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*, Xin Wang*, Stefano Secci†
*School of Computer Science, Fudan University, Shanghai, China
*Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry
{qkun,syhuang14,qyxu15,jzhao,xinw}@fudan.edu.cn
†Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
{stefano.secci@upmc.fr}

***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 design of control plane brings the benefit of network programmability and promises to ease network management. However, it also increases efficiency concerns for large-scale networks. In this paper, our goal is to build a high-performance SDN control plane using multiple controllers. Previous work seeks to improve control plane efficiency by balancing only the load for data plane behaviors among multiple controllers. Deviating from conventional wisdom, we propose the design and implementation of ParaCon, which resorts to parallel computing to speed up the path computation in SDN control plane. We also address the consistency problem and synchronization overhead under the design. To the best of our knowledge, ParaCon is the first attempt that utilizes node parallelism in path computation for SDN control plane. We experimented ParaCon using both Mininet and real-world clusters. Our results show that the path computing time of ParaCon is able to achieve speedup of 10x over POX baseline implementation in a 300-node network with 20 controllers.**

I. Introduction

Recently, *Software-Defined Networking* (SDN), has become 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 behaviors of all the switches in a network. Under the abstraction of OpenFlow, the SDN controller offers a fully functional control plane, including network state monitoring, QoS support, network function virtualization, traffic engineering, etc. Despite its advantages, the single centralized structure may also lead to scalability and performance challenges. 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 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, the 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. In order to achieve better execution efficiency, some high-level languages based controllers were redesigned as low-level languages based controllers. NOX [3], a controller is written in Python has been re-written in C++. Also, to exert the performance of multicore processors, NOX-ML, a multi-threading version of NOX is re-implemented. Another approach is offloading (partial or full) load from the controller to switches, e.g. DIFANE [4] and DevoFlow [5]. Although these techniques can improve the performance of a single-server controller significantly, a theoretical study shows that the solution of the single-server design still suffers from a limited scalability [6].

Alternatively, utilizing multiple controllers in managing the SDN provides yet another choice. HyperFlow [7], Kandoo [8], Beehive [9], Onix [10] and ONOS [11] deploy multiple controllers in one SDN. The distributed control plane designs are becoming increasingly popular. For example, ONOS claims to be the first carrier-grade distributed SDN controller, and Onix is used in Google's B4 [12] as their distributed control plane.

Existing multiple-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 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. Mainstream SDN applications such as traffic engineering and content delivery networking [13] are tightly coupled with the result 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 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.

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 [14] 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 on the updated topology from scratch. More specifically, all-pairs path computation on the updated graph involves roughly $O(n^3)$ time complexity [14]. 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 among involved controllers. Hence, the processes at different controllers will be frequently blocked by each other. With the increase in the number of involved SDN controllers, the synchronization overhead will result in significant performance degradation.

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 novel features to make path computation in multiple controllers efficient. It introduces a new distributed model in SDN that is capable of asynchronous path computation among multiple controllers.

First, it includes an online algorithm that enables all SDN controllers make incremental changes to the computed path results upon receiving a particular topology change, while existing approaches need to restart the computation from scratch. This makes ParaCon more efficient in scenarios with frequent topology updates.

Second, it seeks to minimize the overhead among 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 among controllers during the asynchronous path computation. It utilizes the both *strong consistency* and *eventual consistency* in the maintenance of these information for further improving the performance.

The 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 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

Let us consider a parallel all-pairs path computation model for an SDN control plane with several switches and controllers. Each switch belongs to at most one controller at a given time, and all controllers constitute a logically centralized controller. In the control plane, each controller does single-pair path computation for its switches. Thus, the all-pairs path computation can be calculated by all controllers.

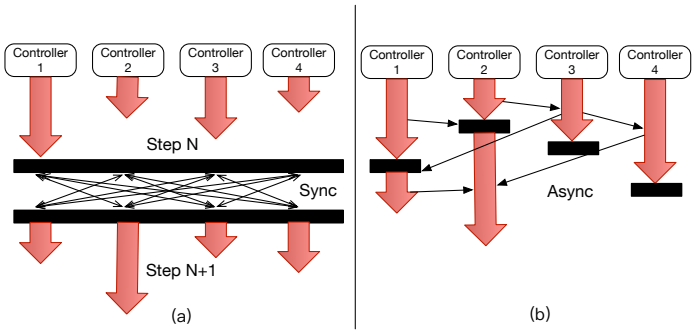


Fig. 1. A synchronous and an asynchronous path computation. In synchronous computation, the synchronization blocks the path computation. In asynchronous computation, the synchronization happened during the computation.

Existing algorithms for parallel path computation are usually based on the Bulk Synchronous Parallel Computing (BSP) model [15]. From Fig. 1(a) we can see the model separates the computation into several steps. In each step, the controller first individually computes the path of the switches that are controlled by itself; then it waits for other controllers to finish their computation; finally, after all controllers finish their computation, they synchronize the intermediate results and prepare the computation in next step.

Clearly, the blocking synchronization in the path computation reduces the efficiency significantly. Moreover, synchronous path computation does not adapt 'online' property in the network either, i.e., if the topology is changed during the computation, controllers cannot compute path based on the new topology.

Hence, we consider an asynchronous model in Fig. 1(b). Similar to the synchronous model, each controller in the control plane does the path computation separately, but controllers can synchronize their intermediate path information during the computation without blocking.

To motivate the asynchronous computing approach, we need to consider several aspects, among which 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) The topology-related information, i.e., the link/switch status.

2) The association between switches and controllers.
3) The path information in each controller during the path computation.

Suppose there are four switches $S1$, $S2$, $S3$ and $S4$ in a network. If controllers $C1$, $C2$ are added to the network with the association as described in Fig. 2. Clearly, 1) topology-related information needs to be synchronized between $C1$ and $C2$ continuously 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 consistency policy formally, we introduce some notations. Let H be a 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 that $p_c \in \{NO, YES\}$: 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 multiple-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 are 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 $\forall c \in C, p_c = YES$.

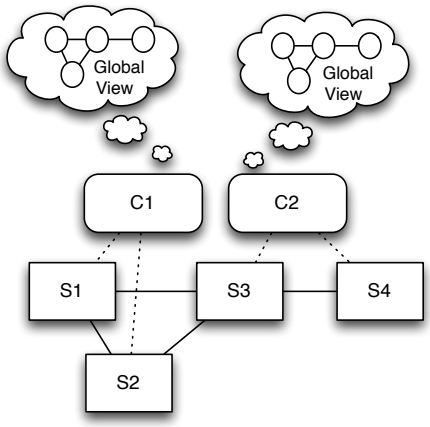


Fig. 2. The topology-related information needs to be synchronized to controller C1 and controller C2 to make sure they have the same global view on switches S1, S2, S3 and S4.

Meanwhile, 2) 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 3) 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 [16]. For example, *Two-Phase Commit* [17] or *Paxos* [18] 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 [19] [20], and we consider the minimum number of messages transmitted for reflecting the change in 1) a link/switch status and 2) a switch-controller assignment is 1 unit. In Fig. 3, these two types of operations only involve 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.

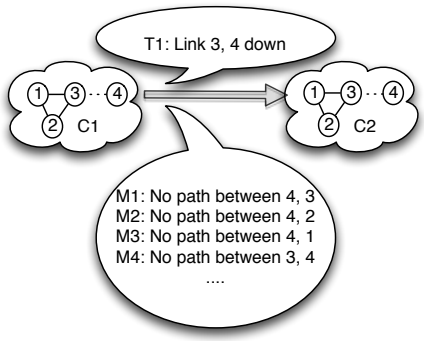


Fig. 3. The different messages that are transmitted between two controllers. C1 only needs to transmit T1 to C2 to modify the link status but has to transmit at least 4 additional messages like M1, M2, M3 and M4 to C2 to change the path information in C2.

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. Differently than *strong consistency* policy, an *eventual consistency* policy is a specific form of weak consistency. The SDN control plane guarantees that if no new updates like 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 $\forall c \in C,\ p_c = YES$.

Naturally, *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 [14].

In summary, finding an equilibrium between synchronization overhead and computation time is difficult. This 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 among multiple controllers. Fig. 4 shows the architecture of ParaCon. Our architecture is based on a distributed structure with an asynchronous parallel algorithm. We will give a brief introduction on 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 [21]. The global topology view includes both *link view* and *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, 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.

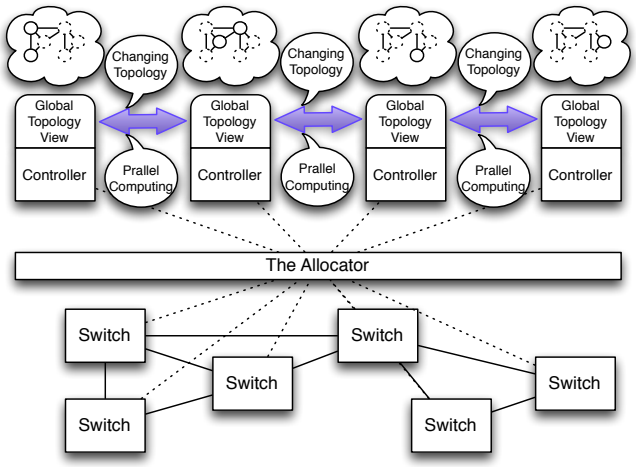


Fig. 4. ParaCon structure. All controllers are distributed and sharing the global topology view. Switches are allocated to different controllers by an allocator, and each controller only computes the path information of its own switches. All the controllers can compute the path together by parallel computing.

B. The asynchronous parallel algorithm

An asynchronous parallel algorithm is introduced, which uses *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 a goal of making a right balance between transmitting overhead and computing overhead.

Furthermore, different from existing algorithms such as Floyd-Warshall [14] or Dijkstra [14], 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. Concept

Switch and controller sets: There are a number of switches and controllers in an SDN. 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 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.

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. 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 [14] and Dijkstra [14] algorithms have a $O(n^3)$ time complexity for 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 [14] 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 centralizd 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, fetching the head of queue and relaxing 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 becomes shorter by passing through this node, its neighbors will be inserted into the queue.

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 $O(K*e*n)$, where e is the number of links and K depends on how many times switches are put into a queue. Intuitively, K 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 large diameter can cause the K 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-parallism advantages compared to the other existing algorithms.

C. Asynchronous Parallel Path Computation

Adopting a parallel path computation approach makes all controllers to compute the path collaboratively. 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 is done. The barrier is used for synchronizing the path computation process at 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 introduce the synchronization delay substantially.

In order to turn our algorithm into an asynchronous one, we redesign the barrier to trade consistency for synchronization overhead. Insteed, we employ eventual consistency. A weak synchronization signal, or SYNC, is introduced which allows a controller to send it *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 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. 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 to 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

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
SYNC is a synchronization signal that makes the current controller synchronize the path view to other controllers.

```
while True do
    while Q_id.empty() is not true do
        now ← Q.get()
        if now is a SYNC then
            for i = 1 → m except the c_id do
                Send Q_i to c_i
                Q_i.clear()
            for i which Con(i) = c_id do
                Send T_i to other controllers.
            continue
        for k which has L_{now,k} ≠ −1 do
            for i = 1 → n do
                temp_topo_{k,i} ← T_{k,i} + L_{now,k}
        for i = 1 → n do
            T_{now,i} ← min(temp_topo_{k,i}) with
              L_{now,k} ≠ −1
            F_{now,i} ← k which minimized prior
              calculation T_{now,i}
        if T_now has changed then
            for k which has L_{now,k} ≠ −1 do
                Q_{Con(k)}.put(k)
            Q_id.put(SYNC)
```

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 check if the head is a synchronization signal. If yes, it sends its path view to other controllers and continue the loop. Otherwise, it will relax its path view of the head and put the adjacency nodes which are relaxed by the head to the queue. At last, it will push a synchronization signal into the queue. The algorithm can do path computation in multiple controllers efficiently.

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_1 to 30, the changes of *path view* and the queue Q_0, Q_1 are shown in Fig. 5.

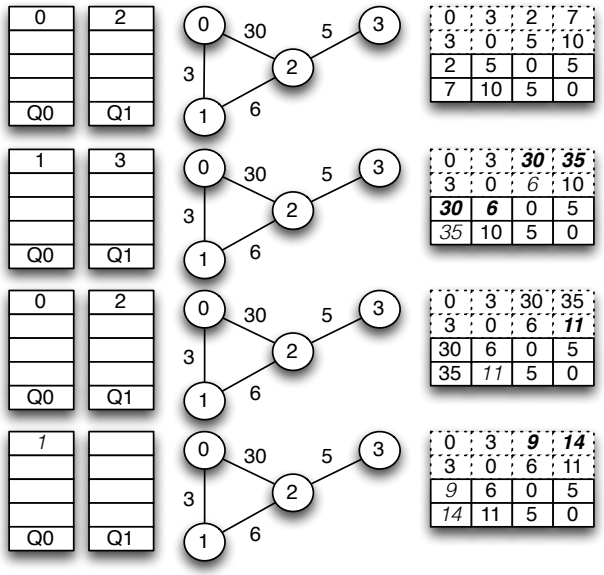


Fig. 5. The distributed version. 4 switches are distributed into 2 controllers. The matrix on the right is separated into two parts. The top two columns are maintained by controller $C1$ and the bottom two columns is maintained by controller $C2$. They only uses 3 steps to finish the algorithm. Each controller computes the path and synchronizes the path information to other controllers at the end of each step.

We discuss the evaluation results in section V.

V. The ParaCon Prototype

Our work is motivated by the intuition that the performance of multiple controllers 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 the real-world implementation.

A. Implementation

We implemented ParaCon structure using commodity hardware. The distributed control plane are based on 10 Dell PowerEdge 2950 (Xeon E5405 with 2G RAM) servers. We build four virtual machines in one server and each virtual machine handle 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 [22]. 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

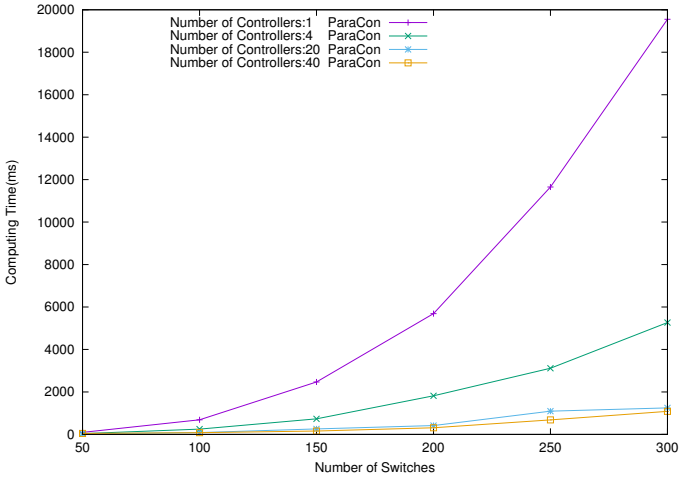


Fig. 6. Average computing time from 1 controller to 40 controllers. With the increasing of the number of switches, more controllers can get more benefit.

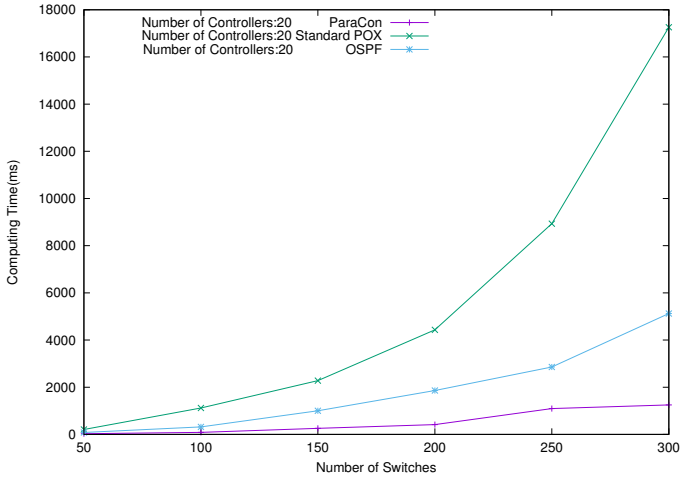


Fig. 7. Average computing time between 3 different modes in 20 controllers. The standard POX uses much more time in path computation by a single controller, and the OSPF uses much more time in waiting for the finish of last computation task.

of switches. 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 [23].

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 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 with a range 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. We randomly shutdown links and measure the time from when the link fails to when the new all-pairs path computation is over. From Fig. 6, we can see that with the increase in controllers, the computing time has been reduced significantly. However, due to the virtual machine overhead, the time shows only sightly improvement 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. 7, 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.

TABLE I
COMMUNICATION OVERHEAD

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

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 [24]. Note that the path computation in ParaCon is in a different way from that in ONOS. ONOS uses an on-demand mechanism of path computation, but ParaCon uses a pre-compute one. In ONOS, the forwarding module (`fwd.ReactiveForwarding`) receives the PacketIn and fetches the source and destination address. Then it calls path computation module to get a path from the source to the destination. ONOS provides a varity 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 a 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 [25]. 'CERNET' is a small topology with 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 definitely 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 resulted from 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 rely on APSP computation despite of 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 fail a link, then we measure the running time from when the link fail 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 fail a link; 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. This 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 $O(e + nlogn)$ if implemented with a Fibonacci heap. However, without loss of generality, we consider the general case $O(n^2)$ [14].

From Fig. 8, 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. Howver, 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 III, 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, path change information propagates slowly. This will increase a larger number of iterations in ParaCon. The result is evidenced by the observation that there are many linear topology subgraphs in USCarrier.

TABLE III
RECOVERY TIME UPON FAILURE

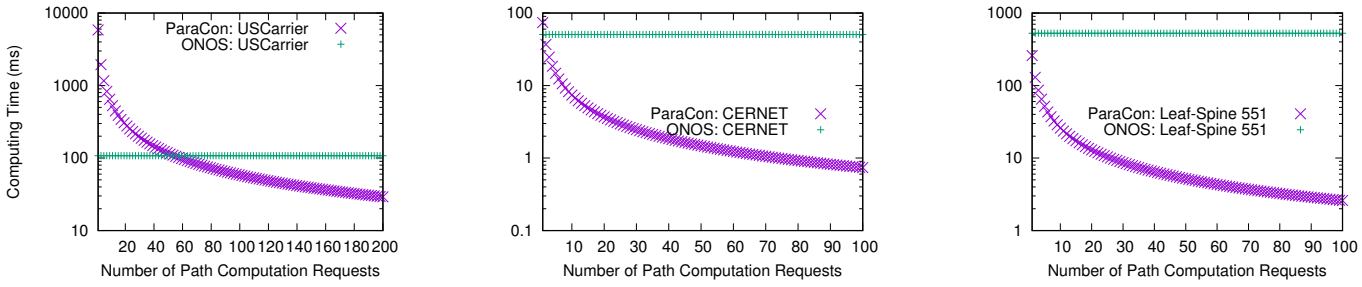
Topology	CERNET	USCarrier	Leaf-Spine
ParaCon Recovery Time (ms)	110.7091	175.2438	115.9289
ONOS Recovery Time (ms)	50.6363	107.5338	526.5933

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 become a concern. GraphChi [26] proposes a disk-based system with a parallel sliding windows method, which enables large-scale graph computation using just one PC. Obviously, single PC has only limited scalability in large-scale graph computation. Thus, Pregel [27] presents a parallel computation model for solving this task. Other solutions such as GraphX [28] and PowerGraph [29] 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) **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 [21]. 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.

*3) **Consistency model**:* The consistency problem is a notable trade-off in the distributed system. As for 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



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

Fig. 8. The computing time per request with ONOS and ParaCon in different topologies and with 4 controllers

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 multiple controllers. 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 achieved high performance over the single-controller design, and it also improved efficiency over the existing algorithms. Furthermore, we also evaluated our design using real-world network topologies and compared with the state-of-the-art solutions. The results demonstrated the effectiveness of our architecture.

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 Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

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

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

[4] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 351–362.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 254–265.

[6] J. Hu, C. Lin, X. Li, and J. Huang, "Scalability of control planes for software defined networks: Modeling and evaluation," in *Proceedings of the IEEE International Symposium on Quality and Service*. IEEE, 2014.

[7] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," in *Proceedings of the internet network management conference on Research on enterprise networking*. USENIX Association, 2010, pp. 3–3.

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

[9] S. H. Yeganeh and Y. Ganjali, "Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014.

[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 *OSDI*, vol. 10, 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 *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. ACM, 2014.

[12] 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 *Proceedings of the ACM SIGCOMM conference on SIGCOMM*. ACM, 2013, pp. 3–14.

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

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

[15] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas, "Towards efficiency and portability: Programming with the bsp model," in *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1996, pp. 1–12.

[16] 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," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

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

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

[19] M. Canini, P. Kuznetsov, D. Levin, S. Schmid *et al.*, "A distributed and

robust sdn control plane for transactional network updates," in *The 34th Annual IEEE International Conference on Computer Communications (INFOCOM 2015)*, 2015.

[20] 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 *Euro-Par 2013: Parallel Processing Workshops*. Springer, 2014, pp. 885–894.

[21] R. Raghavendra, J. Lobo, and K.-W. Lee, "Dynamic graph query primitives for sdn-based cloudnetwork management," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 97–102.

[22] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

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

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

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

[26] 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 12)*, 2012, pp. 31–46.

[27] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[28] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 2.

[29] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.