ParaCon: A Parallel Control Plane for Scaling Up Path Computation in SDN

Kun Qiu, Siyuan Huang, Qiongwen Xu, Jin Zhao, Xin Wang, Stefano Secci

ParaCon: A Parallel Control Plane for Scaling Up Path Computation in SDN

Kun Qiu, Siyuan Huang, Qiongwen Xu,
Jin Zhao, *Member, IEEE*, Xin Wang, *Member, IEEE*, Stefano Secci, *Senior Member, IEEE*

***Abstract*—The fundamental tasks of the control plane in Software Defined Networking (SDN) are to customize forwarding policies for the data plane and to provide global network view for applications. The logically centralized control plane design brings benefits in terms of network programmability and can largely ease network management. However, it also increases efficiency concerns. One practical control plane challenge is path computation, because it can require a significant amount of computation load if the network scale is large and the path requests from applications are frequent. In this paper, our goal is to build a high-performance control plane for path computation using multiple controllers. Previous works attempt to improve control plane efficiency by balancing only the load for data plane behavior between multiple controllers. Going beyond conventional wisdom, we designed ParaCon, a solution we propose to speed up the control plane by distributing the load of path computation. We also address the consistency and synchronization overhead challenges related to ParaCon design. To the best of our knowledge, ParaCon is the first attempt that utilizes node parallelism in SDN path computation. We evaluated ParaCon using both Mininet and real-world clusters. Our results show that the path computing time of ParaCon can achieve a speedup of 10x over Floyd (used in POX) and Dijkstra (used in ONOS) baseline implementations for networks with hundreds of nodes.**

***Index Terms*—software-defined networks, distributed algorithms, performance management.**

I. Introduction

RECENTLY, *Software-Defined Networking* (SDN) became a hot topic in both academia and industry. SDN brings the benefit of network controllability by separating the control plane from the data plane. With OpenFlow [1], a southbound interface specified by the ONF [2], a single centralized SDN controller can control the behavior of all the switches in a network. Despite its advantages, the single centralized structure may lead to scalability and performance challenges [3]. The SDN controller can easily suffer from performance degradation resulting from a rapid increase of the network scale. Thus, such a design cannot scale up for real-world deployments, which is one of the reasons why current SDN deployments do not seem to take place in carrier-grade networks.

Therefore, improving the SDN controller performance and scalability is one of the major concerns in realizing SDN benefits. To address this challenge, academia and industry have proposed a diverse range of methods. One approach is to optimize the system architecture of existing single-server controllers for higher efficiency. To achieve better execution efficiency, some high-level language based controllers were redesigned as low-level language based controllers [4]. Another approach is offloading (partial or full) load from the controller to switches [5], [6]. Alternatively, utilizing multiple controllers in managing the SDN provides yet another choice [7]–[12] and the distributed control plane designs are becoming increasingly popular. For example, ONOS [11] claims to be the first carrier-grade distributed SDN controller, and Onix [10] is used in Google's B4 [13] as their distributed control plane. Moreover, a new control plane design, which is based on a database approach was recently proposed [14].

Existing multi-controller solutions aim to tackle the scalability issue in handling more switches, which may come at a prohibitive cost for maintaining and computing network information. More specifically, the SDN controllers periodically monitor the links, the forwarding tables and compute the network information such as the routing path and spanning tree of the global network. Among the variety of network information to be computed and processed in SDN controllers, the routing path is the most important one. Performance of mainstream SDN applications such as traffic engineering and content delivery networking [15] is tightly coupled with the performance of path computation. Path computation in a large-scale network indeed requires a significant amount of computation load. However, the path computation method in most of the existing SDN controllers is unable to meet the performance needs. As an example, ONOS [11], a widely used SDN controller, cannot achieve the intended throughput in a 205-switch topology due to its inefficiency in path computation. It is reported that the current ONOS design, though with multiple controller instances, only uses one single controller instance to compute paths, thus not fully utilizing the potential of parallelism in path computation [16].

A straightforward approach to scaling path computation is to leverage the multiple controller instances in a distributed SDN control plane. For example, a possible solution is to implement parallel Dijkstra or Floyd algorithms [17] on multiple controllers. However, naïvely applying node parallelism to these algorithms within the context of the SDN control plane has suffered from two major challenges in practice.

First, the computation overhead is sensitive to topology changes. Even when there are only minor changes in the topology, offline algorithms need to start path computation

K. Qiu, S. Huang, Q. Xu, J. Zhao, X. Wang are with the School of Computer Science, Fudan University, Shanghai 201203, China; Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China (mail:qkun;syhuang14;qyxu15;jzhao;xinw@fudan.edu.cn). S. Secci is with the Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, Paris, France (mail: stefano.secci@upmc.fr)

on the updated topology from scratch. More specifically, all-pairs path computation on the updated graph involves roughly $O(n^3)$ time complexity [17]. Due to the potentially large size of SDN networks, frequent or even moderate network changes will pose a substantial amount of computation overhead. Second, parallel path computation needs to synchronize path information between involved controllers. Existing algorithms for parallel path computation are usually based on the Bulk Synchronous Parallel Computing (BSP) model [18]. Fig. 1(a) shows that blocking synchronization in the path computation reduces the efficiency significantly. Moreover, synchronous path computation does not adapt 'online' property in the network either. With the increase in the number of involved SDN controllers, the synchronization overhead will result in significant performance degradation.

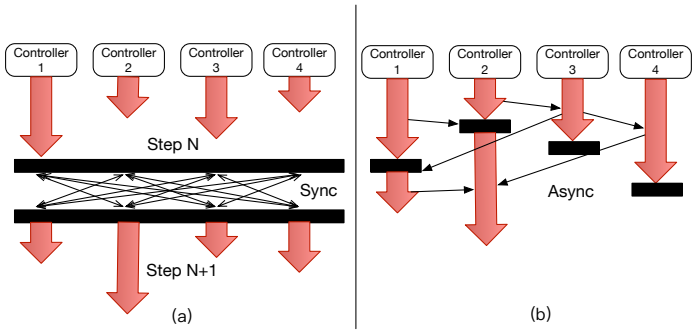


Fig. 1. Synchronous and asynchronous path computation. In synchronous computation, the synchronization process separates path computation into several steps, and synchronization happened between the steps. In asynchronous computation, synchronization happened during the computation.

In this paper, we propose ParaCon, a distributed SDN control plane that addresses the performance challenges of distributed path computation of SDN. Under this design, SDN programmers can settle path computation without worrying about the details of synchronization while achieving the benefits offered by parallel computing. To the best of our knowledge, ParaCon is the first control plane architecture to utilize parallelism in path computation for SDN. More specifically, ParaCon introduces several new features to make path computation in multiple controllers efficient. It introduces a new distributed model in SDN that is capable of asynchronous path computation like Fig. 1(b) between multiple controllers.

ParaCon includes an online algorithm that enables all SDN controllers to make incremental changes to the computed path results upon receiving a particular topology change while existing approaches need to restart the computation from scratch. It makes ParaCon more efficient in scenarios with frequent topology updates. Moreover, it seeks to minimize the overhead between multiple controllers by designing an asynchronous algorithm. In this algorithm, different controllers can synchronize information without blocking. Finally, it introduces a hybrid consistency model to maintain the topology-related information and path information between controllers during the asynchronous path computation. It utilizes both *strong consistency* and *eventual consistency* in the maintenance of this information for further improving the performance. Our theoretical analysis shows that the online algorithm has a bounded convergence time. Moreover, experimental results show that ParaCon can significantly reduce the path computation time when the scale of the network is increasing.

The rest of the paper is organized as follows. Section II gives an abstraction of the problem space and challenges. Section III gives an overview of ParaCon architecture. Section IV gives the details of the algorithm. We present our evaluation in Section V and related work in Section VI. We conclude in Section VII.

II. Problem Statement

As we have mentioned above, we seek to minimize the overhead between multiple controllers by designing an asynchronous path computation algorithm. To motivate the asynchronous computing approach, we need to consider several aspects, among which the consistency problem is the most important one. Namely, we need to consider what information is to be synchronized, when to synchronize it, and how to minimize the synchronization overhead during the path computation.

Generally speaking, to compute path information in a distributed fashion, we need to maintain at least the following information:

1) **Link view:** The topology-related information, i.e., the link/switch status.
2) **Association information:** The association between switches and controllers.
3) **Path view:** The path information in each controller during the path computation.

From the example described in Fig. 2, we can see the link view (topology-related information) needs to be synchronized between all controllers to make sure the applications in different controllers get the same global topology. Otherwise, an inconsistent link/switch status will make an inconsistent result of path computation. We refer to this synchronization policy as *strong consistency* policy.

Before we define the consistency policy formally, we introduce some notations. Let H be some new information updated to a controller such as a link state or path change. For a given controller $c \in C$, where C is controller set, p_c indicates its status such as $p_c \in \{NO, YES\}$: when in status NO, a controller has not received or accepted H, when in YES it has received or accepted H. The control plane is defined as in a consistent state after H only if $\forall c \in C, p_c = YES$. The multi-controller SDN control plane is said to be *available* when it can reply to control plane request messages. Otherwise, it is said to be *not available*. Different consistency models impose diverse levels of strictness on the availability of the control plane during in the inconsistent state. A strong consistency policy only allows new changes to be accessed after they have been applied to all controllers.

Definition 1: (**Strong Consistency Policy**) After a control plane global information update H, supposed to change the status of the receiving controller from NO to YES, with a strong consistency policy, the control plane is available only if $\forall c \in C, p_c = YES$.

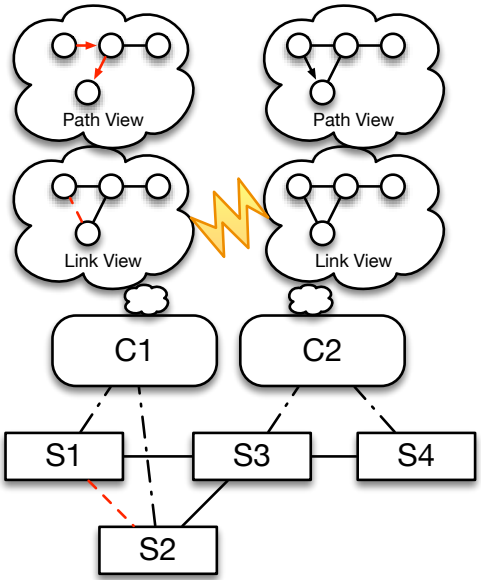


Fig. 2. Suppose there are four switches $S1$, $S2$, $S3$ and $S4$ in a network. Controllers $C1$, $C2$ are added to the network with the association. The topology-related information needs to be synchronized to controller $C1$ and controller $C2$ to make sure they have the same path view on switches $S1$, $S2$, $S3$ and $S4$. Otherwise, an inconsistent link/switch status will make an inconsistent result of path computation. For example, if we do not synchronize the link status when the link between $S1$ and $S2$ fails (we use a red line between $S1$ and $S2$), we may get an inconsistent path view (the path from $S1$ to $S2$ is $S1 \rightarrow S2 \rightarrow S3$ in $C1$, but $S1 \rightarrow S3$ in $C2$).

Meanwhile, association information also needs to be synchronized to each controller with a strong consistency policy, or the assignment between controllers and switches may be incoherent if migrations of switches happened in the meantime.

However, synchronizing path view (path information) during the path computation, even at a minimum required frequency, may eventually lead to network congestion due to the excessive overhead in control message exchange under a strong consistency policy [19]. For example, *Two-Phase Commit* [20] or *Paxos* [21] are revealed to be inefficient by signaling overhead.

Therefore, it is necessary to reduce the synchronization overhead of path information. Considering that the three types of information we mentioned above involve different transmission overhead, we need to study the amount of exchanged messages quantitatively. We assume the exchange process on the control plane adopts a message-passing model [22] [23], and we consider the minimum number of messages transmitted for reflecting the change in a link/switch status and a switch-controller assignment is 1 unit. We have two types of operations: synchronizing topology-related information and synchronizing path information. Synchronizing topology-related information involves one simple message that is associated with one switch. In contrast, synchronizing path information requires more transmission overhead. If there are n switches, the controller needs to transmit up to n^2 messages because the change is associated with the path information that has n^2 paths at most.

Accordingly, synchronizing the path information can be much more resource-intensive than other operations. This motivates us to minimize the overall overhead in synchronization by reducing message transmission for path information. Different from *strong consistency* policy, an *eventual consistency* policy is a specific form of weak consistency. The SDN control plane guarantees that if there are no new updates such as link status change, eventually all controllers will return the same path information, which is computed based on the topology-related status. Consequently, an *eventual consistency* policy can be used for synchronizing path information to reduce the transmission overhead.

Instead of synchronizing path information to all controllers immediately, an eventual consistency policy allows a delay in synchronization. The path information can be accessed during the time with inconsistent results.

Definition 2: (**Eventual Consistency Policy**) After a control plane global information update H, supposed to change the status of the receiving controller from NO to YES, with an eventual consistency policy, the control plane is available also during the inconsistency window going from the instant when $\exists! c \in C \mid p_c = YES$, and the instant when $\forall c \in C,\ p_c \neq YES$.

Naturally, the *eventual consistency* results in inconsistent path information during the inconsistency window. To avoid inconsistency effects as much as possible, each controller should independently compute the path information based on topology-related information to ensure consistency. However, with the increasing scale of the network, the computation overhead will inflate rapidly since the time complexity of all-pairs path computation algorithm is $O(n^3)$ in general [17].

In summary, finding an equilibrium between synchronization overhead and computation time is difficult. It motivates us to propose a distributed architecture, which neither significantly increases synchronization overhead nor obviously increases computation time for parallel path computation.

III. The ParaCon Architecture

The core objective of ParaCon is to offer a high-performance SDN control plane for path computation with minimal transmission overhead between multiple controllers. Fig. 3 shows the architecture of ParaCon. Our architecture is based on a distributed structure with an asynchronous parallel algorithm. We will give a brief introduction to its two components.

A. The Distributed Structure

1) Switch: The data plane consists of switches and links. If a switch needs to forward a packet to other switches, it will request a path between two switches from the controller.

2) Topology Abstraction: We abstract the topology using a graph [24]. The global topology view includes both a *link view* and a *path view*. Link view is composed of nodes and edges, where nodes are switches and edges are links. Link view indeed reflects the topology of the underlying network. Path view provides information on the best route between any two switches. Path view indeed reflects a snapshot of the weights for the all-pairs shortest paths at a given time. As the

algorithm progresses, the path view will be updated till the results at all the controllers converge.

3) Controller and Allocator: The control plane is distributed. Each controller monitors the status of their switches/links, which are allocated by the Allocator. If the status of switches/links has changed, the controller will synchronize the topology-related information to other controllers by utilizing the *strong consistency* policy. Thus, all controllers are sharing the same global topology view based on switches/links status. Meanwhile, each controller also computes the path information of their switches and receives the topology-related/path information from other controllers.

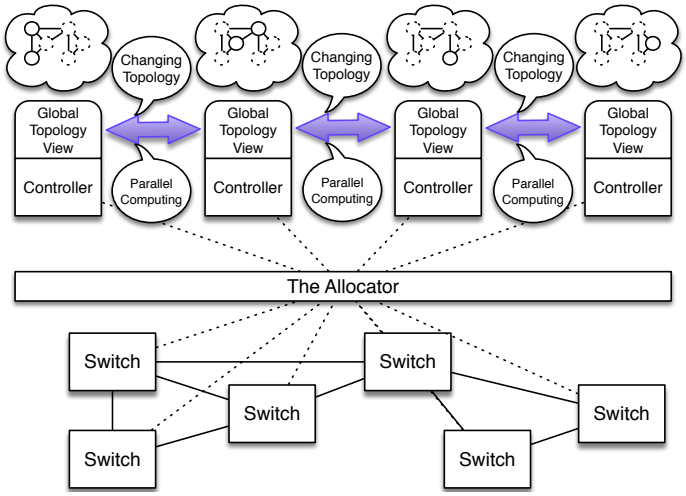


Fig. 3. ParaCon structure. All controllers are distributed and sharing the global topology view. Switches are allocated to different controllers by an allocator, and each controller only computes the path information of its switches. All the controllers can compute the path together by parallel computing.

B. The asynchronous parallel algorithm

An asynchronous parallel algorithm is introduced, which uses the *eventual consistency* policy to improve the efficiency of the distributed controller design and eliminates the effects of the inconsistency due to the *eventual consistency* policy. More specifically, the design choice we choose is to synchronize the topology-related information (link view) and the affiliation with a strong consistency policy and to synchronize path information (path view) with an eventual consistency policy, with the goal of making a correct balance between transmitting overhead and computing overhead.

Furthermore, different from existing algorithms such as Floyd-Warshall [17] or Dijkstra [17], which are used in existing controllers, the proposed algorithm utilizes the 'online' property to reduce synchronization overhead. Upon a new topology change event, it does not need to wait until the end of updating the previous event. Instead, our algorithm can handle the new event immediately.

IV. Parallel Path Computation

As mentioned above, we synchronize the link/switch status and the affiliation of switches with a *strong consistency* policy. To this end, we use *Two-Phase Commit* policy to submit the change so that all controllers can receive it. This section details the asynchronous path computation algorithm. The algorithm is more complicated than the synchronous one because of its interweaving with the *eventual consistency* policy. We use several steps to describe our algorithm.

A. Key notions

Switch and controller set: There are switches and controllers in a Software-Defined Network. The controllers are spatially distributed, and each switch is only controlled by one controller. Set S stands for the switches and C stands for the controllers. There are n switches s_1, s_2, s_3, ..., s_n in set S, and m controllers c_1, c_2, c_3, ..., c_m in set C. The element in vector $Con(s)$ means the controller which controls switch s.

Link view: Link view is a graph $L(S, E)$ where S stands for the switches set and E stands for the links set. All elements in set E are two-tuples $(x, y), x, y \in S$. The link status between two switches is expressed in an integer as the weight w. The link view is thus represented as an *adjacency matrix*. $L_{a,b}$ stands for the weight of the link between s_a and s_b. If there is no link between s_a and s_b, the $L_{a,b} = -1$.

Path view: The path view provides information on the cost of the best (usually the shortest) route between any two switches. A path view can also be regarded as a weighted full graph $T(S, W)$, where S stands for switches and W stands for the weight of the shortest path between two switches. The weight of the shortest path is expressed as an integer, which is calculated based on link view $L(S, E)$. The path view can be implemented as an *adjacency matrix* as well. $T_{a,b}$ stands for the weight of the shortest route between s_a and s_b. $T_{a,*}$ stands for the weights for all the shortest routes beginning with s_a.

Forwarding table: Each switch has a list of forwarding rules to indicate the next hop of the best path to other switches. When we compute *path view*, we can also get the corresponding forwarding table for the shortest paths. The forwarding table is described as a *matrix*. $F_{a,b}$ represents the next hop from s_a to s_b, which is the first constituent edge along the shortest path from s_a to s_b.

In summary, we use 1) link view L as the topology-related information; 2) vector $Con(S)$ as switch-controller association; and 3) path view T and forwarding table F as path information.

B. Centralized approach

We discuss a centralized algorithm first. There are several centralized algorithms for computing the paths in a network. The well-known Floyd-Warshall [17] and Dijkstra [17] algorithms have an $O(n^3)$ time complexity for the all-pairs shortest path (APSP) problem in a network with n nodes. These algorithms have been used in various SDN controllers. For example, POX uses the Floyd-Warshall algorithm. However, both algorithms are offline algorithms, i.e. they can not deal with any link change while the algorithm is in progress. In contrast with existing approaches, we first design an online APSP algorithm based on Moore [17] algorithm and then tailor it for parallel scenarios. Note that Moore algorithm is

an optimized version of the Bellman-Ford algorithm used for solving the single-pair shortest path (SPSP) problem.

The centralized algorithm uses a queue at each controller to store the nodes that will be checked for path cost update. We briefly explain how it works. If the queue is not empty, the algorithm will fetch the head of the queue and relax it. Relaxing a node involves updating the cost of existing paths if the total cost can be reduced by passing through the node. If the node can be relaxed, which means that there is at least one path which becomes shorter by passing through this node, its neighbors will be inserted into the queue.

The correctness of our algorithm can be proved by contradiction. We can prove all-pairs shortest path are computed when the queue is empty. We use a contradiction to give simple descriptive proof: First, we assert the following Lemma: shortest path is composed of shortest path. In other words, for a shortest path $x \rightarrow \ldots \rightarrow a \rightarrow \ldots \rightarrow b \rightarrow \ldots \rightarrow y$, any sub-path $a \rightarrow \ldots \rightarrow b$ must be a shortest path. The proof is based on the notion if there was a path shorter than any sub-path $a \rightarrow \ldots \rightarrow b$, we can replace the sub-path with the shorter path to make the whole path shorter. We want to prove the Proposition: all-pairs shortest paths are computed when the queue is empty. Suppose for the sake of contradiction that is not true: there exists one path that is not the shortest one when the queue is empty. We assume the path is $x \rightarrow \ldots \rightarrow u \rightarrow \ldots \rightarrow y$. With the Lemma we proposed above, we can find u that satisfy the part of the path $x \rightarrow \ldots \rightarrow u$ is the shortest path, and the other part of the path $u \rightarrow \ldots \rightarrow y$ is not the shortest path. Thus, we can always find another path $u \rightarrow v \rightarrow \ldots \rightarrow y$ passing through v to make it shorter. To relax v, the algorithm will put u and its neighbor v into the queue, which is a contradiction to the Proposition.

Example: Suppose we have 4 switches and 1 controller. The *link view* and *path view* are shown in Fig. 4. When we change the link status between s_0 and s_1 to 30. The changes of *topology view* and the queue P and Q are shown in Fig. 4.

The approach can deal with link status change by just putting switches into the queue and waiting for the result. Because of its 'online' property, the time complexity is $O(p*e*n)$, where e is the number of links and p is related to the diameter of the graph. In general, if the network diameter is small, e.g., the diameter of some topology in data-center is less than 3, the iterations will be completed very fast. In contrast, some extreme topologies with a large diameter made the p become large due to an iteration oscillation. For example, a topology with a large linear subgraph (in which most nodes have a degree $\leq$2) will lead to more iterations. Fortunately, the graph of real network topology rarely has a linear subgraph, and the oscillation can be with further optimizations.

The algorithm runs in a centralized way in each controller, which has no additional node-parallelism advantages compared to the other existing algorithms.

C. Asynchronous parallel path computation

Adopting a parallel path computation approach makes it all possible for all controllers to compute the path collaboratively.

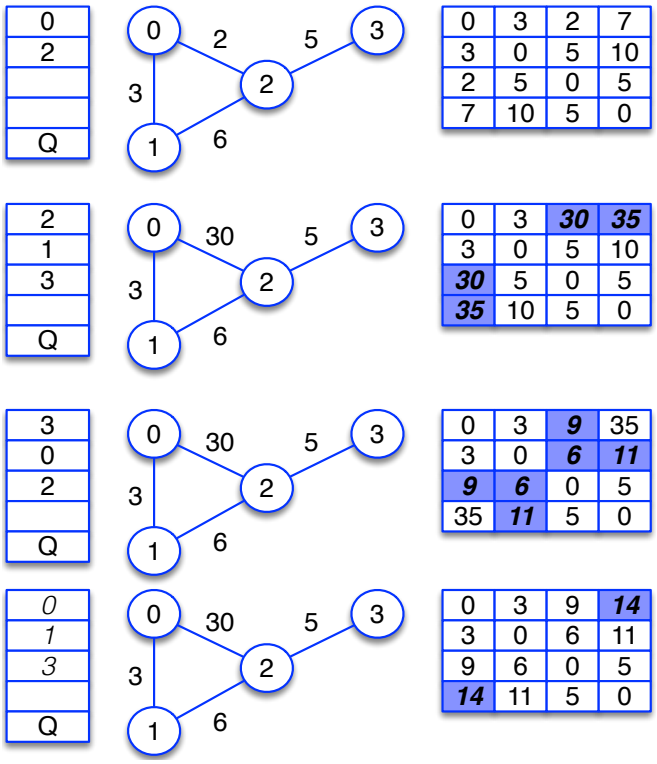


Fig. 4. The centralized version. There are several steps to compute the path. The right matrix shows the weight of shortest routes between switches. We use the blue square with the blue background to indicate that the path view is updated by the controller. After link $0, 2$ has changed to 30, we put 0 and 2 to the queue Q. The algorithm gets the head of Q and computes the shortest route of it. After processing 0, we add the adjacent: 1 and 3 to Q because there are some changes when processing 0. If the queue is empty, the algorithm stops. In this example, the algorithm uses 6 steps: 0, 2, 1, 3, 0, 2 to finish the algorithm.

We assign the switches and computation load to different controllers. However, due to the heterogeneity in computing power and transmission overhead, the inconsistency of *path view* becomes severe under this design. A straightforward approach to consistency is to add a barrier to the queue after an iteration has been done. The barrier is used for synchronizing the path computation process for all controllers. If the head of the queue is a barrier, the path computation thread will be blocked until other controllers have reached the barrier. However, adopting a barrier will increase the synchronization delay substantially.

To turn our algorithm into an asynchronous one, we redesign the barrier to trade consistency for synchronization overhead. Thus, we employ the eventual consistency. A weak synchronization signal, or SYNC, is introduced which allows a controller to send its *path view* to other controllers without blocking any computation thread.

Another concern is that we need to deal with the rising conflicts when synchronizing the *path view* in different controllers with the eventual consistency. It is clear that some conflicts must be overwritten, and some conflicts cannot be overwritten. Otherwise, the algorithm may take extra time to compute the path (e.g., if a new path is overwritten by an old one), and it can significantly increase the overhead in our control plane.

Algorithm 1: Path computation (parallel, online, asynchronous)

Input: Link view L, The number of switches n, The queue of switches need to be maintained Q_{id}
Output: Path view T

1 **SYNC** is a synchronization signal that makes the current controller synchronize the path view to other controllers.
2 **while** *True* **do**
3 **while** *Queue* Q_{id} *is not empty* **do**
4 $now \leftarrow Q_{id}.head()$
5 **if** now *is a* ***SYNC*** **then**
6 **for** *Each controller* i *except current one* id **do**
7 Send queue Q_i to controller i.
8 Send Path view $T_{i,*}$ to controller i.
9 **continue**
10 **for** *Each adjacent node* i *of node* now **do**
11 **if** $T_{now,i}$ *can be relaxed* **then**
12 Relaxing Path View $T_{now,i}$.
13 Get t, which minimized prior calculation $T_{now,i}$.
14 Forwarding Table $F_{now,i} \leftarrow t$
15 **if** T_{now} *has changed* **then**
16 Put all adjacent nodes k into $Q_{Con(k)}$.
17 Put **SYNC** into $Q_{Con(k)}$.

We introduce the following policy for path overwriting.

1) If the received path information is *newer* than the existing one, the existing one needs to be overwritten
2) If the received path information is *older* than the existing one, the existing one need not be overwritten

The chronological order, either *newer* or *older*, is based on the logical clock. This motivates us to use **Vector Clock** to establish the global logical clock for each element in path view T.

Besides *newer* conflicts and *older* conflicts, there still exist other conflicts, which are neither *newer* nor *older* in vector clock that should be solved manually. Fortunately, under ParaCon design, these conflicts can be overwritten immediately, because directly overwriting path view is equivalent to changing the sequence of the queue Q in the centralized approach. This operation does not affect the results of the path computation which is an iteratively based approach.

Algorithm 1 is the ParaCon parallel path computation algorithm we proposed; it keeps the path view *eventually consistent*. The algorithm is deployed into all controllers and makes controllers compute the paths concurrently. The algorithm has a loop and is separated into three parts. First, it fetches the head of the queue and checks if the head is a synchronization signal. If yes, it sends its path view to other controllers and continues the loop. Otherwise, it relaxes its path view of the head and puts the adjacency nodes which are relaxed by the head to the queue. At last, it pushes a synchronization signal into the queue. The algorithm can do path computation in multiple controllers efficiently. During the inconsistency window, the controller can fall back to on-demand mode (utilizing Dijkstra/Floyd algorithm) to obtain correct results.

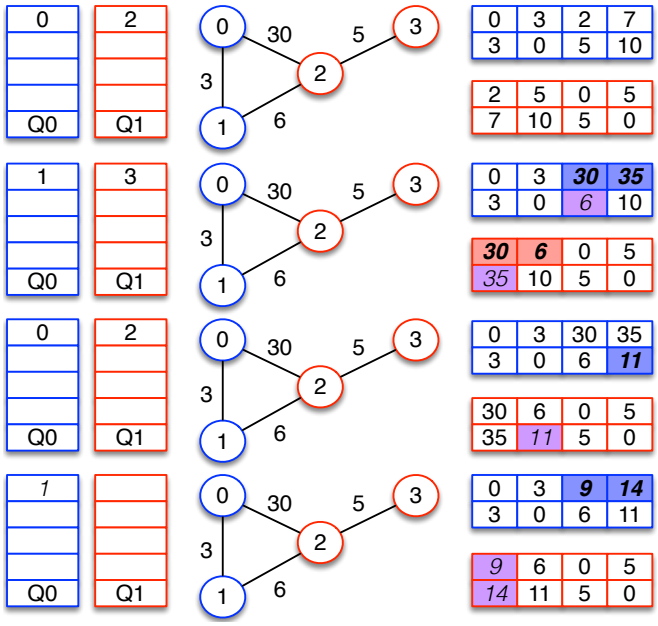


Fig. 5. The distributed version. 4 switches are associated with 2 controllers. We use blue color (blue queue, nodes, and matrix) to indicate controller $C1$ controls these elements; we also use the red color (red queue, nodes and matrix) to indicate controller $C2$ controls these elements. Thus, the matrix on the right is separated into two parts. The top two rows are maintained by controller $C1$ and the bottom two rows are maintained by controller $C2$. We use the square with the blue (red) background to indicate the path view is updated by controller $C1$ ($C2$). We also use the square with the purple background to indicate the path view is updated because of data synchronization from other controllers. For example, in step 2, '6' in $C1$ is synchronized from $C2$, and '35' in $C2$ is synchronized from $C1$. They only use 3 steps to finish the algorithm. Each controller computes the path and synchronizes the path information with other controllers at the end of each step.

Example: Suppose we have a topology with 4 switches and 2 controllers. Switch s_0 and s_1 are controlled by controller c_0, while s_2 and s_3 are controlled by c_1. When we change the link status (weight) between s_0 and s_2 to 30, the changes of *path view* and the queue Q_0, Q_1 are shown in Fig. 5.

D. Convergence analysis

We use an epidemiology model [25]–[27] to analyze our algorithm. Usually, the epidemiology model is used as an analyzing tool for network protocols such as Gossip or RIP (Routing Information Protocol) [28], [29].

Firstly, we define the symbols. We suppose there are n nodes (switches) in the graph (link view), and we assume k as the average number of nodes relaxed in the graph when SYNC is received (from line 5 to line 9 in Algorithm 1). To put this another way, when SYNC is received, the cost of existing paths can be updated as they can be shorter through these k nodes. Thus, we define β as update rate with $\beta = \frac{k}{n}, (0 < \beta < 1)$. As we have mentioned in the centralized approach, each node has two states: it can be relaxed or cannot be relaxed. Nodes will stay in the second state (cannot be relaxed) eventually and will not convert into the first state (can be relaxed).

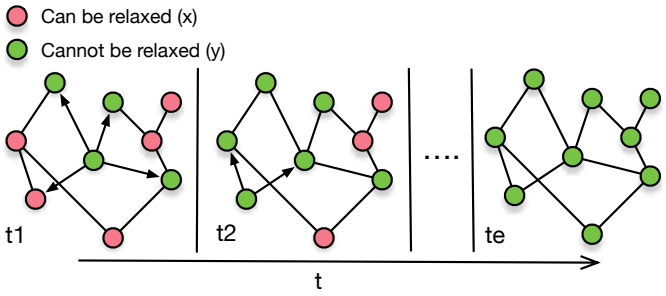


Fig. 6. An example for explaining the convergence analysis. There are $n = 9$ nodes in the graph. Each node has two states: it can be relaxed (red) or cannot be relaxed (green). There are 4 nodes relaxed at time t_1, but only 3 nodes stay in the second state (the left one may have chances to be relaxed by other nodes). At time t_2, 2 nodes relaxed in the graph, and both of them stay in the second state. Finally, all nodes will be in the second state eventually at the last time t_e. We assume k_n as the number of nodes relaxed at time t_n. In this example, at least we have $k_1 = 3$, $k_2 = 2$. k is the average number of nodes relaxed in the graph, which is the average number of $k_1, k_2, \ldots, k_e$. Obviously, k is closely related to the topology, such as the average degree of nodes or the diameter of the graph.

At any time t, we assume the number of nodes in the first state is y_t and the number of nodes in the second state is x_t. Thus, $x_0 = n, y_0 = 1$ when a link status change happened, and at any time, $x_t + y_t = n + 1$. Also, we have

$$\frac{dx}{dt} = -\beta xy$$

and we get

$$y = \frac{n+1}{1 + ne^{-\beta(n+1)t}}$$

Obviously,

$$\lim_{t\to\infty} y = n + 1$$

Thus, all nodes will be in the second state finally.

We assume $t = clog(n)$, we can get

$$y \approx (n+1) - \frac{1}{n^{ck-2}}$$

We give an example to describe these symbols in Fig. 6. As the update rate β is related to k, which is influenced by the property of the graph, i.e., the diameter and the average degree of nodes, the number of nodes that are not in the second state at a given time t, is widely ranged. For example, k in a full mesh topology is larger than k in a linear topology, and obviously, the convergence time of a graph with a small diameter is less than the time of a graph with a large diameter. We have confirmed this conclusion by evaluation results in section V.

V. The ParaCon Prototype

Our work is motivated by the intuition that the performance of multi-controller solution can be improved using parallel computing in a cluster. We believe the most exciting results from the efficiency of the structure can be achieved in a real-world implementation.

A. Implementation

We implemented ParaCon structure using commodity hardware. The distributed control plane is based on 10 Dell PowerEdge 2950 (Xeon E5405 with 2G RAM) servers. We have four virtual machines in one server, and each virtual machine handles one controller. Therefore, we have 40 controllers at most. The controller is based on a modified POX, in which we replace the module of path computation by our parallel version. The switches are provided by Mininet [30]. We convert topologies into a Python program, which is used as topology input by Mininet. Due to its programmability, we can use any topology to test our distributed control plane. We will discuss the details as follows.

1) Path View Module: The path view module is separated into three threads: computing the path, handling event of topology-related information change, transmitting and receiving messages. The module uses an adjacency matrix to store the path view. We also implement a *Two-Phase Commit* to submit the change of link/switch status and assignment of switches. The transmitting and receiving thread uses TCP to communicate with its counterpart in other controllers. To improve efficiency, we employ *eventlet*, a concurrent networking Python library that provides highly scalable non-blocking I/O [31].

2) Modified POX: We implement ParaCon based on POX, a widely used controller in SDN research community. Two modules in POX are modified, namely, `Openflow.Discover`, which controls the topology discovery, and `Forwarding.l2_multi` which controls path computing. Briefly, first, POX catches *link change* event by LLDP protocol using `Openflow.Discover`, and send to path view module. Next, our module sends an event of topology-related information change to all controllers and computes the path using parallel computing. After the computing is finished, we get the route and modify the flow table in OpenFlow switches using module `Forwarding.l2_multi`.

B. Processing path computation after link change

To evaluate ParaCon's performance in processing link change, we generate several full mesh topologies ranging the number of switches from 50 to 300. With the dense topologies, we can balance the heavy path computation load to controllers using even switch assignment. Meanwhile, we also change the number of controllers from 1 to 40 and assign switches to controllers. Even if the common number of controllers is generally less than 10 [11], we still set up to 40 controllers in order to test its scalability. We assign each controller 1 CPU core. We randomly shut down links and measure the time from when the link fails to when the new all-pairs path computation is over. In Fig. 7 we can see that by increasing the number of controllers, the computing time is reduced significantly. However, with 40 controllers, each (PowerEdge 2950) server has to run 4 controllers, each occupying 1 CPU cores; since each server only has 4 CPU cores, and the hypervisor uses at least 1 core for scheduling, the performance of each controller gets decreased. This explains the slight

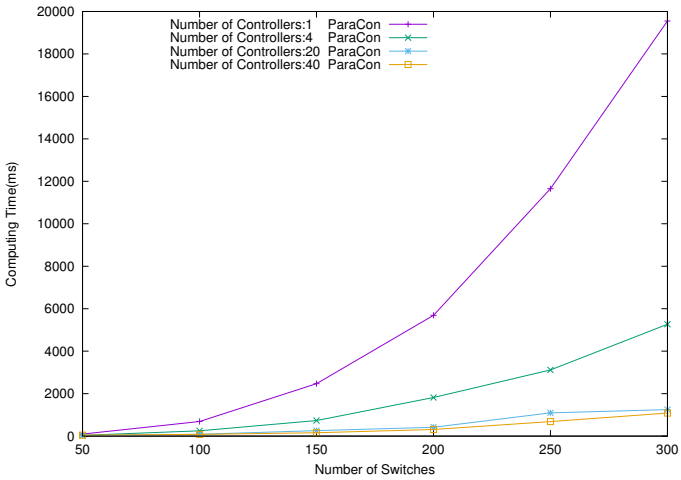


Fig. 7. Average computing time from 1 controller to 40 controllers. With the increasing of the number of switches, more controllers can get more benefit.

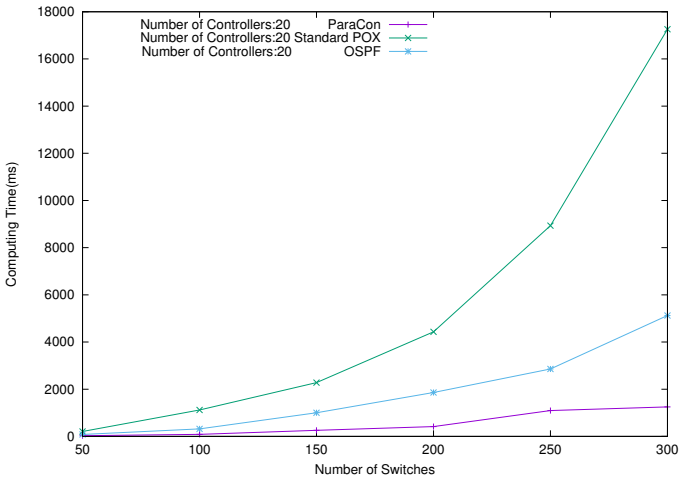


Fig. 8. Average computing time between 3 different modes in 20 controllers. The standard POX uses much more time in path computation by a single controller, and the OSPF uses much more time in waiting for the finish of last computation task.

improvement in computing time between 20 controllers and 40 controllers.

We also compare our module to other existing modules: a standard POX module (implemented by Floyd-Warshall algorithm), and an OSPF module (implemented by a distributed Dijkstra algorithm). From Fig. 8, we can see that with an increasing number of switches, the computing time used by ParaCon is far less than other modules.

We count the average frequency of communication between controllers in one controller. From TABLE I, we can see that with an increasing number of controllers, the communication overhead is well balanced into all controllers.

C. Comparing with ONOS

1) Setup: To study the performance gain of ParaCon over the state-of-the-art solution, we compare it with ONOS v1.3, a popular and stable distributed OpenFlow controller [32].

TABLE I
COMMUNICATION OVERHEAD

Number of controllers	4	20	40
Average communication times (per link change)	19.6917	2.8478	1.4506

Note that the path computation in ParaCon is in a different way than is done in ONOS. ONOS uses an on-demand mechanism of path computation, but ParaCon uses a pre-computed one. In ONOS, the forwarding module (`fwd.ReactiveForwarding`) receives the PacketIn and fetches the source and destination address. Then it calls the path computation modules to get a path from the source to the destination. ONOS provides a variety of algorithms for path computation. The default algorithm `getPaths()` uses a `DijkstraGraphSearch()` method. Finally, the forwarding module converts the path to flows and inserts them into switches.

Therefore, we compare the path computation module of ParaCon with the `getPaths()` (using Dijkstra by default) method in ONOS.

In this evaluation, we do not use full mesh topology because a Dijkstra-based module (e.g., OSPF, ONOS) is obviously slower than ParaCon with an increasing number of nodes in a full mesh topology. Therefore, we compare path computation module between ParaCon and ONOS using real topologies. We set up 4 ParaCon controllers, and we use three typical topologies as shown in TABLE II. 'CERNET' and 'USCarrier' are provided by Topology Zoo [33]. In order to use these topologies, we first get topology files from the Topology Zoo. These files are on a 'GML'/'GraphML' format, and we use Python and its library *iGraph* to convert these files into an adjacency matrix and edge list. For the topologies we made by ourselves ('Leaf-Spine'), we directly generate the adjacency matrix and edge list. 'CERNET' is a small topology with a small diameter; while 'USCarrier' is a large topology with a larger diameter. The third one 'Leaf-Spine' is a generated topology, which is usually used in a data center. Although it has a large number of nodes and links, the diameter is the smallest (only two hops from any hosts to any hosts in this topology).

TABLE II
SIMULATED TOPOLOGIES

Topology	Node	Link	Diameter
CERNET (2006)	41	57	6
USCarrier (Region, 2008)	158	189	35
Leaf-Spine (Core:1, Spine:50, Leaf:500)	551	25050	2

2) Path Computation Latency: It is not straightforward to evaluate the performance between ParaCon and ONOS due to the different mechanisms for path computation (pre-compute v.s. on-demand). If there's no topology change, ParaCon can reply to the path request immediately with no extra latency

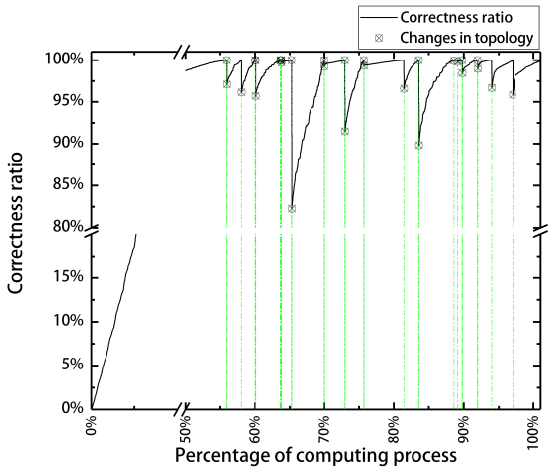
because the results have been pre-computed. Consequently, ParaCon will outperform ONOS in scenarios with infrequent topology change.

For a fair comparison, we consider the case with link failures. Suppose the network topology has changed as a result of a link failure, the controller must re-compute the path. We compare the average latency for answering a batch of path requests. For ParaCon, the latency mainly relies on APSP computation despite the batch size, because ParaCon only computes APSP once, and it does not consume any time in querying a path. Therefore, to evaluate the computing time in ParaCon, we set up the topology and make a link fail, then we measure the running time from when the link fails to when new APSP are computed. Finally, we divide the running time with the number of requests to get the average computing time of one request. For ONOS, we set up the same topology and make a link fail; then we also measure the running time of `getPaths()` method for getting a path in the new topology. The running time for one request is stable, no matter how many requests it received. It is because the running time of `getPaths()` method only depends on the number of nodes (switches). Note that Dijkstra algorithm can have a time complexity of $O(e+n\log n)$ if implemented with a Fibonacci heap. However, since the original ONOS only utilizes the Dijkstra algorithm without Fibonacci heap, and the Fibonacci heap can take a very long time to complete some operations in the worst case [34], we consider the general case $O(n^2)$ [17].

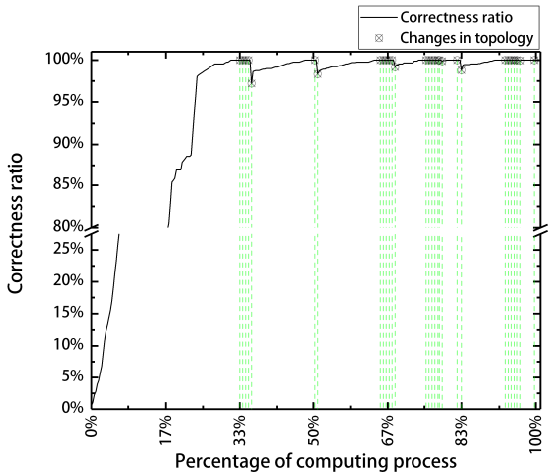
From Fig. 10, we can see that ParaCon performs better than ONOS in 'CERNET' and 'Leaf-Spine' because these topologies have a small diameter. Although the iterations of our algorithm are closely related to the diameter of the graph, the increasing number of nodes and links does not reduce the performance of the algorithm significantly. The 'USCarrier' is the worst case for ParaCon: if the number of queries is smaller than 60, the efficiency of ParaCon is less than ONOS. However, with the increase in path requests, the computing time decreases substantially. In 'CERNET', the efficiency of ParaCon exceeds that of ONOS in the early stage. In 'Leaf-Spine', the ParaCon uses less time to finish the computation of APSP than ONOS, while ONOS uses more time to compute only a single-pair path. We believe ParaCon is more suitable for a data center environment than ONOS, given that data center topologies usually have a small diameter.

3) Recovery Time Upon Failure: We define recovery time as the time between when a link fails and when a new path is obtained. From TABLE IV, we can see that in 'CERNET' and 'USCarrier', ParaCon only increases the overhead slightly, but in 'Leaf-Spine', ParaCon is much faster than ONOS. This is because a larger number of iterations will make ParaCon perform worse. A topology with a large diameter will make the number of iterations larger. In the linear part of the topology, where nodes have fewer degrees than in the non-linear part of the topology, path change information propagates slowly. It will increase a larger number of iterations in ParaCon. The observation provides evidence for the result that there are many linear topology subgraphs in 'USCarrier'.

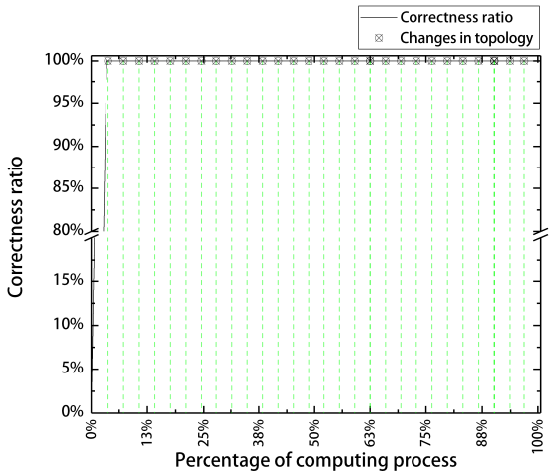
Although the new path upon link failure is computed,



(a) USCarrier

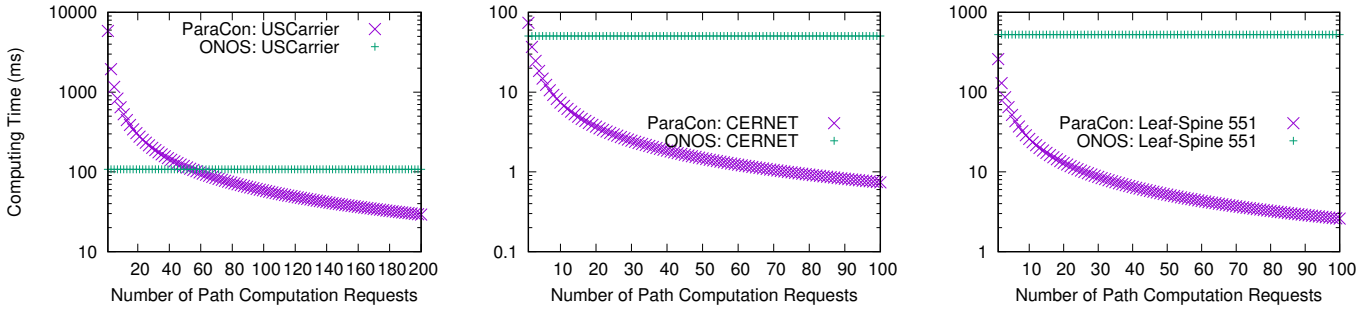


(b) CERNET



(c) Leaf-Spine

Fig. 9. The correctness ratio in different topologies with 30 changes in the topology. The correctness ratio, which is defined as a percentage of consistent results, show the consistency influence on the result. The dot *Changes in topology* with green line indicate the time when our algorithm start processing the topology changes request from the queues in controllers, while controllers receive topology changes from switches and put them into queues in $1s$ interval. The initial process, before the first topology change, makes the correctness ratio start from 0% to 100%.



(a) USCarrier (b) CERNET (c) Leaf-Spine

Fig. 10. The computing time per request with ONOS and ParaCon in different topologies and with 4 controllers

queues in controllers may not be empty. The controllers are still computing until queues are empty. In other words, at the time "Recovery Time Upon Failure", the new path upon link failure is computed, but not all the all-pairs shortest paths are fully computed. Thus, the recovery time is smaller than the computing time for the all-pairs shortest paths, as mentioned in the previous section.

4) Convergence Upon Incoming Requests from Switches: As the eventual consistency application is one of our contributions, whose impact we want to assess precisely, we export the intermediate results during path computation to measure the 'dynamical correctness ratio'. The dynamical correctness ratio is defined as a percentage of consistent results, to show the consistency influence on the results during the computation. A lower correctness ratio may increase the possibility of getting a stale path. We used the same topologies ('USCarrier', 'CERNET' and 'Leaf-Spine') to perform this evaluation. For each topology, we randomly change the topology 30 times by selecting two switches, and randomly adding/removing the link between them with a $1s$ time interval. We measure the correctness ratio during the process. From Fig. 10, we can see that the results are confirming our intuition that a graph with small diameter ('Leaf-Spine') gets a smaller convergence time, and a graph with a large diameter (e.g., 'USCarrier' has a linear part) makes the convergence time larger. Comparing Fig. 9 (a) with Fig. 9 (b), we can see that most topology changes in 'USCarrier' make the correctness ratio go down to $80\% \sim 90\%$, and it uses more time to recover. But in 'CERNET', topology changes only make the correctness ratio going down to 95%, with a faster recovery than in 'USCarrier'. In 'Leaf-Spine', the correctness seldom changes, which is pretty stable around 100%, and the recovering time is the fastest. In summary, a graph with a large diameter (especially with some linear part) will lead to a large number of iterations, which increases the amount of data that needs to be synchronized, besides the convergence time.

5) Synchronization Data Overhead: One way to evaluate the impact of the communication overhead is to quantify the amount of data that needs to be synchronized. As defined in Section II, we consider the minimum number of messages transmitted (such as a link/switch status message) as 1 unit. As for the previous evaluation, we measure the amount of data synchronized in controllers with 30 random topology changes in $1s$ interval, using the 'CERNET' topology. We use a number of controllers (2, 4 and 6) to perform our evaluation. Note that we balance switches (41 switches in 'CERNET') among all controllers. We export the number of messages that need to be synchronized in all controllers during path computation from our program to measure the amount of synchronized data. TABLE III reports the results, showing that the communication overhead due to synchronization marginally increases with an increasing number of controllers, knowing that one would not expect having a very high number of controllers in a common SDN scenario. Moreover, the amount of synchronized data in one controller is decreased with respect to the computation load, which is well balanced among all controllers. Furthermore, looking at the overhead due to one single change (the last row in the table), it is smaller than $1,681$ units (equal to n^2 with $n = 41$, which composes the whole path view) when the number of controllers equals to 4 and 6. The overhead is a bit larger than $1,681$ when the number of controllers equals to 2 (this overhead is introduced by Vector Clock and overwritten, as discussed in Section IV.C)

TABLE III
THE AMOUNT OF SYNCHRONIZED DATA IN 'CERNET' (WITH 30 TOPOLOGY CHANGES IN $1s$ INTERVAL, THE UNIT IS 1 MESSAGE DEFINED IN SECTION II)

Number of controllers	2	4	6
Total data synchronized with 30 changes (global, all controllers)	129,601	149,117	193,069
Average data synchronized with 30 changes (1 controller)	64,780	37,269	32,144
Average data synchronized with 1 change (1 controller)	2,173	1,230	1,066

TABLE IV
RECOVERY TIME UPON FAILURE

Topology	CERNET	USCarrier	Leaf-Spine
ParaCon Recovery Time (ms)	110.7091	175.2438	115.9289
ONOS Recovery Time (ms)	50.6363	107.5338	526.5933

VI. Related Work

*1) **Large-scale graph computation**:* As the scale of many graphs, e.g. web graphs or social network graphs, can reach billions of nodes and trillions of edges, data processing over such graphs becomes a concern. Some research such as GraphChi [35] and BBQ [36] enables large-scale graph computation using just one PC. Obviously one single PC has only limited scalability in large-scale graph computation. Thus, Pregel [37] presents a parallel computation model for solving this task. Other solutions such as GraphX [38] and PowerGraph [39] also focus on increasing the performance of large-scale graph computation. However, most of them are BSP-based algorithms, which are inflexible in networking path computation.

*2) **Distributed shortest path algorithm**:* The shortest path block is the key one of many network applications, such as related to routing or failure recovery. Most of distributed shortest path algorithms are variations of either the Dijkstra or Bellman-Ford algorithm. J. K. Antonio [40] proposed an $O((n/p)\log n)$ (n is the number of nodes, p is the number of processors for each node) algorithm for solving all-pairs shortest path problem on *balanced hierarchically clustered (BHC)* topology. Hence, S. Zhu and G. M. Huang [41] proposed to adopt a newer algorithm for solving the multiple origins shortest path problems (MOSP) on hierarchically clustered topology, where m is an integer related to the topology. It is worth mentioning that there is also a work [42] designed for computing Single Pair Shortest Path (SPSP) in a High Performance Computer (HPC) with $500,000+$ CPU cores system. Moreover, some approximation algorithms can speed up the shortest path computation. Suppose D is the diameter of the graph and n is the number of nodes in the graph, B. Patt-Shamir and C. Lenzen [43] proposed a distributed approximation algorithm with $\tilde{O}(n^{(1/2)+(1/2k)}+D)$-time, $(8k\lceil\log(k+1)\rceil-1)$-approximation ($k\geq 1$) for single-pair shortest path and Danupon [44] give an algorithm with a smaller time complexity $\tilde{O}(n^{(1/2)}D^{(1/4)}+D)$-time, $(1+o(1))$-approximation to solve the weighted single-pair shortest path computation. However, these algorithms are designed for a special topology or getting approximation results, which are less useful for logically centralized SDN routing environment.

*3) **SDN graph modeling**:* In SDN environments, a network can be represented as a graph. SDN controllers thus can compute the network information by utilizing the graph algorithms [24]. Most of the current SDN controllers such as POX, OpenDaylight and ONOS directly use Dijkstra or Floyd algorithm for path computation. Apparently, such designs lack scalability when processing all-pairs shortest path due to the $O(n^3)$ time complexity.

*4) **Consistency model**:* The consistency problem is a notable trade-off in the distributed system. As for the distributed SDN controller, the consistency model can either be coarse-grained or fine-grained. Under coarse-grained design, consistency is guaranteed by controllers and is transparent to applications. For example, HyperFlow [7] manipulates a publish/subscribe system and Kandoo [8] utilizes a hierarchical design (root controller and local controller) for maintaining the consistency of network information. Although ONOS [11] employs a Gossip protocol, which relaxes the constraints in data consistency of topology-related information, it cannot provide effective optimization for some particular operations like path computation. Moreover, ONOS delegates the complexity of maintaining the consistency to an external system while using an external system also increases the management difficulty. A fine-grained consistency policy is firmly correlated to a particular algorithm or application. A proper fine-grained consistency model can increase the synchronization efficiency significantly. That is the reason why Onix [10] does not synchronize network information in a transparent way for the SDN programmers, who have to create and maintain the network information by themselves explicitly.

VII. Conclusion

In this paper, we proposed ParaCon as a solution to address the performance bottleneck in the SDN control plane. To accelerate the path computation, we designed a set of mechanisms that utilize the untapped parallel computing potential of a multi-controller control-plane system. More specifically, ParaCon integrates the following features into an overall design: first, it exploits an asynchronous distributed algorithm for path computation in SDN; second, it reduces the transmission overhead by utilizing a hybrid consistent model. The evaluation results show that our design can achieve higher performance than legacy systems. Furthermore, we analyzed the convergence time, both theoretically and experimentally, comparing ParaCon with the state-of-the-art solutions. The results demonstrated the effectiveness of our architecture.

For future work, we are interested in integrating the design into commodity Open-Source SDN controllers, such as OpenDaylight or ONOS. Also, we plan to leverage the power of heterogeneous computing to further improve the performance.

Acknowledgements

This work was partially funded by the 863 program (Grant no. 2015AA016106) and the EU FP7 IRSES MobileCloud Project (Grant no. 612212)

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[2] Open network foundation homepage. [Online]. Available: http://www.opennetworking.org

[3] J. Hu, C. Lin, X. Li, and J. Huang, "Scalability of control planes for software defined networks: Modeling and evaluation," in *Proc. IEEE IWQoS*, 2014, pp. 147–152.

[4] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.

[5] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable Flow-based Networking with DIFANE," in *Proc. ACM SIGCOMM*, 2010, pp. 351–362.

[6] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," in *Proc. ACM SIGCOMM*, 2011, pp. 254–265.

[7] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," in *Proc. USENIX INM/WREN*, 2010, pp. 3–3.

[8] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proc. ACM SIGCOMM HotSDN*, 2012, pp. 19–24.

[9] S. H. Yeganeh and Y. Ganjali, "Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking," in *Proc. ACM SIGCOMM HotNets*. ACM, 2014, p. 13.

[10] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A Distributed Control Platform for Large-scale Production Networks." in *Proc. USENIX OSDI*, 2010, pp. 1–6.

[11] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proc. ACM SIGCOMM HotSDN*, 2014, pp. 1–6.

[12] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "Scl: Simplifying distributed sdn control planes," in *Proc. USENIX NSDI*, 2017.

[13] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *Proc. ACM SIGCOMM*, 2013, pp. 3–14.

[14] B. Davie, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Gude, A. Padmanabhan, T. Petty, K. Duda, and A. Chanda, "A database approach to sdn control plane design," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 1, pp. 15–26, 2017.

[15] M. Wichtlhuber, R. Reinecke, and D. Hausheer, "An sdn-based cdn/isp collaboration architecture for managing high-volume flows," *IEEE Trans. Networking and Service Management*, vol. 12, no. 1, pp. 48–60, 2015.

[16] H. Yamanaka, E. Kawai, and S. Shimojo, "[demo]scaling-up flow path computation for a large number of virtual sdn/nfv infrastructures," in *Proc. ACM SIGCOMM SOSR*, 2015.

[17] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[18] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas, "Towards efficiency and portability: Programming with the bsp model," in *Proc. ACM SPAA*, 1996, pp. 1–12.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," vol. 41, no. 6, pp. 205–220, 2007.

[20] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the r* distributed database management system," *ACM TODS*, vol. 11, no. 4, pp. 378–396, 1986.

[21] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[22] M. Canini, P. Kuznetsov, D. Levin, S. Schmid *et al.*, "A distributed and robust sdn control plane for transactional network updates," in *Proc. IEEE INFOCOM*, 2015, pp. 190–198.

[23] K. Dashdavaa, S. Date, H. Yamanaka, E. Kawai, Y. Watashiba, K. Ichikawa, H. Abe, and S. Shimojo, "Architecture of a high-speed mpi_bcast leveraging software-defined network," in *Proc. Springer Euro-Par*, 2014, pp. 885–894.

[24] R. Raghavendra, J. Lobo, and K.-W. Lee, "Dynamic graph query primitives for sdn-based cloud network management," in *Proc. ACM SIGCOMM HotSDN*, 2012, pp. 97–102.

[25] M. Jelasity and A. Montresor, "Epidemic-style proactive aggregation in large overlay networks," in *Proc. IEEE ICDCS*, 2004, pp. 102–109.

[26] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh, "Efficient and adaptive epidemic-style protocols for reliable and scalable multicast," *IEEE Trans. Parallel and Distributed Systems*, vol. 17, no. 7, pp. 593–605, 2006.

[27] J. Holliday, R. Steinke, D. Agrawal, and A. El Abbadi, "Epidemic algorithms for replicated databases," *IEEE Trans. Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1218–1238, 2003.

[28] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *IEEE Trans. Networking*, vol. 14, no. SI, pp. 2508–2530, 2006.

[29] M. Haridasan and R. Van Renesse, "Gossip-based distribution estimation in peer-to-peer networks." in *Proc. USENIX IPTPS*, 2008, p. 13.

[30] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proc. ACM SIGCOMM HotNets*, 2010, p. 19.

[31] Eventlet. [Online]. Available: http://eventlet.net/

[32] Onos version 1.3. [Online]. Available: http://onosproject.org/

[33] Topology zoo. [Online]. Available: http://www.topology-zoo.org

[34] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, "The pairing heap: A new form of self-adjusting heap," *Algorithmica*, vol. 1, no. 1, pp. 111–129, 1986.

[35] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 31–46.

[36] Q. Xu, X. Zhang, J. Zhao, X. Wang, and T. Wolf, "Fast shortest-path queries on large-scale graphs," in *Proc. IEEE ICNP*, 2016, pp. 1–10.

[37] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD*, 2010, pp. 135–146.

[38] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *Proc. ACM GRADES*, 2013, p. 2.

[39] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. USENIX OSDI*, 2012, pp. 17–30.

[40] J. K. Antonio, G. M. Huang, and W. K. Tsai, "A fast distributed shortest path algorithm for a class of hierarchically clustered data networks," *IEEE Trans. Computers*, vol. 41, no. 6, pp. 710–724, 1992.

[41] S. Zhu and G. M. Huang, "A new parallel and distributed shortest path algorithm for hierarchically clustered data networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 9, pp. 841–855, 1998.

[42] V. T. Chakaravarthy, F. Checconi, P. Murali, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," *IEEE Trans. PDS*, vol. 28, no. 7, pp. 2031–2045, 2017.

[43] D. Nanongkai, "Distributed approximation algorithms for weighted shortest paths," in *Proc. ACM STOC*, 2014, pp. 565–573.

[44] B. Patt-Shamir and C. Lenzen, "Fast Routing Table Construction Using Small Messages," in *Proc. ACM STOC*, 2013, pp. 381–390.

Kun Qiu received his Bachelor Degree from Fudan University. He is now a Ph.D. student at Fudan University, His research interests include computer network and computer architecture.

Siyuan Huang received his Bachelor Degree from Zhejiang University. He is now a graduate student at Fudan University. His research interest is software-defined networking (SDN), especially hybrid SDN and network convergence.

Qiongwen Xu received her Bachelor Degree from Central China Normal University. She is now a graduate student at Fudan University. Her research interest is software-defined networking (SDN), especially in routing optimization.

Jin Zhao received his B. Eng. Degree in computer communications from Nanjing University of Posts and Telecommunications, China, in 2001, and the Ph.D. Degree in computer science from Nanjing University, China, in 2006. He joined Fudan University in 2006. He stayed at University of Massachusetts Amherst for 1 year as a visiting scholar in 2014. His research interests include software defined networking, media streaming and network coding theory. He is a member of IEEE and ACM.

Xin Wang received his Bachelor Degree and the Master Degree from Xidian University, Xian, China, in 1994 and 1997 respectively, in Information Theory and Communications. He received the Ph.D. Degree from Shizuoka University, Japan in 2002, in Computer Science. Since 2002, he has been with the School of Computer Science at Fudan University, where he is currently a full professor.

Stefano Secci received the M.Sc. Degree in communications engineering from Politecnico di Milano, Milan, Italy, in 2005, and a dual Ph.D. Degree in computer science and networks from Politecnico di Milano and Telecom ParisTech, France. Now he is an Associate Professor at the Universite Pierre et Marie Curie (UPMC - Paris VI - Sorbonne Universites), Paris, France. Webpage: https://lip6.fr/Stefano.Secci.