# Routing Keys

Asaf Samuel[1], Eitan Zahavi[2], Isaac Keslassy[1,3]
[1] *Technion* [2] *Mellanox* [3] *VMware*

***Abstract*—The network plays a key role in High-Performance Computing (HPC) system efficiency. Unfortunately, current HPC routing solutions are not application-aware, and therefore cannot deal with the sudden HPC traffic bursts and their resulting congestion peaks.**

**To address this problem, we introduce *Routing Keys*, a scalable routing paradigm for HPC networks that decouples intra- and inter-application flow contention. Our *Application Routing Key (ARK)* algorithm proactively allows each self-aware application to route its flows according to a predetermined routing key, *i.e.,* its own intra-application contention-free routing. In addition, in our *Network Routing Key (NRK)* algorithm, a centralized scheduler chooses between several routing keys for the communication phases of each application, and therefore reduces inter-application contention while maintaining intra-application contention-free routing and avoiding scalability issues. Using extensive evaluations, we show that both ARK and NRK significantly improve the communication runtime by up to 2.7x.**

***Keywords*-routing keys; HPC; contention-free routing**

## I. Introduction

**Background.** As Moore's law keeps slowing down, High-Performance Computing (HPC) applications naturally become increasingly parallel and involve an ever-larger number of hosts. As a result, they need to rely on an efficient and scalable routing algorithm.

Unfortunately, current routing solutions employed in HPC networks suffer from performance degradation and/or scalability issues. This is because they are either (a) oblivious to contention, *e.g.,* DmodK and ECMP [1], [2], often with poor performance; or (b) centralized [3]–[8], with scalability issues; or (c) reactive to congestion [9]–[13], [13], [14], which are often based on outdated information, especially when the traffic pattern keeps changing [9].

**Inter- versus intra-application contention.** The performance degradation is particularly acute for BSP (Bulk Synchronous Parallel)-based applications, where common barriers force the communication phases of different hosts of the same application to start and end synchronously. As illustrated by a toy example in Figure 1, this strong synchronization can result in significant intra-application contention (*e.g.,* between A1 and A2). As a result, current routing solutions have trouble handling such barrier-based applications.

Figure 2(a) illustrates the contention when using a traffic-pattern-agnostic routing algorithm. Application $A$ runs in hosts 0, 1, 4 and 8, referred to as A1, A2, A3 and A4. Application $B$ runs in hosts 2 and 12, referred to as B1 and B2. Three different flows contend for the same leftmost link: two of them (in continuous blue lines) belong to application $A$ and exhibit intra-application contention. The last one (in the dashed red line) belongs to application $B$, and exhibits inter-application contention with application $A$. While this figure exhibits both intra- and inter-application contention, the intra-application contention between the two flows of $A$ is typically stronger in practice because of the barrier synchronization effects.

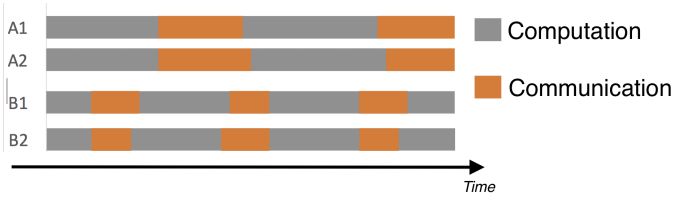


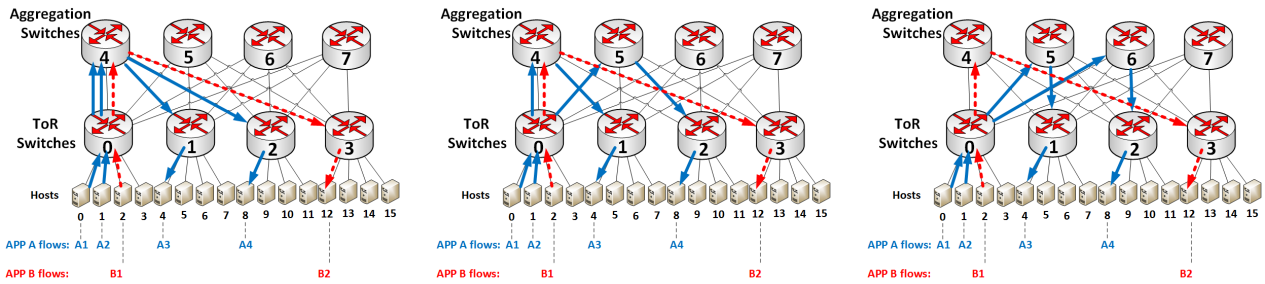

Figure 1. Inter- vs. intra-application contention in a simple example with two barrier-based applications A and B, each with two hosts. The computation and communication phases of hosts that belong to the same application (*e.g.,* A1 and A2) are synchronized, resulting in significant intra-application contention. On the other hand, communication phases of hosts that belong to different applications (*e.g.,* A1 and B1) are typically not synchronized, thus the inter-application contention is weaker.

**Contributions.** *Our key observation is that applications can proactively prevent intra-application contention by requesting contention-free routing paths for their flows.* In other words, instead of independently determining the routing of each flow of a given application, we can jointly determine the routing of all the application flows so that they do not conflict. Our paper targets the wide range of applications that know their main traffic patterns, *e.g.,* applications based on stencil or MapReduce.

We further introduce *Routing Keys*, a new scalable routing paradigm for HPC networks and software-defined networks (SDN) that decouples intra- and inter-application flow contention. Within this paradigm, we present two algorithms:

*(1) ARK.* Our first algorithm is *Application Routing Key*, which allows each self-aware application to route its flows according to a predetermined routing key, *i.e.,* its own intra-application contention-free routing (Section II-A).

In ARK, at placement time, when an application requests placement and informs the centralized scheduler of




(a) Routing with strong contention

(b) ARK intra-application contention-free routing

(c) NRK optimal routing

Figure 2. Illustration of different routing algorithms for flows of application $A$ (continuous blue lines) and $B$ (dashed red line). (a) Traffic-pattern-agnostic routing algorithm, resulting in strong contention between different flows. (b) ARK algorithm, yielding an intra-application contention-free routing. (c) NRK algorithm, resulting in this example in inter- and intra-application contention-free routing.

its computation needs (*e.g.,* $n$ hosts), it can also inform the scheduler of its communication needs (*e.g.,* stencil or permutation pattern). The scheduler then not only returns a list of hosts, but also generates a *routing key*, *i.e.,* a contention-free list of specific routes that these hosts will use to communicate. Thus, ARK proactively prevents any intra-application contention.

Figure 2(b) illustrates how ARK routing prevents intra-application contention between flows of application $A$. However, since ARK considers each application independently, it still allows for contention between two flows of applications $A$ and $B$ on the leftmost link, at the upstream of the first Top-of-Rack (ToR) switch.

*(2) NRK.* Our second algorithm is *Network Routing Keys* (Section II-B). At placement time of each application, NRK generates *a list of routing keys* instead of a single routing key. Each routing key guarantees an intra-application contention-free routing for the application flows. Then, at run-time, at the start of each new communication phase of a given application, the scheduler can pick a routing key in its corresponding list that would yield less contention with other applications. Thus, the centralized scheduler reduces inter-application contention while keeping intra-application contention-free routing.

Figure 2(c) illustrates a possible routing produced by NRK that is both intra- and inter-contention free, *i.e.,* an optimal routing solution for this placement.

We further provide a detailed description of an Infiniband-based implementation design of ARK and NRK using three main architectural components: the *scheduler*, the *subnet manager* and the *application-flow manager*. We also leverage the *Infiniband LMC mechanism* to create different routing paths between each pair of hosts (Section III).

Finally, we extensively evaluate both ARK and NRK using a flit-level simulator on a large InfiniBand network. We extend the simulator to provide MPI semantics for reproducing its distributed communication patterns. The evaluations show that both ARK and NRK significantly reduce communication runtime by up to 2.7× compared to DmodK [1], [2] and Conga [9] (Section IV).

## II. Routing Keys

In this section, we introduce Routing Keys. We first present our simpler ARK solution, then the more advanced NRK algorithm. For ease of exposition, we assume a two-level fat-tree network. We later generalize to a three-level extended generalized fat-tree (XGFT) network (Section III). We also discuss routing keys at the *application granularity*, but all of our techniques equally apply at the *application communication-phase granularity*[1], which we also use in our implementation design (Section III).

### A. ARK: Application Routing Key

The ARK algorithm relies on the centralized scheduler, *i.e.,* the architectural element that handles the placement of the application hosts. At placement time, each application informs the scheduler about its computing and networking requirements. For example, it may mention (a) the number of ordered hosts that the application requires, and (b) the communication pattern among these hosts (*e.g.,* stencil or permutation). Then, the ARK scheduler not only decides about the placement of the application hosts, but also exploits the communication pattern to produce a *routing key* for this application, *i.e.,* an intra-application contention-free routing for the application flows.

Consider again Figure 2(b). In that example, application $A$ informs the scheduler that it requires four hosts $A1, A2, A3, A4$ and that its traffic pattern is $(A1 \rightarrow A3, A2 \rightarrow A4)$. The smaller application $B$ requires only two hosts $B1, B2$ and its traffic pattern is $(B1 \rightarrow B2)$.

[1]In our evaluations, we create a routing key for each communication phase of each application. When such a phase begins, the scheduler chooses the key based on the (application, phase) pair, potentially providing better performance.

The scheduler decides on the host placement and returns a routing key.

**Compressed key.** In order to produce the desired routing key efficiently at placement time, we design an algorithm based on the Birkhoff-von Neumann decomposition. Specifically, we consider a bipartite graph in which the left set of nodes represents the flow source ToR switches, the right set represents the flow destination ToR switches, and the edges represent the requested source-destination application flows. For example, in application $A$ of Figure 2(b), ToR source 0 is connected to ToR destinations 1 and 2. Once the bipartite graph is formed, we *iteratively* decompose it by (1) finding a maximum-cardinality match using the computationally-efficient *hopcroft-karp algorithm* [15], (2) translating this match into a set of intra-application contention-free routes, (3) mapping each obtained match to a different middle (aggregation) switch, and (4) removing the corresponding edges from the graph. This procedure terminates once there are no edges left. For instance, in Figure 2(b), at the end of the procedure, the two flows of application $A$ are mapped to switches 4 and 5, respectively. We are guaranteed to terminate with intra-application contention-free routing if such a contention-free solution exists.

### B. NRK: Network Routing Keys

As can be seen in Figure 2(b), while ARK provides intra-application contention-free routing for each application independently, it does not consider inter-application contention. The goal of the Network Routing Keys (NRK) algorithm is to greedily minimize inter-application contention while preserving the intra-application contention-free routing.

Unlike ARK, NRK returns several routing keys, *i.e.,* several intra-application contention-free routes, for each application at placement time. Each application is then associated with this set of different routing keys. At run-time, at the start of the communication phase of each application, the scheduler can dynamically choose which of its routing keys to employ such that inter-application contention is minimized.

**Simplified key generation.** Generally, an application routing demand can have many potential intra-application contention-free routing keys. Considering all of them would be impractical. Instead, NRK returns only a fixed-size subset of keys for each application. In addition, in many symmetric topologies (*e.g.,* leaf-spine, folded-Clos, XGFT [16]–[18]), we can use a single key to generate many different intra-application contention-free routes. Specifically, we observe that in such symmetric topologies, the intra-application contention-free routing depends only on the relative position of the switches. To illustrate this consider again Figure 2(b). In that example we have an inter-application contention at the leftmost link. However, with NRK, we can leverage the same application routing key that is used by ARK and *shift* it by one step to the right. Namely, use aggregation switches 5 and 6 instead of 4 and 5. By doing so, NRK obtains an optimal solution (*i.e.,* inter- and -intra contention-free routes), as depicted in Figure 2(c).

We want the NRK scheduler to choose the best routing key shift to employ given the current network state. To do so, we consider two optimization criteria: (a) the maximal link load and (b) the load variance over the links. Next, since an exhaustive search for the best routing key shift according to (a) and (b) leads to exponential complexity, we define a *fine-grained* shift optimization. We first consider the most loaded middle switch in the routing key, *i.e.,* the middle switch with the most application flows. Since we want to balance the load across the network middle switches, we assign the flows going through this middle switch to the least-loaded middle switch in the current network state. Then we do the same for the second most-loaded switch, and so on. For instance, assume that a routing key routes all flows through middle switch 1. Assume also that currently middle switch 7 is unused. Then we can shift the routing key and route all these flows through middle switch 7 instead of 1. In this approach, the number of possible shifts for a key grows like the square of the number of switches.

## III. Implementation Design

In this section, we design the Routing Keys implementation for an InfiniBand cluster with an XGFT topology [16]–[18]. We rely on three main architectural components: (a) the *scheduler*, which is in charge of the placement and the routing-key generation, (b) the *subnet manager* (SM), which controls the switch routing tables, and (c) the *application-flow manager*, *i.e.,* a single application host that is responsible for the key management. We detail the implementation of NRK, as ARK only uses a subset of the NRK functions.

**At placement time.** Once an application requests placement, it also provides a list of its traffic patterns that correspond to its different communication phases. In the simplest case, this list may contain only a single pattern, but we allow applications to report several communication patterns if they change between phases. In our design, the SM computes a routing key for each traffic pattern provided by the application, then sends *key identifiers* to the application-flow manager.

**At run-time.** At run-time, the application-flow manager sends to the scheduler the required key identifier that corresponds to the application's next communication stage. The scheduler, in turn, takes the following actions: (a) it first optimizes the routing key using the *fine-grained* shift technique that enables it to generate several alternative routing keys and pick the best one for the current network state; then (b) it sends the required LMC (detailed in the next paragraph) to the application hosts; and finally (c) it requests the SM to configure the forwarding tables accordingly.

**LID Mask Control (LMC).** To enable the compatibility of Routing Keys with an InfiniBand cluster, we leveraged the LID (Local IDentifier) Mask Control (LMC) mechanism that is already supported by InfiniBand. LMC enables the aliasing of the same physical host with an array of LIDs instead of a single one. This, in turn, allows the creation of different routing paths between each pair of hosts. Specifically, we code each routing key for a specific application using a different LMC,[2] and notify the application which LMC to use during its communication stages. When NRK decides on a route, the SM configures the forwarding tables for this specific LMC.

**SM.** We adopted openSM [19] as our SM. We implemented NRK using the *Compressed key* technique and the *hopcroft-karp* subroutine.

**Routing key.** Each routing key is designed as a list of routing paths. For example, consider the following routing key that corresponds to the example in Figure 2(b):

A1: 0→4→1→DLID(4)
A2: 0→5→2→DLID(8)

It states that A1 is routed through the path: $switch\,0 \rightarrow switch\,4 \rightarrow switch\,1$, and finally arrives at its destination, *i.e.,* a host with a Destination LID (DLID) 4.

**3-level XGFT.** We also extend our key generation algorithms to 3-level XGFTs. Specifically, we reduce the three-level XGFT algorithm to two-level fat-tree algorithms using the following reduction. Figure 3 illustrates how each aggregation-spine sub-tree is represented by a single node. The reduced tree links have a one-to-one correspondence with the ToR-to-Aggregation level links. This allows us to use a two-step approach. First, we apply our two-level key generation algorithm to the reduced graph and obtain the upstream and downstream ToR-to-Aggregation links. This solution serves as an input for the second step. Specifically, we apply the same algorithm again for the Aggregation-Spine demand. It is important to note that the second step allows for a completely parallel/distributed implementation, as all the sub-trees are independent (this is illustrated in Figure 3 where each color shows a different sub-tree).

**Discussion.** Our design can be extended not only to additional topologies, as explained above for 3-level XGFT, but also to additional platforms such as Ethernet. In particular, the routing key creation process is platform-independent. In Ethernet-based architectures, key usage can be enabled by algorithms that allocate predetermined routes to specific flows (*e.g.,* source-routing) [4], [9], [20].

Our techniques require additional router table entries to store the route for each key. Yet, this does not appear to be a significant limitation, since the number of required keys is quite small compared to a typical routing table size.

[2]Current technology supports up to 7 bits for LMC, allowing up to 128 different LIDs per host.

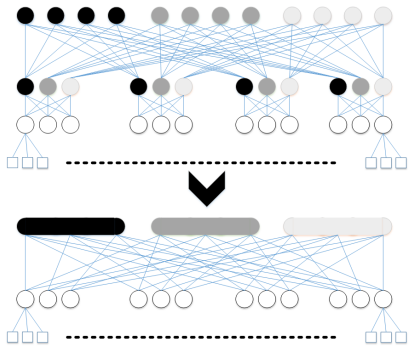


Figure 3. Reduction of a 3-level XGFT into a 2-level Fat-Tree. Each color stands for an independent sub-tree.

## IV. Evaluation

In this section, we first conduct a fundamental evaluation of the load-balancing ability of NRK using a Matlab-based simulation. Then, we compare the performance of several algorithms using an InfiniBand-based simulator running MPI applications.

### A. Load-balancing evaluation of NRK

We use a Matlab-based simulation to compare between the load-balancing abilities of *NRK* and *DmodK*, which is commonly implemented in current HPC networks. At each network switch of radix $k$, DmodK routes packets to destination $d$ through a switch port number that is obtained by applying a modulo function that depends on $d$, $k$, and the switch level and position in the network [1], [2]. Specifically, we conduct an experiment in which we increase the number of hosts, and for each host count, consider a folded-Clos topology with $n$ hosts and $\sqrt{n}$ sources, destinations and middle switches. This is the minimal number of middle switches required to make the network to be rearrangeably-non-blocking. We set the network load to 60% utilization (*e.g.,* if we have 100 hosts, then at least 60 hosts send flows to 60 other hosts), and divide the flows into different applications, with 10 flows per application, on average. The arrival order of applications is chosen randomly. Finally, we measure the maximal contention, *i.e.,* maximum number of allocated flows on a link.

Figure 4 reveals that in DmodK the maximal contention increases relatively fast with the size of the network (as could be expected when using a simple balls-and-bins model), while NRK keeps contention low.

### B. Flit-level simulation model with MPI semantics

**Simulator.** We evaluated ARK and NRK using an InfiniBand flit-level simulator provided by Mellanox Technologies that is implemented within the OMNet++ network simulation framework. It has already been extensively used and described in the literature [21]–[24]. Its switches are based on

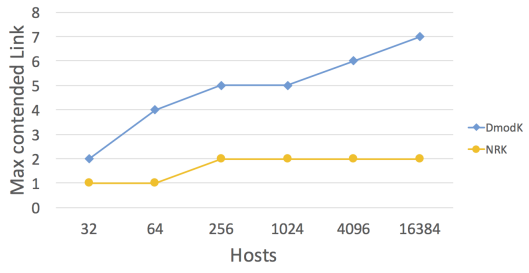



Figure 4. NRK maintains a low contention as the number of hosts increases, while DmodK shows a continuous contention increase.

```
SCOPE ALL ;
Calc [ 45.0 , 55.0 ]us 0.02us;
Send 16K 1 +x ALL ;
Recv 16K 1 -x ALL ;
Send 16K 2 -y ALL ;
Recv 16K 2 +y ALL ;
Send 16K 3 +z ALL ;
Recv 16K 3 -z ALL ;
Barrier ALL ;
```

(a) Simple MPI trace.

```
Calc [ 45.0 , 55.0 ]us 0.02us;
RODRequest 1 1000 "RKEY_ID" ;
Send 16K 1 +x 1 ;
Recv 16K 1 -x 1 ;
Barrier ALL ;
RODDone 1 1000 ;
```

(b) RODrequest.

Figure 5. (a) shows a trace of a 3-dimensional stencil application. *SCOPE ALL* represents the openMPI global communicator *MPI_COMM_WORLD* [25]. *Calc* is compute-time distributed uniformly with extra STD. *Send* and *Recv* operations show the communications along the 3 Cartesian axes and include message size, tag and communicator. A barrier collective is finally invoked to synchronize the application phase. (b) shows an example of MPI trace containing a routing key request for communicator 1 with tag 1000 and requests the route for *RKEY(Routing Key) IDENTIFIER*. When this communication phase ends, *i.e.,* after *BARRIER ALL*, an *RODDone* is sent.

forwarding tables generated by OpenSM. It includes credit propagation times and virtual-output-queue based switching.

**MPI support.** We updated the simulator to support MPI collectives for HPC applications. Unfortunately, this simulator has no support for the replay of MPI traces. We addressed this challenge by extending the simulator and adding this capability. Specifically, we designed a new file format to represent MPI program communications, in a way that allows easy translation of most MPI applications. Moreover, this file format is presented in a coherent compact view and its code supports symbolic and dense syntax for defining MPI application traces. Such a view is useful when analyzing jobs and constructing test-case jobs. Note that while Open Trace Format (OTF), an open source standard, may enable the capture and merging of MPI call traces, it does not support the collapse of the task-specific trace into a coherent compact view, and thus, is harder to analyze. Figure 5(a) illustrates an example of an MPI trace of a 3-dimensional stencil application. We can observe how the trace provides a symbolic representation of Cartesian neighbor coordinates, and recognizes all the MPI communication APIs (collectives, send/recv and waits).

**Route request support.** Another needed feature is to enable the application to request a routing key using the MPI collective. For that purpose, we define two new Route-On-Demand commands. The first is $RODRequest$. Namely, providing a routing key identifier upon a request for route. The second, $RODDone$, is sent when the application has stopped using the routing key (*i.e.,* finished its communication phase). A small example is presented in Figure 5(b).

### C. Performance evaluation of ARK and NRK

**Topology.** We verify our design using a 3-level extended generalized fat-tree (XGFT), as it is a widespread topology in HPC and datacenter networks [16]–[18]. The 3-level XGFT definition uses parameters $(h; m_1, m_2, m_3; w_1, w_2, w_3)$. $h$ is the height of the tree. Node levels are labeled 0 to 3, where 0 represents the host level and 3 the spine switch level. Each node in level $i$ has $w_i$ parents and $m_{i-1}$ children. We simulated two cluster topologies. First, a small-scale network with 216 nodes (hosts) and 108 InfiniBand switches, implemented as a 3-level XGFT with parameters $(3; 6, 6, 6; 1, 6, 6)$. Second, a large-scale network with 1728 nodes and 432 InfiniBand switches [23], [26], implemented as a 3-level XGFT with parameters $(3; 12, 12, 12; 1, 12, 12)$. All links are 40Gbps links.

**Algorithms.** To evaluate our design, we implemented both the key creation and the optimization algorithm as presented in Section II. We compared ARK and NRK with two algorithms from the literature: (1) *DmodK*, which is commonly implemented today in HPC networks; and (2) *Conga* [9], a state-of-the-art load-balancing flowlet-based algorithm. Since Conga was originally designed for a 2-level fat-tree, we enhanced its algorithm for a 3-level fat-tree topology. Specifically, we replicated the algorithm from 2-level into 3-level as described in the original *Conga* paper.

**Workload.** In the experiment, we run $m$ applications using $n$ bare-metal hosts (servers), such that each application runs on $\frac{n}{m}$ bare-metal hosts. Our benchmark workload iterates though a Cartesian MPI permutation (+x/-x/+y/-y/+z/-z), each with a synchronized barrier to ensure that every phase ends before the next one starts. Different applications are independent of each other, and each runs the same phases in a different order.

**Mapping.** We expect intra-application contention to be the strongest when application hosts are close to each other. As a result, we would expect our ARK and NRK algorithms to perform particularly well when the placement is either contiguous or semi-contiguous. Instead, we assume a detrimental *random* mapping algorithm to neutralize the effect of mapping.

**Utilization.** We define the *global network utilization* as the ratio between the communication time and the total (computation and communication) time, when the network does not have an impact, *i.e.,* all sources are connected to all destinations in a full mesh at full link capacity, such that the propagation time is only due to the transmission time.

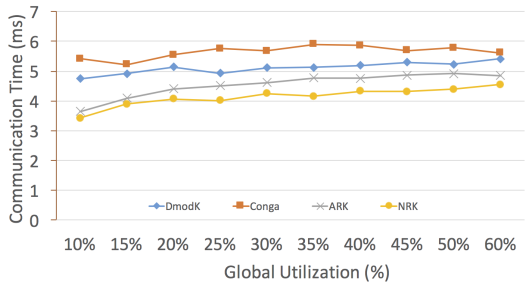



Figure 6. An experiment with 216 hosts and 6 running applications. Each application contains 36 flows spread over the network. The communication runtime is significantly improved with ARK and even more with NRK when compared to DmodK and Conga.

**Measure.** We measure the communication time of all of the phases of each application, and summarize it to obtain the total communication time of each application. Then we compare the maximal (*i.e.,* worst-case) total communication time across all applications.

**Simulations.** The first simulation is over the smaller-scale fat-tree. Each trace contains a mixture of phases with different message sizes, which vary from 4KB up to 256KB, sent through the MPI trace Cartesian axes. The variation is based on application distribution analysis from a Facebook trace [27]. The experiment contains 6 applications each with 36 flows randomly distributed among the hosts. We vary the global utilization in the different simulation runs while keeping the amount of data in each communication phase constant. Therefore, the increase in communication time as utilization increases is only due to network contention.

Figure 6 shows that Conga performs worse than DmodK, because it reacts to past congestion and does not adapt quickly to the fast changes in the traffic pattern. On the contrary, our simple ARK provides better performance because it provides intra-application contention-free routing. In addition, our more advanced NRK optimized algorithm achieves the best communication times across all utilizations, since it also tries to minimize inter-application contention. It is particularly interesting that the most significant improvements are achieved at low utilizations, as these are the typical operating points of current HPC networks.

The second evaluation is conducted on a larger-scale topology with 1728 nodes. As a result, we have more opportunities to divide the flows among the different applications. We repeat the previous experiment on a larger scale, but with a varying distribution of the flows among the applications.

Figure 7 illustrates the results. When the number of applications is small (Figure 7(a)), ARK provides most of the improvement over both DmodK and Conga, and the improvement of NRK over ARK is negligible. For instance, with 10% utilization, both ARK and NRK reduce communication runtime by 2.7× compared to DmodK and Conga. As the number of applications increases (*i.e.,* Figures 7(a) and 7(b)), the dynamic optimizer of NRK further improves its performance compared to ARK.

Figure 8 illustrates the effect of using a single message size. In this experiment, message size is always set at 32KB. As expected, in this scenario, the performance of *DmodK* is comparable to our ARK solution, as intra-application contention is less significant. On the other hand, NRK still shows an advantage due to its two-stage optimization.

Finally, Figure 9 illustrates the effect of the application communication-time duty-cycle[3] on the different routing algorithms. In this experiment, we send 16K-256KB messages while keeping the same amount of data being exchanged across the different runs. We increase the calculation time accordingly and reduce the number of iterations. Since the duty-cycle changes, fewer barriers are presented for larger messages. Therefore, we expect the experiments with larger message sizes to run faster (less synchronization).

Finally, it is notable that NRK outperforms all other algorithms in various scenarios. In addition, when the number of jobs is small, even a static solution like ARK can provide significant improvement over traditional solutions.

## V. Related Work

**Oblivious routing.** The most common way to forward packets in current HPC clusters is by using a static, oblivious routing algorithm. It could be destination-based (*e.g.,* DmodK [1], [2]), source-based, or flow-based (*e.g.,* ECMP). Such oblivious algorithms are known to provide poor load-balancing [28]. More recently, Presto [20] has provided an oblivious load-balancing approach that relies on spreading packets. Unfortunately, it suffers from packet reordering, which can be mitigated using the Juggler [29] mechanism.

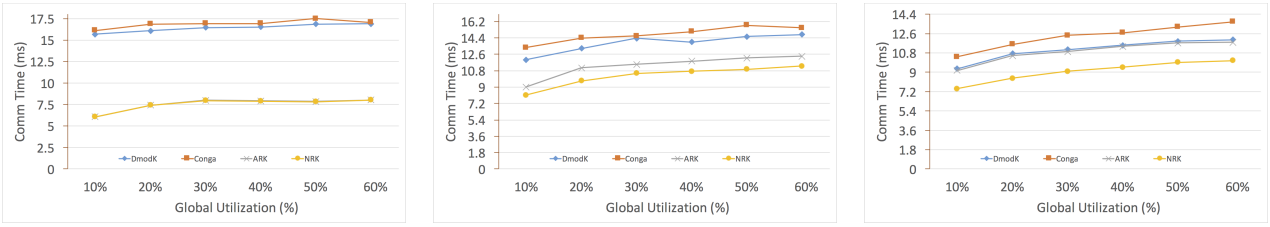
**Adaptive routing.** Adaptive routing aims to provide optimal load-balancing [24]. However, many systems cannot utilize it due to its inherent out-of-order delivery characteristics.

**Centralized routing.** Hedera, MicroTE, SWAN, Fastpass and Flowtune [3]–[8] rely on a centralized scheduler that maintains the global network state and calculates routes for network flows. Such algorithms can be slow to react to changes in traffic patterns at dataplane timescales, and face significant scalability challenges.

**Congestion-aware routing.** Recently, there have been many suggested congestion-aware routing algorithms [9]–[13], [13], [14]. These algorithms take recent congestion into account, either directly or indirectly, in order to change the routing path of packets. Unfortunately, they have had trouble handling applications with fast-changing traffic patterns, and cannot guarantee any intra-application contention-free routing.

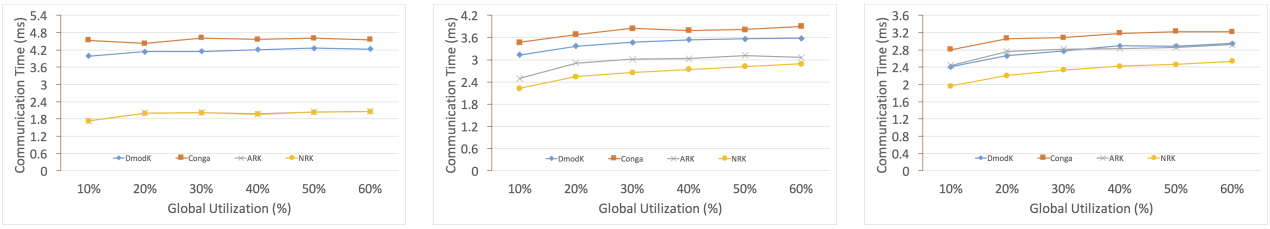
**Application-aware-routing.** Some proposals have been made to deal with application-aware routing [30]–[35]. How-

[3]The duty-cycle is defined as the time between barriers.



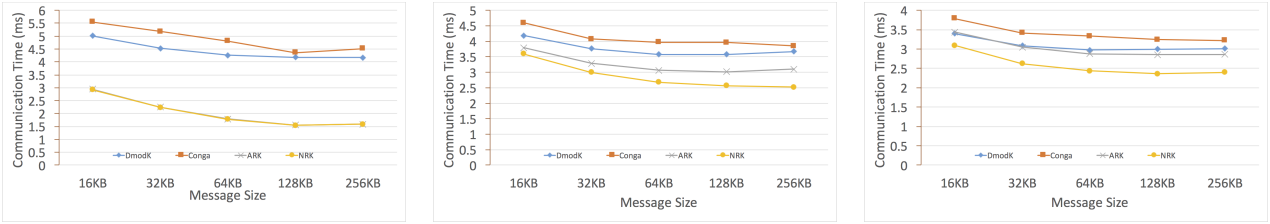

(a) 2 applications (b) 8 applications (c) 32 applications

Figure 7. Communication time as a function of utilization for 1728 hosts with mixed traffic, when flows are divided into: (a) 2 applications, (b) 8 applications, (c) 32 applications.




(a) 2 applications (b) 8 applications (c) 32 applications

Figure 8. Communication time as a function of utilization for 1728 hosts with single message size of 32KB, when flows are divided into: (a) 2 applications, (b) 8 applications, (c) 32 applications.




(a) 2 applications (b) 8 applications (c) 32 applications

Figure 9. Communication time as a function of message size for 1728 hosts with single message size, when flows are divided into: (a) 2 applications, (b) 8 applications, (c) 32 applications.

ever, most of them rely on mixed-integer linear programming, which is hard to scale with network demands. Other approaches do not deal with the whole routing solution, but instead, act as congestion control. A recent work by [34] suggested a linear-program based solution, however the complexity of their algorithm may be as high as a six-order polynomial with respect to the size of the network. Our solution reduces complexity by decoupling the inter- and intra- application contention problems and relies on the lightweight complexity of the maximal cardinality matching algorithm. Finally, LaaS [35] introduces a routing with guaranteed inter-application contention-free routing, at the expense of a slightly lower host utilization.

**Intra-application contention.** The differentiation between intra- and inter-application contention is not new and has already appeared in the literature, *e.g.,* [36].

**MPI-based algorithms.** In [37], the authors propose to use the network closeness between virtual machines to modify the internal structure of the MPI-based algorithm. This approach is complimentary to ours, and can reduce network utilization, although it does not prevent data flows of the same application from creating intra-job-contention. In addition, a recent study [38] proposes a proactive separation of such applications to disjoint buffer resources.

## VI. Conclusions

In this paper, we introduced *Routing Keys*, a new scalable routing paradigm for HPC networks that decouples intra- and inter-application flow contention. We presented both ARK that provides intra-application contention-free routing, and the more advanced NRK that significantly reduces inter-application contention while preserving the intra-application contention-free routing. Using extensive evaluations, we showed that both ARK and NRK achieve a significant

improvement in the application performance when applied to an XGFT topology over InfiniBand.

## ACKNOWLEDGMENT

We would like to thank Shay Vargaftik and Gal Mendelson for their most valuable comments and suggestions.

This work was partly supported by the Shillman Fund for Global Security, the Gordon Fund for Systems Engineering, the Hasso Plattner Institute Research School, the Israel Ministry of Science and Technology, and the Technion Fund for Security Research.

## REFERENCES

[1] B. Towles and W. Dally, "Worst-case traffic for oblivious routing functions," *IEEE Computer Architecture Letters*, 2002.
[2] W. Nienaber, S. Mahapatra, and X. Yuan, "Improving Performance of Deterministic Single-Path Routing on 2-Level Generalized Fat-Trees," *IPDPS*, 2011.
[3] 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," *ACM SIGCOMM CCR*, vol. 43, no. 4, pp. 3–14, 2013.
[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." *NSDI*, 2010.
[5] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," *ACM CoNEXT*, 2011.
[6] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," *ACM SIGCOMM*, 2013.
[7] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *ACM SIGCOMM*, 2014.
[8] J. Perry, H. Balakrishnan, and D. Shah, "Flowtune: Flowlet control for datacenter networks," in *Usenix NSDI*, 2017.
[9] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," *ACM SIGCOMM*, 2014.
[10] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *ACM SIGCOMM CCR*, vol. 37, no. 2, pp. 51–62, 2007.
[11] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, "Scalable, optimal flow routing in datacenters via local link balancing," *ACM CoNEXT*, 2013.
[12] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable load balancing using programmable data planes," *SOSR*, 2016.
[13] N. Katta, M. Hira, A. Ghag, I. Keslassy, J. Rexford, and C. Kim, "CLOVE: How I learned to stop worrying about the core and love the edge," *ACM HotNets*, 2016.
[14] E. Vanini, R. Pan, M. Alizadeh, T. Edsall, and P. Taheri, "Let it flow: Resilient asymmetric load balancing with flowlet switching," *Usenix NSDI*, 2017.
[15] D. B. West *et al.*, *Introduction to graph theory*. Prentice hall Upper Saddle River, 2001, vol. 2.
[16] S. Ohring, M. Ibel, S. Das, and M. Kumar, "On generalized fat trees," in *IPPS*, 1995.
[17] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM*, 2008.
[18] A. Andreyev, "Introducing data center fabric, the next-generation facebook the next generation datacenter network," 2014.
[19] "Opensm release notes, rev 0.3.1," 2004.
[20] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM*, 2015.
[21] E. G. Gran, E. Zahavi, S.-A. Reinemo, T. Skeie, G. Shainer, and O. Lysne, "On the relation between congestion control, switch arbitration and fairness," *CCGrid*, pp. 342–351, May 2011.
[22] J. Domke, T. Hoefler, and W. E. Nagel, "Deadlock-free oblivious routing for arbitrary topologies," in *IEEE IPDPS*, 2011.
[23] E. Zahavi, "Fat-tree routing and node ordering providing contention free traffic for MPI global collectives," *JPDC*, 2012.
[24] E. Zahavi, I. Keslassy, and A. Kolodny, "Distributed adaptive routing convergence to non-blocking DCN routing assignments," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 1, pp. 88–101, 2014.
[25] E. Gabriel, G. E. Fagg, G. Bosilca *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.
[26] E. Zahavi, G. Johnson, D. J. Kerbyson, and M. Lang, "Optimized InfiniBandTM fat-tree routing for shift all-to-all communication patterns," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 2, pp. 217–231, 2010.
[27] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," *ACM SIGCOMM*, 2014.
[28] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen, "Parallel randomized load balancing," in *ACM symposium on Theory of computing*. ACM, 1995, pp. 238–247.
[29] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh, "Juggler: a practical reordering resilient network stack for datacenters," *EuroSys*, 2016.
[30] M. A. Kinsy, M. H. Cho, T. Wen, E. Suh, M. Van Dijk, and S. Devadas, "Application-aware deadlock-free oblivious routing," *ISCA*, 2009.
[31] A. H. Abdel-Gawad and M. Thottethodi, "Transcom: Transforming stream communication for load balance and efficiency in networks-on-chip," in *IEEE/ACM International Symposium on Microarchitecture*, 2011.
[32] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *IEEE Infocom*, 2015, pp. 424–432.
[33] J. Jiang, S. Ma, B. Li, and B. Li, "Tailor: Trimming coflow completion times in datacenter networks," *IEEE ICCCN*, 2016.
[34] A. H. Abdel-Gawad and M. Thottethodi, "Scalable, global, optimal-bandwidth, application-specific routing," in *IEEE HotI*, 2016.
[35] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy, "Links as a Service (LaaS): Guaranteed tenant isolation in the shared cloud," in *ACM/IEEE ANCS*, 2016.
[36] A. Jokanovic, G. Rodriguez, J. Sancho, and J. Labarta, "Impact of inter-application contention in current and future HPC systems," in *IEEE HotI*, 2010.
[37] Y. Gong, B. He, and J. Zhong, "Network performance aware MPI collective communication operations in the cloud," *IEEE TPDS*, 2013.
[38] A. Jokanovic, J. Sancho, J. Labarta, G. Rodriguez, and C. Minkenberg, "Effective quality-of-service policy for capacity high-performance computing systems," in *IEEE HPCC*, 2012.