Evaluation of an Interference-free Node Allocation Policy on Fat-tree Clusters

Samuel D. Pollard*, Nikhil Jain†, Stephen Herbein‡, Abhinav Bhatele†

*Department of Computer and Information Science, University of Oregon, Eugene, OR 97403 USA
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551 USA
‡Livermore Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551 USA
Email: spollard@cs.uoregon.edu, {nikhil, sherbein, bhatele}@llnl.gov

Abstract—Interference between jobs competing for network bandwidth on a fat-tree cluster can cause significant variability and degradation in performance. These performance issues can be mitigated or completely eliminated if the resource allocation policy takes the network topology into account when allocating nodes to jobs. We implement a fat-tree network topology aware node allocation policy that allocates isolated partitions to jobs in order to eliminate inter-job interference. We compare the impact of this node allocation policy to a topology-oblivious policy with respect to the execution time of individual jobs with different communication patterns. We also evaluate the cluster's quality of service using metrics such as system utilization, schedule makespan, and job wait time for both policies. The results obtained for production workloads indicate that a topology-aware node allocation can provide interference-free execution without negatively impacting the cluster's quality of service.

Index Terms—fat-tree, interference, node allocation

I. MOTIVATION

Traditional resource managers typically do not consider the network topology when allocating nodes to jobs. Since most supercomputing centers aim to maximize system utilization, when allocating nodes to a new job in the queue, all currently available nodes are considered as opposed to topologically isolated sets of nodes. However, ignoring the underlying network topology during the node allocation process leads to significant link sharing and inter-job interference because network resources are shared by all running jobs on most HPC systems [1], [2]. Inter-job interference can result in significant degradation and variability in performance. The degradation occurs when heavy traffic from some jobs causes congestion on certain ports/links in the network, and other jobs attempt to use these ports/links for communication [3], [4].

A dynamic job queue and runtime variations within each job further make performance degradations variable and unpredictable. The effect of performance variations is two-fold. First, arbitrary runtime variations not attributable to the code itself can increase the difficulty of pinpointing performance issues in a parallel program. Second, these variations reduce the accuracy of users' estimates of the duration of their jobs, which in turn causes additional strain on resources. For example, a user underestimating the required runtime of a job results in that job being killed by the job scheduler, thus forcing the experiment to be rerun [5].

Historical job queue data (including node allocation information) from a fat-tree cluster, Cab at Lawrence Livermore National Laboratory (LLNL), shows that the deployed resource manager allocates nodes all over the network to individual jobs and disregards the network topology. We can get an idea of the compactness (or inversely spread) of node allocations and in turn, expected inter-job interference by plotting the average number of hops messages within a job must travel on the network. This metric is formally defined in Section IV-C. Figure 1 shows the *average pairwise hops* for different jobs submitted to Cab over a two month period in 2014. We observe that average pairwise hops for most jobs are much higher than the minimum possible, which suggests that jobs are most likely placed in a scattered manner and suffer from inter-job interference.
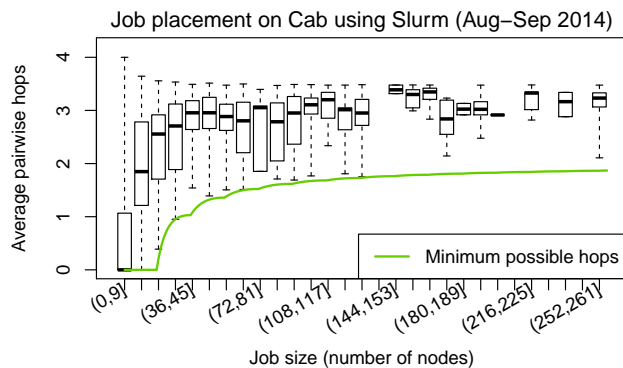

Fig. 1: Average hops for most jobs are significantly higher than the minimum possible, indicating scattered node allocations. We define one hop as a network link connecting two network ports. However, only links that can be shared by multiple jobs are considered, links connecting nodes to the switches are not counted. The maximum number of hops between nodes on Cab is four.

An elegant solution to eliminate inter-job interference is to allocate partitions (sets of nodes) to each job that are *isolated* i.e. do not share any links with other partitions [3], [6]. This has been implemented on IBM machines with torus networks where n-dimensional prisms can be allocated to a

job. Such solutions can eliminate inter-job interference and improve performance, but may increase system fragmentation and job wait times. In this paper, we explore the impact of a similar scheme for creating isolated job partitions on clusters with fat-tree networks.

Our previous work described a set of rules that a topology-aware node allocation policy can follow to eliminate inter-job interference on fat-tree networks [6]. However, that work, among others, does not compare topology-oblivious and topology-aware node allocation schemes with respect to their impact on a cluster's Quality of Service (QoS). We use three metrics to quantify a cluster's QoS: system utilization, schedule makespan, and job wait time (formally defined in Section IV-C). These metrics together represent the overall efficiency of a cluster and are important considerations for deploying a node allocation policy. The general perception is that topology-aware node allocation policies severely impact a fat-tree cluster's QoS and thus limits their adoption in practice.

In this paper, we build upon the rules proposed in [6] to implement a topology-aware node allocation policy for fat-tree clusters and compare its efficacy with a topology-oblivious scheme. In particular, we make the following contributions:

- We implement an interference-free node allocation policy in the Flux resource management framework.
- We compare the impact of our topology-aware node allocation policy with a topology-oblivious policy on the runtime of different types of jobs and overall cluster QoS.
- We identify the scenarios in which a topology-aware node allocation policy can be deployed in production without impacting a cluster's QoS in an effort to break the perception that such policies are not useful in practice.

II. BACKGROUND

In this section, we provide a brief description of the tools used in this study: the Flux resource manager, the Flux simulator, and the TraceR-CODES network simulator.

Flux resource manager: Flux is a relatively new, distributed resource management system for scheduling jobs and allocating resources to them on HPC machines. Flux provides versatile support for scheduling jobs centrally, hierarchically or in a distributed fashion, and supports making decisions based on many types of resources, including I/O and power [7], [8]. There are three components to job scheduling in Flux:

1. Ordering Policy: This creates an ordering of jobs in the queue according to a policy based on inputs such as user priority, historical usage, and resources requested.

2. Resource Selection Policy: This component determines if enough resources exist to satisfy a job's requirements, and then selects the exact resources (nodes) to allocate to or reserve for the job. This is where the node allocation policy resides.

3. Reservation Policy: This determines whether to allocate resources now or to reserve resources in the future for a given job. Example policies that operate on the ordering generated by the ordering policy are first-come, first-served (FCFS), conservative backfilling, and EASY backfilling. In

the FCFS scheme, scheduling stops at the first job for which resources are currently not available. When backfilling is enabled, resources can be tentatively reserved for jobs that cannot be currently executed and subsequent jobs can be scheduled if resources are available for them. In conservative backfilling, reservation is done for every job in the order they appear, while in EASY backfilling, the reservation is done only for one job (the first job that needs it) in the queue.

Our proposed work only affects the second component (resource selection), where we swap the topology-oblivious node allocation policy with a topology-aware scheme. This indicates that our work is compatible with existing job ordering and reservation policies. In our evaluations, we order jobs based on their submission time and schedule jobs with EASY backfilling as is common in many supercomputing centers [9].

Flux simulator: Flux also includes a simulation mode that enables evaluation of new policies without the need for a full system or real world execution. The simulation mode emulates the execution of job queue logs from production clusters by creating a virtual set up for the real world scenario. The Flux simulator is a discrete-event simulator, which natively supports job queue logs from the Slurm resource and job manager [8]. Simulation support in Flux is tightly integrated with the scheduler itself, which allows us to use the same code in both simulated and production environments.

TraceR-CODES network simulator: In addition to real-world tests, we also use packet-level network simulation capabilities provided by CODES [10] to study the impact of topology-aware node allocations on the performance of individual jobs with different communication patterns. The simulations are driven by TraceR [11], which can replay MPI traces collected using ScoreP [12] and Adaptive MPI [13]. The multi-job workload simulation feature of TraceR provides expected runtime for each job executed as part of the workload on a modeled fat-tree network.

III. DESIGN AND IMPLEMENTATION

We first describe the fat-tree topology, then the design and implementation of our interference-free topology-aware node allocation policy as a plugin in Flux.

A. Fat-tree and Inter-job Interference

Large cluster installations that deploy the fat-tree topology typically use a three-level fat-tree network. Hence, we use a three-level fat-tree (illustrated in Figure 2) to describe the topology-aware node allocation algorithm and its implementation. Fat-tree networks are typically built using commodity switches with a fixed radix (say r). Each leaf-level or level 1 (L1) switch has $\frac{r}{2}$ nodes connected to it. The remaining $\frac{r}{2}$ ports are used to connect to the second-level (L2) switches. Level 1 (L1) and level 2 (L2) switches are grouped together to form *pods*. Each pod has $\frac{r}{2}$ L1 and $\frac{r}{2}$ L2 switches forming a bi-partite all-to-all graph. Half of the ports on each L2 switch are connected to level 3 (L3) switches (also called *core switches*). These connections enable traffic to flow across pods. A cluster can have a maximum of r pods.
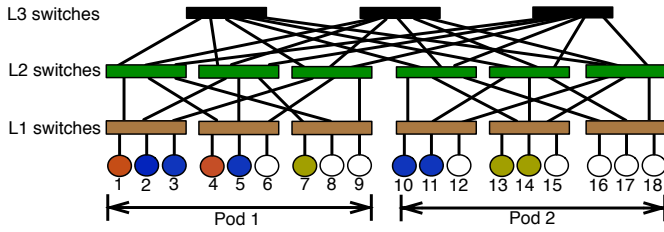
Fig. 2: A fat-tree with a switch radix of six. The numbers underneath the circles indicate node IDs and each color indicates a different job and its placement on certain nodes (three levels are used for illustration only; with radix-6 switches, 18 nodes can be connected using just two levels).

An important property of full bisection fat-tree installations is that each level of the fat-tree has the same amount of bandwidth. However, tapered installations are also possible in which the total bandwidth per level is reduced by using more ports to connect downward than upward. The scheme we describe works for both tapered and full bisection configurations.

Figure 2 demonstrates the potential for interference between jobs on a fat-tree network. We have three jobs in this example: red job on nodes 1 and 4; blue job on nodes 2, 3, 5, 10, and 11; and yellow job on nodes 7, 13, and 14. All other nodes are unoccupied and we assume that within each job, every node communicates with every other node. In this scenario, communication of the red job can interfere with that of the blue job because messages between node 1 and 4, and those between node 2 and 5 can potentially use the same link connecting L1 to L2 switches. This arises because routing policy on fat-trees is typically static and does not change with the job placement. Adaptive routing could potentially reduce link sharing but not eliminate it. Hence, links connecting L1–L2 switches are not dedicated to specific pairs of source-destination nodes connected to L1 switches. Similarly, communication from the blue job can also interfere with that of the yellow job even though those jobs do not share an L1 switch. This interference can happen at links connecting L2–L3 switches that enable communication between the two pods.

In order to avoid inter-job interference, a node allocation policy needs to address potential interference at two levels: links connecting L1–L2 switches and links connecting L2–L3 switches. Since individual nodes are not shared among jobs in most HPC clusters, the links connecting nodes to L1 switches cannot be shared by different jobs.

B. Design of an Interference-free Node Allocation Policy

In [6], we defined the following four rules to ensure that jobs do not share ports/links on a fat-tree network and thus do not interfere with one another:

1) A job can be fully allocated within a single leaf (L1) switch without interfering with other jobs.
2) A job can be fully allocated using all nodes of one or more switches in a single pod without interfering with other jobs. This avoids sharing of L1–L2 links because if different jobs do not share L1 switches, they use different L1–L2 links. Also note that such jobs will not use L2–L3 links.
3) A job can be fully allocated using all nodes in all switches of one or several pods without interfering with other jobs. This avoids link sharing because if different jobs do not share pods, they use different L1–L2 and L2–L3 links.
4) If nodes in one or more pods are allocated to multiple jobs and each job satisfies either condition (1) or (2), the rest of the nodes can be assigned to one single job without causing inter-job interference.

We build upon these rules that avoid L1–L2 and L2–L3 link sharing, and derive a practical way to optimize for job allocations. Assuming a full bisection fat-tree, we propose to divide incoming jobs into three categories based on their sizes (number of nodes requested):

- Type 1 (T1): A job that requests fewer than or equal to $r/2$ nodes; it can fit on a single leaf switch.
- Type 2 (T2): A job that requests up to $(r/2)^2$ nodes; it can fit in a single pod but spans multiple switches.
- Type 3 (T3): A job that requests more than $(r/2)^2$ nodes; it spans multiple pods.

For example, in Figure 2, a job requesting between one and three nodes is of type T1, a job requesting between four and nine nodes is of type T2, and one requesting between ten and eighteen nodes is of type T3. Below, we enumerate the node allocation policies for each type of job.

T1 job allocation: Based on Rule 1), we always allocate a T1 job to the nodes of a single L1 switch in order to avoid its interference with any other job. With this restriction, since a T1 job only uses node to L1 links, multiple T1 jobs can be allocated to the nodes of a single L1 switch and a T1 job can also share an L1 switch with other types of job without interfering with them.

T2 job allocation: Following Rule 2), we restrict the allocation of T2 jobs within a pod. This forces T2 jobs to only use node to L1 and L1–L2 links. Further, we never assign nodes of an L1 switch to more than one T2 job to avoid sharing of L1–L2 links. However, we do allow a T2 job to share an L1 switch with multiple T1 jobs as T1 jobs do not use L1–L2 links. Further, multiple T2 jobs can co-exist in a pod if they use nodes from different L1 switches since then those jobs are guaranteed to use different L1–L2 links.

T3 job allocation: As a T3 job is assigned nodes from multiple pods by definition, based on Rules 3 and 4, we cannot assign nodes of one pod to more than one T3 job. Since a T3 job can use links at all levels, we cannot assign a T3 job to the nodes of an L1 switch whose other nodes are assigned a T2 job. However, T3 and T2 jobs can co-exist in a pod if they use nodes from different L1 switches since those jobs will use different L1–L2 links. T3 jobs can also share an L1 switch with T1 jobs as T1 jobs only use node to L1 links.

An example of a valid, i.e. interference-free, allocation of jobs is shown in Figure 3. The colors represent the jobs and their types are listed in the figure. Note here that although the red and blue jobs share an L1 switch, they do not interfere
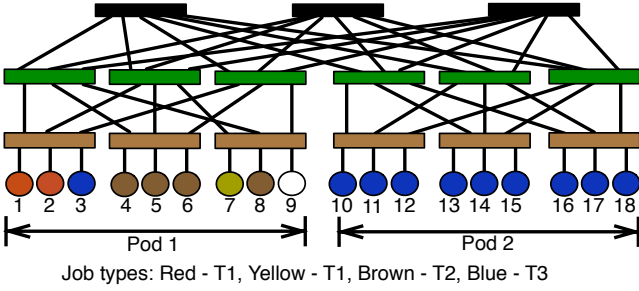
Fig. 3: A topology-aware node allocation on a cluster that eliminates inter-job interference on the links.

since the red job is of type T1 and thus only uses node to L1 links not used by the blue job. Similarly, although the brown and blue jobs share Pod 1, they do not interfere because the L1–L2 links these jobs can use are different: the brown job can only use links from the second and third switch of Pod 1 while the blue job can only use links from the first switch.

In contrast to Figure 3, in Figure 4, the allocation of jobs to nodes 2 and 7 has been swapped. This swapping leads to inter-job interference because now the red and brown jobs can contend for the L1–L2 links in Pod 1, and the red and blue jobs can also contend for the L1–L2 links in Pod 1.
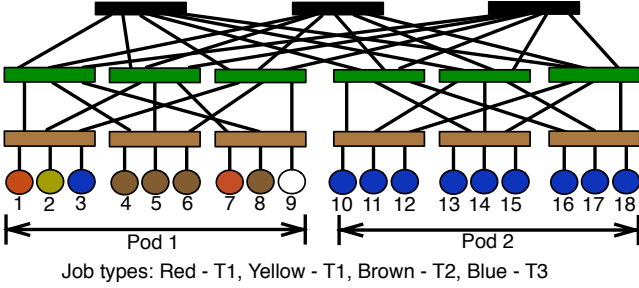

Fig. 4: Topology-oblivious node allocation for the example in Figure 3. Merely swapping the allocation of two nodes (node 2 and 7) can induce inter-job interference between the red and brown jobs, and red and blue jobs.

C. Implementation in Flux

We implement the policy described in the previous section in version 0.8.0 of the Flux framework (available at Github). Our plugin is also open source and available here. In this paper, we focus on the EASY backfilling policy for both topology-oblivious and topology-aware node allocation policies.

As mentioned in Section II, our proposed work only modifies the resource selection component of Flux for which the pseudo-code is presented in Algorithm 1. In this plugin, the following global state is maintained: the set of available nodes, type of jobs allocated to each node, and connectivity of nodes to L1 switches and pods. When the function to allocate nodes is invoked for a new job, the type of job (T1, T2, or T3) is determined based on the job's requested node count, and one of the three functions is invoked. In

---

Algorithm 1 Topology-aware node allocation

Input: job node count N, nodes per L1 switch k, pod size p
Output: allocated nodes S
Maintains: types of jobs allocated to nodes in the system (T1, T2, T3)

1: **function** RESOURCE_ALLOCATION(N, k, p)
2:    **if** $N \leq k$ **then**   ▷ T1 job
3:       **return** resource_allocate_small(N)
4:    **else if** $N \leq p$ **then**   ▷ T2 job
5:       **return** resource_allocate_medium(N)
6:    **else**   ▷ T3 job
7:       **return** resource_allocate_large(N)

8: **function** RESOURCE_ALLOCATE_SMALL(N)
9:    $S = \{\}$
10:   sort pods based on available nodes: least to most available
11:   **for** each pod in the sorted order **do**
12:      sort L1 switches based on available nodes: least to most available
13:      **for** each switch in the sorted order **do**
14:        **if** N nodes available in the switch **then**
15:           add first N available nodes to S
16:           mark type of job for nodes in S to T1
17:           **return** S
18:   **return** NULL

19: **function** RESOURCE_ALLOCATE_MEDIUM(N)
20:   $S = \{\}$
21:   sort pods based on available nodes: least to most available
22:   **for** each pod in the sorted order **do**
23:      sort L1 switches with no T2 and T3 jobs based on available nodes: most to least available
24:      **for** each switch in the sorted order **do**
25:        add first $N - |S|$ available nodes to S
26:        **if** $|S| == N$ **then**
27:           mark type of job for nodes in S to T2
28:           **return** S
29:      $S = \{\}$
30:   **return** NULL

31: **function** RESOURCE_ALLOCATE_LARGE(N)
32:   $S = \{\}$
33:   sort pods without T3 jobs based on available nodes: most to least available
34:   **for** each pod in the sorted order **do**
35:      sort L1 switches without T2 jobs based on available nodes: most to least available
36:      **for** each switch in the sorted order **do**
37:        add first $N - |S|$ available nodes to S
38:        **if** $|S| == N$ **then**
39:           mark type of job for nodes in S to T3
40:           **return** S
41:   **return** NULL

---

order to reduce fragmentation, we have developed heuristics to determine more preferred locations when there are multiple options for where to place a job on the fat-tree topology.

T1 jobs: T1 jobs have the most flexibility due to their small node count requests. So, we place them in pods and L1 switches with the fewest available nodes (lines 10–13).

T2 jobs: T2 jobs can be placed in any pod as long as they do not share an L1 switch with other T2 and T3 jobs. Thus, we prioritize pods with the fewest available nodes in which the T2 job can fit and not cause interference (line 21). Also, since T2 jobs interfere with other jobs if they share L1 switches, we wish to minimize the number of L1 switches they utilize. Thus, within a pod, we prioritize allocating nodes on switches with the most available nodes (line 23).
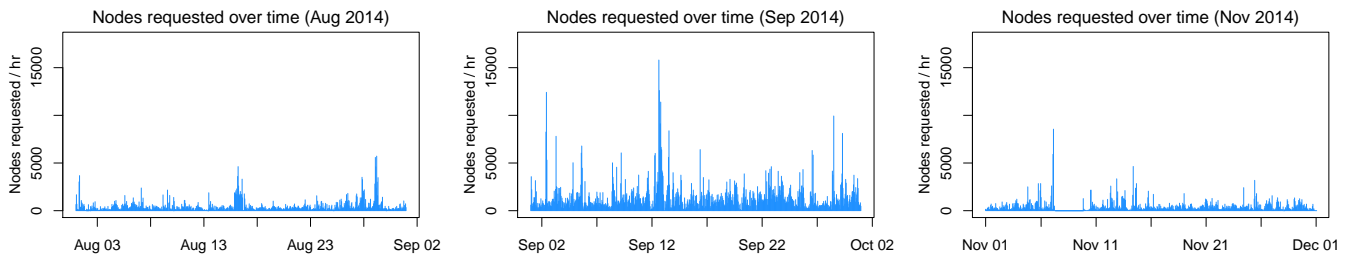
Fig. 5: Nodes requested in different workloads. Bars show the number of nodes requested in every hour during the one-month periods: significantly more nodes are requested in September. Distribution for other workloads can be found in Appendix A.
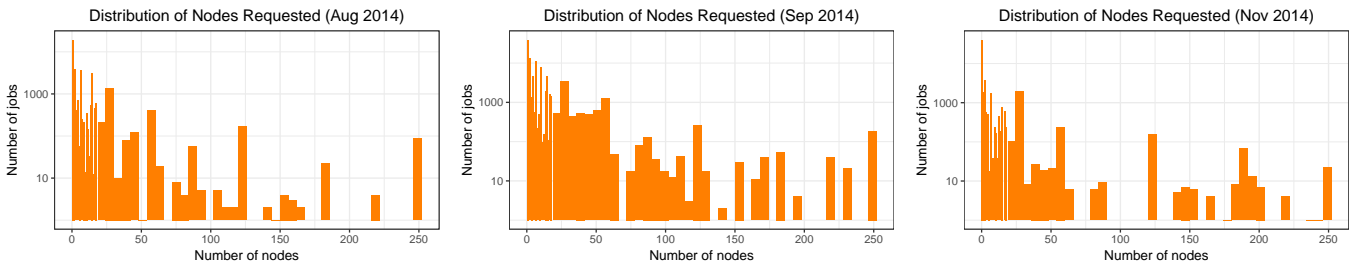

Fig. 6: Histograms showing the distribution of jobs based on the number of nodes requested. The bin size is one for jobs requesting 1–18 nodes and six elsewhere. Note the logarithmic scale on the y-axis (see Appendix A for other workloads).

T3 jobs: Because of their large node count, T3 jobs can cause the most interference. Thus, we prioritize placement on pods with the most available nodes (line 33). Within a pod, we prioritize the switches with the most available nodes (line 35). Note that, for all these selections, we only choose resources such that inter-job interference is guaranteed not to occur.

IV. EXPERIMENTAL SETUP

In order to evaluate the impact of topology-aware node allocation on the performance of individual jobs and the cluster's overall QoS, we run multi-job workloads on a production system and perform simulations, respectively (using the TraceR-CODES and Flux simulators). The Flux framework is ideal for our study because it allows us to use existing job logs from production clusters as input.

A. Machines Simulated

We simulate Cab, a production cluster at LLNL with 1,296 nodes. We obtained the precise network topology for Cab from the system administrators and provide it as an input to the Flux simulator. Cab is a three-level fat-tree with 36-port switches. At the leaf level, 18 nodes are connected to each L1 switch. 18 L1 switches connect to 18 L2 switches to construct a pod. There are four such pods, which are connected to each other by 36 L3 switches. We also simulate a similar hypothetical system, Pinot, which is also a three-level fat-tree with 36-port switches but has twice as many nodes and pods as Cab.

B. Description of Job Logs

We obtained three workloads (job logs) for Cab, one each from the months of August, September, and November in

2014. These logs contain information recorded by the resource manager about each submitted job including number of nodes requested, specific nodes allocated to the job, start time, end time, etc. The logs do not contain information about the application/code being run in the batch jobs or their communication patterns. Job logs from Cab represent a capacity-style workload in which jobs are limited to 256 nodes of the total 1,296, and runtime is limited to 24 hours for individual jobs.

Since the 256-node limit on job sizes on Cab is lower than the number of nodes in one pod (i.e. no T3 jobs), we create additional workloads derived from existing logs by doubling the number of nodes requested by every job. Aug-2x and Nov-2x represent workloads created by doubling the nodes requested by each job in August and November logs, respectively. We also create an Aug-3x workload by tripling the nodes requested by each job in August. As we will see next, September is unusually heavily loaded, and hence we only simulate jobs submitted in the first 15 days of September, and do not use it for creating any new workloads.

Figure 5 shows the total number of nodes requested per hour in August, September, and November. We observe that there are several local peaks for node requests, mostly due to the time of the day: more jobs are submitted during business hours. Further, we see that a much higher number of nodes were requested in September than in August and November. We note that the distributions for the 2x and 3x workloads are similar to their base months, with the height of each bar doubled or tripled (see Appendix A).

Figure 6 shows the distribution of job sizes for the same logs as above. We observe that a large number of jobs request very few nodes. Also note that in September, several more

mid-sized (25-100 nodes) jobs were submitted in comparison to August and November, but overall, the percentage of small-sized jobs (1-18 nodes) is much higher in September. Appendix A presents distributions for all other workloads.

In addition to comparing topology-oblivious and topology-aware node allocation policies for the job logs described above, we create additional scenarios for experimentation that speed up the jobs (reduce their execution time) by a certain percentage. Based on the results presented in Section V-B, we make an assumption that jobs requesting more than four nodes would have modest speedups due to the complete elimination of interference and compact allocations of nodes to jobs. Hence, for the topology-aware node allocation, we conduct five additional simulations that use shorter execution times according to different schemes. The first three scenarios reduce the execution times by $x\%$ for each job that requests more than four nodes ($x = 5, 10, 20$).

We also create two additional scenarios for speeding up the jobs – in these, the execution time reductions are a function of the job size and a randomized job categorization. In the first scenario, called V1, every job is randomly categorized into one of three speedup bins: 0–10%, 0–20%, and 0–30%. In the second scenario, called V2, jobs that request between 5–128 nodes are randomly categorized into speedup bins of 0–10% and 0–20%, and jobs that request more than 128 nodes are randomly categorized into speedup bins of 0–10%, 10–20%, or 10–30%. Within a speedup bin, the execution time for a job is reduced linearly based on its node count request, i.e., larger jobs have larger speedups. The reduction in execution time for all jobs that request ≥ 512 nodes is the upper range of the job's bin. These two scenarios help us reproduce real-world situations in which topology-aware node allocations improve the performance of different application codes differently including a dependence on the job size.

C. Metrics for Comparison

We compare topology-oblivious and topology-aware node allocation policies using various metrics. The first metric, average pairwise hops, is an estimate of the compactness of jobs under the two node allocation policies and is an indicator of the expected improvement in execution time of jobs due to reduced interference [6], [14]. The remaining metrics are used to measure the cluster's quality of service (QoS).

Average Pairwise Hops (APH) quantifies the compactness of nodes allocated to a job by calculating the number of links messages travel through between pairs of nodes in a job. Given a network topology T, we have an associated function H_T, which maps a pair of nodes to the number of inter-switch hops between them. For example, if nodes i and j are in the same pod but on different switches within the pod, then $H_T(i, j) = 2$. Specifically, we do not count hops between nodes and L1 switches because such hops do not contribute to inter-job interference. Given H_T, we compute the average across all ordered pairs of nodes in a job. Thus, for a job J

that is assigned n nodes, we calculate APH as:

$$APH(J) = \frac{\sum_{i \neq j} H_T(i, j)}{n \times (n - 1)}. \tag{1}$$

System Utilization is a straightforward metric expressing the percentage of nodes in a cluster allocated for running jobs at a given time. In our evaluation, we record utilization every minute of simulation time. Specifically, it is defined as:

$$utilization_t = \frac{N_t}{N} \tag{2}$$

where N_t is the number of nodes allocated at time t and N is the total number of nodes in the cluster. Note that the utilization of a cluster can be low for two reasons: 1. the job scheduling algorithm is not able to find suitable nodes for pending jobs, or 2. not enough jobs are submitted to fill the cluster's capacity.

Schedule Makespan is the time interval from when the first job is submitted to the completion of the last job. This is a way to quantify the compactness and efficiency of a job schedule; the job scheduler usually aims to run a given number of experiments with a minimal makespan. This can be a challenging metric for real-world logs since in the real-world, jobs are submitted continually over the entire time period contained in the logs. For example, in a month-long job queue log, there will be jobs submitted on day 30. This means our makespan measurements can show limited improvement for a given log regardless of how efficiently jobs on the first 29 days are scheduled. When a significantly large number of nodes are requested, this metric reflects the efficiency of a scheme.

Job Wait Time measures the interval between when a job is submitted and when it starts running. It is defined as:

$$T_{\text{wait}} = T_{\text{start}} - T_{\text{submit}} \tag{3}$$

where T_{start} is the time when the job begins execution and T_{submit} is the time when the job is submitted to the scheduler. The lower the wait time, the better it is for the users.

Utilization and job wait time can be correlated. For instance, increasing utilization typically decreases wait time. There also exists a potential trade-off between these metrics and APH. On the one hand, delaying the execution of a job to wait for an interference-free allocation (i.e., lower APH) can increase the job's wait time. On the other hand, lower APH allocations can reduce network interference experienced by the application, thereby reducing its execution time. If the job speedups outweigh the increased wait time, then the job's and workload's makespan will improve.

V. EVALUATION RESULTS

We now evaluate the impact of the previously described interference-free node allocation policy on average pairwise hops, execution time of common HPC communication patterns, and metrics defined in Section IV-C that represent the QoS of a cluster.
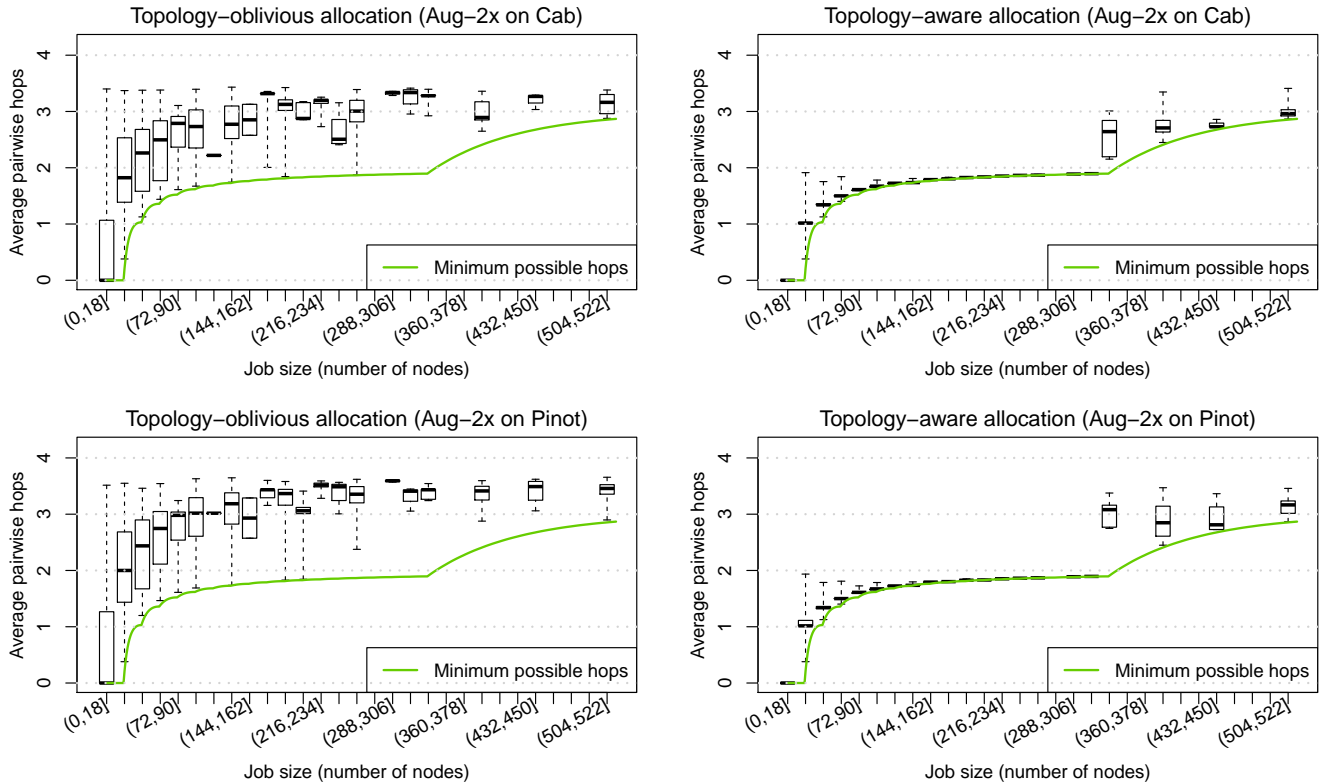
Fig. 7: Average pairwise hops for the Aug-2x logs on Cab (top row) and Pinot (bottom row). The topology-aware node allocation policy significantly reduces the average hops, resulting in fewer links on which jobs can interfere on both systems.

A. Reduction in Network Hops

We group jobs into bins based on the number of nodes requested and present the distribution of APH for all jobs in each bin using bars and whiskers. Figure 7 presents representative results for the Aug-2x workload simulated on Cab and Pinot. For each bin, the five data points shown are the minimum, 25th percentile, median, 75th percentile, and the maximum APH of the distribution for the jobs in that bin. We observed similar results for all other workloads that were simulated, and thus only these representative results are shown.

On both systems, we observe that the topology-aware node allocation policy is able to reduce APH for jobs with up to 324 nodes significantly. The maximum APH is less than two for all jobs with less than 128 nodes, and for jobs with 128–324 nodes, the maximum APH is close to the minimum possible hops. In contrast, the median APH for the topology-oblivious policy is higher than the ideal APH. This indicates that for T1 and T2 jobs that can be fit within a pod, the topology-aware policy finds compact allocations while the topology-oblivious policy spreads them. We also note that for the topology-oblivious policy, the APH for these job sizes is typically higher for the larger system (Pinot). This indicates that the jobs are more spread out on the larger system, and thus are more likely to suffer from inter-job interference.

For T3 jobs that span multiple pods (>324 nodes), the APH is sometimes higher than the ideal APH for the topology-aware

scheme because nodes for such jobs may not be compacted in the best possible manner across multiple pods if interference-free execution is guaranteed. Nonetheless, the median APH of jobs in the topology-aware policy is lower than that in the topology-oblivious policy, and gets closer to the ideal APH as the node count increases.

B. Impact on Execution Time of Individual Jobs

Previous work has shown that the execution time of production applications reduces when interference-free and/or topology-aware node allocations are used on a fat-tree network [4], [15], [16]. In order to quantify the impact of our interference-free topology-aware node allocations on the execution time of individual jobs, we run TraceR-CODES network simulations as well as real-world multi-job executions on Cab. For these comparisons, we use proxy applications with four different HPC communication patterns from the Chatter-bug communication proxy application suite [17]: structured near-neighbor communication on a 4D process grid (Stencil), unstructured near-neighbor communication on a 3D process grid (Unstr-mesh), random-pairs communication (Pairs), and sub-communicator based all-to-all communication on a 3D process grid (Sub-a2a).

Simulation results: For the network simulations, we model a three-level fat-tree system that consists of 10,648 nodes connected using radix-44 switches arranged in 22 pods each
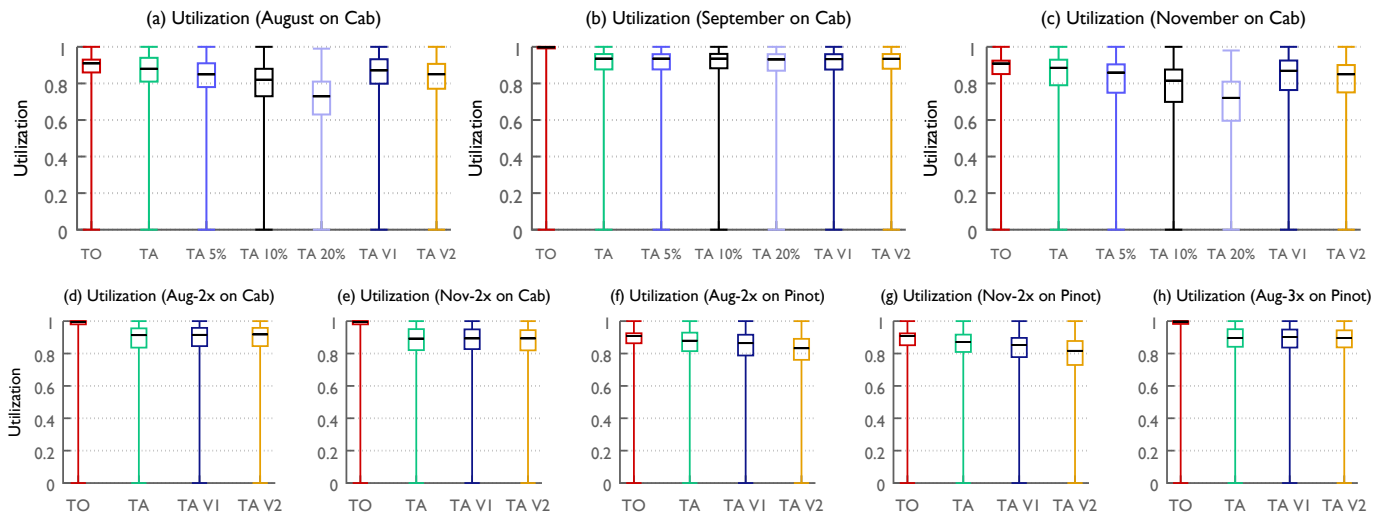
Fig. 8: Comparison of system utilization – a 10% drop is observed because of topology-aware node allocation when the system is overloaded (in (b), (d), (e), and (h)). In other cases, the drop in utilization is due to the lack of jobs available for execution.

with 22 switches. We simulate 15 randomly generated multi-job workloads, each of which consists of jobs running on 512–4096 nodes (5-40% of the system size). Each job in these multi-job workloads is randomly assigned one of the four communication patterns. For jobs performing structured and unstructured near-neighbor communication, our multi-job workload simulations predict an average speedup of $1.58\times$ and $1.68\times$ respectively when using an interference-free node allocation. The average speedup obtained for Pairs is only $1.06\times$, the lowest among the four patterns. For Sub-a2a, with all-to-all performed along two dimensions, the speedup is $1.14\times$. The average speedup obtained over all jobs is $1.36\times$.

Experimental results: We compare the performance impact of topology-oblivious and topology-aware node allocations on a production system, Cab, with 1,296 nodes under a full-system reservation. The workloads for these experiments are generated by randomly packing jobs of sizes between 4 and 512 nodes on the entire system. Each job is assigned one of the four communication patterns listed above, which it repeats in a `for` loop. The input parameters are configured so that each jobs runs for approximately 15 minutes and reports the time taken per iteration. The same set of workloads are run with both topology-aware and topology-oblivious policies in order to compare their performance.

When jobs of all sizes (4–512 nodes) are allowed, a typical workload generated consists of ~20 jobs. In this case, we find that the average speedups for Stencil and Unstr-mesh are $1.26\times$ and $1.14\times$, with maximum speedups of $1.5\times$ and $1.4\times$ respectively. For Pairs, the mean speedup is $1.1\times$, but we also observe performance degradation in some cases. This is because static routing on fat-tree can adversely impact performance for some node allocations, especially when the communication pattern is sparse. For Sub-a2a, in which all-to-all is performed along all dimensions, we observe a mean speedup of $1.55\times$. Overall, the average speedup over all jobs is

$1.3\times$. These empirical results come close to what we observe in the simulations above despite the relatively smaller size of Cab and smaller node counts for individual jobs.

If we limit the size of jobs to be between 4 and 128 nodes (so that more jobs execute simultaneously as is the common case), we obtain a workload with ~36 jobs. In this case, the average speedup over all jobs is $1.35\times$. The mean speedups of individual patterns are as follows: Stencil – $1.32\times$, Unstr-mesh – $1.29\times$, Pairs – $1.08\times$, and Sub-a2a – $1.74\times$. For the same workload, when we add idle computation to the jobs such that the communication time is expected to be around 30% of the total time in interference-free scenarios, the mean and maximum speedups obtained are $1.08\times$ and $1.31\times$.

In summary, we find that the performance improvements on a production system vary significantly not only with the type of communication pattern, but also with the specific node allocation policy chosen for a workload. However, the expected gains predicted by the simulations and experiments are similar and significant.

C. Impact on System Utilization

Next, we compare the two node allocation policies in terms of their impact on various QoS metrics defined in Section IV-C. Figure 8 compares the topology-oblivious (TO) and topology-aware (TA) allocation policies in terms of the system utilization for several combinations of datasets and systems. Utilization is calculated every minute, and box and whisker plots are used to show the distribution over the entire simulation. In four scenarios, (a) August on Cab, (c) November on Cab, (f) Aug-2x on Pinot, and (g) Nov-2x on Pinot, the utilization for TO and TA are similar. These are the scenarios in which the system is not overloaded with jobs (as indicated by <100% utilization for the topology-oblivious scheme). As a result, even the topology-aware scheme is able to find the specific nodes it needs for incoming jobs and maintain high utilization. In these scenarios, as we introduce speedups for
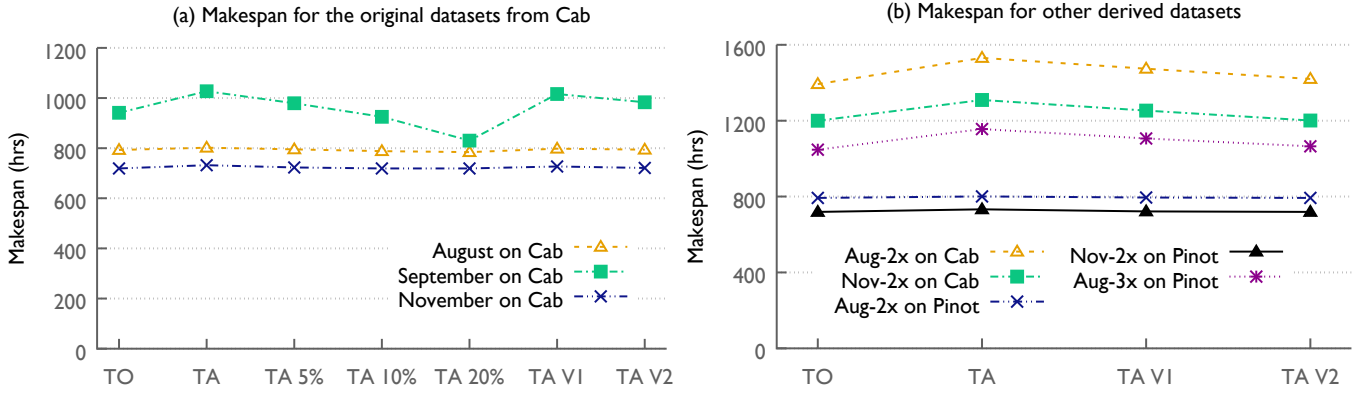
Fig. 9: The topology-aware node allocation increases the makespan when the system is overloaded. However, when job speedups due to better allocations are considered, the makespan is similar to the topology-oblivious scheme in most scenarios.
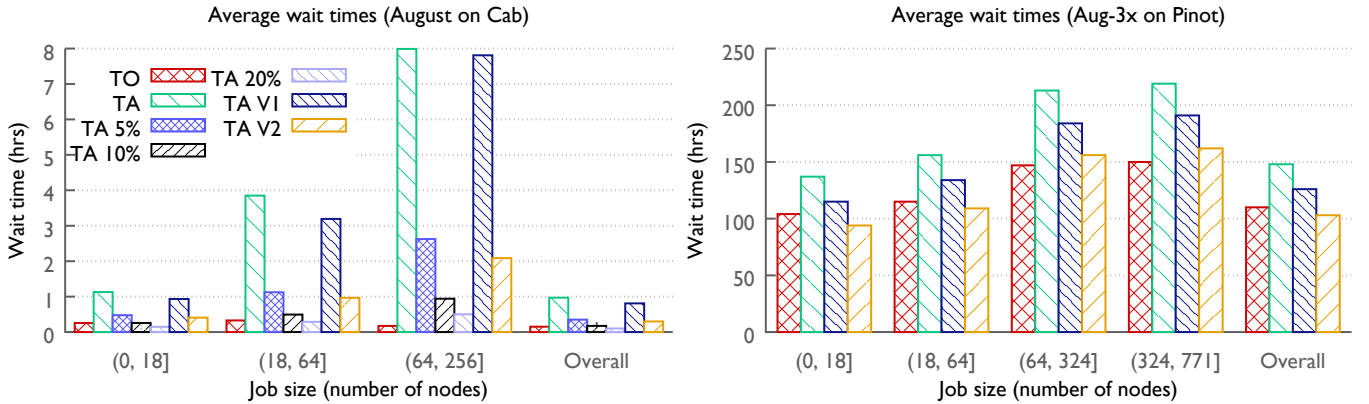

Fig. 10: Breakdown of the wait times by job size for two representative scenarios: (a) August on Cab, and (b) Aug-3x on Pinot. With job speedups, the wait times for jobs are not significantly higher than the topology-oblivious scheme.

jobs in TA 5%, TA V1, etc., the utilization begins to drop as the jobs begin to complete sooner, and sufficiently large number of jobs are not available to maintain high utilization.

In the remaining scenarios with heavily loaded job queues, (b) September on Cab, (d) Aug-2x on Cab, (e) Nov-2x on Cab, and (h) Aug-3x on Pinot, the topology-aware node allocation scheme results in a ~10% drop in the system utilization. In these scenarios, even though more jobs are available for scheduling, the requirements posed by the topology-aware scheme prevent newer jobs from being allocated nodes, and leave a fraction of the system unused. Further, speeding up individual jobs does not impact the utilization as several jobs are waiting to be queued even when some jobs finish early.

D. Impact on Schedule Makespan

Figure 9 compares the makespan for the two node allocation policies. As expected, for scenarios with lighter load such as August on Cab, the makespan is close to the length of a month (720-800 hours) for both policies across different simulation setups. For heavily loaded scenarios such as September on Cab, the makespan increases by up to ~9% for TA if individual job speedups are not considered. However as job speedups are introduced, we find that the makespan improves for the TA-

based scenarios and is close to the makespan of the topology-oblivious scheme in most cases. The most notable exception is the half-month simulation of September on Cab, where even TA V1 and TA V2 provide 9% and 4% longer makespan. We suspect this is because 87% of jobs in September request less than 18 nodes, and thus are expected to have minimal speedups for TA V1 and TA V2. In contrast, for other workloads, less than 80% jobs request fewer than 18 nodes.

E. Impact on Job Wait Time

Job wait time is an important QoS metric from the point of view of the end user. Figure 10 shows a breakdown of the wait times by job size for two representative scenarios: August on Cab and Aug-3x on Pinot. All but the last cluster show wait times for different job sizes, and the last cluster shows the overall wait times. It is interesting to note that depending on the system load, the wait time can be as low as a few minutes (in August) and as high as hundreds of hours (in Aug-3x). We observe that, on average, jobs have to wait an additional 45 minutes in the August dataset and 38 hours in the Aug-3x dataset when the topology-aware node allocation is used as opposed to the topology-oblivious node allocation. However,

Policy	TO	TA	TA V1	TA V2
Aug on Cab	.15	.98	.8	.3
Sept on Cab	295	330	327	307
Nov on Cab	.01	.41	.31	.10
Aug-2x on Cab	269	322	296	270
Nov-2x on Cab	240	300	266	240
Aug-2x on Pinot	.15	1.2	.65	.29
Nov-2x on Pinot	.01	.69	.13	.04
Aug-3x on Pinot	110	148	126	103

TABLE I: Comparison of mean wait times (in hours) for different workload and system combinations.

as job speedups are introduced, the mean wait times for both the scenarios reduce significantly, in particular for TA V2.

If we look at the breakdown by job size, we observe that the trends for small-sized jobs (first cluster in the plots) are similar to the overall trends. This is because small node jobs constitute the majority of jobs in the system. The wait time increases significantly with job size for the topology-aware node allocation assuming no speedups. For example 65-256 node jobs have to wait for almost eight hours with the topology-aware node allocation as opposed to 10 minutes with the topology-oblivious policy (for the August dataset). However, for the 10% and 20% speedup scenarios, the wait times are comparable or even smaller with the topology-aware node allocation. When jobs speed up by 20%, 65-256 node jobs in the August dataset only wait for 30 minutes in the queue on average. With TA V2, the wait times obtained are similar to the scenario with 10% speedup.

Table I presents the mean wait times over all jobs for all datasets. As noted previously, we find that in comparison to the topology-oblivious scheme, the wait times are higher for the topology-aware scheme. However, the job speedups that are expected due to the use of the topology-aware scheme have a significant impact on these wait times. For all scenarios with wait time of more than an hour, the wait time for TA V2 is similar to the topology-oblivious scheme. For other scenarios, the wait time increase is reasonable with the mean wait time increasing to only 18 minutes in the worst case.

VI. RELATED WORK

With communication on shared interconnection networks becoming a common performance bottleneck, there is much recent work on optimizing job schedulers for specific topologies, including 3D and 5D torus networks [18], [19], [20]. One common approach used by schedulers on tori networks is to enforce convexity of all allocations. This ensures that each job has exclusive access to the network links between its nodes but causes global system fragmentation. Another approach is to linearize the n-D torus of nodes using a space-filling curve, on which the scheduler then performs a contiguous allocation [21]. This approach does not guarantee an interference-free allocation, but the allocated nodes are physically close to one another (i.e., high locality). Slurm provides this approach for 3D tori topologies via the "topology/3d_torus" plugin [22]. As these approaches leverage unique features of tori, they are not applicable to hierarchical networks such as fat-trees.

Several job schedulers do support optimizations for fat-tree topologies. The Slurm workload manager provides a plugin for fat-tree topologies denoted "topology/tree." Its general strategy is to find the smallest sub-tree in which the request can be satisfied, then select the nodes within that tree using a best-fit algorithm. Specifically, best-fit allocates nodes subject to minimizing the number of free nodes [22]. Jokanovic et al. propose "quiet neighborhoods," which are virtual blocks of nodes designed to minimize fragmentation among jobs. For example, since many jobs request powers-of-two nodes, the scheduler can prevent 16-node partitions from being fragmented between multiple jobs [3]. An approach by Soner and Özturan, AUC-SCHED3, builds on an auctioning strategy which prioritizes contiguous allocations within the smallest sub-tree possible but requires solving an expensive integer programming at each scheduling interval [23]. Subramoni et al. propose a technique that collects dynamic congestion data and schedules jobs to minimize congestion on an InfiniBand cluster [24]. This differs from our approach because an additional service must be run to measure congestion while our scheduler relies on the static configuration. Despite being topology-aware, these job scheduling and node allocation techniques still permit situations where jobs may interfere with one another.

Considerable research also exists in exploring topology-aware mapping of tasks to nodes within a node allocation [25], [26], [27]. Routing Algorithm-aware Hierarchical Task Mapping (RAHTM) maps tasks using not only communication graphs and network topology but also knowledge of the routing algorithm [28]. These techniques help ensure better application performance within a given resource allocation, but they were not designed to prevent a topologically poor resource selection.

VII. CONCLUSION

We have presented an implementation of a topology-aware node allocation policy in the Flux resource manager for clusters with a fat-tree network. The topology-aware policy allocates isolated partitions to jobs for eliminating inter-job interference completely. We showed that QoS metrics such as schedule makespan and job wait time increase under this new allocation policy because jobs might have to wait longer for the right set of nodes. However, we also found that such a scheme results in better execution time for proxy applications with common HPC communication patterns. Assuming modest speedups based on these results, we found that the negative impact of the proposed topology-aware policy on both makespan and job wait times is neutralized. Overall, our results suggest that the topology-aware policy could provide better QoS if, in comparison to the topology-oblivious policy, the average execution time of individual jobs reduces by more than 10%.

References

[1] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013, LLNL-CONF-635776.

[2] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch Out for the Bully! Job Interference Study on Dragonfly Network," in *Supercomputing 2016 (SC'16)*, Salt Lake City, UT, November 13-18 2016.

[3] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, "Quiet neighborhoods: Key to protect job performance predictability," in *International Parallel and Distributed Processing Symposium*, May 2015.

[4] N. Jain, A. Bhatele, L. Howell, D. Böhme, I. Karlin, E. Leon, M. Mubarak, N. Wolfe, T. Gamblin, and M. Leininger, "Predicting the performance impact of different fat-tree configurations," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. IEEE Computer Society, Nov. 2017, lLNL-CONF-736289. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126967

[5] Y. Fan, P. Rich, W. E. Allcock, M. E. Papka, and Z. Lan, "Trade-off between prediction accuracy and underestimation rate in job runtime estimates," in *2017 IEEE International Conference on Cluster Computing*, ser. Cluster '17. IEEE, sep 2017. [Online]. Available: https://doi.org/10.1109%2Fcluster.2017.11

[6] N. Jain, A. Bhatele, X. Ni, T. Gamblin, and L. V. Kale, "Partitioning low-diameter networks to eliminate inter-job interference," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '17. IEEE Computer Society, May 2017, lLNL-CONF-706801.

[7] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A next-generation resource management framework for Large hpc centers," in *Proceedings of the 10th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, September 2014.

[8] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[9] D. A. Lifka, "The ANL/IBM SP scheduling system," in *Job Scheduling Strategies for Parallel Processing*. Springer Berlin Heidelberg, 1995, pp. 295–303. [Online]. Available: https://doi.org/10.1007%2F3-540-60153-8_35

[10] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Enabling parallel simulation of large-scale HPC network systems," *IEEE Trans. Parallel Distrib. Syst.*, 2016.

[11] N. Jain, A. Bhatele, S. T. White, T. Gamblin, and L. V. Kale, "Evaluating HPC networks via simulation of parallel workloads," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Computer Society, Nov. 2016, LLNL-CONF-690662.

[12] "Score-p user manual," 2015. [Online]. Available: https://silc.zih.tu-dresden.de/scorep-current/pdf/scorep.pdf

[13] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance Evaluation of Adaptive MPI," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[14] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 75–84.

[15] A. Jokanovic, G. Rodriguez, J. C. Sancho, and J. Labarta, "Impact of Inter-application Contention in Current and Future HPC Systems," in *18th Annual Symposium on High Performance Interconnects (HOTI)*, Mountain View, CA, USA, August 18-20 2010.

[16] G. Michelogiannakis, K. Ibrahim, J. Shalf, John anWilke, S. Knight, and J. Kenny, "Aphid: Hierarchical task placement to enable a tapered fat tree topology for lower power and cost in hpc networks," *CCGrid 2017 (to appear)*, 2017.

[17] N. Jain and A. Bhatele, "Chatterbug communication proxy applications suite," https://github.com/LLNL/chatterbug, LLNL-CODE-756471.

[18] K. Li, M. Malawski, and J. Nabrzyski, "Topology-aware job allocation in 3d torus-based hpc systems with hard job priority constraints," *Procedia Computer Science*, vol. 108, pp. 515 – 524, 2017, international Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050917305100

[19] Z. Zhou, X. Yang, Z. Lan, P. Rich, W. Tang, V. Morozov, and N. Desai, "Improving batch scheduling on blue gene/q by relaxing 5d torus network allocation constraints," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 439–448.

[20] J. Enos, G. Bauer, R. Brunner, and S. Islam, "Topology-aware job scheduling strategies for torus networks," in *Proceedings of the Cray User Group*, 2014.

[21] J. A. Pascual, J. A. Lozano, and J. Miguel-Alonso, "Analyzing the performance of allocation strategies based on space-filling curves," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2017, pp. 232–251.

[22] SchedMD LLC, "Topology guide," may. [Online]. Available: https://slurm.schedmd.com/topology.html

[23] S. Soner and C. Özturan, "Topologically aware job scheduling for SLURM," Tech. Rep., 2014. [Online]. Available: http://www.prace-ri.eu/IMG/pdf/WP180.pdf

[24] H. Subramoni, D. Bureddy, K. Kandalla, K. Schulz, B. Barth, J. Perkins, M. Arnold, and D. K. Panda, "Design of network topology aware scheduling services for large infiniband clusters," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–8.

[25] A. Bhatele, "Automating topology aware mapping for supercomputers," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, Aug. 2010. [Online]. Available: http://hdl.handle.net/2142/16578

[26] A. Bhatele, N. Jain, K. E. Isaacs, R. Buch, T. Gamblin, S. H. Langer, and L. V. Kale, "Improving application performance via task mapping on IBM Blue Gene/Q," in *Proceedings of IEEE International Conference on High Performance Computing*, ser. HiPC '14. IEEE Computer Society, Dec. 2014, LLNL-CONF-655465.

[27] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still, "Mapping applications with collectives over sub-communicators on torus networks," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society, Nov. 2012, LLNL-CONF-556491.

[28] A. Abdel-Gawad, M. Thottethodi, and A. Bhatele, "RAHTM: Routing-algorithm aware hierarchical task mapping," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Computer Society, Nov. 2014, l,LNL-CONF-653568. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SC.2014.32
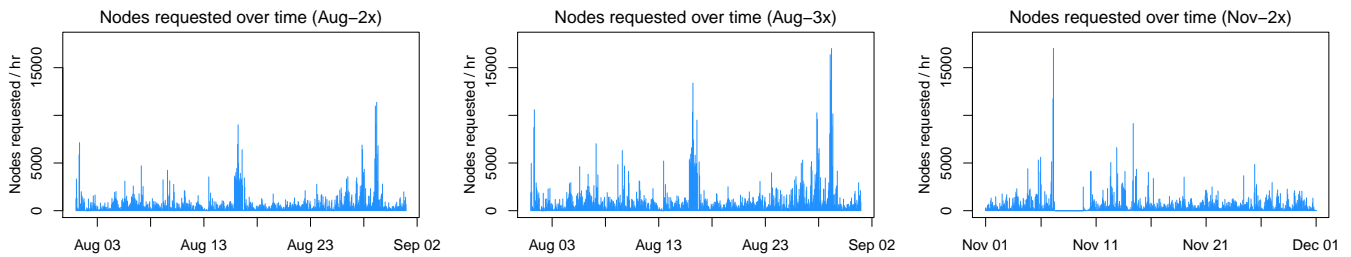
Fig. 11: Nodes requested in different workloads. Bars show the number of nodes requested in every hour during the one-month periods.
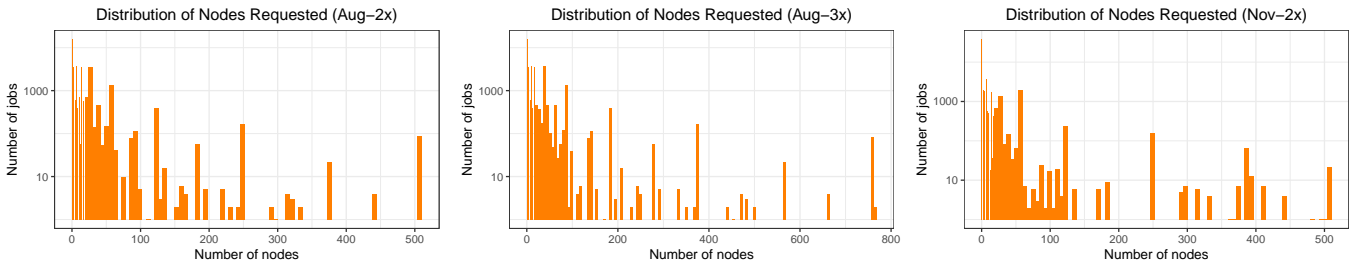

Fig. 12: Histograms of the number of jobs binned based on the number of nodes requested. The bin size is one for jobs requesting 1–18 nodes and six elsewhere.

APPENDIX

A. Additional Data on Job Traces

Figure 11 shows the total number of nodes requested per hour in the Aug-2x, Aug-3x, and Nov-2x logs. Figure 12 shows the distribution of job sizes for the same logs as above.

B. Artifact Description: [Evaluation of an Interference-free Node Allocation Policy on Fat-tree Clusters]

1) Abstract: In this work, job scheduling has been simulated for job queue logs collected from Cab using the open source Flux framework. These logs can be obtained by contacting the authors. The new interference-free topology-aware job allocation policy has been implemented in Flux and can be downloaded using the link below. The network simulation results for analyzing the impact of topology-aware node allocations on specific communication patterns are obtained using the production version of the TraceR-CODES simulation framework. The benchmarks used in these simulations are available publicly in the Chatterbug communication proxy suite.

2) Check-list:
- **Program: Flux, TraceR-CODES**
- **Compilation: using default options; compilation options do not affect the results.**
- **Data set: job queue logs from August, September, and November 2014 for system Cab.**
- **Run-time environment: single node Linux; does not affect the results.**
- **Hardware: Intel Xeon node; does not affect the results.**
- **Output: standard output from the Flux simulator and TraceR.**
- **Experiment workflow: simulate job queue logs using default Flux options; simulate communication traces using TraceR.**
- **Experiment customization: none.**

- **Publicly available: yes.**

3) How software can be obtained: The Flux plugin developed as part of this work is available in a public git repository. For network simulations, we used the current production version of TraceR-CODES. Here are the links to these software:

- https://github.com/flux-framework/flux-core
- https://github.com/flux-framework/flux-sched
- https://github.com/sampollard/flux-sched/tree/topo
- https://github.com/LLNL/chatterbug
- https://github.com/LLNL/tracer
- https://xgitlab.cels.anl.gov/codes/codes

4) Hardware dependencies: None.

5) Software dependencies: Flux-core, Flux-sched, TraceR, CODES, ROSS, MPI.

6) Installation: Standard installation process described in the documentation of Flux, ROSS, CODES, and TraceR have been followed.

7) Experiment workflow: The job queue log simulations in this paper follow a two step workflow:

- Obtain job queue logs for specific time frame by contacting the HPC facility at Lawrence Livermore National Laboratory, where Cab is hosted.
- For different node allocation policies, simulate the job scheduling using the Flux simulator for the network configuration of system Cab.

The network simulations performed in this paper also follow a two step workflow:

- Collect traces for the available communication-proxy codes that are used in the multi-job workload.

- Generate job-mapping using the Flux simulator and predict runtime using TraceR.

8) Evaluation and expected result: Most of the results in this paper are based on the job properties output from the Flux simulations: job start time, allocated nodes, wait time, etc. Other results are based on the timing output from the simulations performed using TraceR. The users should be able to use the released logs/traces, and system configuration files to reproduce them.