Mitigating Inter-Job Interference via Process-Level Quality-of-Service

Lee Savoie and David K. Lowenthal
Deptartment of Computer Science, The University of Arizona

Bronis R. de Supinski and Kathryn Mohror
Lawrence Livermore National Laboratory

Nikhil Jain
Nvidia Corporation

***Abstract*—Jobs on most high-performance computing (HPC) systems share the network with other concurrently executing jobs. Network sharing leads to contention that can severely degrade performance. This paper investigates the use of Quality of Service (QoS) mechanisms to reduce the negative impacts of network contention. QoS allows users to reduce resource sharing between network flows and to provide bandwidth guarantees to specific flows. Our results show that careful use of QoS reduces the impact of network contention for specific jobs, resulting in up to a 40% performance improvement. In some cases the impact of contention is completely eliminated. These improvements are achieved with limited negative impact to other jobs; any job that experiences performance loss typically degrades less than 5%, often much less. Our approach can help ensure that HPC machines maintain high levels of throughput as per-node compute power continues to increase faster than network bandwidth.**

I. Introduction

High performance computing (HPC) systems typically use thousands of nodes to execute multiple jobs concurrently. HPC installations use *space-shared* scheduling in which each node is assigned to at most one job at any time. This strategy prevents contention for per-node resources such as cores or memory. However, on most HPC systems, the interconnect is shared among all jobs and can become a severely contended resource, regardless of the topology, including fat trees [1], dragonfly networks [1], [2], [3], and tori [4].

Processes that execute on different nodes create contention by concurrently sending messages that share network links and, thus, their bandwidth. Network contention increases job communication time, which degrades performance. Since the degradation depends on other running jobs, it is different every time a job runs, which interferes with accurately estimating the job's run time. Thus, schedulers that use run time estimates to improve efficiency will likely make suboptimal scheduling decisions due to network contention.

We explore using Quality of Service (QoS) to manage network contention. Many modern networks provide QoS capabilities, including popular HPC network fabrics such as InfiniBand. Although QoS has been used to prioritize traffic in other contexts [5], [6], [7], [8], it is not commonly used on HPC systems. By intelligently allocating bandwidth between jobs to reduce network contention, QoS could improve job performance and, thus, HPC system throughput.

This paper presents the design, implementation, and evaluation of MPI process-based QoS prioritization to reduce contention on fat-tree networks. We use an iterative, feedback-directed approach that takes input from all concurrent jobs and prioritizes MPI processes to improve performance. The algorithm generally ensures that processes that receive an increase in priority have a limited impact on the performance of other processes from the same or other jobs. Our technique handles job arrivals and departures as well as jobs with changing computation and/or communication patterns. Our algorithm only requires knowledge of per-process times per job timestep, which are easily determined if the application identifies the timesteps. With this assumption, our algorithm can easily be integrated into a global run-time layer.

We have implemented our system, called TraceR-QoS, using the TraceR network simulator [9]. Our results show that it reduces or eliminates the impact of contention for several different workloads and job placements. This improvement has relatively little cost to other concurrently executing jobs.

The contributions of this paper are as follows:

- A demonstration that QoS can selectively prioritize MPI processes to improve performance of individual jobs with limited impact on other jobs;
- An algorithm that transparently and effectively applies QoS with little application information;
- Its open-source implementation, TraceR-QoS; and
- A thorough evaluation that demonstrates TraceR-QoS improves job performance up to 40%.

In some cases, TraceR-QoS completely eliminates the impact of network contention on a job (i.e., that job executes as if it were completely isolated). Further, TraceR-QoS rarely degrades the performance of any job more than 5%.

The rest of this paper is organized as follows. Section II discusses the problem of network contention and provides background on QoS and fat-tree networks. Section III describes our per-process service level assignment algorithm, and Section IV presents the new TraceR-QoS simulator framework in which we explore its performance. Section V details our experimental setup, while Section VI presents our results.

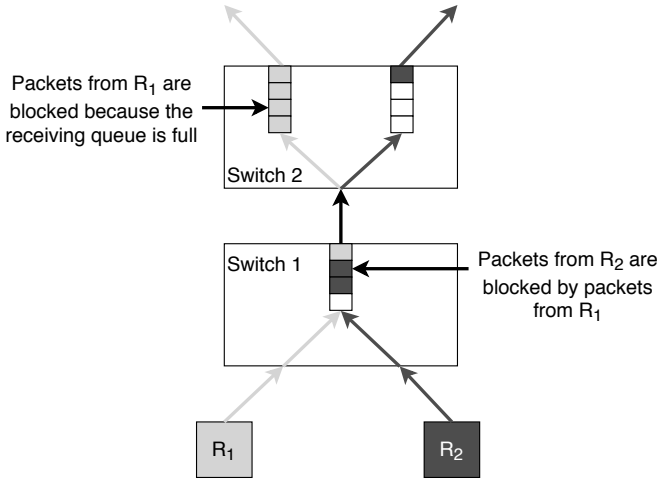


Fig. 1: Head-of-line blocking

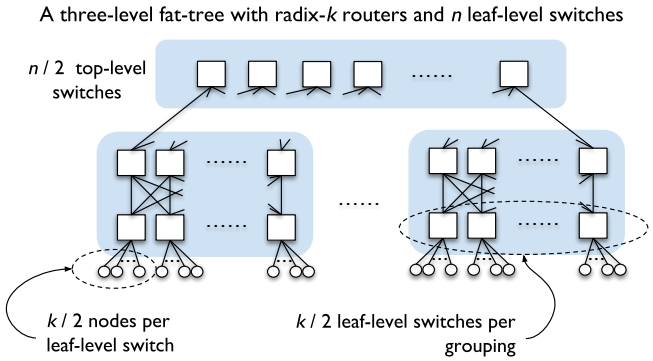


Fig. 2: Typical fat-tree network topology

II. Overview

A. Network Contention

Network contention occurs when multiple network flows concurrently use the same network link. This sharing causes the length of the switch output queue that connects to the link to grow. Thus, packets have longer queue wait times and increased latency. In addition, all flows that share the link experience reduced bandwidth. This slower message delivery increases job execution time and reduces system throughput.

Contention effects can propagate across a job. For example, process R_1, which does not experience contention, may be slowed because it waits for a message from process R_2 that does, even if no contention occurs between R_1 and R_2. Further, if contention causes a message to process R to arrive later than it otherwise would, R is likely to send subsequent messages later even if they use different links, which will delay the processes that receive those messages. These perturbations may propagate across the system and manifest as second and higher order effects that are difficult to predict.

Head of Line (HOL) blocking [10] can produce other unexpected effects. HOL blocking occurs when a switch cannot send a packet because its receiver's queue is full due to heavy contention. Because switch queues are typically FIFOs, other packets in the queue wait even if they could be sent immediately. Thus, contention elsewhere can delay a packet that experiences minimal contention as Figure 1 illustrates.

As node processing power grows relative to network bandwidth in future systems, the time in communication will increase. Also, messages will likely become larger and travel farther. Due to the dynamic nature of network contention, jobs will experience quite different contention environments in different runs. These factors will increase the likelihood and impact of contention, making it increasingly important.

Space-sharing the network would avoid network contention. In this scheme, jobs use disjoint parts of the network, so they do not share links and, thus, contention does not occur. BlueGene systems [11]. for example, use this strategy. However, its job placement restrictions reduce node utilization and, typically, overall throughput [12], [13]. Low utilization is antithetical to the goals of many HPC centers so most systems use shared interconnects.

B. Quality of Service

Most modern networks provide Quality of Service (QoS) mechanisms to manage their traffic. QoS has many forms; in this paper we focus on traffic prioritization by assigning relative priorities to different network flows. The network forwards packets based on the priorities of their flows, giving more bandwidth to those with higher priority.

While our approach applies to any network with traffic prioritization, we focus on InfiniBand, which implements QoS with *service levels*. Each service level corresponds to a priority, and each packet is assigned to a service level. InfiniBand nodes and switches send packets in a weighted round-robin order in which the weights correspond to the service level priorities [14]. For example, if service level S_1 has a priority of 7 and service level S_2 has a priority of 2, a switch sends 7 packets from S_1 followed by 2 packets from S_2, and so on. Since packets from different service levels use separate queues, HOL blocking does not occur between service levels.

C. Fat Trees

Many HPC systems use a fat-tree network topology [15], in which switches are arranged as a k-ary tree and the bandwidth of links between switches increases towards the root of the tree. This extra bandwidth gracefully handles cases in which many nodes in different subtrees communicate. Building true fat trees is infeasible since they require different types of switches and links with different bandwidths. Most systems instead use folded Clos networks; Figure 2 shows an example. At the root level, all links connect to lower level switches. At all other levels, half of the links connect to higher level switches and half connect either to lower level switches or to nodes. Thus, a fat tree with n leaf switches has $\frac{n}{2}$ top level switches. Systems typically have two or three levels and tens of ports per switch, which supports thousands of nodes.

Fat trees have many routes between node pairs. Most fat trees use static routing to choose among them. In the typical

routing algorithm, "destination mod k" or D-mod-k [16], the final destination of a message determines its next hop. This algorithm disperses routes across the tree to improve performance but leads to potentially severe network contention [1].

III. Algorithm

Our goal is to mitigate network contention using Quality of Service (QoS). We define a job as an MPI application execution. Each job consists of many MPI processes. We reduce network contention by applying QoS at the process level. All messages that a process sends or receives will have the priority assigned to that process. Our previous work [17] showed that job-level priorities provide little benefit. Assigning priorities at the flow or message level requires significant per-job analysis, because most jobs have many more flows and messages than processes. Since we assign per-process priorities, we can obtain significant performance improvement (as we show in Section VI), while avoiding per-job analysis.

We first motivate the issues with per-process priorities and discuss our assumptions. We then present our per-process algorithm that reduces the effect of contention.

A. Challenges

As we discussed in Section II-B, InfiniBand assigns a service level to each packet, which determines the relative priority of that packet. Thus, our algorithm, which assigns service levels to processes, must determine the priorities of the service levels such that we improve performance for at least some jobs. We consider two main issues.

a) Approximating the application critical path: Considering every process for a priority increase would be too expensive. To reduce the cost, we could consider only the slowest process from each job. However, we would ideally consider processes that spend significant time on the critical paths of their jobs, and the slow processes may not. Since the cost of finding the critical path of every job would also be too high, we must determine *candidate* processes that are likely to be on the critical paths.

b) Handling contending ranks: Our initial set of candidates does not consider the network topology, which can guide further pruning of the candidate set. Suppose two processes, denoted R_1 and R_2, share a link and thus contend with each other. Prioritizing R_1 will slow down R_2. If R_2 is on its job's critical path, or if this change causes it to become a part of the critical path, the prioritization will reduce the performance of R_2's job. Similarly, prioritizing R_2 reduces the performance of R_1's job. Thus, we may need to avoid prioritizing a process that shares links with other candidates.

We address these two key challenges as follows. First, given the difficulty of finding the critical path at run time, we approximate it by choosing *candidates* that are within a tolerance of the slowest process. These processes probably have spent time on the critical path. Second, we use an online feedback-directed algorithm to identify processes that may share links. Our algorithm rolls back the priority of a candidate process if its increase caused significant slowdowns elsewhere. These choices greatly simplify our algorithm and allow it to handle diverse contention scenarios.

B. Assumptions

We strive to minimize information required from the executing jobs. Information about a single job has limited value since a job is typically affected by *other* jobs. We assume that our algorithm can measure the amount of work that each process performs in a given time period. We can easily obtain this information if each process sends a signal to the run time system each time it completes an iteration. We also assume that each process can send packets from any service level and that we can change service level priorities at run time. While most systems do not currently provide these capabilities, they are easily supported. We only need to set a few bits in the packet header to assign its service level. Changing the priority of a service level is more complicated since we must update the priority table at every node and switch in the network. However, InfiniBand already includes a subnet manager that must transmit priority table updates across the network. Our algorithm simply uses this capability more often than is currently the norm.

C. Rank Prioritization Algorithm

Algorithm 1 shows that we initially run all jobs in the default configuration (all jobs share a single service level). The user must provide P (line 6) to determine how long the algorithm allows the jobs to run before adjusting service levels. We then record the default performance, which is the amount of work completed by each job (line 7). Subsequent iterations of the algorithm adjust service levels by increasing the priority of a candidate (we discuss how we select candidates below). If the selected process is still in the default service level, we put it in a separate, unused service level (line 32). Otherwise, we increase the priority of its current service level (line 30).

We then run the jobs with this service level change. The resulting performance guides the next step of the algorithm. If the new service level improves performance relative to the current best service level assignments (line 16), the current set becomes the new best and we choose new candidates. Otherwise, we prioritize a different candidate. We determine the benefit of the new service level by calculating each job's improvement over the default run and averaging the results.

The algorithm iterates until either no service levels or no candidates remain (line 19). We then use the best set of service levels until a job arrives or departs (line 35), at which point the algorithm restarts. It may also restart from scratch if the overall performance becomes worse than the default run (line 23).

The *SelectCandidates* function selects the algorithm's candidates, which are all processes within a threshold (line 39) of the slowest process (the one that makes the least progress). The user provides T. While this heuristic may omit some processes on the critical path, it limits the size of the candidate set without being too selective.

Other aspects of the algorithm are specific to the QoS implementation. Variable *NUM_SLS* (line 19) is the number of

Algorithm 1 Rank Prioritization Algorithm

```
1: function RANK_PRIORITIZATION_ALGORITHM
2:     state ← DEFAULT
3:     assignments ← no ranks prioritized
4:     repeat
5:         SetServiceLevels(assignments)
6:         Allow the jobs to run for P seconds
7:         workDone ← work completed by all jobs
8:         if state = DEFAULT then
9:             default.workDone ← workDone
10:            best.workDone ← workDone
11:            best.assignments ← assignments
12:            candidates ← SelectCandidates()
13:            state ← RUNNING
14:        else if state = RUNNING then
15:            if workDone > best.workDone then
16:                best.workDone ← workDone
17:                best.assignments ← assignments
18:                candidates ← SelectCandidates()
19:            if |assignments| = NUM_SLS  or
20:              |candidates| = 0 then
21:                state ← IDLE
22:        else if state = IDLE then
23:            if workDone < default.workDone then
24:                state ← DEFAULT
25:        if state = DEFAULT then
26:            assignments ← no ranks prioritized
27:        else if state = RUNNING then
28:            rank ← candidates.pop()
29:            if rank in assignments then
30:                increase assignments[rank].priority
31:            else
32:                assignments[rank] ← new service level
33:        else if state = IDLE then
34:            assignments ← best.assignments
35:    until the job mix changes
36: function SELECTCANDIDATES
37:    candidates ← ∅
38:    for all jobs do
39:        worst ← min(job.ranks.workDone) ∗ (1/T)
40:        for all ranks in job do
41:            if rank.workDone < thresh then
42:                candidates ← candidates ∪ {rank}
       return candidates
```

available service levels. For example, InfiniBand allows 16 service levels, although most current hardware only implements eight [18]. The *SetServiceLevels* function (line 5) updates the hardware's service levels and priorities.

IV. Simulator

This section describes the simulator that we use to evaluate our algorithm. We use a simulator (rather than a real system) for four reasons. First, we can use per-process QoS, which is not implemented on any HPC cluster to which we have access. Second, we can run many tests on systems of varying size. Third, the results are repeatable. Fourth, we can more easily investigate and understand the low-level effects of QoS.

We use TraceR [9], which is an HPC application trace replay tool. It is built on the CODES framework [19], which is built on the ROSS parallel discrete event simulator [20]. TraceR performs packet-level simulation of MPI applications on HPC networks. It efficiently and accurately predicts the performance of important applications on networks with different properties as previous work [21] has extensively validated.

For our experiments, we add QoS to TraceR. For clarity, we refer to the baseline TraceR implementation as TraceR while TraceR-QoS represents our modifications to TraceR and CODES along with additional scripts that implement our algorithm. Our modifications are publicly available and have been submitted for inclusion in their respective projects. Our updates to TraceR are available at https://github.com/LLNL/TraceR/pull/23 and our updates to CODES are available at https://xgitlab.cels.anl.gov/codes/codes/merge_requests/81.

A. Requesting Per-Rank Service Levels

Our first modifications allow applications to request a specific service level for each process. For each job, TraceR-QoS reads a text file that specifies an (optional) service level for each process and a default service level for any unspecified processes. When a process sends a message, TraceR-QoS looks up its service level and notifies CODES that the message should be sent at the specified service level. If the service level file indicates that a process should use a specific service level, then any message sent to or from that rank will use the specified service level. If the sending and receiving ranks are explicitly assigned different service levels, TraceR-QoS uses the sender's service level.

B. Modifying Packet Injection

A node may have several packets in its output queue between which it must choose when it is ready to inject one into the network. This situation arises, for example, when a process rapidly sends multiple messages. CODES implements several packet selection methods, including first-come, first-served, round robin, and priority. The built-in priority scheme sends all packets in the high priority queue before sending any packets in lower priority queues. With QoS, each queue corresponds to a service level, and thus each queue may send a fixed number of packets after which packets from other queues are sent if available. We extended the existing priority mechanism to implement QoS with separate queues for every service level, between which we arbitrate based on priorities.

C. Handling Packets Within the Network

Finally, we modified how switches handle packets in the CODES framework. In CODES, each switch has a single output queue for each port. When a switch receives a packet it is immediately placed in the output queue of the appropriate sending port. It removes packets from the output queue

Machine	Nodes	Levels	Link B/W	Num. Jobs	Ranks/Job
small	64	2	3.2	8	8
medium	324	2	3.2	16	20

TABLE I: Experimental systems; *Levels* is the height of the fat tree; bandwidth is in GB/s.

and sends them to their destinations in FIFO order. QoS requires packets to be sent based on their priorities. Thus, we implemented switches that use multiple queues per port, one per service level. When a switch receives a packet, it is immediately placed in one of the output queues of the appropriate output port, specifically, the one for the packet's service level. TraceR-QoS uses an arbitration algorithm in which a switch selects a packet to send from one of its queues in accordance with the priorities of the service levels. This change only applies to the fat-tree network implementation in TraceR-QoS; we did not add QoS support to the other network topologies that TraceR supports.

D. Additional Software

To avoid unnecessary changes to TraceR, TraceR-QoS contains multiple components. One is TraceR itself, which we use to calculate job performance under QoS. The other is a Python component that selects candidates and updates service levels and priorities. This Python code is available at https://bitbucket.org/lesavoie/cluster-2019-mitigating-inter-job-interference-via-process.

V. Experimental Setup

We evaluate the effectiveness of our system for several MPI microbenchmarks under various configurations. In the following sections, we describe our experimental environment, experimental methodology, and microbenchmarks.

A. Environment

Table I describes the two simulated systems with fat-tree topologies, with which we test our algorithm in TraceR-QoS. We denote these systems as *small* and *medium* in this paper, as they represent a small- and medium-scale cluster, respectively. We base the link bandwidth for both systems on the measured bandwidth of actual systems. Table I also includes the number of jobs and the number of processes per job that we use on each system. We run more tests on *small* than on *medium* because TraceR-QoS simulates those tests much faster, which allows us to run a broader range of tests. We include selected results from *medium* for comparison.

B. Methodology

As we noted in Section IV-D, we did not integrate our entire algorithm into TraceR. Instead, we use the simpler approach of running several short simulations, one for each iteration of our algorithm. Thus, only line 6 of Algorithm 1 actually runs within the simulator. We run the the rest of the algorithm in wrapper scripts. The general pattern of a test is as follows:

- Run a short simulation with all jobs and the set of service level assignments;
- Analyze the results from the simulation; and
- Select new service levels.

We repeat this pattern for the desired number of iterations. We run 500 iterations of our algorithm for each experiment except the arrival/departure tests, in which we run 2500 iterations to allow many jobs to arrive and to depart.

We use 0.06 seconds for P (the period of measurement before we adjust service levels) so each iteration of our algorithm takes less than 1/10th of a second. We select 0.06 seconds to give our algorithm sufficient time to evaluate the performance of the jobs but to keep the simulation time reasonable. After each simulation, we measure the amount of progress each job made by calculating the fewest number of iterations any process in that job completed during P.

TraceR-QoS replays traces of job executions to simulate them. We collected our traces on Catalyst, a system at Lawrence Livermore National Laboratory (LLNL) that has 300 compute nodes, each with two 12-core Intel Xeon E5-2695 CPUs and 128 GB of RAM. We use ScoreP [22] to collect traces. We compile ScoreP into each application as a library. During a tracing run, it intercepts MPI calls and records them in a standard trace format. We use the OTF2 format [23].

Because we repeatedly run similar simulations, we could over-fit our results to our traces. Since even uniform applications behave slightly differently over time, we use five different traces of every application, which makes our simulations more realistic. We run the applications with the same parameters for each trace but we run them at different times and/or on different nodes. Thus, the traces are similar but have slightly different characteristics, much like repeated iterations of a job over time. We randomly select a trace for every job in every iteration of our algorithm. The *Mini-ParaDiS* test is the only exception; for it we use a different trace for every iteration, which allows its computational imbalance to grow over time.

We run three tests for every iteration of our algorithm. First, we run each job in isolation to measure un-contended performance. Second, we run all jobs together with a single service level, which is our default. Third, we use the service levels that our algorithm selects. We use the same version of all traces for these three runs so we can accurately compare the performance of our algorithm to default and isolated runs.

We ran all tests on eight different randomly chosen node allocations, or assignments of processes to nodes. This choice approximates typical space-shared HPC systems in the steady state. Since schedulers run jobs on any available nodes, job allocations (in the steady state) are essentially random. We simulate this effect with different node allocations, which also tests our algorithm with different contention characteristics.

Most of our tests use a T of 0.9 so processes that are within roughly 10% of the slowest rank are included in the candidate set. A few (clearly delineated) tests vary this threshold. We set *NUM_SLS* to 16, which allows our algorithm to use up to 16 service levels, the maximum that InfiniBand allows. All of our tests assign one process to each node with the communication scaled to match that of a hybrid MPI/OpenMP model.

C. Microbenchmarks

We test our algorithm with six microbenchmarks that implement common communication patterns. These benchmarks can be configured with different message sizes, computation amounts, and computation imbalances. We call these benchmarks *Flood-Pairs*, *Nearest-Neighbor*, *Random-Pairs*, *All-to-all*, *Mini-ParaDiS*, and *Mini-AMG*.

Flood-Pairs divides processes into pairs that exchange many messages. We adapt it from the CORAL bisection bandwidth test [24]. Our tests pair process x with process $\frac{n}{2} + x$, where n is the total number of processes. Despite its simple communication pattern, *Flood-Pairs* has significant network usage.

Nearest-Neighbor implements a 2D nearest neighbor communication pattern without wraparound. Each process exchanges messages with its 4 neighbors in a 2D grid (processes on the edges of the grid have fewer neighbors). This communication pattern is common in HPC applications.

Like *Flood-Pairs*, *Random-Pairs* groups processes in pairs that exchange many messages. However, in *Random-Pairs*, we match each process in the lower half of the processes with a randomly selected process in the upper half. This test represents applications with static communication patterns that are not pre-determined at compile time.

In *All-to-all*, each process repeatedly exchanges messages with every other process. We use it to examine how QoS affects applications with periodic global synchronization.

Mini-ParaDiS is a load-imbalanced microbenchmark that mimics the ParaDiS dislocation dynamics simulation [25]. It has nearest neighbor communication but the work steadily becomes more imbalanced over time. The full ParaDiS application can periodically use a dynamic load balancer, but in that case it behaves similarly to *Nearest-Neighbor*.

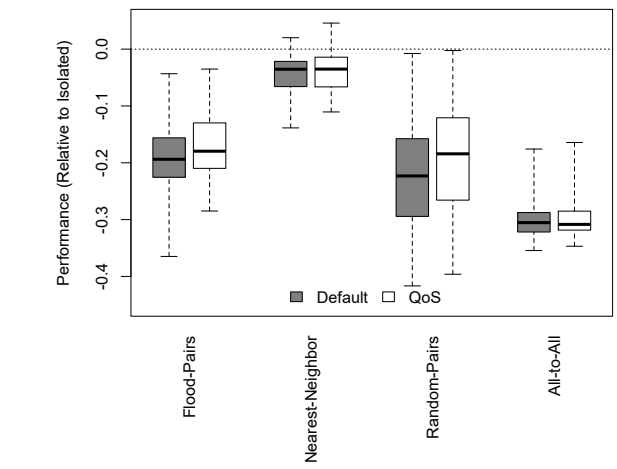
Mini-AMG is a micro-benchmark that mimics AMG [26], in which communication occurs between neighbors in changing grids (because of coarsening and interpolation). We base its communication on a trace of AMG but the microbenchmark version allows us to experiment with different message sizes and computation-to-communication ratios.

VI. Results

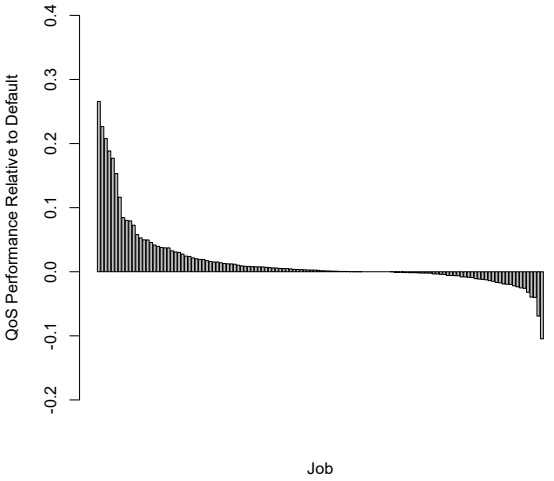
This section documents the performance of our algorithm. We test with different applications, different computation-to-communication ratios, different values of T, and different job arrival and departure patterns,

A. Static Communication Patterns

Our base results use jobs that have static (i.e., unchanging) communication patterns and predictable computation. Figures 3 and 4 show the results on the *medium* and *small* systems. On *medium*, we run 16 jobs: four instances each of *Flood-Pairs*, *Nearest-Neighbor*, *Random-Pairs*, and *All-to-all*; each job has 20 processes. On *small*, we run eight jobs: two instances each of the same four applications; each job has 8 processes. We introduce some computational imbalance in these jobs. The computation time for each rank was chosen randomly between 0 and Y, and Y was chosen such that the process that computes the longest spends roughly 65% of its time in computation. For a given process, the amount of computation is the same for an entire run. We set T to 0.9 and run eight tests on different random assignments of processes to nodes. We refer to these as the *static* results for the rest of this section.



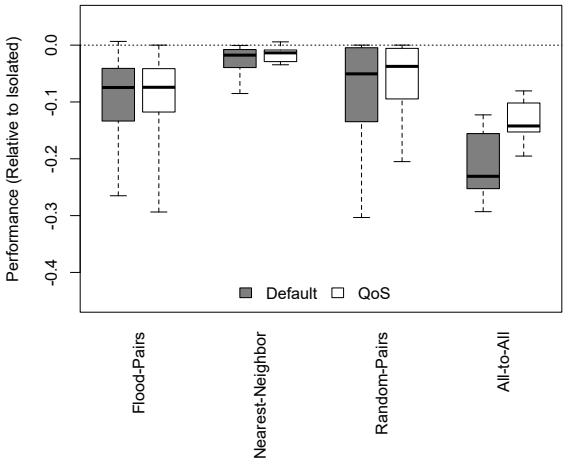
(a) Overall results



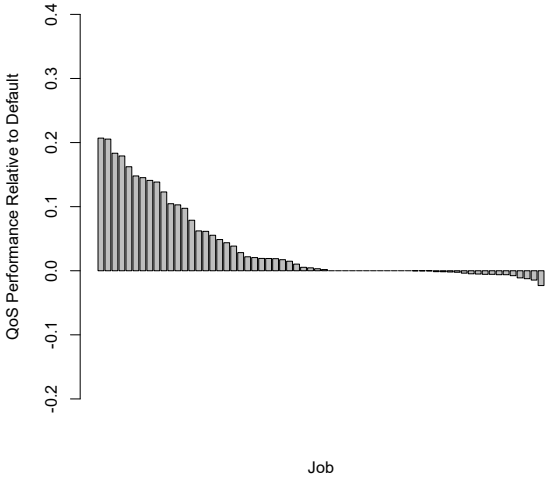
(b) Individual job performance

Fig. 3: Results on the *Medium* System (324 Nodes)

These results include both the running and idle time of our algorithm, so they accurately depict its benefits and overheads. We include two charts for each set of results. The first chart shows the overall results for each job type as a box plot; we include results with all jobs in the same service level (labeled *Default*) and with our algorithm (labeled *QoS*). The x-axis lists the microbenchmark and the y-axis shows the performance of each job relative to the same job run in isolation (negative numbers indicate that the job completed less work than it did on the isolated run, the most likely scenario). We mark $y = 0$, which indicates that a job achieved the same performance as it did on the isolated run, with a dashed line. We combine



(a) Overall results



(b) Individual job performance

Fig. 4: Results on the *Small* System (64 Nodes)

multiple instances of the same microbenchmark because they are identical. Thus, each box plot shows the results from several jobs across several placements. The second chart shows the improvement that our algorithm achieves over the default performance for individual jobs, sorted from the greatest improvement to the greatest degradation.

Our algorithm improves performance on the *medium* system for more than half of the jobs, while incurring relatively small slowdowns in the other jobs. The largest single-job performance improvement over default is 27%, and many jobs have improvements over 10%. All single-job performance degradations are 4% or less, with the exception of two jobs with 7% and 10% degradation. Further, our algorithm reduces the impact of contention by up to 77% (for one of the *Flood-Pairs* jobs) and completely eliminates it (for some of the *Nearest-Neighbor* jobs). The results on the *small* system are similar; performance improves for most jobs—over 20% in two cases. Several had at least 10% improvement, with the worst case only 4% degradation. As with *medium*, in some cases our algorithm eliminates the impact of contention.

On *medium*, some *Nearest-Neighbor* jobs ran faster in default and QoS cases than in isolation despite incurring contention with other jobs. Although we expect performance to be worse, in this case contention with other jobs reduces the impact of contention within a job. Suppose processes R_1 and R_2 are part of the same job and share a link, so they contend with each other. If R_2 experiences contention from another job, R_1 can send more packets and complete its message faster. If R_1 is on the critical path and R_2 is not, the performance of the job will improve. This situation occurs when the default or QoS case is faster than the isolated case.

Our remaining experiments vary system or workload characteristics. These results are on the *small* system because those simulations run much faster than simulations of *medium*.

B. Imbalance

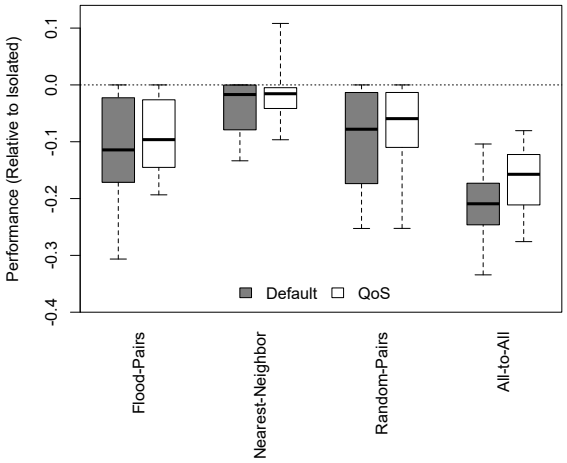
Our next tests use traces with different amounts of computational imbalance than the static ones. In our static tests, the computation of each process is randomly chosen to be between 0 and Y such that the process that does the most computation is expected to spend 65% of its time in computation. We next experiment with significant load imbalance (see Figure 5. In these results, one process does Y computation while others in the job do none. Note that we shift the y-axis

The results with large computational imbalance are not much better than the static results, with the exception of one case in which our algorithm achieves an 11% improvement over the isolated case. This 11% improvement occurs because QoS changes message timing such that messages on the critical path overlap with fewer non-critical messages and thus experience less contention. These results show that our algorithm does not need extreme imbalance to improve job performance. They also indicate that the potential additional benefit of large imbalances is limited with a QoS-based scheme.

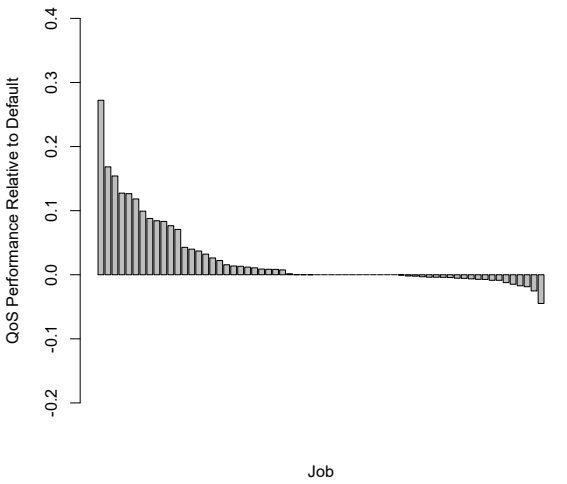
C. Irregular Applications

The applications in our experiments so far have static communication patterns and predictable computational load. We now present tests with applications with more variable computation and communication patterns to understand how our algorithm works on irregular applications. Recall that *Mini-ParaDiS* implements a nearest neighbor communication pattern, but the computational imbalance grows as it executes. We use the same test setup as the static test except that we replace the *Random-Pairs* jobs with *Mini-ParaDiS* jobs. Figure 6 shows that our algorithm speeds up many of the jobs, although not as much as in the static tests. Thus, our algorithm handles irregular computation, although the improvements may be smaller than with more regular applications.

Our second irregular application is *Mini-AMG*. To obtain a nontrivial communication pattern, we must use 32 ranks for *Mini-AMG*, so we have one instance of the four static benchmarks. Figure 7 shows that our algorithm's impact on *Mini-AMG* is minimal because it incurs minimal slowdown from contention. Our experience is that AMG's message

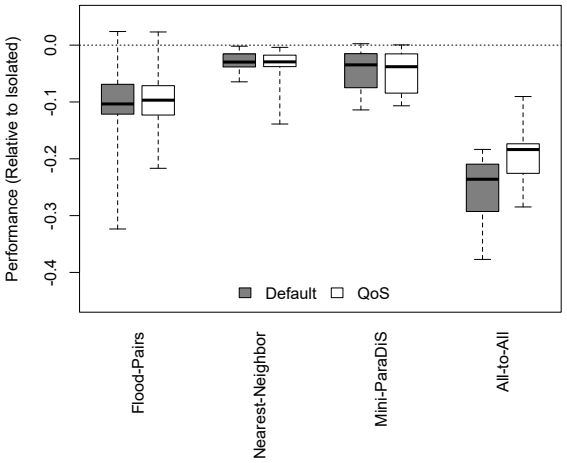


(a) Overall results

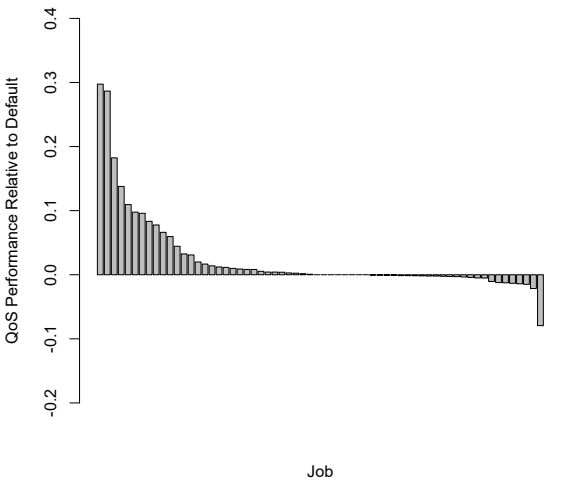


(b) Individual job performance

Fig. 5: Results with computational imbalance



(a) Overall results



(b) Individual job performance

Fig. 6: Results with *Mini-ParaDiS*

pattern does not generate much contention. However, the static benchmarks obtain improvement and our algorithm handles the irregular communication pattern of *Mini-AMG*.

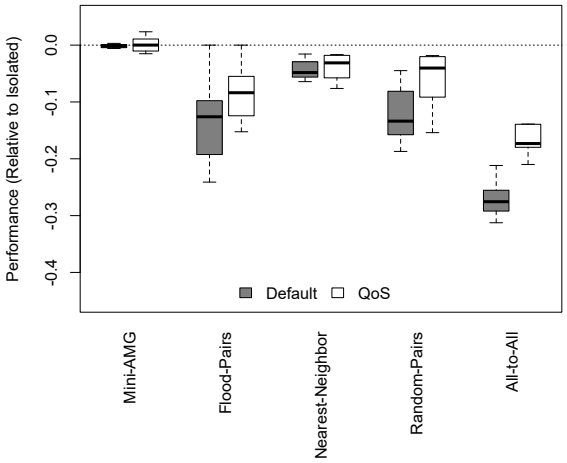
D. Computation-to-Communication Ratio

The percentage of time that a job communicates affects the amount of network contention that it experiences and, thus, the impact of QoS. For example, a job that spends 100% of its time in computation does not use the network, so our algorithm will not affect it. At the other extreme, network contention will significantly affect a job that does no computation. However, the job uses the network so heavily that any prioritization will likely negatively affect some jobs. Fortunately, real jobs lie between these extremes. We further discuss the relationship between computation-to-communication ratio and the impact of our algorithm in Section VI-G.

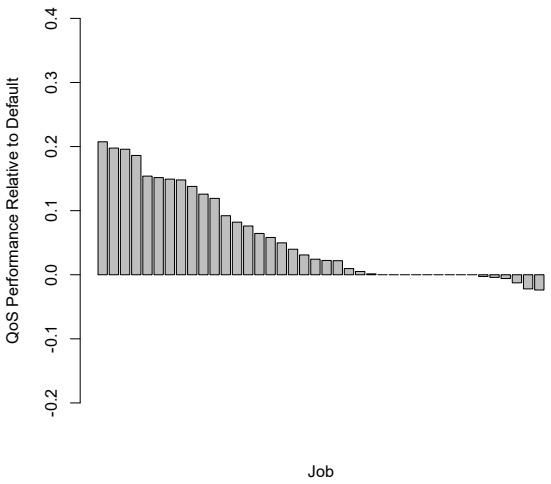
Figures 8 and 9 show the impact of the computation-to-communication ratio on our algorithm; note the expanded y-axis in Figure 8. The slowest process of each job spends approximately either 35% or 90% of its time in computation in these tests, which are otherwise the same as the static tests. Jobs with 35% computation experience up to 57% performance degradation from the isolated case due to contention, while jobs with 90% computation experience up to 12% degradation. However, our algorithm improves some of the jobs' performance relative to default. In the 35% computation case our algorithm improves performance up to 33% at the expense of 15% slowdowns in other jobs. The 35% computation case is the primary situation in which our algorithm degrades performance by more than 10%. However, this ratio is unlikely to occur in real jobs. In the 90% computation case our algorithm improves performance by up to 7%, but no job experiences a performance degradation greater than 1%.

E. Candidate Threshold

Here, we vary the candidate threshold, T. Section III described that T determines how close (in work completed) a process must be to the slowest process to be a candidate. These tests set T to 0.5 and 1.0, to complement the static tests with $T = 0.9$. Figures 10 and 11 show that most jobs

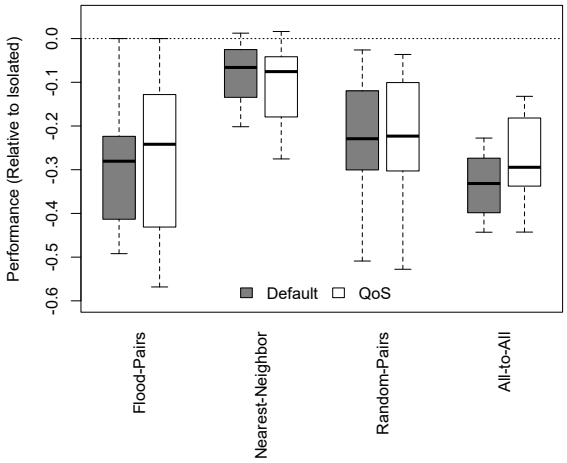


(a) Overall results

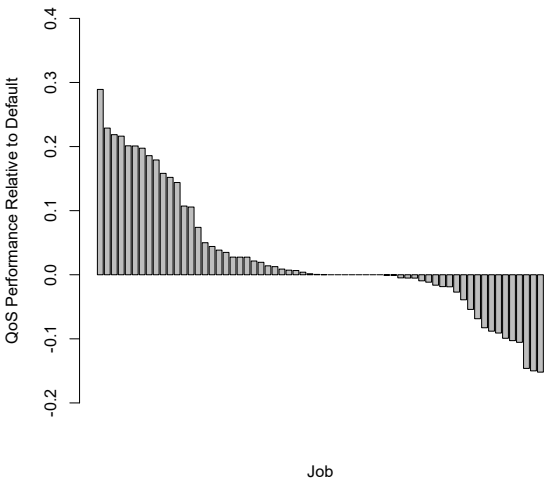


(b) Individual job performance

Fig. 7: Results with *Mini-AMG*



(a) Overall results



(b) Individual job performance

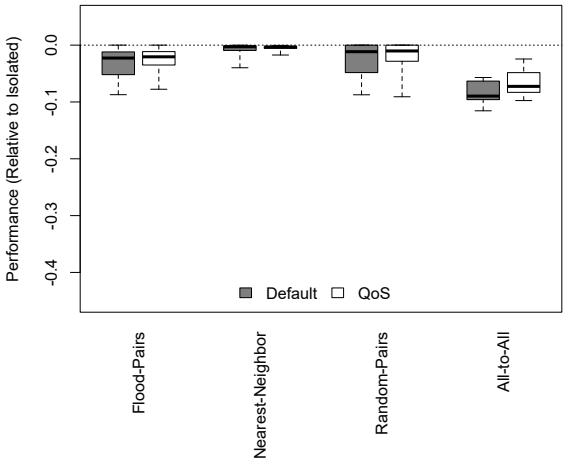
Fig. 8: Results with 35% computation

have slightly smaller performance improvements with $T = 0.5$ than with $T = 0.9$. A lower threshold means our algorithm runs for more iterations before converging, which dampens performance improvements. With $T = 1.0$, our algorithm improves performance as much as 40%, although the overall improvements are smaller than in the static results. These results show that our algorithm works best with a moderate number of candidates.

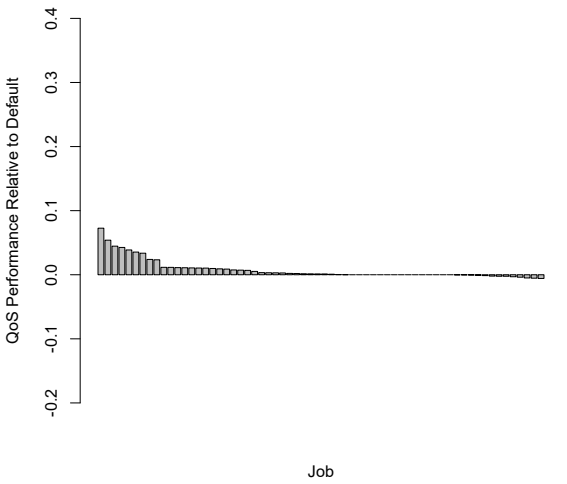
F. Job Arrivals and Departures

On HPC systems, jobs can arrive and depart at any time. Algorithm 1 handles arrivals and departures by using the default service level for all jobs and starting over. Analysis of MPI traces from LLNL shows that the time between arrivals and departures averages about a minute. However, the average is reduced by similar small-node jobs that perform uncertainty quantification. These jobs could, in principle, be handled together, and the average arrival/departure would be longer. Since we use a P of 0.06, our algorithm can complete many iterations in a few minutes, so in most cases it will have plenty of time to converge before the next arrival or departure. Further, our algorithm continually applies incremental improvements, so full convergence is not required.

Figure 12 shows the results for a test with arrivals and departures. Because each job executes for a different amount of time in this test, each individual *job* (rather than each *job type*) is represented by a boxplot. The test starts identically to our static test, but after 500 iterations of our algorithm we replace the first *Flood-Pairs* job with another *Flood-Pairs* job that does 35% computation. The new job is the 9th job in the chart. After 1000 iterations, we replaced the first *Nearest-Neighbor* job with a *Nearest-Neighbor* job that does 35% computation, which is the 10th job in the chart. We perform similar replacements with *Random-Pairs* and *All-to-all* after 1500 and 2000 iterations. Our algorithm again improves performance for most jobs by as much as 25% while limiting degradations to 6% or less. These results are similar to those achieved in the static tests. Thus, our algorithm improves job performance in the presence of arrivals and departures.

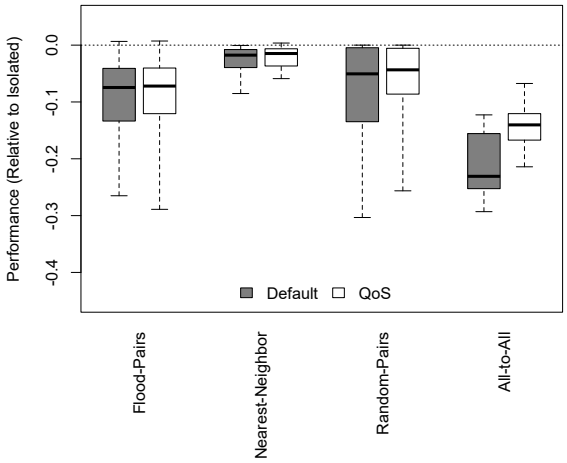


(a) Overall results

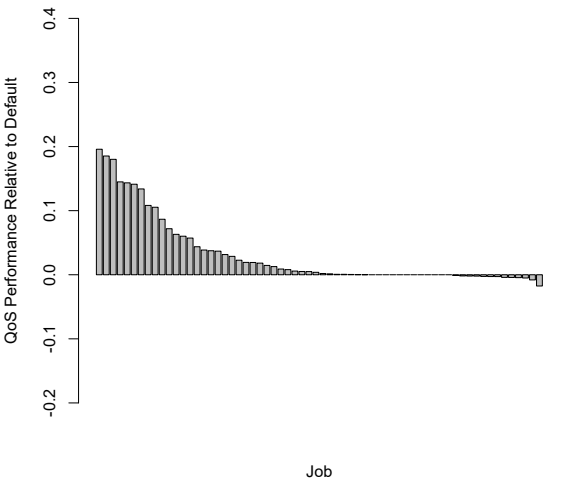


(b) Individual job performance

Fig. 9: Results with 90% computation



(a) Overall results



(b) Individual job performance

Fig. 10: Results with $T = 0.5$

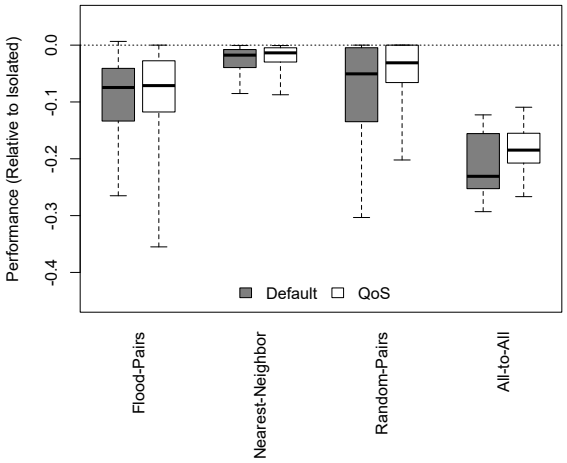
G. Discussion

Our results yield some interesting insights. First, the effects of network contention are complex and difficult to predict. Contention can propagate as processes send messages within a job, and as messages share links between jobs and within jobs. Localized contention can affect jobs across the network. Further, the impact of contention depends on the timing of messages as well as on the critical path of each job. These effects form complicated interactions. Thus, we implement a feedback-directed algorithm as the most reliable way to determine the effect of contention is to observe it.

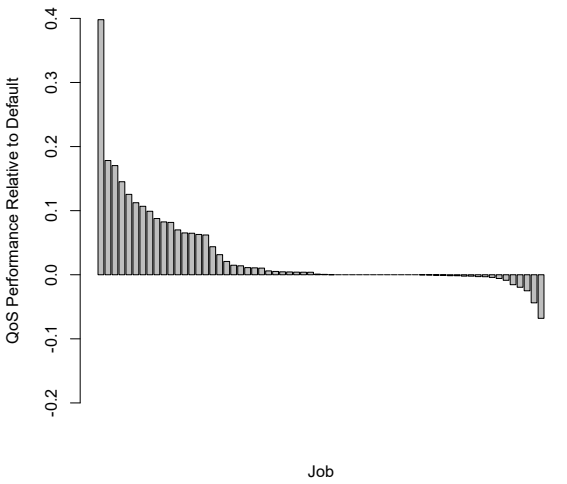
Our results also provide information on scenarios that can benefit from QoS. We have found that QoS is most useful when jobs have some computational imbalance, as our algorithm de-prioritizes messages to nodes with little computation in favor of messages to nodes with more computation. QoS can probably benefit some cases with jobs with perfectly balanced computation, mostly by eliminating HOL blocking, but these improvements are likely to be relatively modest.

QoS is most useful when jobs have a moderate amount of communication. Contention hardly impacts jobs with little communication and thus they can gain little from QoS. Alternatively, jobs that spend most of their time communicating experience a large (negative) impact from network contention. However, these jobs have little computational imbalance since they perform little computation. So, QoS has little opportunity to exploit. Further, these jobs use sufficient network bandwidth that adjusting network flows is likely to slow some of the jobs.

QoS also has more potential if each process performs a similar amount of computation across successive iterations. If each process performs a truly unpredictable amount of computation, by definition we cannot predetermine the impact of QoS. Fortunately, the amount of computation within a process tends to change slowly, which our algorithm handles. Overall, QoS is most helpful for jobs with moderate communication and moderate computational imbalance and for which the computational imbalance changes (relatively) slowly over time, which describes many HPC workloads.

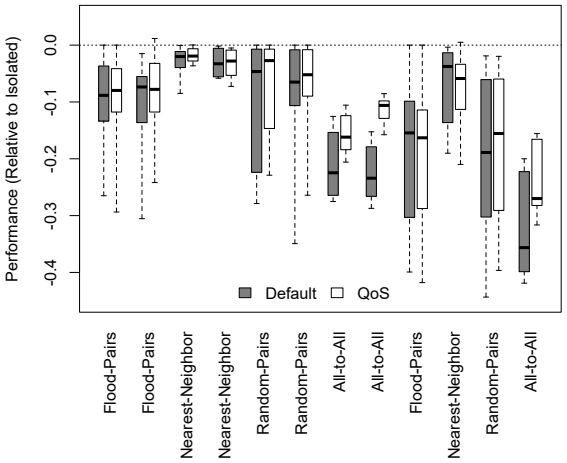


(a) Overall results

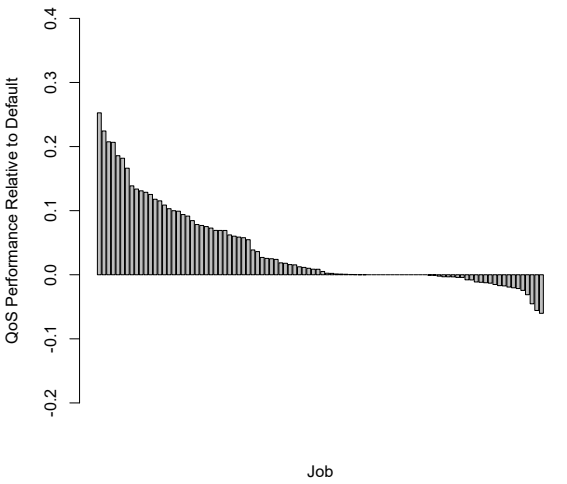


(b) Individual job performance

Fig. 11: Results with $T = 1.0$



(a) Overall results



(b) Individual job performance

Fig. 12: Results with arrivals and departures

VII. Related Work

QoS research has a long history. The internet protocol (IP) implements differentiated services [27], in which flows are divided into classes that are handled differently. This is an example of traffic prioritization, the QoS mechanism that we use. QoS has also been applied in areas as diverse as wireless networking [5], data centers [6], and video streaming [7], [8].

QoS use in HPC networks is limited. Researchers have used QoS to minimize HOL blocking [28], [29], but these schemes use service levels with identical priorities. Our work prioritizes some flows. A different approach uses QoS to prioritize traffic at the job level [30]. This scheme prioritizes jobs based on their network utilization. With more jobs than service levels, jobs with similar network utilization are grouped in the same service level. This approach improved job performance in a simulated environment. Others have used QoS on a simulated Megafly network (which is a variant of the dragonfly) [31]. They applied QoS at the job level and by separating collective and point-to-point communication into different service levels. The job-level QoS scheme is similar to our prior work (see next paragraph); this paper is fundamentally different in that we use rank-level granularity.

We previously presented the first empirical investigation of QoS and its impact on network contention on an HPC system [17]. We showed that network contention reduces performance up to 70% and that giving high priority to a job avoids this issue. However, we also showed that the coarse-grained prioritization degraded the performance of other jobs, which often reduced overall throughput. Our study is in contrast to the simulation-based work [30], [31] that found an overall improvement from per-job QoS. We found that assigning a high priority to an entire job prioritizes some processes that do not need to be prioritized, increasing the impact of network contention on other jobs. Thus, we explore per-process QoS in this paper.

Researchers have considered several other methods to reduce network contention on fat trees [1], dragonfly networks [1], [2], [3], and tori [4]. Adaptive routing [32] seeks

to route traffic away from congested links. However, adaptive routing on dragonflies does not necessarily reduce contention significantly [1]. Hardware adaptive routing on fat trees has been introduced recently on Summit and Sierra [33]. However, unlike our QoS solution, hardware adaptive routing necessarily lacks full global information. Other research into adaptive routing at the global level [1] involves rewriting routing tables based on application communication patterns, so it is more intrusive and requires more information than our solution. Researchers have also studied job placement strategies that minimize contention [3], [34], [12]. However, these approaches do not eliminate contention within a job, and they typically reduce machine utilization. Further, any of these approaches could be combined with QoS.

VIII. Summary

Node processing power improvements will continue to outpace network bandwidth improvements in HPC systems. Thus, network performance and network contention will become more important. In this paper, we have explored how QoS mechanisms can mitigate the impacts of network contention. We have introduced an algorithm that uses QoS to improve the performance of individual jobs by up to 40%, usually without degrading the performance of other jobs by more than 5%. Further, our algorithm sometimes eliminates or reduces the impact of contention, especially for jobs that have imbalanced computation. Future HPC systems must employ network management techniques such as those we have introduced in this paper as network performance becomes a larger component of overall system performance.

References

[1] S. Smith, C. Cromey, D. K. Lowenthal, J. Domke, N. Jain, and A. Bhatele, "Mitigating inter-job interference using adaptive flow-aware routing," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18. IEEE Computer Society, Nov. 2018.

[2] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on Xeon Phi based Cray XC systems," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, USA, November 12-17 2017.

[3] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully! Job interference study on dragonfly network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16*, 2016.

[4] 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.

[5] M. Andrews, K. Kumaran, K. Ramanan, A. Stolyar, P. Whiting, and R. Vijayakumar, "Providing quality of service over a shared wireless link," *Comm. Mag.*, vol. 39, no. 2, pp. 150–154, Feb. 2001. [Online]. Available: http://dx.doi.org/10.1109/35.900644

[6] T. Voith, K. Oberle, and M. Stein, "Quality of service provisioning for distributed data center inter-connectivity enabled by network virtualization," *Future Generation Computer Systems*, vol. 28, no. 3, pp. 554 – 562, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X11000392

[7] C.-H. Ke, C.-K. Shieh, W.-S. Hwang, and A. Ziviani, "A two markers system for improved MPEG video delivery in a diffserv network," *IEEE Communications Letters*, vol. 9, no. 4, pp. 381–383, April 2005.

[8] W. Kumwilaisak, Y. T. Hou, Q. Zhang, W. Zhu, C. C. J. Kuo, and Y.-Q. Zhang, "A cross-layer quality-of-service mapping architecture for video delivery in wireless networks," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 10, pp. 1685–1698, Dec 2003.

[9] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, "Evaluating HPC networks via simulation of parallel workloads," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 14:1–14:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014923

[10] M. J. Karol, M. G. Hluchyj, and S. P. Morgan, "Input versus output queuing on a space-division packet switch," *Communications, IEEE Transactions on*, vol. COM-35, pp. 1347 – 1356, 01 1988.

[11] E. Krevat, J. G. Castaños, and J. E. Moreira, "Job scheduling for the BlueGene/L system," in *8th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP '02. London, UK, UK: Springer-Verlag, 2002, pp. 38–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=646383.689703

[12] E. Zahavi, A. Shpiner, O. Rottenstreich, and I. Keslassy, "Links as a service (LaaS): Guaranteed tenant isolation in the shared cloud," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, March 2016.

[13] S. A. Pollard, N. Jain, S. Herbein, and A. Bhatele, "Evaluation of an interference-free node allocation policy on fat-tree clusters," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18. IEEE Computer Society, Nov. 2018.

[14] S. A. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken, "An overview of QoS capabilities in Infiniband, advanced switching interconnect, and Ethernet," *IEEE Communications Magazine*, vol. 44, no. 7, pp. 32–38, July 2006.

[15] C. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. 34, no. 10, October 1985.

[16] C. Gomez, F. Gilabert, M. Gomez, P. Lopez, and J. Duato, "Deterministic versus adaptive routing in fat-trees," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS'07)*, Rome, Italy, March 26-30 2007.

[17] L. Savoie, D. K. Lowenthal, B. R. de Supinski, and K. Mohror, "A study of network quality of service in many-core MPI applications," in *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, 8 2018, pp. 1313–1322.

[18] D. Crupnicoff, S. Das, and E. Zahavi, "White paper: Deploying quality of service and congestion control in Infiniband-based data center networks," Mellanox Technologies Inc., 2900 Stender Way, Santa Clara, CA 95054, Tech. Rep. 2379, November 2005.

[19] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross, "CODES: Enabling co-design of multilayer exascale storage architectures," in *Proceedings of the Workshop on Emerging Supercomputing Technologies*, May 2011.

[20] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: A high-performance, low memory, modular time warp system," in *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, May 2000.

[21] N. Jain, A. Bhatele, L. H. Howell, D. Böhme, I. Karlin, E. A. León, M. Mubarak, N. Wolfe, T. Gamblin, and M. L. Leininger, "Predicting the performance impact of different fat-tree configurations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017.

[22] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[23] ParaTools, Inc. (2019) Open Trace Format. [Online]. Available: http://www.paratools.com/otf/

[24] (2014, Jun.) Coral benchmark codes. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[25] V. Bulatov, W. Cai, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis, "Scalable line dynamics in ParaDiS,"

in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2004.

[26] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on HPC platforms," in *Proceedings of the International Conference on Supercomputing*, May 2011.

[27] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated service," United States, 1998.

[28] H. Subramoni, P. Lai, S. Sur, and D. K. D. Panda, "Improving application performance and predictability using multiple virtual lanes in modern multi-core Infiniband clusters," in *Proceedings of the 2010 39th International Conference on Parallel Processing*, ser. ICPP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 462–471. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2010.54

[29] W. L. Guay, B. Bogdanski, S. A. Reinemo, O. Lysne, and T. Skeie, "vFtree - A fat-tree routing algorithm using virtual lanes to alleviate congestion," in *International Parallel Distributed Processing Symposium (IPDPS)*, May 2011, pp. 197–208.

[30] A. Jokanovic, J. C. Sancho, J. Labarta, G. Rodriguez, and C. Minkenberg, "Effective quality-of-service policy for capacity high-performance computing systems," in *International Conference on High Performance Computing and Communication and International Conference on Embedded Software and Systems (HPCC-ICESS)*, June 2012, pp. 598–607.

[31] M. Mubarak, N. McGlohon, M. Musleh, E. Borch, R. Ross, R. Huggahalli, S. Chunduri, S. Parker, K. Kalyan, and C. Carothers, "Evaluating quality of service traffic classes on the megafly network," in *International Supercomputer Conference*, Jun. 2019.

[32] N. Jain, A. Bhatele, X. Ni, N. J. Wright, and L. V. Kale, "Maximizing throughput on a dragonfly network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 336–347. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.33

[33] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. H. Oral, D. E. Maxwell, V. G. V. Larrea, A. Bertsch, R. Goldstone, W. Joubert, C. Chambreau, D. Appelhans, R. Blackmore, B. Casses, G. Chochia, G. Davison, M. A. Ezell, T. Gooding, E. Gonsiorowski, L. Grinberg, B. Hanson, B. Hartner, I. Karlin, M. L. Leininger, D. Leverman, C. Marroquin, A. Moody, M. Ohmacht, R. Pankajakshan, F. Pizzano, J. H. Rogers, B. Rosenburg, D. Schmidt, M. Shankar, F. Wang, P. Watson, B. Walkup, L. D. Weems, and J. Yin, "The design, deployment, and evaluation of the CORAL pre-exascale systems," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA, November 11-16 2018.

[34] 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 (IPDPS)*, May 2015, pp. 449–459.