

Boosting HPC Applications in the Cloud Through JIT Traffic-Aware Path Provisioning

Guilherme R. Pretto[1], Bruno L. Dalmazo[1(✉)], Jonatas A. Marques[1], Zhongke Wu[2], Xingce Wang[2], Vladimir Korkhov[3,4], Philippe O. A. Navaux[1], and Luciano P. Gaspary[1]

[1] Federal University of Rio Grande do Sul, Porto Alegre, Brazil
{grpretto,bldalmazo,jamarques,navaux,paschoal}@inf.ufrgs.br
[2] Beijing Normal University, Beijing, China
{zwu,wangxingce}@bnu.edu.cn
[3] Saint Petersburg State University, Saint Petersburg, Russia
[4] Plekhanov Russian University of Economics, Moscow, Russia
v.korkhov@spbu.ru

Abstract. Data centers, clusters and grids have historically supported High-Performance Computing (HPC) applications. Due to the high capital and operational expenditures associated with such infrastructures, in recent past, we have witnessed consistent efforts to run HPC applications in the cloud. The potential advantages of this shift include higher scalability and lower costs. If on the one hand, app instantiation – through customized Virtual Machines (VMs) – is a well-solved issue, on the other, the network still represents a significant bottleneck. When switching HPC applications to be executed on the cloud, we lose control of where VMs will be positioned and of the paths that will be traversed for processes to communicate with one another. To alleviate this problem, and taking advantage of new advances in programmable networks, we propose a mechanism for dynamic, just-in-time path provisioning in cloud infrastructures. It continuously monitors the network conditions and, given the current communication patterns of the application, systematically (re)programs paths to avoid uncongested links and reduce end-to-end delays. The proposed mechanism achieves a speedup of up to 44.24% regarding application runtime when compared to the traditional shortest-path, static approach.

Keywords: HPC applications · Cloud infrastructures · Link usage-aware path provisioning

1 Introduction

Handling massive amounts of data is commonplace for most modern scientific, engineering, and business applications. As these applications need to target

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 702–716, 2019.
https://doi.org/10.1007/978-3-030-24305-0_52

big data-related challenges, while delivering expected results promptly, they frequently pose large computing power requirements. In this context, High-Performance Computing (HPC) becomes a key factor for speeding up data processing. To this end, HPC solutions have traditionally taken advantage of cluster, grid and data center infrastructures for running applications having those computing power requirements [6]. More recently, we have witnessed consistent efforts to run HPC applications in the cloud. The pay-per-use cost model makes cloud computing a promising environment for HPC, which can be provided with instant availability and flexible scaling of resources (*i.e.*, elasticity).

Although the allocation of virtual machines in the cloud (for HPC application execution) has been extensively studied, the inter-process communication rates still represent the main performance bottleneck [10]. In addition to the possibility of virtual machines to be deployed physically far from each other into the cloud, the provisioned paths are typically static and based on shortest paths (potentially traversing congested, high delay links). The dynamic nature of communication patterns observed in most HPC applications makes these limitations even more prominent. In summary, subjecting HPC flows to high network latencies is *still* one of the leading open research challenges that cloud environments have to cope with to be able to offer a suitable infrastructure for HPC.

Previous studies [3,6,13,15,18] have assessed the feasibility of using public clouds for HPC. Their findings suggest that clouds were not designed for running tightly coupled HPC applications. The main limitation is the poor network performance resulting from I/O virtualization overhead, processor sharing, and usage of commodity interconnection technologies. Lee *et al.* [9] and Faizian *et al.* [4] have identified limitations of network technologies used by cloud infrastructures, namely the use of simplistic routing schemes, which may result in degraded communication performance. To overcome this limitation, they have proposed routing mechanisms that avoid congested paths, which however are computed priorly to the execution of an HPC application and therefore can cope neither with network traffic fluctuation nor with varying communication patterns.

In this paper, we advance the state-of-the-art by proposing a novel mechanism for dynamic, just-in-time path provisioning in cloud infrastructures. It monitors cloud infrastructure links and keeps track of their utilization. Taking advantage of new advances in programmable networks, the proposed mechanism dynamically computes and programs the best paths (*i.e.*, shortest uncongested paths) between processes according to their momentaneous communication needs. In addition to the mechanism, another equally important contribution is the formalization of a *Link Usage-Aware Routing* (LUAR) strategy, which plays a crucial role in reducing end-to-end communication delays. To prove the concept and technical feasibility, we have evaluated our solution using the experimental setup of a realistic cloud infrastructure with different levels of background traffic and multiple instances of the NAS Parallel Benchmarks [2].

The remainder of the paper is organized as follows. In Sect. 2, we cover some of the most prominent related work. In Sect. 3, we describe the overarching

conceptual solution and its main components. In Sect. 4, we present the evaluation and discuss the obtained results. In Sect. 5, we conclude the paper with some final remarks and prospective directions for future research.

2 Related Work

Several feasibility studies about using public clouds for HPC point out the network communication overheads as a significant barrier. Other investigations propose strategies to reduce these overheads. Two classes of solutions stand out: those resorting on topologies that attempt to maximize connectivity between nodes and those that explore resource and/or network management. Given the nature of our proposed approach, next, we revisit work on the general problem of executing HPC applications in the cloud and recent proposals that apply Software-Defined Networks (SDNs) as a crucial component to configure efficient inter-process communication paths.

Gupta *et al.* [6] investigate network bottlenecks in commercial cloud infrastructures and propose new mechanisms to improve the performance experienced by HPC applications. Essentially, they employ lightweight virtualization and grant Virtual Machines (VMs) native access to physical network interfaces. Despite the overhead reduction in average turnaround time (up to 2x) and throughput (up to 6x), the authors recognize that their proposal may not be enough to improve the performance of HPC applications. Similarly, Ramakrishnan *et al.* [14] claim that virtualization of network resources accounts for the main performance issue, being responsible for at least 60% delay and throughput degradation if compared with typical local cluster and supercomputer options.

Mauch *et al.* [11] introduce an approach to use InfiniBand in a private virtualized environment. It allows for an individual network configuration to operate with QoS mechanisms. Similarly, Alsmadi *et al.* [1] propose a method for logically slicing the network and isolating applications from one another. While the former is based on a complex architecture that requires interfacing with several APIs and protocols to configure the network, the latter cannot handle path reallocation in case of a switch or link failure. Moreover, both proposals deploy static paths that cannot change during application execution.

Motivated by the fact that InfiniBand is increasingly available in commercial clouds, Lee *et al.* [9] address one of its central weakness for HPC, namely the employment of simple routing schemes such as *destination-mod-k* routing. The authors propose an enhancement to InfiniBand, incorporating OpenFlow-style SDN capability to provision uncongested paths. While the proposal is a cornerstone towards the employment of programmable networks to speed up HPC applications in the cloud, it fails by requiring routes to be determined before the execution of an application. It, therefore, wastes a valuable opportunity to adapt the network to meet fluctuating application communication demands.

Along the same lines as the work by Lee *et al.*, Faizian *et al.* [4] include SDN-related functionality to Dragonfly networks and propose a set of routing schemes to achieve the performance required by HPC applications. One limitation of this

work is the lack of a mechanism for accurate traffic demand estimation, making it difficult to make optimal global decisions. Another limitation is that even when several paths are available, only two (*i.e.*, a primary and a backup) are considered for forwarding a packet. A final shortcoming is that these paths are provisioned upon application instantiation and cannot change.

3 The Proposed Path Provisioning Approach

As a relevant research step further in comparison with the related work, we propose a mechanism based on SDN for *just-in-time path selection and provisioning* in cloud infrastructures. It continuously monitors the network conditions and, given the current communication patterns of an HPC application, systematically (re)programs paths to avoid congested links and reduce end-to-end delays. The proposed approach is *complementary* to the VM placement and instantiation process, which we assume will be taken care of by a third party resource management system such as OpenStack [17] or OpenNebula [12].

Figure 1 introduces the basis of our solution, highlighting its main conceptual components and their interactions. In the following subsections, we describe them in detail, starting with (*i*) the SDN-based Path Provisioning Controller, then (*ii*) illustrating a working example, and finally (*iii*) presenting the accompanying proposed routing strategy.

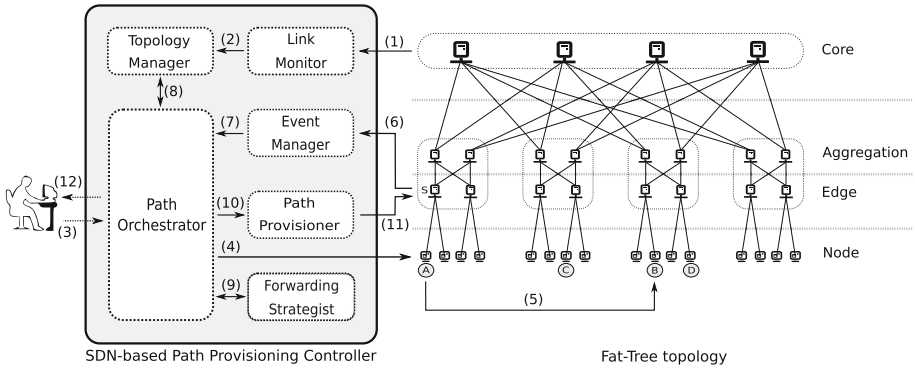


Fig. 1. Components of the proposed solution and their interactions

3.1 SDN-Based Path Provisioning Controller

The SDN-based Path Provisioning Controller consists of six components: *Link Monitor*, *Topology Manager*, *Event Manager*, *Path Orchestrator*, *Forwarding Strategist*, and *Path Provisioner*. Next we summarize the role of each of these components.

Link Monitor. This component plays a vital role in periodically gathering traffic statistics from the forwarding devices. Assuming the infrastructure is OpenFlow-enabled, this information can be obtained via `OFPPortStatsRequest` requests. The infrastructure operator must tune the frequency of this process. Highly frequent requests favor accuracy while sparsed ones favor low intrusiveness. After each round of information collection, the Link Monitor calculates the amount of traffic that traversed each link and derives its average utilization. This information is calculated according to Eq. 1 and sent to the Topology Manager component.

$$LkUsage_i(t, t-1) = LkCapacity_i - \frac{[tx + rx_bytes](t) - [tx + rx_bytes](t-1)}{\Delta(t, t-1)} \quad (1)$$

Topology Manager. It maintains updated information about the topology of the cloud network infrastructure in the form of an *annotated graph*. Forwarding devices (*e.g.*, OpenFlow switches) are abstracted as vertices while physical links are represented as edges. The average utilization of each link, measured by the Link Monitor, is used to denote the edge weight. The reader should keep in mind that (*i*) the representation of the topology might be updated in response to changes in the infrastructure (*e.g.*, due to a link failure – event `OFPErrorMsg`) and (*ii*) the weights are updated regularly in response to periodic link usage measurements carried out by the Link Monitor.

Event Manager. Whenever a new flow starts between two processes of an HPC application or a previously existing one remained idle for a long time, there will be no path provisioned in the network to route its packets. The forwarding device that is closer to the source host then triggers an event (`OFPPacketIn`) that is received by the Event Manager component. This component intermediates communication with the Path Orchestrator for a path selection procedure.

Path Orchestrator. The orchestrator, as the name implies, coordinates the process of path selection and provisioning. The component reacts to both changes in the physical topology (informed by the Topology Manager) and events of the type `OFPPacketIn` (reported by the Event Manager). It consumes (updated) information made available by these components, invokes the Forwarding Strategist for the determination of a path and resorts to the Path Provisioner for its deployment. The Path Orchestrator is also responsible for interfacing with the human network operator (*e.g*, to receive configuration commands and send him/her notifications about the path provisioning process).

Forwarding Strategist. The Forwarding Strategist can implement different approaches for determining a path between a source and a destination (HPC application) process. It uses an updated view of the annotated graph and location information of the processes to be "interconnected". The problem addressed by

this component is then of a *graph-based path finding* one. As a proof of concept, in this paper, we propose a *Link Usage-Aware Routing* strategy, which favors shortest and uncongested paths. It is described in detail in Subsect. 3.3.

Path Provisioner. This component is responsible for performing the actual deployment of end-to-end paths on the network infrastructure. This process unfolds in the invocation of a coordinated set of `OFPFlowMod` commands to install forwarding rules on the devices that (are calculated to) compose the paths. The duration of these rules is determined by the operator, who can favor either path freshness/efficiency or stability/reuse.

3.2 A Working Example

To illustrate how the described components interact, we introduce now a working example. Initially, recall that the Link Monitor is expected to be continuously monitoring the traffic to consolidate statistics from the cloud network infrastructure (Flow 1 in Fig. 1). The gathered information (*tx_bytes* and *rx_bytes*) is sent to and stored by the Topology Manager (Flow 2).

The process of deploying an HPC application consists of (*i*) instantiating and configuring the VMs (with any off-the-shelf resource management system), as well as (*ii*) requesting our proposed SDN-based Path Provisioning Controller (via the Path Orchestrator component) to manage the path provisioning process (Flow 3). Should the third-party resource management system expose an API that can be used to control the resource provisioning life-cycle, then it is possible to integrate (*i*) and (*ii*) to have the whole application deployment/execution process dealt with by the SDN-based Path Provisioning Controller (3 and 4).

Consider the situation in which an HPC application has been deployed and that the process running on node A needs to communicate with the process on node B (5). When the first packet from A reaches its closest forwarding device (*s* in the figure), the device will not know how to route it further and will send an `OFPPacketIn` event to the Event Manager (6). This component will then pass the event to the Path Orchestrator (7).

At this point, the Path Orchestrator consults the Topology Manager to receive updated information about the current topology and the utilization of the links (8). Next, it invokes the Forwarding Strategist to determine an optimized route to connect processes running on A and B (9). Finally, it requests the Path Provisioner (10) to deploy the forwarding rules on the network devices (11). From this moment on, the packet that triggered the event is reintroduced by the controller into the network, and the remaining flow packets will be transmitted through the configured route.

The process above is executed for each new application communication flow (or when the physical topology changes). To avoid frequent communications with (and interventions from) the external controller, the operator can resort to two system parametrization options. The first is setting up a higher value for the `idle_timeout` associated with the flow rules configured on the forwarding devices. The consequence will be that provisioned paths will be kept active

for more extended periods (assuming that short inter-process *no* communication periods will be succeeded by new message exchanges). The second option is *proactively* provisioning paths for which there exists a single possible route, or that will be undoubtedly necessary along the execution of a whole application (*e.g.*, master/worker communication channels). An in-depth analysis of the additional performance gains (*i.e.*, in application runtime) reached with these configuration options are out of the scope of this work and are left as future work.

3.3 Link Usage-Aware Routing (LUAR)

In this subsection, we propose and formalize our strategy – called *Link Usage-Aware Routing* (LUAR) – designed to efficiently compute the best path from a source to a destination point in a cloud network topology. LUAR is one of the possible approaches to be implemented by the Forwarding Strategist. It considers the best path to be the shortest, least utilized, currently available path. In the future, we plan to investigate other algorithms and approaches to this task.

Algorithm 1 shows the pseudo-code of LUAR. The algorithm has five input parameters: a set V of nodes in the topology graph; a vector of adjacency lists E, where $E(v)$ lists the nodes connected to node v; a link utilization matrix U, where $U(v, u)$ indicates the utilization of the link that connects node v to u; and the source and destination nodes s and t. The algorithm can be logically divided into two blocks. Next, we detail each block.

The first block (Lines 1–12) is an adaptation of the Breadth-First Search (BFS) algorithm extended to record, for every node, its possible predecessors in any of the shortest paths starting from the source. The algorithm maintains two main data structures in Block 1, Predecessors and Seen, that indicate, respectively, the possible predecessors for each node v and the first level in which they were seen during the search. These structures are initialized with *nil* values for every node in the graph (Line 2). In Line 3, we set Predecessors(s) of source node s to an empty list (since it has no predecessors) and Seen(s) to level 0 (since the search will start from it). The algorithm also maintains three auxiliary variables l, N and C (Lines 4 and 6), where l indicates the current level of the search, N is the set of nodes to explore in the next level, and C is the set of nodes being explored in the current level.

Lines 5–12 contain the main repeat loop of Block 1. Every iteration of the outermost loop represents the exploration of a level in the graph. Each iteration starts by updating C to be what was previously N and resetting N to an empty set (Line 6). The algorithm then checks, for each node v currently in C (Line 7), if any of its adjacent nodes $u \in E(v)$ has not been visited (Line 9). In the positive case (Line 10), the Predecessor(u) is initialized to a list containing only the node currently being explored (*i.e.*, v), u is added to the set N of nodes to explore in the next level and is also marked as to have been seen in next level $l + 1$. Otherwise, LUAR checks if node u has already been seen in the next level (Line 11), in which case it simply appends v to the list of predecessors of u. This procedure is repeated until all nodes in the graph have been visited, *i.e.*, no item

Algorithm 1. Compute the shortest, least utilized, currently available path from source s to destination t.

Input: V, E, U, s, t

1: ▷ ***Block 1***: Generate predecessor tree for the graph starting from node s.
2: Predecessors$(v) \leftarrow nil, \forall v \in V$; Seen$(v) \leftarrow nil, \forall v \in V$
3: Predecessors$(s) \leftarrow$ EmptyArrayList(); Seen$(s) \leftarrow 0$
4: $l \leftarrow 0$; $N \leftarrow \{s\}$
5: **while** $N \neq \varnothing$ **do**
6: $C \leftarrow N$; $N \leftarrow \varnothing$
7: **for** $v \in C$ **do**
8: **for** $u \in E(v)$ **do**
9: **if** Seen$(u) = nil$ **then**
10: Predecessors$(u) \leftarrow$ ArrayList$([v])$; $N \leftarrow N \cup \{u\}$; Seen$(u) \leftarrow l+1$
11: **else if** Seen$(u) = l + 1$ **then** Append(Predecessors$(u), v$)
12: $l \leftarrow l + 1$
13: ▷ ***Block 2:*** Compose the shortest paths and find the least utilized one.
14: $W \leftarrow$ArrayList$([(t, 0, 0)])$; $\rho_{best} \leftarrow nil$; $au_{best} \leftarrow \infty$
15: **while** Length$(W) > 0$ **do**
16: $v, i, au \leftarrow$Last(W)
17: **if** $v = s$ **and** $au < au_{min}$ **then**
18: $\rho_{best} \leftarrow$ Reversed(List({FirstItem(w): $w \in W$}))
19: $au_{best} \leftarrow au$
20: **if** $i <$ Length(Predecessors(v)) **then**
21: Append(W, (Predecessors$(v)_i, 0, au + U$(Predecessors$(v)_i, v$)))
22: **else**
23: RemoveLast(W)
24: **if** Length$(W) > 0$ **then**
25: SecondItem(Last(W)) $\leftarrow$ SecondItem(Last(W)) +1

Output: ρ_{best}

is added to N during an outermost iteration (Line 5). In the end of Block 1, data structure Predecessors has been populated with the predecessors of all nodes in the shortest paths starting from node s.

Block 2 (Lines 13–31) is a modified version of the non-recursive Depth-First Search (DFS) algorithm. The search is started from the destination node t and targets source node s (*i.e.*, a reverse search) and the Predecessors data structure (populated in Block 1) is used instead of the original adjacency list. Along with the search, LUAR records the path being traversed and its aggregate utilization.

The path is recorded using an array list W of triples (v, i, au_{cur}), where v is a node, i is index of the next predecessor to explore for v, and au is the current aggregate utilization in the path ending on node v (Line 4). Block 2 also maintains two auxiliary variables (Line 14): the best path ρ_{best} found in the search (initialized as nil); and the aggregate utilization au_{best} of this best path (initialized as ∞).

Lines 15–25 contain the main loop of Block 2. Each iteration starts by loading the last triple from W into variables v, i, and au (Line 16). Line 17 checks whether the current node v is the source node s and the current aggregate path utilization au is smaller than the best known value au_{best}. In case both conditions are true, LUAR builds the current path by creating a list from the nodes listed in W, which are the first items of each triple (Line 18), and updates the best known path ρ_{best} to be it. Line 19 also updates the minimum known aggregate utilization au_{best} to be the current one au. Regardless of the previous conditions, if there is predecessors that should still be explored for node v (Line 20), a new triple is added to W containing the next predecessor of v to be explored and adding up the utilization from the predecessor to v to the current aggregate utilization (Lines 21). Otherwise, LUAR removes the last triple from W, and if W does not become empty, it increments by one the index of the next predecessor to explore from the last triple. This process is repeated until all of the shortest paths from source node s to destination node t have been analyzed. At the end of the algorithm, variable ρ_{best} is returned and indicates the shortest, least utilized, currently available path.

The worst-case complexity of LUAR is given by $\mathcal{O}(n + m)$, where n is the number of nodes and m refers to the number of links in the topology, since both of the adapted algorithms (BFS and DFS) have complexity $\mathcal{O}(n + m)$ and are executed sequentially in our algorithm. Therefore, LUAR runs in polynomial linear time to the number of devices and links in the network and is suited to quickly compute the best path to route traffic from two processes of an HPC application.

4 Evaluation

In this section, we evaluate the proposed solution aiming at assessing application execution speedup under varying conditions. We start by detailing the experimental setup. Next, we characterize the workloads employed. Finally, we present and discuss the results obtained with the SDN-based Path Provisioning Controller (with LUAR).

4.1 Experimental Setup

We implemented the cloud infrastructure with Mininet [8]. Hosts were instantiated as full-fledged lightweight Linux VM instances. The network infrastructure was comprised of Open vSwitch version 2.0.2 switches (that support OpenFlow version 1.3) and links of 1 Gbps. The SDN-based Path Provisioning Controller

was implemented as an application that runs on top of Ryu [16]. This experimental environment was deployed on a 3.4 GHz, 64-bit quad-core Intel (i7-8550U) processor, with 16 GB of RAM and running Ubuntu 18.04 LTS.

The right-most side of Fig. 1 illustrates a simplified version of the cloud network topology used in the experiments. It is a *fat-tree* that, as usual, is organized in three hierarchical layers: *edge*, *aggregation*, and *core*. In our topology, 32 hosts ($h1 \ldots h32$) were attached to edge switches. In the aggregation layer, we instantiated 16 switches, while in the core, 4 (totaling 20: $s1 \ldots s20$). The switches in the aggregation layer were organized in 4 PoDs (Points of Delivery).

Throughout the evaluation, two distinct scenarios were considered: one named *Dense*, in which the VMs allocated to execute an HPC application were randomly provisioned within the same PoD, and one entitled *Sparse*, meaning the VMs were spread across multiple PoDs. The workloads to which the experiments were submitted are described in the next Subsect. 4.2. We compared the results obtained using LUAR with those observed with more traditional forwarding strategies, namely *Shortest Path* and *Random*. The metric employed for the comparison was *application runtime*. Each experiment was repeated 30 times. Results have a confidence level of 90%.

4.2 Workloads

Regarding the HPC application used to carry out the experiments, we employed the NAS Parallel Benchmarks (NPB) version 3.3 [2]. The benchmarks are designed to support the performance evaluation of HPC infrastructure and systems. They are derived from computational fluid dynamics and consist of five kernels and three compact applications. The choice for NPB was due to, in addition to encompassing a wide range of HPC application profiles, it is regarded as a reference and is extensively employed for performance evaluation in the literature.

We selected six representative NPB applications, whose communication patterns are illustrated in Fig. 2 (considering their instantiation with four nodes). Thick edges denote intensive communication demands, while thin ones represent the opposite. Observe that in BT all nodes exchange data with one another, but at low amounts. Similarly to BT, in EP data exchange is also low, but not all nodes communicate directly (*e.g.*, 2 and 3 in the example). The LU and MG applications take the form of a ring-based topology, with massive information exchange. Finally, CG and SP comprise an initial data distribution step, followed by data-intensive communication between half of the process pairs.

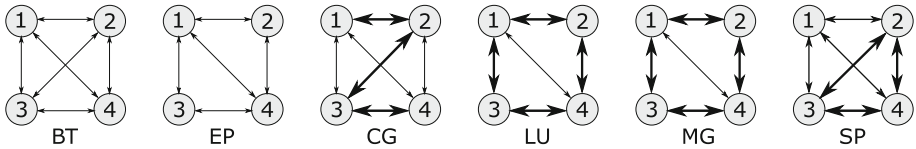


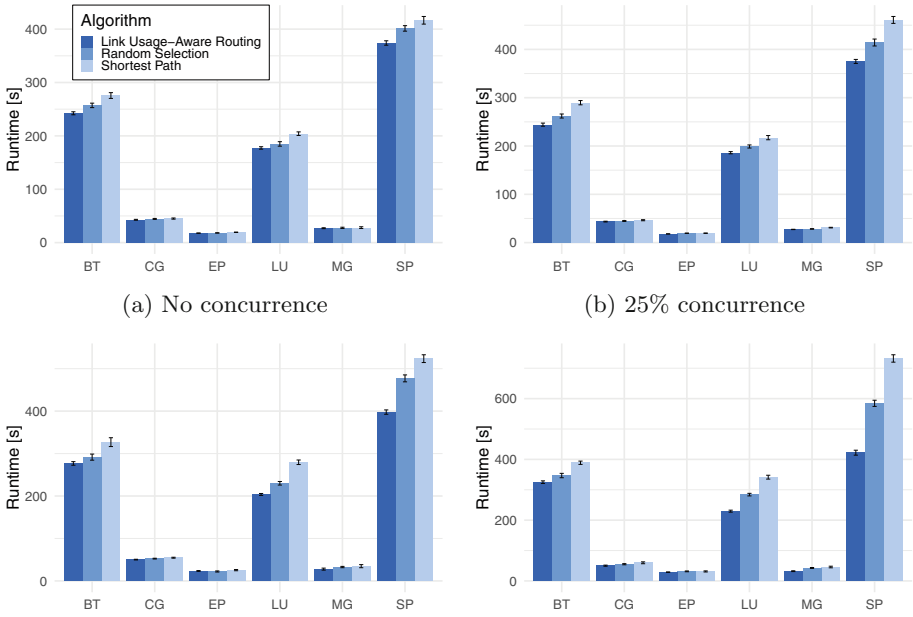
Fig. 2. Communication patterns of six NPB applications

According to recent data center measurements [5], a large portion of data center traffic is carried by a small fraction of long-lived, voluminous flows (called *elephant* flows). Similarly, Joy *et al.* [7] report that nine out of ten flows transfer less than 1 MB of data (*mice* flows), while 90% of the total bytes transferred belongs to flows larger than 100 MB (*i.e.*, elephant).

In line with the studies mentioned above, we emulated a realistic traffic workload composed of mice and elephant flows. Each experiment had a duration of 15 min and was composed of two states: *warmup* (1–3 min) and *steady* (4–15 min) states. During the former, the number of ongoing background flows increased gradually until reaching the desired level of resource competition. During the latter, whenever a background flow ended, a new one was started to maintain the expected overall concurrence level. To create such a new flow, we randomly selected a pair of *producer* and *consumer* processes, and determined its duration to be of up to 4 min (respecting the proportion of 9/10 mice and 1/10 elephant flows). It was also during the steady state that we instantiated the HPC application, *i.e.*, one of the NAS benchmarks shown in Fig. 2, complying with either the Dense or the Sparse scenario.

4.3 Results

We now present and discuss the results obtained starting with the Sparse scenario, followed by the Dense one. Figure 3 summarizes the application performance when the VMs (and, as a consequence, the HPC application processes)



(a) No concurrence (b) 25% concurrence

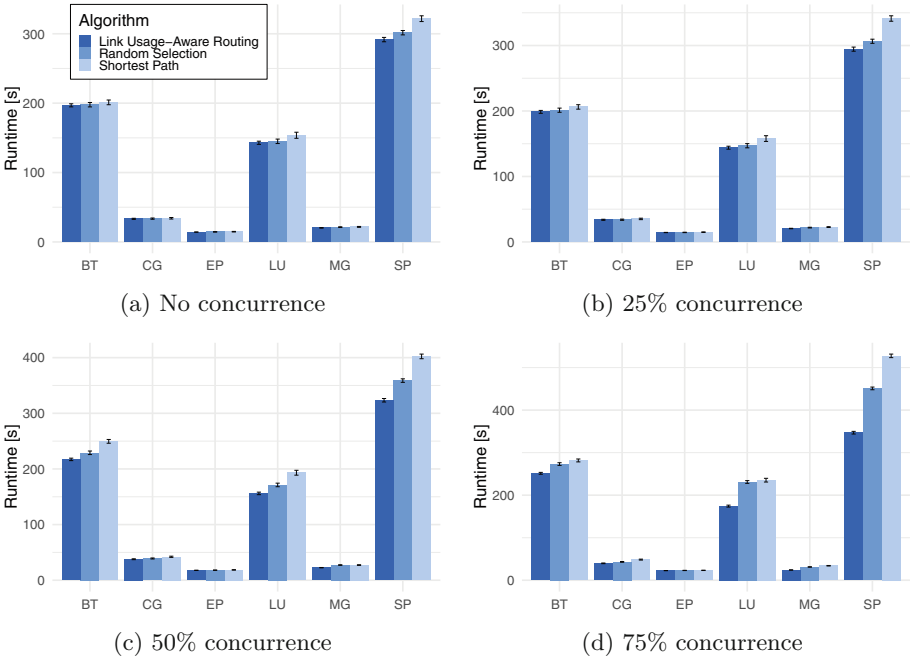
(c) 50% concurrence (d) 75% concurrence

Fig. 3. Application performance observed for the Sparse scenario

are spread across multiple PoDs. Notice that larger applications tend to take more advantage of our proposed approach. The application runtime observed for BT, LU, and SP is, on average, 22.86 s lower (8.79% speedup) than what is achieved with Shortest Path and Random. The gains of LUAR for CG, EP, and MG are more modest, being of nearly 1-second runtime improvement.

Another important angle from which to analyze the results is the performance gains as a function of the concurrency level. The higher the contention of the cloud network, the higher the advantage and the value of employing LUAR. On average, it outperforms Shortest Path by 8.61% in (a – no concurrence), 11.23% in (b), 17.82% in (c), and 31.35% in (d – 75% concurrence). Taking both the size of an application and the level of concurrence into consideration simultaneously, we observe that the shortest application, EP, exhibits nearly constant 8% runtime improvement with LUAR (against Shortest Path) regardless of concurrency level. Conversely, the largest application, SP, presents consistently increasing gains of 9.87% (a), 20.23% (b), 26.61% (c), and 44.24% (d).

We move now to the scenario in which the VMs (and the HPC application processes) are allocated on hosts of a single PoD. This scenario is expected to be more common, but with fewer opportunities to speed up application performance through just-in-time traffic-aware path provisioning. Figure 4 shows the results. Despite the reduction of path diversity on the network edge for the Dense scenario, LUAR beats its counterparts. For example, SP executes 24.36% faster than when using Shortest Path under the condition of 50% concurrency.



(a) No concurrence (b) 25% concurrence

(c) 50% concurrence (d) 75% concurrence

Fig. 4. Application performance observed for the Dense scenario

Other applications are too short-lived to take full advantage of LUAR but still run faster. This is the case of CG, which took 34.12 s to complete with Shortest Path and 33.20 s with LUAR (2.7% faster).

Finally, it is worth analyzing the performance observed with Random. The reason is that this strategy mimics load balancing, which is currently used by state-of-the-art cloud infrastructures. Although its results are better than the Shortest Path, it performs consistently worse, *i.e.*, for all applications and concurrency levels, than LUAR. Our gains range, on average, from 2.23% to 17.47% for the Dense scenario (and from 4.51% to 19.1% for the Sparse one). These results underscore that deciding paths dynamically and with accurate knowledge of network traffic has the potential to allow better routing decisions.

5 Final Considerations and Future Work

In this paper, we proposed a mechanism for *dynamic* path provisioning in cloud network infrastructures. It keeps track of *link utilization* and deploys *shortest, least utilized paths* to "interconnect" the processes of a running HPC application. The mechanism unfolds into *two main research contributions*: (*i*) an SDN-centric path provisioning architecture and (*ii*) a link usage-aware routing strategy.

The results suggest that the proposed solution is promising. LUAR performs particularly well under challenging, yet ordinary circumstances such as when VMs are dispersed across the infrastructure, and the network exhibits some degree of contention. Large applications with a high degree of communication between its processes – the norm in high-performance computing – have the potential to benefit the most. For example, LU and SP experiment an average of 23.14% runtime speedup (considering all analyzed factors). Under network contention (*i.e.*, 26.82%), the gains are, on average, as high as 44.24%. To achieve these results our approach needs to be executed for the first packet of a new flow between two application processes, demanding negligible communication delays with the external SDN controller (less than a dozen milliseconds).

As future work, we intend to evaluate the effectiveness of the proposed approach to accelerate heavier, real HPC applications being weather forecasting a potential candidate. We also plan improvements to the proposed mechanism that may lead to even higher application runtime speedups. Examples include pre-provisioning paths leveraging potentially known (or expected) application communication patterns and determining the optimal period for flow rules to remain active.

Acknowledgements. This work was carried out in the context of the project CloudHPC – Harnessing Cloud Computing to Power Up HPC Applications, BRICS Pilot Call 2016. It was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq), Project Number 441892/2016-7, Call CNPq/MCTIC/BRICS-STI No 18/2016, as well as the National Key Cooperation between the BRICS Program of China (No. 2017YE0100500) and the Beijing Natural Science Foundation of China (No. 4172033).

References

1. Alsmadi, I., Khamaiseh, S., Xu, D.: Network parallelization in HPC clusters. In: International Conference on Computational Science and Computational Intelligence (CSCI 2016), pp. 584–589, December 2016. https://doi.org/10.1109/CSCI.2016.0116
2. Bailey, D.H.: NAS parallel benchmarks. In: Padua, D. (eds) Encyclopedia of Parallel Computing, pp. 1254–1259. Springer, Boston (2011). https://doi.org/10.1007/978-0-387-09766-4
3. Evangelinos, C., Hill, C.N.: Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on amazon's EC2. In: The 1st Workshop on Cloud Computing and its Applications (CCA) (2008)
4. Faizian, P., Mollah, M.A., Tong, Z., Yuan, X., Lang, M.: A comparative study of SDN and adaptive routing on dragonfly networks. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 51. ACM (2017)
5. Guo, Z., Liu, R., Xu, Y., Gushchin, A., Walid, A., Chao, H.J.: STAR: preventing flow-table overflow in software-defined networks. Comput. Netw. **125**, 15–25 (2017)
6. Gupta, A., et al.: Evaluating and improving the performance and scheduling of HPC applications in cloud. IEEE Trans. Cloud Comput. **4**(3), 307–321 (2016). https://doi.org/10.1109/TCC.2014.2339858
7. Joy, S., Nayak, A.: Improving flow completion time for short flows in datacenter networks. In: IFIP/IEEE International Symposium on Integrated Network Management (IM 2015), pp. 700–705, May 2015. https://doi.org/10.1109/INM.2015.7140358
8. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, p. 19. ACM (2010)
9. Lee, J., Tong, Z., Achalkar, K., Yuan, X., Lang, M.: Enhancing infiniband with openflow-style SDN capability. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC 2016, pp. 36:1–36:12. IEEE Press, Piscataway (2016). http://dl.acm.org/citation.cfm?id=3014904.3014953
10. Li, C., Zhang, J., Luo, Y.: Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm. J. Netw. Comput. Appl. **87**, 100–115 (2017)
11. Mauch, V., Kunze, M., Hillenbrand, M.: High performance cloud computing. Future Gener. Comput. Syst. **29**(6), 1408–1416 (2013)
12. Milojičić, D., Llorente, I.M., Montero, R.S.: Opennebula: a cloud management tool. IEEE Internet Comput. **15**(2), 11–14 (2011)
13. Netto, M.A.S., Calheiros, R.N., Rodrigues, E.R., Cunha, R.L.F., Buyya, R.: HPC cloud for scientific and business applications: taxonomy, vision, and research challenges. ACM Comput. Surv. **51**(1), 8:1–8:29 (2018). https://doi.org/10.1145/3150224
14. Ramakrishnan, L., Canon, R.S., Muriki, K., Sakrejda, I., Wright, N.J.: Evaluating interconnect and virtualization performance for high performance computing. ACM SIGMETRICS Perform. Eval. Rev. **40**(2), 55–60 (2012)

15. Roloff, E., Diener, M., Diaz Carreño, E., Gaspary, L.P., Navaux, P.O.A.: Leveraging cloud heterogeneity for cost-efficient execution of parallel applications. In: Rivera, F.F., Pena, T.F., Cabaleiro, J.C. (eds.) Euro-Par 2017. LNCS, vol. 10417, pp. 399–411. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64203-1_29
16. RYU: Ryu, a component-based software defined networking framework (2019). http://osrg.github.io/ryu/. Accessed 26 Jan 2019
17. Sefraoui, O., Aissaoui, M., Eleuldj, M.: Openstack: toward an open-source solution for cloud computing. Int. J. Comput. Appl. **55**(3), 38–42 (2012)
18. Walker, E.: Benchmarking amazon EC2 for high-performance scientific computing. Login: Mag. USENIX & SAGE **33**(5), 18–23 (2008)