diameters of FiConn, Bcube, and hyper-fat-tree network (HFN) in the case that $N = 4$ and $M = 8$.

DOI: 10.1002/cpe

be bisected. To bisect this HFN_{i-1}, we need to remove at least $(N-1)/2$ links in each of its N^{i-1} switches at level $i-1$ and bisect one of its N HFN_{i-2}s. This process repeats until $i=0$. Furthermore, at least two links need to be removed to bisect an HFN_0. Thus, under these circumstances, the lower bound of the bisection width of an HFN_i is $2+\sum_{g=1}^{i}[(N-1)/2]\times N^g$, namely $2+[(N-1)/2]\times\sum_{g=1}^{i}N^g$. Theorem 7 is proven. □

Figures 6 and 7 show the comparison result among the lower bounds on bisection width of FiConn, BCube, and HFN in the case that $N=6$ and $N=5$, respectively. When N is an even integer, the bisection width of HFN equals to that of BCube. When N is an odd integer, the lower bound of the bisection width of HFN is a little more or less than that of BCube. So we can hardly establish their difference from Figure 7. However, the bisection width of HFN is larger than that of FiConn with the increase of i, no matter if N is an even or odd integer.

Table I lists the topological properties of FiConn, BCube, and HFN. From the preceding comparison and analysis, we can derive that HFN connects many more servers at the cost of a lower diameter and larger bisection width, irrespective of the value of i. Therefore, HFN exhibits better performance than other typical recursively defined topologies.

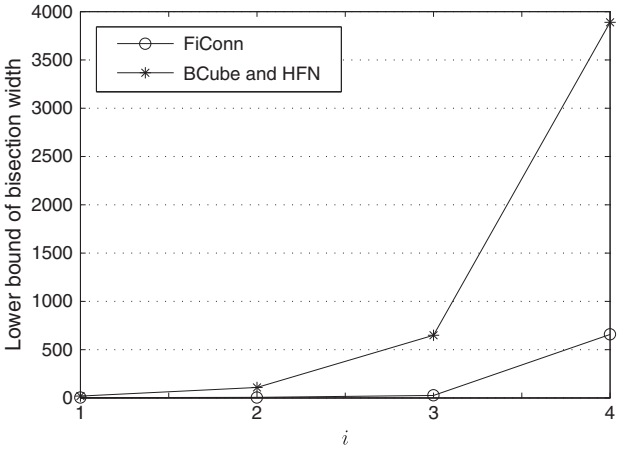

Figure 6. The lower bound of bisection width of FiConn, Bcube, and hyper-fat-tree network (HFN) in the case that $N=6$.

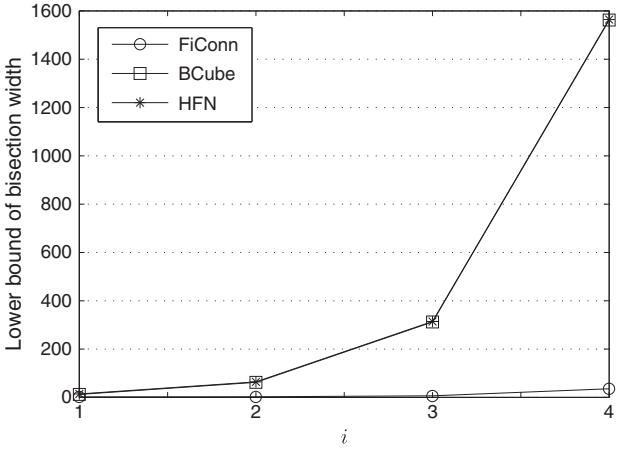

Figure 7. The lower bound of bisection width of FiConn, Bcube, and hyper-fat-tree network (HFN) in the case that $N=5$.

DOI: 10.1002/cpe

Table I. Comparison of topological properties.

Property	FiConn	BCube	HFN
Diameter	$2^{i+1}-1$	$i+1$	$i+1+(N/2)$
Number of servers	$2^{i+2}\times(N/4)^{2^i}$	N^{i+1}	$(M+1)\times N^{i+1}$
Number of switches	$\left[2^{i+2}\times(N/4)^{2^i}\right]/N$	$(i+1)\times N^i$	$(i+N)\times N^i$
Bisection width (N is even)	$(N/4)^{2^i}$	$N^{i+1}/2$	$N^{i+1}/2$
Bisection width (N is odd)	$(N/4)^{2^i}$	$[(N-1)/2]\times\sum_{g=0}^{i}N^g$	$2+[(N-1)/2]\times\sum_{g=1}^{i}N^g$

7. EXECUTION TIME OF MAPREDUCE ON HFN

According to the popular distributed file systems [4, 5], this paper reasonably supposes that all the data are stored at worker servers. Thus, the execution time of map and reduce tasks will be much longer than that of the communication between servers. For ease of explanation, this paper focuses on the execution time of MapReduce and omits the transmission time of necessary packets between servers.

7.1. Execution time

The execution time of a MapReduce procedure depends on the execution time of each job, which further depends on the execution time of each map and reduce task performed by a worker server. The amount of data processed by a task is fixed (64M or 128M) [8]. The execution time of a task will also be fixed in a homogeneous DCN. Consequently, the execution time of a MapReduce procedure is finally determined by the task execution efficiency of each worker server. For a data center, the arrival of users' MapReduce requests is a stochastic progress; thus, map and reduce tasks arrive to a worker server in a stochastic manner as well. For a worker server, tasks are executed according to a first-come-first-served rule. Therefore, we employ a queueing theory to calculate the expected mean execution time of a MapReduce procedure.

Assume that the arrival process of map and reduce tasks to a worker server follows the Poisson distribution. Let the arrival rates of map and reduce tasks be λ_1 and λ_2, respectively. Assume that the service time of a map or reduce task follows a negative exponential distribution. Let the mean service time of a map task be $1/\mu_1$ and that of a reduce task be $1/\mu_2$.

Theorem 8
Let t denote the mean execution time of a job, including the waiting time and the service time, then t is given by

$$t=\frac{1}{\mu_1-\lambda_1}+\frac{1}{\mu_2-\lambda_2}. \tag{9}$$

Proof
According to the queueing theory, the mean execution time for a map task is $1/(\mu_1-\lambda_1)$. All the map tasks in a map phase are performed in a parallel manner; hence, the mean execution time of a map phase equals $1/(\mu_1-\lambda_1)$. Similarly, the mean execution time of a reduce phase equals $1/(\mu_2-\lambda_2)$. A job includes one map phase and one reduce phase, and the reduce phase starts only when the map phase is finished [8, 21]. Theorem 8 is proven. □

DOI: 10.1002/cpe

For a complex MapReduce procedure, some jobs are parallel, whereas others are sequential. Let C denote the number of jobs in a complex MapReduce procedure. We define the weighted adjacency matrix of the jobs as $U = (u_{x,y})$,

$$u_{x,y} = \begin{cases} 1 & Job_x \text{ is immediately followed by } Job_y \\ 0 & Job_x \text{ is not immediately followed by } Job_y \end{cases}, \tag{10}$$

where $0 \leq x < C$ and $0 \leq y < C$.

Let $V = (v_{x,y})$ denote a weighted reachability matrix of the jobs. It records the number of jobs, which are executed one by one from Job_x to Job_y. From the Floyd algorithm [22], we define $v_{x,y}$ in Algorithm 5. So that the value of $v_{x,y}$ can be achieved, Algorithm 5 traverses each job from Job_x to Job_y and accumulates the corresponding value in the weighted adjacency matrix of that job.

Theorem 9

Let T denote the mean execution time of a MapReduce procedure on HFN; then T is given by

$$T = \max_{\substack{0 \leq x < C \\ 0 \leq y < C}} \{v_{x,y}\} \times t. \tag{11}$$

Proof

According to Algorithm 5, $\max\{v_{x,y}\}$ records the number of jobs that are executed one by one from the first job to the last job. Therefore, the execution time of MapReduce equals the sum of the execution times of these jobs. The mean execution time of a single job is t. Thus, the mean execution time of MapReduce on HFN is $\max\{v_{x,y}\} \times t$. Theorem 9 is proven. □

7.2. Fault-tolerant execution time

Let P_m, P_w, and P_s denote the probabilities that a master server, a worker server, or a switch fails when they are executing a MapReduce procedure, respectively.

Theorem 10

Let P_1 denote the probability that a master server keeps on working when it is executing a MapReduce procedure; then P_1 is given by

$$P_1 = \begin{cases} 1 - P_m & i = 0 \\ (1 - P_m) \times (1 - \sum_{n=1}^{i} (\binom{i}{n} \times P_s^n)) & i > 0 \end{cases}. \tag{12}$$

Proof

Let $MServer_{i,j}$ denote any master server in HFN. When $i = 0$, $MServer_{i,j}$ only connects to the switches at level 0, whose failures do not affect $MServer_{i,j}$. Hence, the probability that $MServer_{i,j}$ keeps on working is equal to the probability that $MServer_{i,j}$ does not fail, namely $1 - P_m$. When $i > 0$, $\binom{i}{n} \times P_s^n$ denotes the probability that n of i switches fail, where those switches are out of HFN_0 and connect to $MServer_{i,j}$. Then the probability that these i switches keep on working is $1 - \sum_{n=1}^{i} (\binom{i}{n} \times P_s^n)$. It is clear that $MServer_{i,j}$ can keep on working only if $MServer_{i,j}$ and all these i switches do not fail. Therefore, the probability that $MServer_{i,j}$ keeps on working is $(1 - P_m) \times (1 - \sum_{n=1}^{i} (\binom{i}{n} \times P_s^n))$. Theorem 10 is proven. □

Algorithm 5 : CreateMatrixV

```
1: for   z = 0; z < C; z++
2:   for   x = 0; x < C; x++
3:     for   y = 0; y < C; y++
4:       v_{x,y} = u_{x,y} = max{u_{x,y}, u_{x,z} + u_{z,y}};
```


DOI: 10.1002/cpe

Theorem 11
Let P_2 denote the probability that a worker server keeps on working when it is executing a MapReduce procedure. We can derive that

$$P_2 = (1 - P_w) \times (1 - P_s). \tag{13}$$

Proof
$1 - P_w$ and $1 - P_s$ are the probabilities that a worker server and a switch do not fail, respectively. Consider that a worker server only connects to one switch. Only when both the worker server and the switch do not break down can the worker server keep on working. So the probability that a worker server keeps on working is given by $(1 - P_w) \times (1 - P_s)$. Theorem 11 is proven. □

Theorem 12
When node failures are taken into account, let t' denote the mean execution time of a job; then t' is given by

$$t' = \frac{t}{P_2}. \tag{14}$$

Proof
Suppose there are M map tasks in the map phase of a job, which are run by M worker servers. Because $1 - P_2$ is the probability that a worker server cannot keep on working, $M \times (1 - P_2)$ of the M map tasks need to be reperformed. Furthermore, $M \times (1 - P_2)^2$ of these $M \times (1 - P_2)$ map tasks need to be performed for the third time. The rest of the processes are performed in the same manner. As a result, according to Theorem 8, the mean execution time of a map phase is $M \times \sum_{g=0}^{\infty}(1-P_2)^g \times 1/(\mu_1 - \lambda_1) \times 1/M$, namely $1/\mu_1 - \lambda_1 \times 1/P_2$. Similarly, the mean execution time of a reduce phase is $1/(\mu_2 - \lambda_2) \times 1/P_2$. Thus, the mean execution time of a job is t/P_2. Theorem 12 is proven. □

Theorem 13
When node failures are taken into account, let T' denote the mean running time of a MapReduce procedure on HFN. Then T' is given by

$$T' = \max_{\substack{0 \le x < C \\ 0 \le y < C}} \{v_{x,y}\} \times \frac{t'}{P_1}. \tag{15}$$

Proof
According to Theorem 9, the execution time of a MapReduce on HFN equals the sum of the execution times of $\max\{v_{x,y}\}$ jobs. Because $1 - P_1$ is the probability that a master server cannot keep on working, $\max\{v_{x,y}\} \times (1 - P_1)$ of the $\max\{v_{x,y}\}$ jobs need to be reperformed. Furthermore, $\max\{v_{x,y}\} \times (1 - P_1)^2$ of these $\max\{v_{x,y}\} \times (1 - P_1)$ jobs need to be performed for the third time. The rest of the processes are performed in the same manner. As a result, when node failures are taken into account, the running time of a MapReduce is given by $\max\{v_{x,y}\} \times t' \times \sum_{g=0}^{\infty}(1 - P_1)^g$, namely $\max\{v_{x,y}\} \times \frac{t'}{P_1}$. Theorem 13 is proven. □

We substitute Equations 9 and 14 into Equation 15 and then derive

$$T' = \max_{\substack{0 \le x < C \\ 0 \le y < C}} \{v_{x,y}\} \times \left(\frac{1}{\mu_1 - \lambda_1} + \frac{1}{\mu_2 - \lambda_2} \right) \times \frac{1}{P_1 \times P_2}. \tag{16}$$

Here, P_1 and P_2 are given by Equations 12 and 13, respectively.

DOI: 10.1002/cpe

7.3. Case study

For the case study of our research, a test bed is built. It consists of four Dell PowerEdge 2900 (Dell, Inc., Austin, TX, USA) servers acting as master servers and 16 Dell Vostro 420 commodity computers acting as worker servers. These servers are interconnected by six 8-port Linksys EtherFast Gable/DSL (Linksys by Cisco, Irvine, CA, USA) mini switches obeying the structure of $HFN_1(2, 4)$.

To show that our method is reasonable for complex MapReduce with multijob, we employ the enumerating triangles [16] as an example. Enumerating triangles are a kind of MapReduce applications in graph theory used for enumerating all the triangles in a large graph. It includes four map phases and four reduce phases, namely four jobs.

As discussed in [16], the procedure in our case can be introduced as follows. The input file records the edges of a graph, and every edge is represented by its two vertices. The first map phase comprises three tasks, which send all the vertices and their adjacent edges as the keys and values of intermediate data. The first reduce phase comprises two tasks, which calculate the degree of each vertex. The second map phase comprises two tasks, which send the edges and the degree of a vertex of each edge as the keys and values. The second reduce phase comprises three tasks, which calculate the degrees of the two vertices for each edge. The third map phase comprises three tasks, which send the low-degree vertex of each edge and the edges as the keys and values. The third reduce phase comprises two tasks, each of which calculates every pair of adjacent edges whose common vertex is their low-degree vertex. The fourth map phase comprises two tasks, which send the edges as keys and values. They also send the two nonadjacent vertices of the obtained adjacent edges and these adjacent edges as keys and values. Finally, the fourth reduce phase comprises three tasks, which calculate the triangles.

According to the definitions in Section 7.1, μ_1 and μ_2 separately denote the mean number of map and reduce tasks that a worker server performs in a unit time. In our case, two object lists, denoted as *MapTimeList* and *ReduceTimeList*, are set for each worker server. They respectively record the execution times of map and reduce tasks that a worker server implements.

Algorithm 6 shows the scheme of achieving μ_1, where $Time_{a,b}$ is the element of $MapTimeList_a$ and denotes the execution time of the bth map task performed by the ath worker server. The scheme of achieving μ_2 can be given by similar pseudo-codes.

For each worker server, Algorithm 6 first calculates the total number of map tasks that have been executed and their total execution time and then calculates the mean number of map tasks executed in a unit time by each worker server. Finally, Algorithm 6 adds together all the number of map tasks executed in a unit time by every worker server and divides the value of the sum by the total number of worker servers, so as to achieve the value of μ_1.

Because there are four serial jobs in our case, according to Equation 10 and Algorithm 5, we can easily derive that $\max\{v_{x,y}\} = 4$. When second is set as time unit and the case has been run for 20 times, we derive that the values of μ_1 and μ_2 are 4.5 and 5.3. Suppose $P_m = 0.01$, $P_w = 0.02$, and $P_s = 0.01$, according to Equations 12 and 13, we obtain $P_1 = 0.98$ and $P_2 = 0.97$. In this case, the variation of T' along with the increase of λ_1 and λ_2 can be illustrated by Figures 8 and 9, respectively.

Algorithm 6

```
1: float μ1 = 0;
2: for  a = 0; a < 16; a++
3:    int Count_a = 0; float Time_a = 0
4:    foreach  Time_{a,b} in MapTimeList_a
5:       if  Time_{a,b} > 0
6:          Time_a = Time_a + Time_{a,b};
7:          Count_a++;
8:    μ1 = μ1 + Count_a/Time_a;
9: return μ1 = μ1/16;
```


DOI: 10.1002/cpe

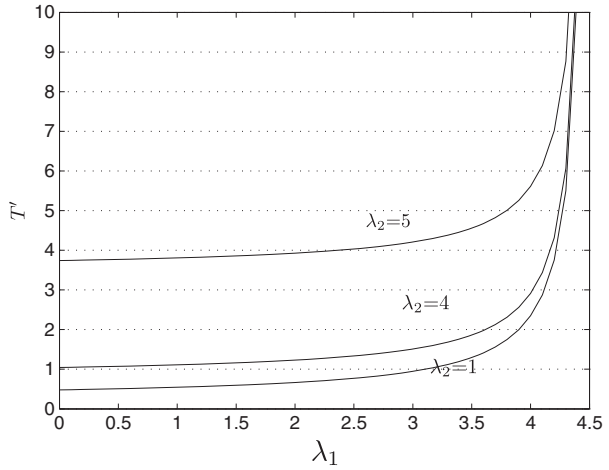

Figure 8. The variation of T' along with the increase of λ_1.

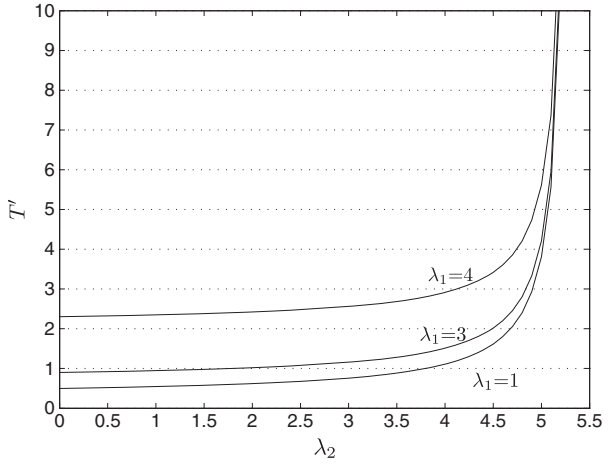

Figure 9. The variation of T' along with the increase of λ_2.

The values of λ_1 and λ_2 depend on the frequency of MapReduce service requests. If they are larger than the values of μ_1 and μ_2, the current data center is unable to support the growth of users' MapReduce service requirements. Hence, the scale of the DCN needs to be expanded, namely the data center needs to be equipped with more servers. Figures 8 and 9 indicate that, when the value of μ_1 minus λ_1 or the value of μ_2 minus λ_2 is larger than 1, T' is able to remain at relatively low values. From the case, we can derive that even with node faults and low recursive level, HFN is competent for MapReduce.

8. DATA FORWARDING PERFORMANCE OF HFN FOR MAPREDUCE

In this section, we evaluate the data forwarding performance of MapReduce on HFN, including the throughput and bandwidth, to further prove that our method is feasible in practice.

8.1. Data forwarding performance

A DCN mainly consists of servers, switches, and links. In a hierarchical data center, such as BCube and HFN, servers are used for not only executing map and reduce tasks but also forwarding necessary data. However, the servers are not network devices specially designed for data forwarding. The

DOI: 10.1002/cpe

data forwarding capacity of servers is usually lower than that of switches and links. Consequently, in this work, we assume that switches and links can provide sufficient bandwidth by considering bandwidth limitation on the servers only.

We performed our simulation for evaluating data forwarding performance as follows. We assumed that each server can forward only one data packet within an extremely short period of time. According to literature [11], in a hierarchical data center, a server forwards at 750 Mbit/s in an all-to-all traffic pattern. The maximum transmission unit is usually set to 1.5 kB for a commodity computer. Thus, on this condition, we can easily calculate that the short period is 1.6×10^{-5} s. In such a period, a server with a packet for transmitting chooses an adjacent server as the destination server, according to our routing scheme. Here, the adjacent server refers to a server that connects to the same switch with the former server. If this destination server is available for receiving a packet, the former server will forward that packet to the destination server successfully. Thus, the destination server will be unavailable for receiving any other packets in this short period. If there is a third server having a packet for transmitting to the same destination server in the current short period, it has to wait until the destination server is available in another short period. We vary the number of servers that can simultaneously generate packets to calculate the best possible data forwarding performance. This procedure is recursively performed in our simulation so that data forwarding performance can be evaluated on a steady working condition.

To compare the data forwarding throughput of HFN with that of BCube, we performed the above simulation in the case that $N = 4$ and $M = 4$ and i varies from 0 to 4. For each value of i, we repeat our simulation 30 times to obtain the mean and range of maximum data forwarding throughput, as shown in Figure 10. We derived that the maximum data forwarding throughput of HFN is much higher and increases more rapidly than that of BCube. The reason for that is, with the same number of levels, HFN involves $M \times N^{i+1}$ servers more than BCube.

In order to further compare the data forwarding throughput of HFN with that of BCube when the two structures hold the same number of servers, we calculated the variations of their data forwarding throughput along with the number of servers. Figure 11 illustrates the result, where the total number of servers varies from 4 to 625. We can find that the data forwarding throughput of the two structures does not strictly increase with the number of servers. For both HFN and BCube, the throughput of 81 servers is lower than that of 64 servers. This is because data forwarding throughput depends not only on the number of servers but also on the number of hops for data forwarding between a pair of servers. 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, HFN and BCube with 81 servers have one level more than those with 64 servers. Given a fixed number of servers in a hierarchical data

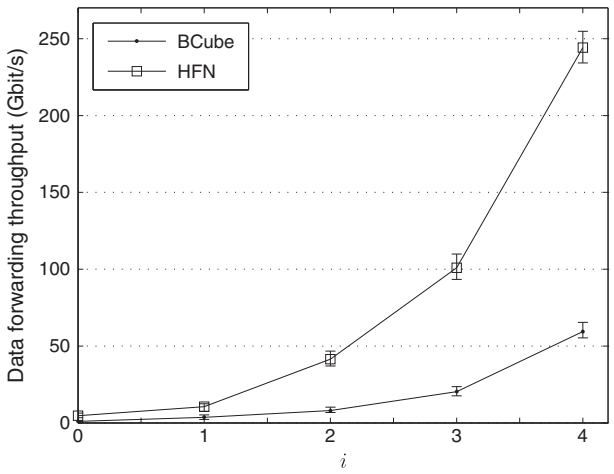

Figure 10. Variation of maximum data forwarding throughput along with the total number of levels; HFN, hyper-fat-tree network.

DOI: 10.1002/cpe

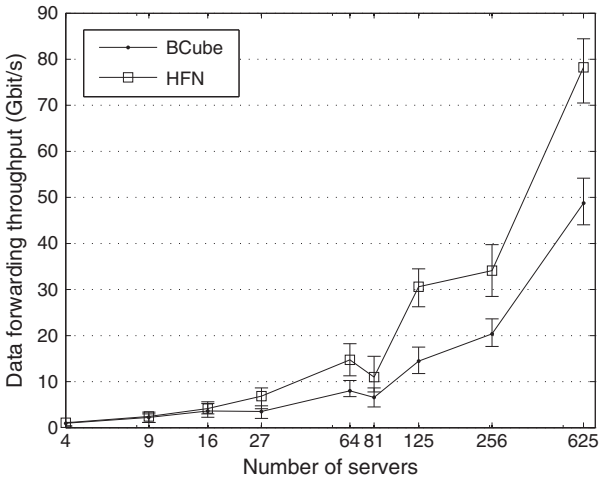

Figure 11. Variation of maximum data forwarding throughput along with the total number of servers; HFN, hyper-fat-tree network.

center, it is crucial to make a trade-off between the number of levels and the number of servers in the smallest recursive unit, so as to achieve the desired performance. As shown in Figure 11, in the case of possessing the same number of servers, the data forwarding throughput of HFN increases rapidly and is higher than that of BCube. This indicates that HFN can better support throughput-hungry MapReduce applications on hierarchical data centers.

To evaluate the bandwidth between servers, we calculated the mean of the maximum bandwidth that each server can achieve for sending data to another server through maximum number of hops, subject to the constraint that data forwarding throughput remains at the maximum value. Figure 12 illustrates the result when $N = 4$ and $M = 4$ and $i = 2$, namely there are 320 servers. As shown in Figure 12, most values of the bandwidth are larger than 0.4 Gbit/s and less than 0.7 Gbit/s. This variance is acceptable for MapReduce applications. In reality, different servers store different types of data, and Internet service providers may store popular data on certain servers to save power [23]. Hence, some of the map or reduce tasks in a MapReduce procedure process and generate more data

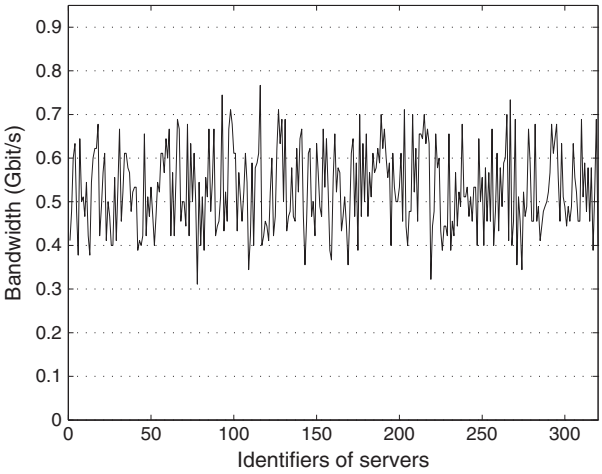

Figure 12. Mean of the maximum bandwidth that each server can achieve for sending data to another server through maximum number of hops, when there are 320 servers and data forwarding throughput remains at the maximum value.

DOI: 10.1002/cpe

than other tasks. Overall, the result of our simulation implies that the bandwidth between servers is sufficient and can be evenly distributed on HFN.

8.2. Fault-tolerant data forwarding performance

We evaluated the data forwarding performance of HFN under given failure rates of links and nodes, to further validate the fault-tolerant capability of our methodology in practice. Here, we regard a link failure as the failure of adjacent nodes and conduct our evaluation according to the analysis discussed in Section 7.2.

We modified the above simulation to evaluate the fault-tolerant data forwarding performance. In each short period, a server with 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 packet for transmitting to the same destination server in the same short period, it has to wait until the destination server is available in another upcoming period. We calculate the probabilities that a master server and a worker server keep on working in a MapReduce procedure according to Theorems 10 and 11, respectively.

We compared the maximum data forwarding throughput of HFN in three cases: ($P_1 = 0.98$, $P_2 = 0.97$), ($P_1 = 0.96$, $P_2 = 0.95$), and ($P_1 = 0.94$, $P_2 = 0.93$). For each case, we repeat the modified simulation 30 times to obtain its mean value. Figure 13 indicates the variation of maximum data forwarding throughput along with the number of servers in these cases. We can derive an interesting result from Figure 13. When the number of servers is relatively small or large, there is no significant difference among the data forwarding throughput in these cases. This is because the throughput is very low when there are a small number of servers, and the difference is not notable. When there are a large number of servers, HFN has good fault-tolerant capability, so the difference is also small. Consequently, with enough servers, HFN can provide satisfied data forwarding throughput against failures of links and nodes.

Because of the failures of links and nodes, we recalculated the mean of the maximum bandwidth that each server can achieve for sending data to another server through the maximum number of hops. Figure 14 illustrates the result when there are 320 servers and $P_1 = 0.95$, $P_2 = 0.94$. We can derive that most values of the bandwidth are larger than 0.3 Gbit/s and less than 0.58 Gbit/s. Although the probability that each server cannot keep on working is 0.05 or 0.06, which are still high in practice, the values of the bandwidth is only 0.1 Gbit/s lower than that shown in Figure 12. Hence, the bandwidth between servers is abundant against failures of links and nodes. Moreover, the range of variance, as shown in Figure 14, is less than that shown in Figure 12. Therefore, the bandwidth between servers can be evenly distributed on HFN under high failure rates of links and nodes.

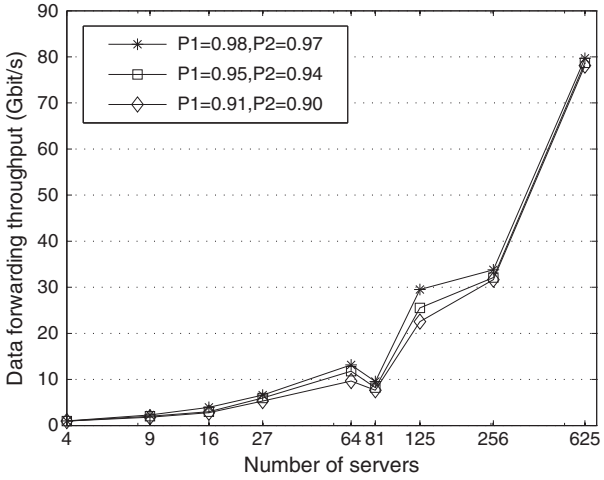

Figure 13. Variation of maximum data forwarding throughput along with the total number of servers considering node and link failures.

DOI: 10.1002/cpe

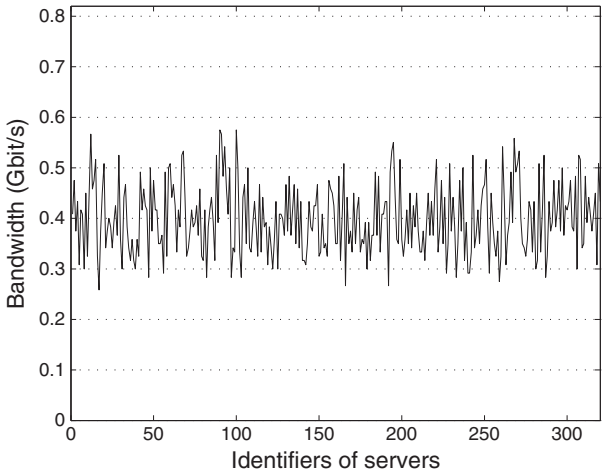

Figure 14. Mean of the maximum bandwidth that each server can achieve for sending data to another server through maximum number of hops, when there are 320 servers and $P_1 = 0.95$, $P_2 = 0.94$.

9. CONCLUSION

Although several novel DCN structures have been proposed to improve the topological properties of data centers, they do not match well with the specific requirements of some dedicated applications. This paper presents a MapReduce-supported DCN structure, named HFN. Through comprehensive analysis and evaluation, we showed that HFN is a reliable and scalable structure with excellent topological properties and data forwarding performance. It is proven that HFN is competent for MapReduce under node failures. Following this work, we plan to study the implementation of distributed file systems on HFN.

ACKNOWLEDGEMENTS

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, and no. 61070216.

REFERENCES

1. Greenberg A, Hamilton JR, Jain N, Kandula S, Kim C, Lahiri P, Maltz DA, Patel P, Sengupta S. VL2: a scalable and flexible data center network. *ACM SIGCOMM Computer Communication Review* 2009; **39**(4):51–62.
2. Greenberg A, Hamilton J, Maltz DA, Patel P. The cost of a cloud research problems in data center networks. *ACM SIGCOMM Computer Communication Review* 2009; **39**(1):68–73.
3. Guo C, Wu H, Tan K, Shiy L, Zhang Y, Lu S. DCell: a scalable and fault-tolerant network structure for data centers. *Proceedings of the ACM SIGCOMM*, Aug. 2008; 75–86.
4. Ghemawat S, Gobioff H, Leung ST. The Google file system. *Proceedings of the ACM 19th Symposium on Operating Systems Principles*, Dec. 2003; 29–43.
5. Borthakur D. The Hadoop distributed file system: architecture and design. http://hadoop.apache.org/core/docs/current/hdfsdesign.pdf [accessed on March 2010].
6. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: a distributed storage system for structured data. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006; 205–218.
7. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Jun. 2007; 59–72.
8. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2004; 137–150.
9. Al-Fares M, Loukissas A, Vahdat A. A scalable, commodity data center network architecture. *Proceedings of the ACM SIGCOMM*, Aug. 2008; 63–74.

DOI: 10.1002/cpe

10. Li D, Guo C, Wu H, Tan K, Zhang Y, Lu S. FiConn: using backup port for server interconnection in data centers. *Proceedings of the IEEE INFOCOM*, Apr. 2009; 2276–2285.
11. Guo C, Lu G, Li D, Wu H, Zhang X, Shi Y, Tian C, Zhang Y, Lu S. BCube: a high performance, server-centric network architecture for modular data centers. *Proceedings of the ACM SIGCOMM*, Aug. 2009; 63–74.
12. Sandholm T, Lai K. MapReduce optimization using regulated dynamic prioritization. *Proceedings of the ACM 11th Joint International Conference on Measurement and Modeling of Computer Systems*, Jun. 2009; 299–310.
13. Kruijf M, Sankaralingam K. MapReduce for the cell B.E. architecture. *IBM Journal of Research and Development* 2009; **53**(5):747–758.
14. Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C. Evaluating MapReduce for multi-core and multiprocessor systems. *Proceedings of the 13th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2007; 13–24.
15. Hsu LH, Lin CK. *Graph Theory and Interconnection Networks*. CRC Press: Boca Raton, 2009.
16. Cohen J. Graph twiddling in a MapReduce world. *IEEE Computing in Science and Engineering* 2009; **2**(4):29–41.
17. Abts D, Marty MR, Wells PM, Klausler P, Liu H. Energy proportional datacenter networks. *Proceedings of the ACM 37th Annual International Symposium on Computer Architecture (ISCA)*, Jul. 2010; 338–347.
18. Guo D, Chen T, Li D, Liu Y, Liu X, Chen G. BCN: expansible network structures for data centers using hierarchical compound graphs. *Proceedings of the IEEE INFOCOM*, Apr. 2011. Mini conference.
19. Bastoul C, Feautrier P. Improving data locality by chunking. *Springer Lecture Notes in Computer Science* 2003; **2622**:320–334.
20. Kumar A, Manjunath D, Kuri J. *Communication Networking an Analytical Approach*. Elsevier Inc.: Amsterdam, 2004.
21. Dean J, Ghemawat S. MapReduce: a flexible data processing tool. *Communications of the ACM* 2010; **53**(1):72–77.
22. Weisstein E. Floyd-Warshall Algorithm. Wolfram MathWorld, 2009. http://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html [accessed on November 2009].
23. Rao L, Liu X, Xie L, Liu W. Minimizing electricity cost: optimization of distributed Internet data centers in a multi-electricity-market environment. *Proceedings of the IEEE INFOCOM*, Mar. 2010.

DOI: 10.1002/cpe