Dynamic Parallel Flow Algorithms With Centralized Scheduling for Load Balancing in Cloud Data Center Networks

Wei-Kang Chung, Yun Li, Chih-Heng Ke, Sun-Yuan Hsieh, *Senior Member, IEEE*, Albert Y. Zomaya, *Fellow, IEEE*, and Rajkumar Buyya, *Fellow, IEEE*

Abstract—BCube is a well-known recursively defined network structure. It provides multiple low-diameter paths and good fault-tolerance for data center networks (DCNs). Its distributed routing algorithm, BCube Source Routing (BSR), can be deployed rapidly and conveniently to build multiple parallel path sets. But in the worst case, BSR may suffer from flow collisions and waste 50% of the capacity of each BCube link. In this paper, to decrease collisions and improve bandwidth utilization, we supplement the BCube topology with a central master computer and design two centralized dynamic parallel flow scheduling algorithms: CDPFS and CDPFSMP, for single-path and multi-path respectively. We focus on finding the least congested path for each flow by analyzing the information about the state of the global network. Furthermore, we allocate those paths to each flow in parallel. The simulation result shows that our proposed algorithms take advantage of BCube structure and deliver high-performance solutions for load balancing problems, which have improved 44.1% of the throughput in random bijective traffic pattern and 36.2% of throughput in data shuffle compared with BSR algorithm.

Index Terms—Algorithms, performance, design, cloud data center networks, server-centric networks, software-defined networks

1 Introduction

The dawn of the twenty-first century has witnessed the development of popular Internet services and major trends in information technology, such as Cloud Computing [1], Web Search, and Big Data [2]. These pivotal technologies all require resilient networks of wide-ranging data centers containing hundreds of thousands of servers and switches. These data centers also provide structure-based services such as distributed file systems [3], structured storage [4]. The networking infrastructure inside a data center is called a data center network (DCN). The prevalent trends of the twenty-first century are motivating scholars to propose new DCN architectures. Numerous scholars are publishing novel designs for flexible, scalable data center interconnection. A favorable DCN architecture should have high scalability, efficient switch and server utilization, and high fault tolerance.

In terms of the reconfigurability of network topology after the deployment of a DCN, the existing popular DCNs are mainly divided into two types: fixed architectures and flexible architectures. The category of fixed architectures can be further classified into two categories: multi-rooted tree-based (switch-centric) networks and recursively-defined (server-centric) networks. In a typical multi-rooted tree-based network, such as Fat-tree [5] and VL2 [6], servers act as endpoint hosts that send and receive data, and switches are responsible for packet forwarding operations, including routing and addressing. In a recursively-defined network, such as BCube [7] and MDCube [8], servers with multiple network ports (NICs) connect to multiple layers of mini-switches that only behave like crossbars. The servers are also responsible for computing-intensive operations like routing; the core task of routing is choosing the next forwarder. That is, servers not only act as end hosts but also act as relays for other servers. Unlike fixed architectures, flexible architectures such as c-Through [9] and Helios [10] enable reconfigurability of their network topology. Every network architecture is characterized by its own unique topology, construction, routing, fault tolerance, and fault recovery.

In this paper, we choose BCube [7], which is a recursively-defined topology and a server-centric network structure, as our network model. BCube has good properties that

- *Wei-Kang Chung and Yun Li are with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 701, Taiwan, R.O.C. E-mail: wkchung@csie.ncku.edu.tw, f94006050@gmail.com.*
- *Chih-Heng Ke is with the Department of Computer Science and Information Engineering, National Quemoy University, Jinning, Kinmen 892, Taiwan, R.O.C. E-mail: chke@csie.nqu.edu.tw.*
- *Sun-Yuan Hsieh is with the Department of Computer Science and Information Engineering and Institute of Medical Informatics, National Cheng Kung University, Tainan 701, Taiwan, R.O.C. E-mail: hsiehsy@mail.ncku.edu.tw.*
- *Albert Y. Zomaya is with the Department of Computer Science and Information Systems, The University of Sydney, Camperdown, NSW 2006, Australia. E-mail: albert.zomaya@sydney.edu.au.*
- *Rajkumar Buyya is with the Department of Computer Science and Information Systems, Cloud Computing and Distributed Systems (CLOUDS) Laboratory, The University of Melbourne, Parkville, VIC 3010, Australia. E-mail: rbuyya@unimelb.edu.au.*

Manuscript received 12 May 2021; revised 8 Nov. 2021; accepted 14 Nov. 2021. Date of publication 22 Nov. 2021; date of current version 8 Mar. 2023.
(Corresponding author: Sun-Yuan Hsieh.)
Recommended for acceptance by C. Wu.
Digital Object Identifier no. 10.1109/TCC.2021.3129768

can provide multiple parallel low-diameter paths. In particular, BCube features graceful degradation; that is to say, when numerous links or switches have failed, the aggregate bandwidth reduces slowly and there are no significant throughput falls. Numerous scholars have proposed centralized algorithms for tree-based network structures such as Fat trees, VL2s, and Clos networks, but few have discussed central management for recursively defined networks. They usually execute a distributed routing algorithm by executing that algorithms on each server in a server-centric network because it is fast and easy. In a server-centric network, each source server has already determined its routing path, which is included in the specific packet header of each packet sent from that server. When the intermediate server receives its packet, it will get the next hop from the specific packet header, forwarding the packet until the destination server has received the packet. If global network status is ignored, it is likely to cause flow collision in the DCN. For example, two source servers may seek to minimize delay time with probe packets; these two servers may choose the same server as an intermediate node, and thus two flows might collide with predefined routing path. To reduce those unnecessary flow collision and improve load balancing of all flows in BCube, we design two variants of a heuristic centralized dynamic parallel flow scheduling algorithm: CDPFS and CDPFSMP, to reduce computation time by parallel methods. The simulation results show that our proposed algorithms can effectively mitigate most flow collisions and improve overall performance.

The main contributions of this paper are as follows. First, to concurrently compute the appropriate paths for each flow that contains a source server paired with a destination server, we propose a fast algorithm to construct two disjoint graphs: SP and NSP, which are subgraphs of BCube. We use those graphs to build a global view interface table with the congestion value of each server network port in BCube. Second, we propose two efficient heuristic centralized flow scheduling algorithms in parallel. The first algorithm finds a single path for each flow, and uses a greedy strategy to have each flow choosing the least congested path in its SP and NSP graphs, so each flow can always find the lowest congestion value on each iteration. When the algorithm decides an appropriate path, we remove this corresponding path from its SP and NSP graphs and update the congestion value of the global view interface table until we allocate all flows to its available path. Furthermore, we know that using Multipath TCP (MPTCP) [11] can use available bandwidth with each link effectively and optimally, giving improved throughput and better fairness on BCube. So we propose a second algorithm to find multiple node disjoint paths for each flow, which is better than finding edge disjoint paths when server or switch failures occurred; it also takes the balance of each server network port used into account. We use a probability model to obtain an approximate solution by the greedy strategy of CDPFS, and our simulation results show that it does not decrease too much throughput compared with the best solution, but it can significantly reduce the calculation overhead for each flow.

The rest of this paper is organized as follows. Section 2 briefly describes BCube structure and discusses the flow scheduling problem. Section 3 presents the building algorithm of the disjoint SP and NSP graphs. Section 4 presents our proposed algorithms: CDPFS with single-path and CDPFSMP with multi-path. Section 5 presents our simulation environment and architecture, the implementation of dynamic flow scheduling and our simulation results. Section 6 concludes this paper.

2 Related Work

2.1 Routing Algorithm

Along with the rapid growth of large-scale data computing, DCNs transport larger traffic and handle longer routing path. As a result of this increased flow bandwidth, the network experiences more network flow collision, which causes throughput reduction. It is challenging to balance the load on such advanced networks with conventional static routing. Scholars have published adaptive routing solutions to this load-balancing problem [12], [13]. In this approach, each switch can make its own routing decisions. However, self-routing causes out-of-order packet forwarding, which degrades reliability [14]. To address this problem, more and more research papers [15], [16], [17] have advocated central management and scheduling with a global view of network-wide communication, which can improve load balancing in data center traffic more effectively than other methods. Scholars designed a centralized routing algorithm using flow-based switches enabled by the OpenFlow [18] framework; their system combined central management and a controller. Several publications have described the software-defined networks (SDN) [19], [20], [21], in which the controller allows for feedback control with information exchange between different switch layers in a DCN architecture, supporting global flow-level control of ethernet switching. The controller has the visibility over all network flows, allowing for near optimal flow scheduling of network traffic flows. A study [22] reported that centralized heuristic approaches with parallel computing can determine routing paths rapidly, manage numerous flows, and apply big-data techniques to large-scale DCNs.

2.2 BCube Network Structure

To make this paper self-contained and easy to understand, we briefly describe the network structure, construction, and distributed routing algorithm: BCube Source Routing (BSR) of BCube in this section.

$BCube(n,k)$ is a recursively defined structure $(k \geq 1)$, which is constructed from n $BCube(n,k-1)$ and n^k n-port switches. Each server has $k+1$ network ports in $BCube(n,k)$, which are connected to switches from 0 to k level. The total number of servers in $BCube(n,k)$ is n^{k+1}, and the total number of switches is $(k+1) * n^k$, which having n^k n-port switches in each level. The example $BCube(4,1)$ can be seen in Fig. 1, a $BCube(4,0)$ has 4 servers with 2-port connecting to 4-port switch, and $BCube(4,1)$ is constructed from 4 $BCube(4,0)$ and 4 4-port switches.

Each server uses an address array $a_k a_{k-1} \ldots a_0$ from the highest level bit k to the lowest level bit 0 where the a_i value belonging $[0, n-1]$ and length (digits) is $k+1$. Each switch uses another format $\langle level_{number}, s_{k-1} s_{k-2} \ldots s_0 \rangle$ where $level_{number}$ is the level of the switch and s_i value belonging

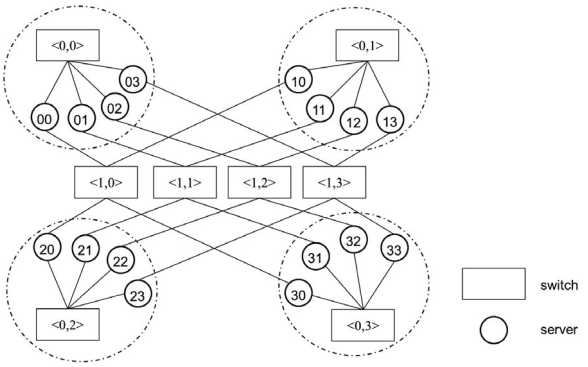


Fig. 1. $BCube(n,k)$ where $k=1$ and switch port $n=4$, each server has $k+1=2$ ports.

$[0, n-1]$. From Fig. 1, we can see the level 0 switches, $\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 0,3\rangle$, connecting to the level bit 0 server with the different values of a_0 from 0 to 3, and the level 1 of switches: $\langle 1,0\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle$ connecting to the level bit 1 server with the different values of a_1 from 0 to 3. More generically, the connecting rule is the k level port of the jth servers $a_j a_{j-1} \dots a_0$ connects to the level j switch $< j, s_{j-1} s_{j-2} \dots s_0 >$ where j is from 0 to k. More details can be found in [7].

Given source server $A = a_k a_{k-1} \dots a_0$ and destination server $B = b_k b_{k-1} \dots b_0$, BSR generates $k+1$ parallel paths in a $BCube(n,k)$; the intermediate servers and switches on one path do not appear any other path. The method uses a permutation set that starts from a different level bit of the A address array (the highest level bit k to the lowest level bit 0), and then shifts right one digit sequentially and modifies the value of its location to the value of B address array when $a_i \neq b_i$. If the value of starting level bit is $a_i = b_i$, it will choose an available neighbor at level i where $c_i \neq a_i$, and set the value a_i to c_i, and correct this level i value to the original value b_i at last iteration. The example can be seen in Fig. 2.

In BSR, the source server decides which path a packet should traverse with its parallel path set by probing the network and encodes the path in the specific packet header. When a new flow comes, the source server sends probe packets over its multiple parallel paths. Those probe packet will collect the network information such as the minimum available bandwidth values of its input/output network ports, or delay time from the intermediate servers of those paths. After the source server receives the probe responses

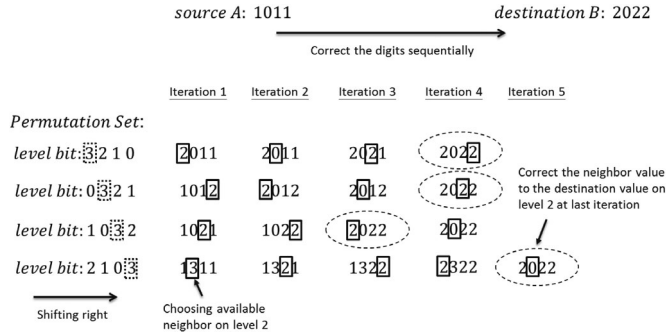


Fig. 2. BSR generates a parallel path set with A and B.

TABLE 1
Notation Table of Section 3

Notation	Description
$h(S,D)$	Hamming distance of two servers, S and D
sp_{count}	Number of different digits of S and D
sp_{bit}	Record the address value of SP level bit
$permutations$	All permutations of 0 to $(sp_{count}-1)$
p	One permutation in $permutations$
$path_p$	A path generated by p
SP_{set}	Contains all shortest paths between S and D

from the destination servers, the path selection completes and the source server chooses a better path from its parallel path set. Since servers need to probe all parallel paths at first and then select the most appropriate path, such as a path with the least end-to-end delay, leading to an increase in the latency of flows.

3 The Path Set of SP, NSP, and AltSP

In this section, we present how to build the the graph and path set of SP, NSP, and AltSP. Description of the notations of Section 3 is shown in Table 1.

3.1 The SP and NSP Categories

To improve the load balancing in the data traffic flows, we consider the total data traffic flows in DCN. Under the premise of the same link capacities and server computing capabilities, when a flow passes through fewer intermediate servers and fewer links, we can have more link spaces to put other flows in the DCN. In our centralized scheduling strategy, we measure all of the data traffic flows and find the shortest path to put those appropriate flows into the DCN. In general, there must be remaining flows that cannot be put into the DCN when many other servers' communications require data transfer. Thus the system uses alternate and longer paths to put those flows into the DCN.

In $BCube(n,k)$ [7], the source server S uses an BCube address array $s_k s_{k-1} \dots s_0$ ($s_i \in [0, n-1], i \in [0,k]$)) and the destination server D uses an BCube address array $d_k d_{k-1} \dots d_0$ ($d_i \in [0, n-1], i \in [0,k]$)). We designate two categories: the Shortest Path (SP) level bit with the digits $s_i \neq d_i$ and Non-Shortest Path (NSP) level bit with the digits $s_i = d_i$. In SP category, $h(S,D)$ denotes the Hamming distance of two servers, S and D, which is the number of different digits of their address array. It is equal to the number of SP level bits and also equal to the shortest path lengths between the the server S and the server D. We know that the server S and server D have $k+1$ digits of address array in $BCube(n,k)$, so there are $h(S,D)$ level bits in the SP category and $k+1-h(S,D)$ level bits in the NSP category. We first focus on the SP level bit category.

3.2 The Path Set and Graph of SP

The SP Path Set contains all of the shortest paths between the source server S and the destination server D. According to each flow with the pair (S,D) in traffic flow patterns, we use this as input to create the SP Path Set by Algorithm 1.

Algorithm 1. CREATE SP PATH SET

Input: Source Server $S = s_k s_{k-1} \dots s_0$ and Destination Server $D = d_k d_{k-1} \dots d_0$; $S[i] = s_i$; $D[i] = d_i$
Output: sp_{count}, the number of SP level bits; SP_{set} which contains all shortest paths between S and D

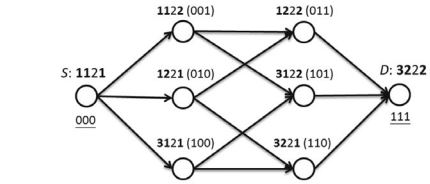
1: Initialization: $SP_{set} = \emptyset$; $sp_{count} = 0$ and $sp_{bit} = []$;
2: **for** each $i \in [0, k]$ **do**
3: **if** $s_i \neq d_i$ **then**
4: $sp_{count}++$;
5: $sp_{bit}append(i)$;
6: $permutations$ = **Heaps_Algorithm**($list$(range[0..$(sp_{count} - 1)$]));
7: **for** each $p \in permutations$ **do**
8: $path_p = [S]$;
9: $N = S$; /* The next hop BCube server */
10: **for** each $dig \in p$ **do**
11: $N[sp_{bit}[dig]] = D[sp_{bit}[dig]]$;
12: $path_p append(N)$;
13: $SP_{set} append(path_p)$;
14: **return** SP_{set}

In Algorithm 1: CREATE SP PATH SET, sp_{count} denotes the number of SP level bits, that is, the number of different digits of S and D. We record the address value of SP level bit in the sp_{bit} array. We use a list array that ranges from 0 to $sp_{count} - 1$ as input to run Heap's Algorithm. Heap's Algorithm is an effective algorithm for generating permutations by computer. It was first proposed by B. R. Heap in 1963 [23]. In the second for loop, we use $permutations$ to create the shortest paths between the S and D. We sequentially create the next server N by modifying one digit of the previous server with p in $permutations$, and then append this path $path_p$ to the SP_{set}.

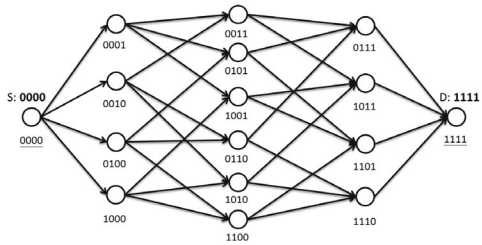
After we get the SP_{set} in Algorithm 1: CREATE SP PATH SET, which means we find out all the shortest paths between the source server S and the destination server D. We can build the SP graph by all the shortest paths in the SP_{set}. Then we use the n-bit binary address to transfer the server's address array in the SP graph ($n = sp_{count}$). We call this binary address a node address. The node address is constructed by the digits from SP level bits from high level bit to low level bit. It can also be viewed as, the n-dimensional hypercube graph Q_n with 2^n nodes (servers). We show an example in Fig. 3a, if the $n = sp_{count} = 3$, the node address of the source server S is (000) and node address of the destination server D is (111). The other node address are one digit different from the previous node which start from S. Fig. 3b shows $n = sp_{count} = 4$ in $BCube(4, 3)$, it is same as the 4-dimensional hypercube graph Q_4 with 2^4 nodes. We will use the SP graph to select appropriate path to improve the load balancing with this flow pattern (S, D) in $BCube(n, k)$.

3.3 The First, Intermediate and Last Forwarders

With a flow pair (S, D) in traffic flow patterns, the neighbors of the source server S are the first forwarders, the neighbors of the destination server D are the last forwarders and the other servers in the flow path are intermediate forwarders. There must be one digit that is different from the first forwarder to source S in the SP level bit, and there must be one digit that is different from the last forwarder to destination D in the SP level bit as well. We can see Fig. 3a as an example, The first forwarders $1122(001)$, $1221(010)$ and $3121(100)$ are the neighbors of S. The last forwarders $1222(011)$, $3122(101)$ and $3221(110)$ are the neighbors of D. In the SP graph where $sp_{count} = n$, We have C_1^n first forwarders , $C_2^n + \dots + C_{n-2}^n$ intermediate forwarders and C_{n-1}^n last forwarders. If we want to find the sp_{count} node-disjoint parallel paths for the flow traffic between the server S and D in the SP graph, we must choose all of the first forwarders and last forwarders in the sp_{count} multi-paths. For $sp_{count} \geq 4$, we have more intermediate forwarders to select appropriate ones in the multi-paths. With numerous traffic flows, we concern about how to select those intermediate forwarders to reduce flow collision as possible.



(a) BCube address (node address) of the server S and D are $1121(000)$ and $3222(111)$ where $sp_{count} = 3$



(b) One example for the BCube address is same as the node address where $sp_{count} = 4$

Fig. 3. SP Graph with BCube Address and node address between the server S and D in $BCube(n, k)$ where $k = 3$ and $n = 4$.

3.4 The Path Set and Graph of NSP and AltSP

In [7], a system had $k + 1$ multi-paths between any two servers in a $BCube(n, k)$, and these $k + 1$ multi-paths would be node-disjoint parallel paths. In the Section 3.1, we recognize both the SP category and the NSP category. With a flow pair (S, D) in traffic flow patterns, if $sp_{count} < (k + 1)$, we have to create the NSP path set to maintain the $k + 1$ multi-paths. To make the NSP paths and the SP paths being disjoint in the $k + 1$ parallel paths, our method is choosing all appropriate neighbor pairs (S', D') that $S'[j] = D'[j]$ where $j =$ the NSP level bit between the server S and D. We use the neighbor pair (S', D') as the input to execute Algorithm 1, and the output would be (S', D') SP path set. As a result, the NSP paths is S to S', SP graph of (S', D'), and D' to D.

We show that why NSP path set and SP path set are disjoint. Choose a neighbor pair (S', D') that $S'[j] = D'[j]$ where $j =$ the NSP level bit of (S, D). We use Algorithm 1 to modify one digit to find the next server on each iteration and put those servers into the NSP path set. It is obvious that we must not modify the digit which located at j level bit. That is, the NSP path set and SP path set of (S, D) are disjoint because there are different servers in the paths. The neighbor pairs (S', D') are shifting neighbors in the same level switch as (S, D), which means the shortest path length

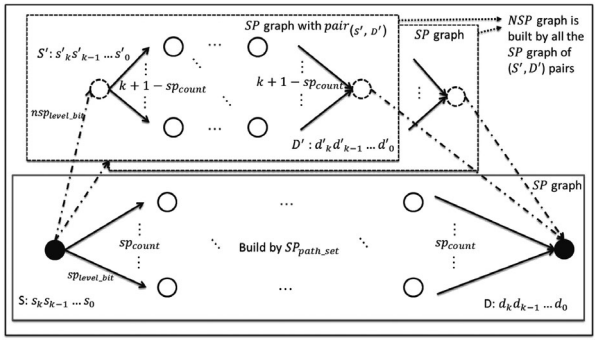


Fig. 4. NSP Graph of $BCube(n,k)$ when $sp_{count} < (k+1)$.

between S and D is equal to the shortest path length between S' and D'. That is why the NSP path lengths are $sp_{count}+2$.

We use NSP path set to create NSP Graph which can be seen in Fig. 4. If $sp_{count} = k+1$, the system has no NSP level bits between the address of source server S and destination server D. If the $sp_{count} < k+1$, the system has $(k+1) - sp_{count}$ NSP level bits. For every NSP level bit j, the system has neighbor pairs (S', D') where $S'[j] = D'[j]$. The NSP graph is built by all the SP graph of those (S', D') pairs with different NSP level bit. We can see Fig. 5 as an example of NSP graph, with $sp_{count} = 3$, the system has $(k+1) - sp_{count} = 1$ NSP level bits, and the index of NSP level bit $j = 1$ in $BCube(n,k)$ where $k = 3$ and $n = 4$. The server address value set are $\{0,1,2,3\}$ since $n = 4$, and the original value $\{2\}$ need to be changed, so we have three candidate neighbor pairs $(S'_1, D'_1) = (1101, 3202)$, $(S'_2, D'_2) = (1111, 3212)$, $(S'_3, D'_3) = (1131, 3232)$. For our method in next section, we will see the global interface table to determine available neighbor pairs. For this example case, assuming that the neighbor pair $(S'_2, D'_2) = (1111, 3212)$ is not available due to the failure or congestion links, we choose the other two neighbor pairs. The NSP graph in Fig. 5 is built by those two neighbor pairs and its SP graph obtained by Algorithm 1.

We consider other alternating paths on SP level bits, which would be AltSP path set. The building method of AltSP graph on the SP level bit is the same as NSP graph. That is, we create AltSP path set which are built by those two neighbor pairs on the SP level bit. They are also disjoint with the NSP path set by the same proof that the address digit $j =$ SP level bit is locked where $S'_{alt}[j] = D'_{alt}[j]$ and different position on the NSP level bit in the previous paragraph. Notably, SP and AltSP have paths of different lengths: $h(S,D)$ and $h(S,D)+1$, because we choose a SP level bit to find its available neighbor pairs. The AltSP graph has $sp_{count} - 1$ nodes between the neighbor pair $S'_{alt}[j] = D'_{alt}[j]$ of the source and destination (S, D).

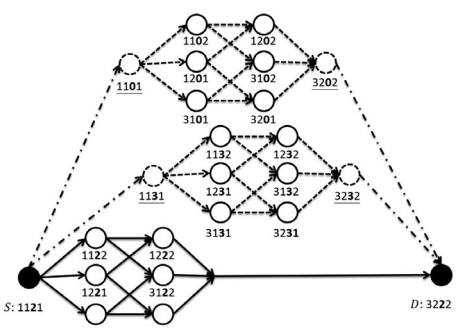


Fig. 5. NSP graph of $BCube(n,k)$ where $k = 3$, $n = 4$, $sp_{count} = 3$ and NSP level bit $j = 1$. Bit value 1 is not available, and bit value 0 and 3 are chose.

TABLE 2
Notation Table of Section 4

Notation	Description
G_d	A bipartite graph with two partite set U_{FP} and V_{Interf}
FP	A vertex in U_{FP}, designated as (f_i, p_j)
$Interf$	A vertex in V_{Interf}, designated as $(level_{bit}, BCube_{addr})$
$Interf_{capa}$	Capacity of each server's interfaces
$Demand$	The number of FP vertex's neighbors which $Interf$ vertex's degree $> Interf_{capa}$
FP_{chose}	The FP vertex with minimum $Demand$ in U_{FP}
G_{chose}	A subgraph of G_d which contains all of the FP_{chose} vertex in U_{FP} and FP_{chose}'s neighbors when $Interf_{capa} = 1$ in V_{Interf}
$Path_{single}$	Contain m paths for each flow in the data traffic patterns
$Actual$	A BCube topology contains capacity W_{actual} of each server's interfaces
$Congestion$	A BCube topology contains weight values W_{conge} of each server's interfaces
$Path_{MP}$	Contain $m*(k+1)$ paths where each flow has $k+1$ node disjoint paths
$MPathSet_i$	Contain $SP_{set[i]}$ and $NSP_{set[i]}$
Sum_{Interf}	Sum of W_{actual} of each interfaces in $Actual$
W_{bound}	Minimum of W_{conge} in $Congestion$
PR_{chose}	A probability to choose the interfaces where $W_{conge} > W_{bound}$
$Weighted$	A BCube topology with weight values on each server interfaces

4 The Proposed Algorithms

In this section, we present how to use CDPFS and CDPFSMP to find appropriate paths for each flow for load balancing. Description of the notations of Section 4 is shown in Table 2.

4.1 The Centralized Dynamic Parallel Flow Scheduling Algorithm with Single-Path

In this section, we propose Centralized Dynamic Parallel Flow Scheduling Algorithm (CDPFS) to improve load balancing of single-path traffic flows in BCube. We consider all of the traffic flows and choose an appropriate path, which is a disjoint path with fewer collisions for each of them. We know that each source server have $k+1$ NICs which can pass through $k+1$ different level bit to the first forwarder, and we say that source server has $k+1$ interfaces in $BCube(n,k)$. In the data traffic patterns, each server may be a source, a forwarder and a destination server in the same time, so we concern about how to allocate these $k+1$ interfaces to different flows with different paths. To reduce the computing time in CDPFS, we use the m multi-thread to deal with m flows in the data traffic patterns.

Algorithm 2. CDPFS FOR SINGLE PATH

Input: All of the m data traffic flows
Output: $Path_{single}$ which have m paths for each flow in the data traffic patterns

1: Global graph : A bipartite graph G_d, with two partite set U_{FP} and V_{Interf}.
2: **for** each $i \in [0, m-1]$ **do**
3: Each $thread_i$ uses (S_i, D_i) of f_i to create $SP_{set[i]}$ by Algorithm 1 and FP vertices in U_{FP}.
4: Each $thread_i$ uses $SP_{set[i]}$ to add the edges connected by FP vertices and $Interf$ vertices.
5: $thread_{flow}.join()$;
6: **while** U_{FP} is not empty **do**
7: FP_{chose} = the FP vertex with min $Demand$.
8: Create G_{chose} subgraph;
9: $G_d = G_d - G_{chose}$;
10: **for** each $FP_{competitor}$ in U_{FP} **do**
11: Remove $FP_{competitor}$ vertices and corresponding $path$ in their path set;
12: **if** $SP_{set[i]} = \emptyset$ **then**
13: $FlowSet_{Re}.append(f_i)$;
14: Remove all FP vertices with f_i in U_{FP};
15: **for** each f_i in $FlowSet_{Re}$ **do**
16: Each $thread_i$ will build the NSP and AltSP path set by Algorithm 1 with available neighbor pairs in the remaining G_d and append to $SP_{set[i]}$.
17: Each $thread_i$ uses $SP_{set[i]}$ to asynchronously add edges connected to the new FP vertices and $Interf$ vertices.
18: $thread_{flow}.join()$;
19: **for** each $i \in [0, m]$ **do**
20: $Path_{single}[i] = FP_{chose}$ with f_i;
21: **if** $Path_{single}[i] = \emptyset$ **then**
22: $Path_{single}[i] =$ **Algorithm 5** $(f_i, Weighted, W_{max})$.
23: /* Using Algorithm 5 to deal with those remaining flows.*/
24: **return** $Path_{single}$

In Algorithm 2, our key idea is using a greedy strategy, the Least Demand First (LDF) method. We build a global bipartite graph $BCubeInterface_{global}$ $DemandGraphG_d$ which contains two partite set U_{FP} and V_{Interf}. Assume the network has total m flows in the data traffic patterns, each flow has its own path set with several paths. Each path is a FP vertex in U_{FP} designated as (f_i, p_j), which means the j_{th} path in the path set of i_{th} flow. The network has n^{k+1} servers in $BCube(n, k)$ and each server has $k+1$ interfaces, each interface is a $Interf$ vertex in V_{Interf}, and designated as $Interf = (level_{bit}, BCube_{addr})$.

At line 2–4, we first use m threads to create SP path set of m flows in parallel to reduce the building time. Each $thread_i$ has the input (S_i, D_i) to create the $SP_{set[i]}$ by Algorithm 1, and creating the FP vertices in U_{FP}. A path has source, destination and many intermediate nodes, and every edges represent a flow pass a switch with two endpoints which has two different interfaces. For example, there are two paths $path_0 = (00, 01, 11)$ and $path_1 = (00, 10, 11)$ in $SP_{set[i]}$ with $(S, D) = (00, 11)$ of f_i. In $path_0$, two edges are $e_0 =$ (00 to 01) and $e_1 =$ (01 to 11), e_0 is a flow passing $switch\langle 0, 0\rangle$ from interface=0 at 0 level bit on 00 server to interface=0 at 0 level bit on 01 server, and e_1 is a flow passing $switch\langle 1, 1\rangle$ from interface=1 at 1 level bit on 01 server to interface=1 at 1 level bit on 11 server, and so on. We use $Interf$ to represent every node in $path_1$ which are vertices $(1, 00), (1, 10), (0, 10), (0, 11)$ in V_{Interf} and passing $switch\langle 1, 0\rangle, switch\langle 0, 1\rangle$. So each $thread_i$ uses the $SP_{set[i]}$ to add edges which connected by the FP vertices and $Interf$ vertices in each $path_x$ with those path nodes where $x \in [0..(sp_{count}! - 1)]$.

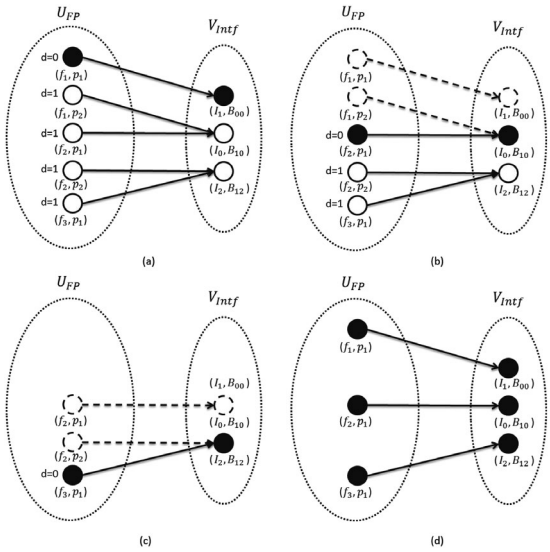


Fig. 6. Least Demand First (LDF) method.

At line 5, $join()$ is a synchronous function and we have to wait all $thread_i$ finishing their work. The computer experiences a small time gap with each computation of $thread_i$ in general. We continually find m appropriate paths to each f_i with the candidate paths in U_{FP} until it is empty. The LDF method is at line 7–8, which can be seen in Fig. 6. $Interf_{capa}$ denotes capacity of each server's interfaces, and we first set the value of $Interf_{capa}$ as $\lceil \sum_i (df_{num[i]} * sp_{count[i]})/((k+1)(n^{k+1})/2) \rceil$. The $df_{num[i]}$ is the number of i_{th} different (S, D) pair flows and $sp_{count[i]}$ is the sp_{count} of i_{th} different (S, D) pair when the same (S, D) pair is smaller than $k+1$. Many flows with same (S, D) pair are limited on their interfaces, so these flows have notably small bandwidth values; this is necessary because a limited total quantity of bandwidth must be allocated to an excessive number of flows. We also call these flows has local property. $Demand$ is the number of FP vertex's neighbors which the neighbor vertex's degree $> Interf_{capa}$, and each iteration we choose the FP vertex with minimum $Demand$ in U_{FP} as FP_{chose}. G_{chose} is subgraph of G_d which contains all of the FP_{chose} vertex in U_{FP} and FP_{chose}'s neighbors when $Interf_{capa} = 1$ in V_{Interf}, otherwise $Interf_{capa}$ of FP_{chose}'s neighbors reduce those capacity by 1.

In Fig. 6, we have three flows f_1, f_2, f_3 and $PathSet_{f1} = \{p_1, p_2\}$, $PathSet_{f2} = \{p_1, p_2\}$, $PathSet_{f3} = \{p_1\}$. We create FP vertices in U_{FP} and $Interf$ vertices in V_{Interf} which can be seen in Fig. 6 (a). Assume we set all of $Interf_{capa}$ of $Interf$ vertices as 1, and each demand value of FP vertices would be as it can be seen in Fig. 6 (a). FP_{chose} would be (f_1, p_1), which is the vertex having the minimum demand value $Demand = 0$, since the degree and $Interf_{capa}$ of its neighbor (I_1, B_{00}) are both one. We allocate $Interf$ vertex (I_1, B_{00}) to FP_{chose} vertex (f_1, p_1), representing that f_1 selects p_1 in $PathSet_{f1}$ as its data transfer path, so we remove other vertices of $PathSet_{f1}$ in U_{FP} and remove the $Interf$ vertices when $Interf_{capa} = 1$. In the case of a vertex with $Interf_{capa} >$

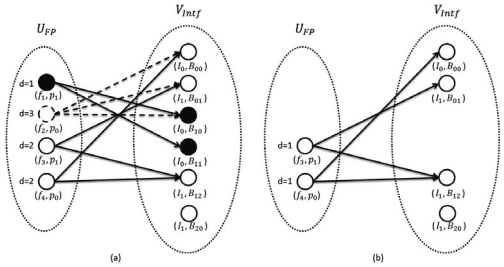


Fig. 7. (a) Flow collision and removing $FP_{competitor}$ of other flows by the neighbors of FP_{chose} in $Interf_{chose}$. (b) The scenarios for choosing FP_{chose} with the same minimum demand value of FP vertices.

1, we reduce those capacity by 1, and releasing those resources of FP_{chose}. As a result, the demand value of FP vertices would be updated as it can be seen in Fig. 6 (b). For example, the demand value of (f_2, p_1) would be updated from 1 to 0, since the degree of its neighbor (I_0, B_{10}) reduces from 2 to 1 and it is no longer bigger than the $Interf_{capa} = 1$ of (I_0, B_{10}). Now we find (f_2, p_1) as FP_{chose} which has the minimum $Demand$ and we allocate $Interf$ vertex (I_0, B_{10}) to f_2 which selects p_1 path, and so on. Fig. 6 (d) shows three flows with those selected paths and used interfaces in V_{Interf}. These used interfaces, which are $Interf$ vertices in V_{Interf} are called $Interf_{chose}$, have no capacity to allocate other flows.

Another example can be seen in Fig. 7. We set all of $Interf_{capa}$ of $Interf$ vertices in V_{Interf} as 1 and each demand value of FP vertices would be as it can be seen in Fig. 7 (a). The FP_{chose} vertex (f_1, p_1) has the minimum demand value $Demand = 1$ and we allocate $Interf$ vertices $(I_0, B_{10}), (I_0, B_{11})$ to FP_{chose} in U_{FP}. When we remove those $Interf$ vertices, it affects FP vertex (f_2, p_0) which called $FP_{competitor}$. Because the p_0 of f_2 has no capacity on (I_0, B_{10}) vertex in V_{Interf}, we also remove those $FP_{competitor}$ vertices and release the $Interf$ vertices as their neighbors. The result can be seen in Fig. 7 (b) which shows the scenario of choosing FP_{chose} with the same minimum demand value of FP vertices. In our method, we will choose FP vertex with higher priority. We maintain a priority table with all active flows and set the initial priority value as 1 where the key is FP vertex and delete flow entry when the flow is left or removed. The lower value of FP vertex has a higher priority to be selected. When we select this FP vertex as FP_{chose}, we will add 1 to its priority value. At next timeout for running Algorithm 2, this method has more fairness to choose other FP vertices with higher priority.

At line 10–14, we remove $FP_{competitor}$ vertices and corresponding $path$ in their $PathSet$, which may result in $PathSet = \emptyset$. If the $PathSet$ of those flows are SP_{set}, which means that we first create SP path set for those flows but no interfaces are available for assignment to these paths. So we put those flows to $FlowSet_{Re}$, and create their NSP and AltSP path set. At line 15–18, we use multi-threads to create NSP and AltSP path set which is built by available neighbor pairs on NSP and SP level bits. Because we have removed the subgraph G_{chose} which contains FP_{chose} and $Interf_{chose}$ in G_d, each $thread$ creates those path set with the remaining G_d which including the existing $Interf$ vertices with left capacity of interfaces in V_{Interf}. After the while loop at line 6, we can find our $Path_{single}$ with m paths for each flow in the data traffic patterns.

Algorithm 3. CDPFSMP FOR MULTI PATH

Input: All of the m data traffic flows
Output: $Path_{MP}$ which have $m * (k+1)$ paths where each flow has $k+1$ node disjoint paths in the data traffic patterns

1: **global** : Two $BCube(n, k)$ topology $Actual$ and $Congestion$.
2: **for** each $i \in [0, m]$ **do**
3: Each $thread_i$ creates $SP_{set[i]}$ and $NSP_{set[i]}$, which appends to $MPathSet_i$.
4: Each $thread_i$ uses the $MPathSet_i$ to add W_{conge} value.
5: $thread_{flow}.join()$;
6: Sum_{Interf} = the sum of W_{actual} of each interfaces in $Actual$;
7: W_{bound} = minimum of W_{conge} in $Congestion$;
8: **while** $W_{bound} \leq$ (maximum of W_{conge}) **do**
9: $Pre\_Sum_{Interf} = Sum_{Interf}$;
10: **for** each $i \in [0, m]$ **do**
11: $Path_{MP}$ = **Algorithm 4** (f_i, W_{bound});
12: /* Each $thread$ executes Algorithm 4 to find the multiple node disjoint paths. */
13: $thread_{flow}.join()$;
14: Update Sum_{Interf};
15: **if** $Pre\_Sum_{Interf} \neq Sum_{Interf}$ **then**
16: W_{bound} = min W_{conge} in $Congestion$;
17: **else**
18: $W_{bound} = W_{bound} * 2$;
19: **for** each $i \in [0, m]$ **do**
20: **if** (the number of $Path_{MP}[i]$ paths) $< (k+1)$ **then**
21: $Path_{MP}[i]$ = **Algorithm 5** $(f_i, Weighted, W_{max})$;
22: /* Using Algorithm 5 to find disjoint paths with remaining flows. */
23: **return** $Path_{MP}$

Although we create the first category of SP and the second category of NSP and AltSP path set to choose appropriate path for flows by running the LDF method, there is still a chance to have f_i with $Path_{single}[i] = \emptyset$. One possibility is those flows have local property that there are many same source or destination servers with each other, and we don't allocate more capacity of $Interf_{capa}$ given our previous computing of df_{num}. Some of the flows have no available path because we choose those paths by the LDF method when selecting the same minimum demand value of FP vertex with its priority value, it may also have to randomly choose due to the same priority value because those flows are new data traffic patterns or the duration of data communications are short. We deal with all of those remaining flows by using Algorithm 5 and obtain the $Path_{single}$ to balance all of our data traffic patterns in $BCube(n, k)$.

4.2 The Centralized Dynamic Parallel Flow Scheduling Algorithm with Multi-Path

In this section, we propose the Centralized Dynamic Parallel Flow Scheduling Algorithm for Multi-Path (CDPFSMP) in Algorithm 3, it is different from CDPFS for single-path which is dedicated to finding a path having less collision to each flow. If we determine a flow using its appropriate path between its source and destination server to data communication, the flow will fill this path with high bandwidth when no other flows share this path, which can be termed a large flow. We can imagine that many large flows exist on some specific links; this situation may leave many broken

links adjacent to those links because many interfaces must be occupied by some large flows and the remaining interfaces cannot find a path to those remaining large flows. We know that each server has $k+1$ interfaces (NICs) in $BCube(n,k)$ and we can find $k+1$ node disjoint paths to each server because numerous vacant edges are present in $BCube(n,k)$ topology. So if we divide a large flow by $k+1$ sub flows, meaning that it only uses section of capacity (bandwidth) of each link in $BCube(n,k)$, those $k+1$ sub flows will use $k+1$ disjoint paths from source to destination with $1/(k+1)$ bandwidth, and it can also achieve the original bandwidth $(1/(k+1))*(k+1)=1$ of flow with single path. The system has numerous link spaces to share among other flows using same interfaces with the remaining $k/(k+1)$ capacity of links. It can reduce the number of broken links and improve the load balancing with all of the data traffic patterns in $BCube(n,k)$. Therefore, we focus on how to select those node disjoint paths for each (S_i,D_i) pair. We will determine how many sub flows should share the bandwidth of bounded interface of links. Under the best conditions, we can speed up $k+1$ times of bandwidth for each (S_i,D_i) pair with $k+1$ disjoint paths when no other flows share those paths.

In Algorithm 3, the system constructs two $BCube(n,k)$ topology $Actual$ and $Congestion$. $Actual$ contains all of the capacity W_{actual} of each server's interfaces. Because each (S_i,D_i) pair has $(k+1)$ paths, we determine $W_{actual}=(\lceil\sum_i(df_{num[i]}*sp_{count[i]})/(k+1)(n^{k+1}/2))\rceil)*(k+1)=(\lceil\sum_i(df_{num[i]}*sp_{count[i]})/(n^{k+1}/2)\rceil)$ for each interface. $Congestion$ initially contains all of the weight values $W_{conge}=0$ of each server's interfaces. W_{conge} denotes the sum of the interface used by all paths of f_i. If many (S,D) pairs struggle to use an interface, the W_{conge} of this interface will become large. Our goal is to improve load balancing with each (S,D) pair of all data traffic patterns. By LDF method, if we first allocate some of the (S,D) pairs to interfaces with lower W_{conge}, we can release other non-used paths in those remaining path set. In next iteration, we will update W_{conge} of each interface, and we continually find appropriate paths for other (S,D) pairs. Finally, we can find our $Path_{MP}$, which allocates nearly maximum interfaces to all (S,D) pairs with appropriate paths.

We count W_{conge} value in each $thread_i$ at Algorithm 3 line 2–4. Each $thread_i$ uses (S_i,D_i) as input to create the $SP_{set[i]}$ and $NSP_{set[i]}$ by Algorithm 1 and append to $MPathSet_i$. Each $thread_i$ uses the $MPathSet_i$ to add the W_{conge} value by exclusive accessing to the $Congestion$ topology which has all of the weight value W_{conge} counted by interfaces in $SP_{set[i]}$ and $NSP_{set[i]}$. We use Sum_{Interf} to record the sum of W_{actual} which are counted by all of currently selecting interfaces of paths in $Path_{MP}$ path set on each interfaces in $Actual$, and Pre_Sum_{Interf} represents the previous Sum_{Interf} value on last executing iteration. Sum_{Interf} value of each iteration will be higher because we find more available paths in $Path_{MP}$ and release more interfaces of unselected candidate paths. W_{bound} denotes the threshold value of interface, which initially equals to minimum of W_{conge} in $Congestion$.

When we use LDF method to find multiple paths for each flow, time is likely to be wasted because too many flows must be considered. Therefore, we reduce the number of iterations by a sequential method at Algorithm 3 line 8–18, using Algorithm 4 executed by each $thread_i$ to find those $k+1$ node disjoint paths for each (S_i,D_i) pair. Once we have the same value of Sum_{Interf} and Pre_Sum_{Interf} on an iteration, representing that we do not find any suitable paths in $thread_i$ for (S_i,D_i) pair on this iteration, then we might waste execution time for Algorithm 4 because we choose a bad W_{bound} for higher W_{conge}. We multiple it twice to expand search space and accelerate each $thread_i$ to find the maximum number disjoint paths. It can reduce much time and this solution is not far from the best solution by LDF method releasing minimum W_{conge} of interfaces sequentially. At Algorithm 3 line 19–22, when the number of $Path_{MP}[i]$ paths is less than $(k+1)$, we obtain $Path_{MP}$ by Algorithm 5, which sequentially removes current existing paths and do next iteration to find disjoint paths with those remaining flows.

Algorithm 4. FIND MAX NODE DISJOINT PATHS

Input: $MPathSet_i$ and W_{bound}
Output: $PathSet_{max}$, the currently maximum node disjoint paths of f_i

1: **local** : Flow network is built by all of the vertices in SP and NSP graph with $MPathSet_i$.
2: **for** each $edge_f$ in SP and NSP graph **do**
3: $W_{ce}=$ maximum W_{conge} of endpoint of interface on edge;
4: **if** $W_{ce}\le W_{bound}$ **then**
5: Add $edge_f$ to flow network and remove it from SP or NSP graph;
6: **else**
7: $PR_{chose}=(HT_p*W_{Actual})/W_{ce}$;
8: **if** $P_{Random}[0..1]\le PR_{chose}$ **then**
9: Add $edge_f$ to flow network and remove it from SP or NSP graph;
10: Using non-node-split max flow method to find node disjoint paths with (S_i,D_i) and append to Set_{temp}.
11: **for** each $p_j\in Set_{temp}$ **do**
12: **if** $W_{actual}>0$ in p_j **then**
13: $thread.acquire()$;
14: W_{actual} reduced by 1 in $Actual$;
15: $thread.release()$;
16: $PathSet_{max}.append(p_j)$;
17: **if** category = 'SP' **then**
18: $sp_{now}++$;
19: **if** $sp_{now}==sp_{count}$ **then**
20: Removing p_j from the flow network;
21: $thread.acquire()$;
22: Release all interfaces with $SP_{set[i]}$ and W_{conge} reduced by 1 in $Congestion$;
23: $thread.release()$;
24: **else**
25: Removing NSP graph on j NSP level bit from the flow network;
26: $thread.acquire()$;
27: Release all interfaces with $NSP_{set[i]}$ and W_{conge} reduced by 1 in $Congestion$;
28: $thread.release()$;
29: **else**
30: Removing $W_{actual}=0$ from the flow network;
31: **return** $PathSet_{max}$

In Algorithm 4, each $thread_i$ has its local flow network which is initially built by all of the vertices in SP and NSP

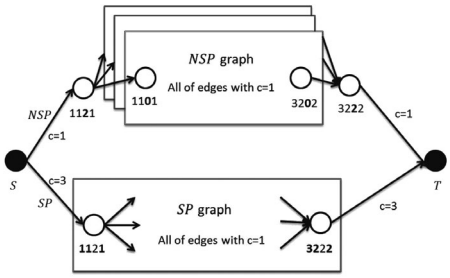


Fig. 8. Flow network with all vertices in SP and NSP graph with $MPathSet_i$ of $(S_i, D_i) = (1121, 3222)$.

graph. At line 2–9, we give each (S_i, D_i) pair in flow network a probability PR_{chose} to choose the interfaces where $W_{conge} > W_{bound}$. Edges are added with higher W_{conge} in advance to reduce execution time. $PR_{chose} = (HT_p * W_{actual})/W_{ce}$ where $HT_p = 1$ if the edge is connected to source and first forwarder or the edge is connected to destination and last forwarder, otherwise $HT_p = 0.5$. W_{ce} is the currently maximum W_{conge} of endpoint of interface on edge. If W_{ce} is high, PR_{chose} would be low. Because we only have $k+1$ interfaces of source or destination server, and we have many intermediate servers to choose a suitable path in the flow network, a higher PR_{chose} is needed to select those edges. The probability model would add less edge to find path in $thread_i$ but fast to avoid unnecessary executing iteration. The result of this method is similar with the method of adding edges sequentially by W_{bound}. To find node disjoint paths in the flow network, we first set the capacity of each edge and adding S, T nodes which can be seen in Fig. 8. We use the non-node split method in Algorithm 4 to find the maximum number of independent paths from S to T and append to Set_{temp}. It is good for us since we would not need additional time to combine those intermediate nodes when we find paths.

At line 11–30 in Algorithm 4, when we find the maximum number of disjoint paths in Set_{temp}, each $thread_i$ will check W_{actual} of available interfaces in $Actual$ by every p_j. We allocate available interfaces to those paths and each W_{actual} value of available interface decreased by one by mutually exclusive access. We use $thread.acquire()$ and $thread.release()$ to access global values exclusively. We deal with the first 'SP' category and the second 'NSP' category by different methods. We find the maximum number of disjoint SP paths which is equal to sp_{count} in 'SP' category. If we just find $sp_{now} < sp_{count}$ disjoint SP paths, where sp_{now} is the maximum number of disjoint paths in currently SP graph of flow network when some of essential edges with high value W_{bound} are not added, we will select recent p_j and remove this path from the flow network to guarantee that we can find other existing disjoint paths in next iteration. We find the NSP path in the current NSP graph of the flow network and we know that the SP and NSP graphs are disjoint. If we have $(k+1) - sp_{count} > 1$ NSP level bits, the system has numerous NSP graphs and all NSP graphs are disjoint because we lock the NSP level bit with different neighbor pairs. The number of maximum flow has only one in each NSP graph with those j NSP level bit. When we find path starting from j NSP level bit in its NSP graph, we have an available neighbor pair on j NSP level bit of (S_i, D_i) pair, and we remove NSP graph on j NSP level bit from the flow network. We append each selected p_j to $PathSet_{max}$. In 'SP' category, we release all of interfaces in $SP_{set[i]}$, so those W_{conge} value in $Congestion$ will be decreased by one when $sp_{now} = sp_{count}$. In 'NSP' category, we release all of interfaces in $NSP_{set[i]}$ and those W_{conge} value will be decreased by one in $Congestion$. So we can reduce some of the W_{conge} value of interfaces to unselected paths of (S_i, D_i) pair, meaning that we can add more edges to other (S_i, D_i) pairs of the flow network in next iterations and find those available paths.

Algorithm 5. FIND PATH IN TWO ITERATIONS

Input: (S_i, D_i); $Weighted$ topology; W_{max}
Output: $Path_{selected}$, the founded path of f_i
1: **initial** : $W_{min} = 0$, $iteration = 0$, $Path_{selected} = \emptyset$;
2: $Weighted_{copy} = Weighted$ removes the edges whose weight value $\geq W_{max}$;
3: $Path_{selected} = Random\_BFS(Weighted_{copy}, S_i, D_i)$;
4: **if** $Path_{selected} == \emptyset$ **then**
5: $Path_{selected} = Random\_BFS(BCube_k, S_i, D_i)$;
6: **else**
7: **while** $iteration < 2$ **do**
8: $W_{mid} = (W_{min} + W_{max})/2$;
9: $Weighted_{copy} = Weighted$ removes the edges whose weight value $\geq W_{mid}$;
10: **if** $Random\_BFS(Weighted_{copy}, S_i, D_i) \neq \emptyset$ **then**
11: Update $Path_{selected}$;
12: $W_{max} = W_{mid}$
13: **else**
14: $W_{min} = W_{mid}$
15: $iteration++$;
16: **return** $Path_{selected}$

In Algorithm 5, we first define the weight value of servers which is the total flows passing through its paths built by currently path set $Path_{single}$ and $Path_{MP}$ on each interfaces, and we call this $BCube(n, k)$ topology with weight values on each server interfaces is $Weighted$ topology. W_{max} is the maximum weight value when f_i uses its path in $Path_{single}[i]$ and passes through each interface of servers with those weight value. For example, a $(S_i, D_i) = (00, 11)$ pair is using its path (00,10,11) in $Path_{single}[i]$; we use $Interf$ to represent every interface which are $(1, 00), (1, 10), (0, 10), (0, 11)$ in $Path_{single}[i]$ and every weight value of interfaces are $3, 2, 4, 1$, so W_{max} value of f_i is 4, which means f_i will occur flow collision with other four flows on the $Interf$ vertex (0,10). According to the principle of fairness, f_i can be allocated to the available bandwidth which is the total of bandwidth divided by $(4+1)$. If the total of bandwidth is $10Mbps$, f_i can be allocated $2Mbps$ because the path of f_i is bounded on the $Interf$ vertex (0,10). Our target is to improve load balancing with the total flows and speed up the total traffic in $BCube(n, k)$. The definition of this problem is to find the minimum W_{max} value for the remaining flows on $Weighted$ topology. If we want to find the optimal solution on this problem, the system may require excessive execution time. So we propose Algorithm 5 which is effective to find the approximate optimal solution because the weight value is a small number.

We initialize the lower bound $W_{min} = 0$ and upper bound W_{max} is the maximum weight value in $Weighted$

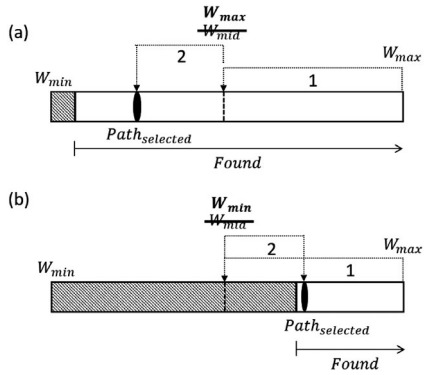


Fig. 9. Two cases of Algorithm 5 which is effective to find the approximate optimal solution.

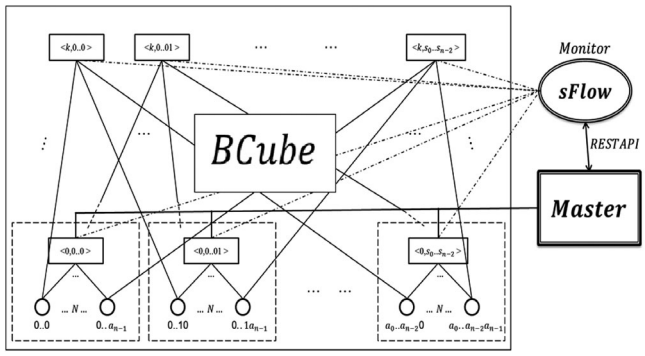


Fig. 10. The Architecture of $BCube(n,k)$ with Master (Large Parallel Computing Computer) and Monitor (sFlow).

topology. At Algorithm 5 line 2, each $thread_i$ removes the edges whose weight value $\geq W_{max}$. For example, if the W_{max} is 4 in the $Weighted$ topology where $n=2$, we will remove its edge which connected with $switch\langle 1,0\rangle$ and the $Interf$ vertex (0,10) of $BCube(n,k)$ server. We use the remaining $Weighted$ topology as input to find $Path_{selected}$ by $Random\_BFS()$. $Random\_BFS()$ function is randomly chooses vertex which is a neighbor of the current vertex and put into queue in each iteration. For example in Fig. 3a, if we sequentially choose vertices by using for loop and put into queue, the path of flow (1121,3222) must be (1121,1122,1222,3222) according to First-in First-out principle. Owing to the fact that we have to deal with some of flows which has local property, we can assume that there are many same source or destination servers with f_i. The calculation considers each flow in one thread with a random probability distribution; it equally disperses those flows with different paths.

If $Path_{selected}$ is empty which indicates that f_i may be one of flows with local property, we deal with those flows by using original $BCube(n,k)$ topology as input by $Random\_BFS()$ to find paths. In another case, when we use SP, NSP and AltSP path sets to allocate most of flow's paths, the $Weighted$ topology exists many irregular alternating paths in some interface area with lower weight value. At line 7–15, our key idea is very similar to the Binary Search [24], we first get $Path_{selected}$ which is found in $Weighted$ topology. We set the middle of weight value $W_{mid} = (W_{min} + W_{max})/2$, continuing to find the more appropriate path in $Weighted$ topology which removes the edges with weight value $\geq W_{mid}$. If this path is found, we update $Path_{selected}$ and continue to reduce the search space by replacing W_{max} to W_{mid}, that is, $W_{max} = W_{mid}$. If no path is found, this $Weighted$ topology has higher W_{max} value for this f_i, and we have to increase W_{mid} value to be close to boundary value by replacing W_{min} to W_{mid}, that is, $W_{min} = W_{mid}$.

Two cases can be seen in Fig. 9, the white block represents the range of weight value which can find the path of f_i. In Fig. 9 (a), we find a path in $Weighted_{copy}$ topology so we reduce the search space by replacing W_{max} to W_{mid}. Then we get $Path_{selected}$ in the second iteration. In Fig. 9 (b), there is no path in $Weighted_{copy}$ topology so we increase W_{mid} value by replacing W_{min} to W_{mid}. Then we get $Path_{selected}$ in the second iteration. Because the weight value is a small number, we use two iterations. This choice causes the system to find the path rapidly and saves a great deal of run time. Algorithm 3 deals with multiple flows sequentially on each iteration which removing the previous path to maintain the disjoint property.

We will analyze the worst-case time complexity of the proposed algorithms briefly. We start with Algorithms 1, 4, and 5, since they are the subfunctions of Algorithm 2-CDPFS and Algorithm 3-CDPFSMP. In a $BCube(n,k)$, the complexity of Algorithm 1, 4, and 5 is $O(k*k!)$, $O(k*k!)$, and $O(k*n^k)$. Hence, with m data traffic flows, Algorithm 2-CDPFS and Algorithm 3-CDPFSMP have the same time complexity $O(m*k*(k!+n^k))$.

5 Simulation on Mininet

In this section, the performance of CDPFS and CDPFSMP is evaluated using Mininet [25], an open-source network simulator. First, we illustrate our simulation environment and architecture of BCube with Master and Monitor. Second, we use a particular method that enables the packet forwarding of each server (host) as router [26] to send packets instead of BSR protocol, and we use network socket which is an endpoint of an inter-process communication across from each server in BCube to Master acquiring global information. Third, we compare the performance of our proposed algorithms to the algorithm of the baseline paper with different traffic patterns.

5.1 Simulation Environment and Architecture

We use a virtual Linux system that installs Mininet and Ryu controller [27] to simulate DCN environment. Our VM equipment is VMWare 5.1 version which has four CPUs and 50GB memory. Mininet is a network emulator which creates a network topology of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software so we can write applications to communicate with each server, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking (SDN).

Fig. 10 shows three components in our centralized architecture: BCube, Master and Monitor. $BCube(n,k)$ is a server-centric network structure with servers connecting to k layers of switches. We add the Master which is a centralized large-scale parallel computing computer. The Master has eight CPUs, 64 GB memory and connects to the lowest layer switches which are 0-level switches. The Master

maintains a global view of the BCube network topology. We add Monitor to measure the real-time flow status and communicate with the Master to compute the best flow scheduling by CDPFS and CDPFSMP. For our simulation, we use sFlow [28] which is the leading, multi-vendor, standard for monitoring high-speed switched and routed networks. The sFlow uses sampling of ways to remove the summary information, and we can customize their sample size and sampling period. By customization, we can optimize network traffic flows and sFlow sampled data to streamline the required information. By doing so we can maintain correct information and ensure that the back-end processing can analyze all of the traffic flows correctly.

We use a Ryu controller as our Master, and each server communicates with controller by using a non-standard protocol, which is called a $Packet-in$ event function. When a flow starts, the packet stream has 10-tuple of packet's header, and controller will capture the header to maintain a global view with all of the flows in BCube. Our proposed algorithms are computed for flow scheduling on a controller by using network socket. The network socket is an endpoint of an inter-process communication across from each server in a BCube. Each server sets the routing rules with available paths which are computed by controller. But for our experiment results, Mininet which uses controller would not simultaneously deal with numerous active flows in large-scale DCN topology. It may be our future work to solve this problem. So we have additional costs using the centralized large-scale parallel computing computer as our Master and links which are connected to the lowest layer switches.

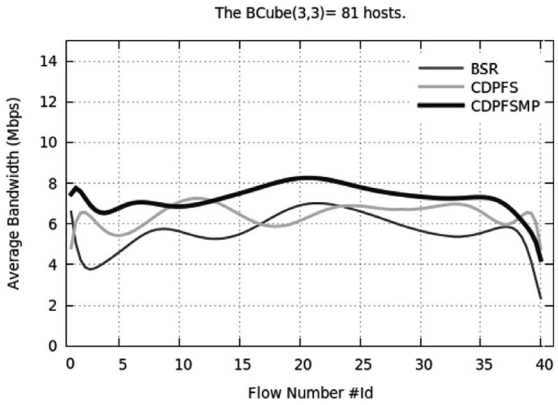
5.2 Dynamic Flow Scheduling

Our central Master periodically (which means every 10 seconds) executes our proposed algorithms to find appropriate paths for each flow. In this duration time (until timeout), if a source server wants to send data to another destination server, the source server will first send the flow information like (S, D) pair to the Master. The Master deals with this new flow by Algorithm 3, and sends the routing rules to the intermediate servers of its path. Finally, the Master notify this source server to start sending its data. We acquire the information from flows sent by the source servers, and we maintain the lifetime of those flows by Monitor. The timeout of TCP connection between source and destination server is 3 seconds when they are not in communication with each other. Our central Master periodically (which means every 3 seconds) acquires all currently flow information from the Monitor, when the Master finds that 30% of the bandwidth of flows is lower than target value (the link bandwidth$/Interf_{capa}$) in the global interface table, which means that numerous new flows occur during this time. The Master will trigger the timeout and immediately execute our proposed algorithm. For all kinds of network conditions and failures, we perform dynamic path selection to adapt to them. During the lifetime of flows, the paths may occur flow collision or break due to various network failures. We deal with the link (switch) failures by Neighbor Maintenance Protocol in the baseline paper, that each server maintains the Neighbor Status Table to record the unreachable neighbors of server. We initialize routing rule and each server communicates with the Master by the lowest level switches and we choose the leftmost server as a manager in each lowest level pod of $BCube(n, 0)$. The Master will periodically poll each manager to acquire the information from Neighbor Status Table. If some of the servers fail or access unreachable due to link failures, we will set a new available path which can communicate with the Master. If the manager fails in some pod on the lowest level of $BCube(n, 0)$, the Master will choose another server as the new manager, and this manager must have information from Neighbor Status Table which records the previous failed manager. For example, if we choose the server 00 as manager in the first pod $BCube(n, 0)$ which is $\{00, 01, 02, 03\}$, when the server 00 fails to reach its lowest level switch $\langle 0, 0\rangle$, the Master chooses another server 01 and acquires the failure information of server 00, and we find the available path $00, \langle 1, 0\rangle, 10, \langle 0, 1\rangle$ to server 00 and using another lowest level switch $\langle 0, 1\rangle$ to communicate with Master.

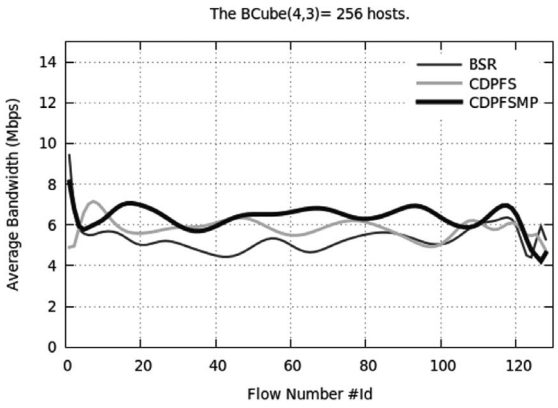
In this paragraph, we illustrate a different way which can let each server (host) as router do the packet forwarding by routing rules to send packets instead of BSR protocol. In the baseline method, BSR uses BCube packet header which stores paths in next hop index (NHI) of every packet. NHI is divided into two parts: DP and DV. DP indicates which digit of the next hop is different from the current relay server, and DV is the value of that digit for single-path, one-to-one traffic such as TCP. The method we use does not modify the BCube header that we can also create the Permutation Set which transfers each server of paths to the same with NHI format by CDPFS. For example, a path 00,01,11 is transferred to $(DP_0 = 0, DV_0 = 1), (DP_1 = 1, DV_1 = 1)$ and the Master send to source server 00. But for one-to-one traffic using multi-path, BSR first divides sending data by m equal chunks for m multi-paths, and each interface of source server sends $1/m$ of the sending data to its destination server with its path by TCP. This method might be undesirable, because each path has a different transmission rate due to various network statuses. So we use MPTCP which allows a TCP connection to use multiple paths and different current congestion control algorithms to adjust the transmission rate until timeout, then we execute new suitable m multi-paths by CDPFSMP. Each server sets and updates the routing rules of those paths which is received from the Master in the period time. The experiment results show that this method is efficient and it can reduce the packet header length, since BCube header is not required.

5.3 Simulation Results

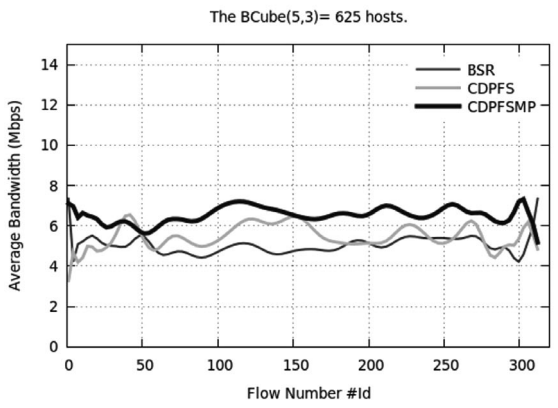
In Fig. 11, we compare the average bandwidth (Mbps) of each flow in BSR, CDPFS with single path and CDPFSMP with multi-path on $BCube(n, k)$ where $k = 3$ by different network topology size: 81, 256, 625, 1296 hosts corresponding to switch port $n = \{3, 4, 5, 6\}$. The maximum bandwidth of each link was 10Mbps in our simulation. Random bijective traffic pattern is applied, which randomly divides servers into equal number of servers by two set A and B, and randomly selects one-to-one mappings which means a host in A sends to any another host in B with uniform probability. This traffic patterns is a good benchmark for us to compare our proposed algorithm with BSR because each host



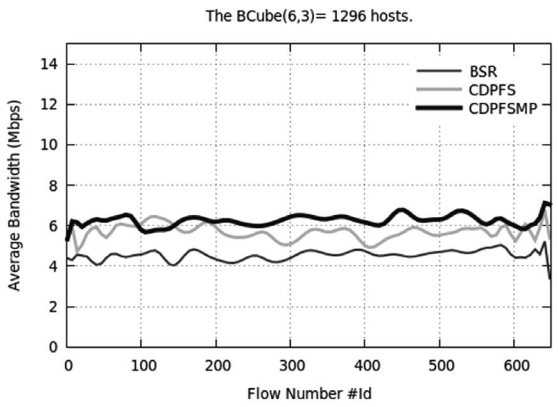
(a) 40 flows



(b) 128 flows



(c) 312 flows



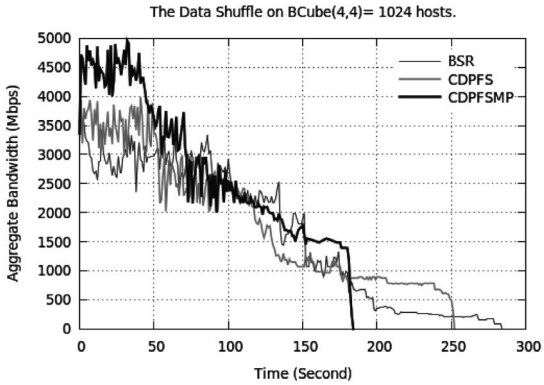
(d) 648 flows

Fig. 11. Average Bandwidth (Mbps) and each Flow ($Flow\#Id$) with Random Bijective Traffic Patterns in $BCube(n,k)$ where $k=3$ and different switch port $n=3,4,5,6$.

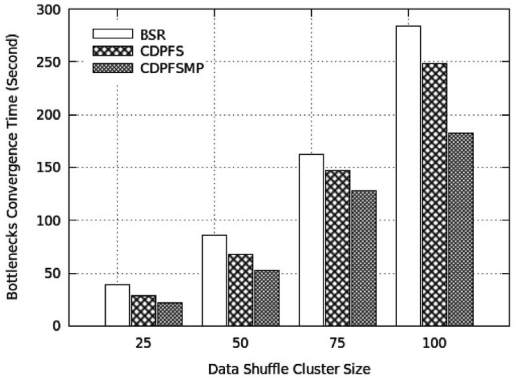
will use at least one interface (NICs), acting as source or destination server in BCube. We know that each path of (S,D) pair must use other servers as the intermediate servers, which means these traffic patterns have high risk of flow collisions. We use different flow size: 40,128,312,648 by random bijective traffic patterns. These flows are sent continuously and the average bandwidth of each flow during 600 seconds is measured which can be seen in Fig. 11 $(a)-(d)$. We can see that CDPFSMP has the highest average bandwidth of each flow. CDPFS is also better than BSR because Master provides a global view, from which we can select an appropriate path with minimal flow collision, instead of using a greedy distributed algorithm to select a path with less delay time by each server. We observed that smaller gaps (smooth line of CDPFSMP in Fig. 11 (d)) of the average bandwidth of each flow when the number of switch port n becomes bigger, because we have more available neighbor pairs to build our NSP graph and have more candidate path set to find the appropriate (less used) paths. We set period time to 10 seconds for our proposed algorithm and for the BSR. Master performs path selection by LDF method of each S,D pair with fairness and selects the same value W_{conge} of interfaces in some of iterations when we choose those higher priority value of interfaces in our proposed algorithms. So some (S,D) pairs have high bandwidth in their paths at start, but their priority value will become lower because counting value increases as time goes. Some (S,D) pairs that fail to compete in previous iteration will be able to select those interfaces currently and achieve higher bandwidth. The result shows that our proposed algorithm is good for load balancing and average bandwidth of all flows achieves 6.2 Mbps with CDPFSMP and 5.8 Mbps with CDPFS, which is higher than 4.3 Mbps with BSR. We have improved 44.1% of the throughput in data traffic patterns because it notably reduces flow collision by our centralized algorithms compared with BSR.

We tested data shuffle operations with various cluster sizes on $BCube(n,k)$ where $k=4$ and $n=4$ in our simulation. We know that in some cloud computing like MapReduce/Hadoop, the time between the end of the Map phase and the beginning of the Reduce phase is known as Data Shuffle process, meaning that data from the mapper tasks is prepared to transfer to the reducer and the tasks of reducer will be run. When the mapper task completes, the format of results are (key, value) which is sorted according to key. If there are multiple reducers, the results of data will be partitioned, and will be written to those disk. For our simulation, we randomly choose 10 servers as a cluster which contains of mappers and reducers and if each server acts as both mapper and reducer, so every server transfers 50MB to every other server in the data shuffle (a 500MB shuffle in a cluster). We simulate this scenario with numerous users using different servers of cluster with cloud computing during data shuffling in $Bube_k$. So the number of cluster size is the number of users using different 10 servers in the same time.

In Fig. 12a, we simulate the 100 of clusters which contains 10 different servers doing data shuffling with 500 MB shuffle in $BCube(4,4)$. The results show the highest aggregate bandwidth is 5 Gbps by CDPFSMP, aggregate bandwidth of CDPFS is 4 Gbps and the aggregate bandwidth of BSR is

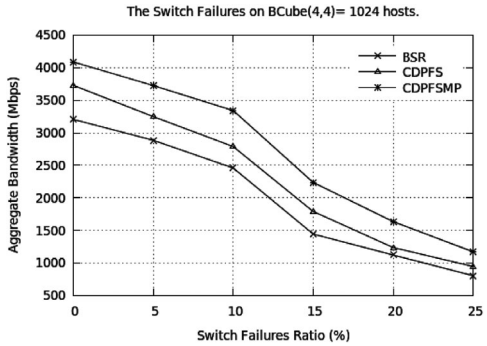


(a) Results for 100 clusters, such that each cluster contained 10 servers

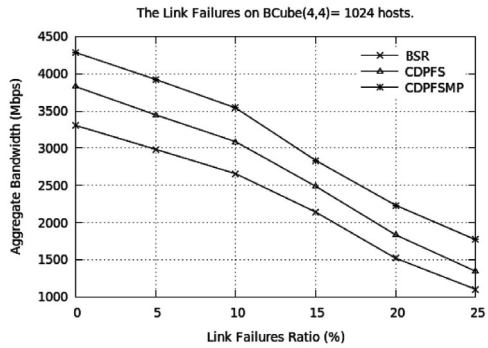


(b) Bottleneck convergence time with different cluster size in data shuffle

Fig. 12. Data Shuffle traffic patterns with 500MB shuffle for our proposed algorithms CDPFS and CDPFSMP, which compared with BSR in $BCube(4,4)$.



(a) Failure of switches



(b) Failure of links

Fig. 13. Aggregate bandwidth throughput in $BCube(4,4)$ with failure of switches and links.

3.5 Gbps at time $0-50$. Because we select suitable paths for each server of cluster with global view in Master, and BSR has many flow collisions due to the high local property in data shuffle. After time 50, the performance of aggregate bandwidth suddenly drops because some of clusters stop transferring transfer data in data shuffle, which means those clusters has higher bandwidth with less used paths in BCube by our proposed algorithms. It may lead to some unfairness for some (S, D) pairs when we first allocate available interfaces to them, but the lifetime of flows are short. So some flows that need high bandwidth to transfer amount of data are allocated insufficient bandwidth. The bottleneck convergence time is that the flow has the smallest throughput and the long convergence time in data shuffle. We can see the bottleneck flows with BSR due to many flow collision in those lifetime of flows at time $200-280$ when cluster size $=100$. Fig. 12b shows the bottlenecks convergence time with different cluster size in data shuffle. We can see that the convergence time of CDPFSMP is 183 seconds, and CDPFS is 252 seconds, and BSR is 287 seconds when the data shuffle cluster size is 100. We improve 36.2% of throughput in data shuffle, which can reduce the convergence time by our proposed algorithms compared with BSR. We also observe a big gap of convergence time with cluster size is 75 and 100, because there are numerous non-active servers (which are not chosen in clusters) with cluster size $=75$ (750 servers) in $BCube(4,4)$, and we can find many available neighbor pairs in NSP graph (more paths with N) and many available intermediate servers in SP graph (more paths with k). But when cluster size grows, we have many same source or destination servers, meaning that the value of $Interf_{capa}$ is not bigger. We deal with those flows with local property by Algorithm 3; the result shows it also better than BSR and efficiently approximates a solution in a short time.

It was convenient to simulate the aggregate bandwidth throughput in $BCube(4,4)$ with failure of switches and links in our Mininet simulator. We used the command: $LinkUp/Down$ to connect/disconnect servers and switches and we disconnected all switch links as switch failures in $BCube(4,4)$ topology. In Figs. 13a and 13b, random data traffic patterns with 1000 flows (which can be chosen repeatedly rather than bijectively) measure the aggregate bandwidth. The results show that the aggregate bandwidth values of our proposed algorithms were better than those of the BSR in spite of numerous switch failures and link failures. When there were no failures, CDPFS, CDPFSMP and BSR provided high aggregate bandwidth throughput, namely 4.1Gbps, 3.7Gbps and 3.2Gbps. However, in situations with numerous failures of switches and links, aggregate bandwidth decreased. We know that the BCube topology is highly fault tolerant because it demonstrates graceful degradation. Because it has numerous NICs for servers and many vacant links, it is highly fault tolerant. So under random switch or link failures, the aggregate bandwidth throughput does not fall drastically. We can see the aggregate bandwidth sharply declining when switch failure ratio changes from 10% up to 15%. It may be unfortunately selection of switches because we randomly chose some of

crucial switches as failures with our traffic patterns, and some (S, D) pairs may be allocated to other alive paths. Various flows take small portions of the total bandwidth; using the same paths can reduce the aggregate bandwidth throughput. The aggregate bandwidth values of our proposed algorithms are close to those of the BSR when the switch failure ratio $= 25\%$, because our algorithms reduce numerous available neighbor pairs for some flows in SP or NSP graph. Many (S, D) pairs find available paths by Algorithm 3 with approximate solutions. Because our algorithms have numerous vacant links that can be used to find available paths, they do not suffer from drastic declines in performance when the link failure ratio increases.

6 Conclusion

In this paper, we propose two variants of centralized dynamic parallel flow scheduling algorithms, CDPFS and CDPFSMP. We use the good properties of BCube topology to divide the appropriate BCube into disjoint subgraphs for each flow computing in parallel. By considering the global network status, we allocate each flow to appropriate paths on BCube. With low-cost, low-level switches, we connect a Master computer to the BCube. The simulation results show that our proposed algorithms can effectively mitigate most flow collisions. Our algorithms improve the load balancing of all data traffic flows and have better overall performance compared to BSR method.

One direction for future research is to use a cluster of physical machines to run large-scale simulations of DCNs with central controllers. Another direction is to use multiple controllers to manage servers in groups of each level within a single DCN. We also propose to analyze recursive defined structures such as DCell and MDCube; ultimately, we propose to extend our algorithms to manage those different data center topologies.

References

[1] P. Kumar and R. Kumar, "Issues and challenges of load balancing techniques in cloud computing: A survey," *ACM Comput. Surv.*, vol. 51, pp. 1–35, 2019.

[2] A. Oussous, F. Z. Benjelloun, A. A. Lahcen, and S. Belfkih, "Big data technologies: A survey," *J. King Saud Uni. Comput. Inf. Sci.*, vol. 30, no. 4, pp. 431–448, 2018.

[3] L. Acquaviva, P. Bellavista, A. Corradi, L. Foschini, L. Gioia, and P. C. M. Picone, "Cloud distributed file systems: A benchmark of HDFS, Ceph, GlusterFS, and XtremeFS," in *Proc. IEEE Global Commun. Conf.*, 2018, pp. 1–6.

[4] L. Li, D. Li, Z. Su, L. Jin, and G. Huang, "Performance analysis and framework optimization of open source cloud storage system," *China Commun.*, vol. 13, no. 6, pp. 110–122, 2016.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.*, New York, NY, USA, 2008, pp. 63–74.

[6] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Conf. Data Commun.*, New York, NY, USA, 2009, pp. 51–62.

[7] C. Guo *et al.*, "BCube: A high performance, server-centric network architecture for modular data centers," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, New York, NY, USA, 2009, pp. 63–74.

[8] H. Wu, G. Lu, D. Li, C. Guo, and Y. Zhang, "MDCube: A high performance network structure for modular data center interconnection," in *Proc. 5th Int. Conf. Emerging Netw. Exp. Technol.*, 2009, pp. 25–36.

[9] G. Wang *et al.*, "C-Through: Part-time optics in data centers," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2010, pp. 327–338.

[10] N. Farrington *et al.*, "Helios: a hybrid electrical/optical switch architecture for modular data centers," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 339–350, 2010.

[11] A. Frömmgen *et al.*, "A programming model for application-defined multipath TCP scheduling," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, 2017, pp. 134–146.

[12] E. Zahavi, I. Keslassy and A. Kolodny, "Distributed adaptive routing convergence to non-blocking DCN routing assignments," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 1, pp. 88–101, Jan. 2014.

[13] F. Zahid, A. Taherkordi, E. G. Gran, T. Skeie, and B. D. Johnsen, "A self-adaptive network for HPC clouds: Architecture, framework, and implementation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2658–2671, Dec. 2018.

[14] S. Chen, Y. Chen, and S. Kuo, "CLB: A novel load balancing architecture and algorithm for cloud services," *Comput. Elect. Eng.*, vol. 58, pp. 154–160, 2017.

[15] H. Zhong, Y. Fang, and J. Cui, "LBBSRT: An efficient SDN load balancing scheme based on server response time," *Future Gener. Comput. Syst.*, vol. 68, pp. 183–190, 2017.

[16] R. D. Devapriya and S. I. Gandhi, "Enhanced load balancing and QoS provisioning algorithm for a software defined network," in *Proc. Int. Conf. Emerg. Trends Inf. Technol. Eng.*, 2020, pp. 1–5.

[17] H. Ghalwash and C. H. Huang, "A congestion control mechanism for SDN-based fat-tree networks," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2020, pp. 1–7.

[18] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," in *Proc. SIGCOMM Comput. Commun. Rev.* vol. 38, no. 2, pp. 69–74, 2008.

[19] R Masoudi and A Ghaffari, "Software defined networks: A survey," *J. Netw. Comput. Appl.*, vol. 67, pp. 1–25, 2016.

[20] T. N. Nishanbayev and M. M. Abdullayev, "Evaluating the effectiveness of a software-defined cloud data center with a distributed structure," in *Proc. Int. Conf. Inf. Sci. Commun. Technol.*, 2020, pp. 1–5.

[21] M. Hamdan *et al.*, "Flow-Aware elephant flow detection for software-defined networks," *IEEE Access*, vol. 8, pp. 72 585–72 597, 2020.

[22] B. Ke, P. Tien, and Y. Hsiao, "Parallel prioritized flow scheduling for software defined data center network," in *Proc. IEEE 14th Int. Conf. High Perform. Switching Routing*, 2013, pp. 217–218.

[23] B. R. Heap, "Permutations by interchanges," *Comput. J.*, vol. 6, pp. 293–298, 1963.

[24] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Cambridge, MA, USA: Cambridge Univ. Press, 1988.

[25] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. 9th ACM SIGCOMM Workshop Hot Top. Netw.*, New York, NY, USA, 2010, pp. 1–6.

[26] G. Wang *et al.*, "Your data center is a router: The case for reconfigurable optical circuit switched paths," in *Proc. ACM Hotnets-VIII*, 2009, pp. 1–6.

[27] Ryu SDN Framework. Accessed: Oct. 18, 2021. [Online] Available: https://ryu-sdn.org/

[28] sFlow-RT. Accessed: Oct. 18, 2021. [Online] Available: https://inmon.com/

Wei-Kang Chung received the MS degree from the Institute of Computer Science and Information Engineering, National Cheng Kung University, Tainan, in 2015. His research interests include data center network and software defined networks.

Yun Li received the BS degree from the Department of Biomedical Engineering, National Taiwan Normal University, in 2015. She is currently working toward the master's degree with the Institute of Computer Science and Information Engineering, National Cheng Kung University, Taiwan. Her research interests include data center networks and software defined networks.

Chih-Heng Ke received the BS and PhD degrees in electrical engineering from National Cheng-Kung University in 1999 and 2007, respectively. He is currently an associate professor of computer science and information engineering, National Quemoy University, Kinmen, Taiwan. His research interests include multimedia communications, wireless network, and software defined networks.

Sun-Yuan Hsieh (Senior Member, IEEE) received the PhD degree in computer science from National Taiwan University, Taipei, Taiwan, in June 1998. He was with compulsory two-year military service. From August 2000 to January 2002, he was an assistant professor with the Department of Computer Science and Information Engineering, National Chi Nan University. In February 2002, he joined the Department of Computer Science and Information Engineering, National Cheng Kung University, where he is currently a chair professor. His research interests include design and analysis of algorithms, fault-tolerant computing, bioinformatics, parallel and distributed computing, and algorithmic graph theory. He was the recipient of K. T. Lee Research Award in 2007, President's Citation Award (American Biographical Institute) in 2007, Engineering Professor Award of Chinese Institute of Engineers (Kaohsiung Branch) in 2008, National Science Council's Outstanding Research Award in 2009, IEEE Outstanding Technical Achievement Award (IEEE Tainan Section) in 2011, Outstanding Electronic Engineering Professor Award of Chinese Institute of Electrical Engineers in 2013, and Outstanding Engineering Professor Award of Chinese Institute of Engineers in 2014. He is an experienced editor with editorial services to a number of journals, including an associate editor of *IEEE Transactions on Computers*, *IEEE Transactions on Reliability*, *IEEE Access*, *Journal of Computer and System Science* (Elsevier), *Theoretical Computer Science* (Elsevier), *Discrete Applied Mathematics* (Elsevier), *Journal of Supercomputing* (Springer), *International Journal of Computer Mathematics* (Taylor & Francis Group), *Parallel Processing Letters* (World Scientific), *Discrete Mathematics*, *Algorithms and Applications* (World Scientific), *Fundamental Informaticae* (Polish Mathematical Society), and *Journal of Interconnection Networks* (World Scientific). He was on the organization committee and or program committee of several dozens international conferences in computer science and computer engineering. He is a fellow of the British Computer Society (BCS) and the Institution of Engineering and Technology (IET).

Albert Y. ZOMAYA (Fellow, IEEE) is currently a chair professor of high-performance computing and networking with the School of Computer Science and the director of the Centre for Distributed and High-Performance Computing, University of Sydney. He has authored or coauthored more than 600 scientific papers and articles and has coauthored and edited more than 30 books. A sought-after speaker, he has delivered more than 190 keynote addresses, invited seminars, and media briefings. His research interests include several areas in parallel and distributed computing and complex systems. He was the editor-in-chief of *IEEE Transactions on Computers* from 2010 to 2014 and *IEEE Transactions on Sustainable Computing* from 2016 to 2020. He is currently the editor-in-chief of *ACM Computing Surveys*. He was the recipient of 1997 Edgeworth David Medal from the Royal Society of New South Wales for outstanding contributions to Australian Science, the IEEE Technical Committee on Parallel Processing Outstanding Service Award (2011), IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing (2011), IEEE Computer Society Technical Achievement Award (2014), ACM MSWIM Reginald A. Fessenden Award (2017), and the New South Wales Premier's Prize of Excellence in Engineering and Information and Communications Technology (2019). He is a decorated scholar with numerous accolades, including American Association for the Advancement of Science and the Institution of Engineering and Technology (U.K.). He is an elected fellow of the Royal Society of New South Wales and an elected foreign member of Academia Europaea.

Dr. Rajkumar Buyya (Fellow, IEEE) is currently a professor of computer science and software engineering, a future fellow of the Australian Research Council, and the director of the Cloud Computing and Distributed Systems Laboratory, University of Melbourne, Australia. He is also the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He has authored more than 500 publications and four text books including *Mastering Cloud Computing* published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He also edited several books including *Cloud Computing: Principles and Paradigms* (Wiley Press, USA, Feb 2011). He is one of the highly cited authors in computer science and software engineering worldwide with h-index=108, g-index=225, and 55800+ citations. He has led the establishment and development of key community activities including the foundation chair of the IEEE Technical Committee on Scalable Computing and five IEEE ACM conferences. He was the recipient of Award of 2009 IEEE TCSC Medal for Excellence in Scalable Computing from the IEEE Computer Society TCSC and 2010 Frost and Sullivan New Product Innovation Award. Manjrasoft has been recognised as one of the Top 20 Cloud Computing companies by the Silicon Review Magazine. He was the foundation editor-in-chief of *IEEE Transactions on Cloud Computing*. He is currently the co-editor-in-chief of *Journal of Software: Practice and Experience*, which was established more than 40 years ago. Microsoft Academic Search Index ranked him as the world's top author in distributed and parallel computing between 2007 and 2015. "A Scientometric Analysis of Cloud Computing Literature" by German scientists ranked Dr. Buyya as the World's Top-Cited Author and the World's Most-Productive Author in Cloud Computing. Software technologies for Grid and Cloud computing developed under Dr. Buyya's leadership have gained rapid acceptance and are in use at several academic institutions and commercial enterprises in 40 countries around the world.