

# Mitigating Inter-Job Interference via Process-Level Quality-of-Service \*

LEE SAVOIE and DAVID K. LOWENTHAL, Department of Computer Science, The University of Arizona

BRONIS R. DE SUPINSKI and KATHRYN MOHROR, Lawrence Livermore National Laboratory

NIKHIL JAIN, Nvidia Corporation

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 manage 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 it completely eliminates the impact of contention. It achieves these improvements 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.

CCS Concepts: • **Networks** → **Network performance evaluation**; • **Computing methodologies** → *Parallel computing methodologies*.

Additional Key Words and Phrases: high-performance computing, network contention, quality of service

## ACM Reference Format:

Lee Savoie, David K. Lowenthal, Bronis R. de Supinski, Kathryn Mohror, and Nikhil Jain. 2020. Mitigating Inter-Job Interference via Process-Level Quality-of-Service . 1, 1 (August 2020), 27 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 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 avoids contention for per-node resources such as cores or memory. However, the interconnect is usually shared among all jobs and can become a severely contended resource on all modern topologies, including fat trees [Smith et al. 2018], dragonfly networks [Chunduri et al. 2017; Smith et al. 2018; Yang et al. 2016], and tori [Bhatele et al. 2013].

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

---

\*A short paper with the same title was published in IEEE International Conference on Cluster Computing, 2019 [Savoie et al. 2019].

---

Authors' addresses: Lee Savoie, [lsavoie@cs.arizona.edu](mailto:lsavoie@cs.arizona.edu); David K. Lowenthal, [dkl@cs.arizona.edu](mailto:dkl@cs.arizona.edu), Department of Computer Science, The University of Arizona, Tucson, Arizona, 85721; Bronis R. de Supinski, [bronis@llnl.gov](mailto:bronis@llnl.gov); Kathryn Mohror, [mohror1@llnl.gov](mailto:mohror1@llnl.gov), Lawrence Livermore National Laboratory, Livermore, California, 94550; Nikhil Jain, [nikhil.jain@acm.org](mailto:nikhil.jain@acm.org), Nvidia Corporation, Santa Clara, California, 95050.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2020 Association for Computing Machinery.

Manuscript submitted to ACM

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 [Andrews et al. 2001; Ke et al. 2005; Kumwilaisak et al. 2003; Voith et al. 2012], 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. Our iterative, feedback-directed approach takes input from all concurrent jobs and prioritizes individual MPI processes to improve performance. The algorithm generally ensures that processes that receive an increase in priority have limited impact on the performance of other processes from the same or other jobs. Our technique handles job arrivals and departures and 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 [Jain et al. 2016]. 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, which means that the 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 2 discusses the problem of network contention and provides background on QoS and fat-tree networks. Section 3 describes our per-process service level assignment algorithm, and Section 4 presents the new TraceR-QoS simulator framework in which we explore its performance. Section 5 details our experimental setup, while Section 6 presents our results.

## 2 OVERVIEW

### 2.1 Network Contention

Network contention occurs when multiple network flows use the same network link concurrently. This sharing causes growth in the length of the switch output queue that connects to the link. 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 [Chunduri et al. 2019].

Contention effects can propagate across a job. For example, process  $P_1$ , which does not experience contention, may be slowed because it waits for a message from process  $P_2$  that does, even if no contention occurs between  $P_1$  and

$P_2$ . Further, if contention causes a message to process  $P$  to arrive later than it otherwise would,  $P$  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 [Guay et al. 2011; Karol et al. 1988; Subramoni et al. 2010] 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 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, for example, allocate jobs in contiguous, non-overlapping network partitions that avoid contention [Krevat et al. 2002]. However, network space-sharing imposes job placement restrictions that reduce node utilization and, typically, overall throughput [Pollard et al. 2018; Zahavi et al. 2016]. Low utilization is antithetical to the goals of many HPC centers, so most systems use shared interconnects.

## 2.2 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* [Reinemo et al. 2006]. 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. 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.

## 2.3 Fat-Trees

Many HPC systems use a fat-tree network topology [Leiserson 1985], 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 1 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  $n$  top-level switches has  $n \times 2$  leaf 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 and typically use static routing to choose among them. The standard “destination mod  $k$ ” or D-mod- $k$  [Gomez et al. 2007] routing algorithm selects the next hop for a message based on

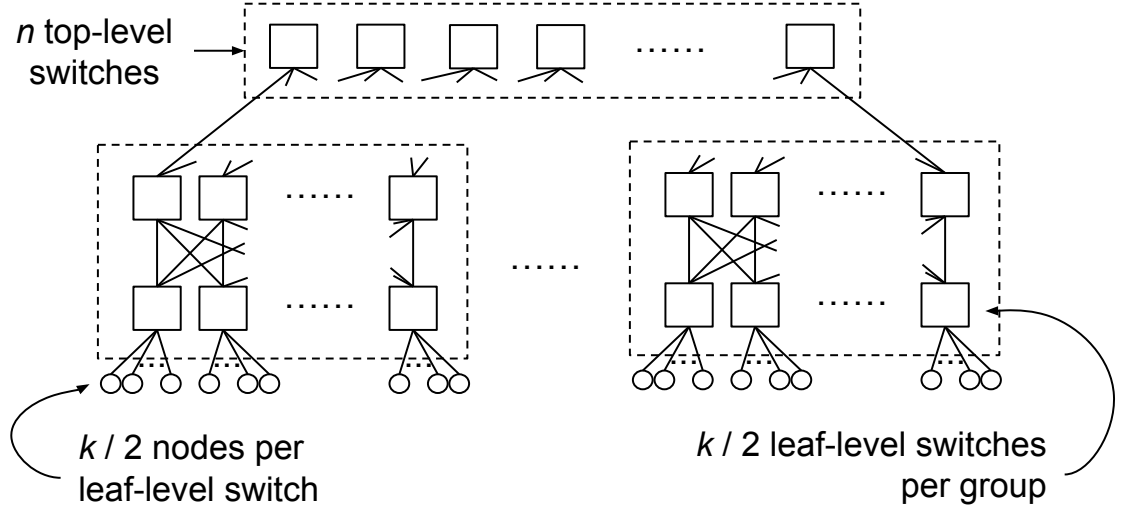


Fig. 1. Three-Level Clos Network,  $n$  Top-Level Switches, Radix  $k$

its final destination. This algorithm disperses routes across the tree to improve performance, but leads to potentially severe network contention [Smith et al. 2018].

### 3 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 use the priority that our algorithm assigns to that process. Our previous work [Savoie et al. 2018] 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 6), while avoiding per-message 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.

#### 3.1 Challenges

As we discussed in Section 2.2, 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.

*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, which the slow processes may not. Since the

```

function MAIN(size)
  INITIALIZE
  for i in iters: do
    DoWORK1
    process results from DoWork1
    DoWORK2
    process results from DoWork2
    ...call more functions and process results ...
  MARKPROGRESS(i)

```

Fig. 2. An example application modified to communicate progress to our system

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.

*Handling Contending Processes:* Our initial set of candidates does not consider the network topology, which can guide further pruning of the candidate set. Suppose two processes, denoted  $P_1$  and  $P_2$ , share a link and thus contend with each other. Prioritizing  $P_1$  will slow down  $P_2$ . If  $P_2$  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  $P_2$ 's job. Similarly, prioritizing  $P_2$  reduces the performance of  $P_1$ '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 on-line 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.

### 3.2 Assumptions

First, we assume a hybrid MPI-OpenMP model for each job, so OpenMP is used to parallelize within a node, and MPI is used to parallelize between nodes. As this work targets fat-tree systems, we assume that the number of nodes is on order of thousands; the fastest supercomputers in the world (Sierra and Summit) currently have less than 5,000 [TOP500.org 2019].

We strive to minimize the 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 marks its progress after each iteration. Figure 2 shows the application-level interface; the call to MarkProgress is placed at the bottom of the loop that is measured. The placement of the call is generally straightforward, and prior work supports automation of its insertion [Kennedy and Kremer 1998]. However, if iterations are extremely coarse grained (e.g., many minutes or even hours), the call must be placed in a nested loop that has finer computational granularity. In this case, we assume that the application programmer performs that task.

We also assume that each process can send packets using 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

**Algorithm 1** Process Prioritization Algorithm

---

```

1: function PROCESS_PRIORITIZATION_ALGORITHM
2:   state ← DEFAULT
3:   assignments ← no processes 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 processes prioritized
27:         else if state = RUNNING then
28:           process ← candidates.pop()
29:           if process in assignments then
30:             increase assignments[process].priority
31:           else
32:             assignments[process] ← 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:     thresh ← min(job.processes.workDone) * (1/T)
40:     for all processes in job do
41:       if process.workDone < thresh then
42:         candidates ← candidates ∪ {process}
return candidates

```

---

uses this capability more often than is currently the norm. Our algorithm currently uses a single, dynamically chosen service level at a time between any pair of processes. If overhead is an issue, we can create a set of service levels statically and have processes choose the proper service level dynamically.

### 3.3 Process Prioritization Algorithm

Algorithm 1 shows that we initially run all jobs in the default configuration (all jobs share a single service level). The cluster administrator must provide  $P$  (line 6) to determine how long the algorithm allows the jobs to run before adjusting service levels. In practice, the administrator would execute benchmarks once to determine the cost of accumulating per-process information over each job and then choose  $P$  so as to amortize overhead. We then record the default performance, which is the amount of work completed by each job (line 9). 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 assignments as the average of each job's improvement over the default run.

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 overall performance becomes worse than the default run (line 23), which handles issues such as large computational noise events that the algorithm otherwise could misinterpret as network contention.

The *SelectCandidates* function selects the algorithm's candidates, which are all processes within a threshold ( $T$ ) of the process that makes the least progress (line 39). The cluster administrator provides  $T$ , which must be a number between 0 and 1. 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 implementation-specific. Variable *NUM\_SLS* (line 19) is the number of available service levels. For example, InfiniBand allows 16 service levels, although most current hardware only implements eight [Crupnicoff et al. 2005]. The *SetServiceLevels* function (line 5) updates the hardware's service levels.

## 4 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 system to which we have access and might require specific vendor support. 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 [Jain et al. 2016], which is an HPC application trace replay tool. It is built on the CODES framework [Cope et al. 2011], which is built on the ROSS parallel discrete event simulator [Carothers et al. 2000]. TraceR simulates MPI applications on HPC networks at the packet level. It efficiently and accurately predicts the performance of important applications on networks with different properties as previous work [Jain et al. 2017] 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://xgbitlab.cels.anl.gov/codes/codes/merge\\_requests/81](https://xgbitlab.cels.anl.gov/codes/codes/merge_requests/81).

#### 4.1 Requesting Per-Process 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 process will use the specified service level. If the sending and receiving processes are explicitly assigned different service levels, TraceR-QoS uses the sender’s service level.

#### 4.2 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. Within each queue we select packets to inject into the network using the round-robin scheme.

#### 4.3 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 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. We applied this change to TraceR-QoS’s fat-tree network implementation but not to the other network topologies that TraceR supports.

#### 4.4 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>.

### 5 EXPERIMENTAL SETUP

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

System	Nodes	Levels	Link B/W	Num. Jobs	Processes/Job
<i>small</i>	64	2	3.2	8	8
<i>medium</i>	324	2	3.2	16	20
<i>large</i>	1296	3	4.0	16	80

Table 1. Experimental Systems; *Levels* is the Height of the Fat-Tree; Bandwidth is in GB/s.

## 5.1 Environment

Table 1 describes the simulated systems with which we test our algorithm in TraceR-QoS. We denote these systems as *small*, *medium*, and *large*. All three systems have fat-tree topologies. We base the link bandwidth on the measured bandwidth of actual systems. Table 1 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* and *large* because TraceR-QoS simulates those tests much faster, which allows us to run a broader range of tests. We include selected results from *medium* and *large* for comparison.

## 5.2 Methodology

As we noted in Section 4.4, we did not integrate our entire algorithm into TraceR. Instead, we run several short simulations, one for each iteration of our algorithm. Thus, only line 6 of Algorithm 1 runs within the simulator. We run 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 where noted.

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 that 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 Lawrence Livermore National Laboratory (LLNL) system with 300 compute nodes, each with two 12-core Intel Xeon E5-2695 CPUs and 128 GB RAM. We compile ScoreP [Knüpfer et al. 2012] into each application as a library to collect traces. During a tracing run, it intercepts MPI calls and records them in the OTF2 format [ParaTools, Inc. 2019].

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 each application, which makes our simulations more realistic. Each application trace uses the same parameters but is run at a different time or on different nodes. Thus, the traces exhibit performance variability, much like repeated iterations of a job over time. For each iteration of our algorithm, we randomly select a trace for each job. 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. These isolation runs do not have inter-job interference; however, congestion can occur due to *intra-job*

interference. This orthogonal problem has been studied previously, especially in terms of communication scheduling [Faraj and Yuan 2005; Patarasuk and Yuan 2007] and topology mapping [Bhatele and Kale 2008; Galvez et al. 2017; Hoefler and Snir 2011]. 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 and the same assignment of jobs to nodes for each of these three runs so we can accurately compare the performance of our algorithm to default and isolated runs. We did not perform experiments in which each *job* was placed in a different service level, because our prior work showed that this use of QoS is never better than executing all jobs at the default service level [Savoie et al. 2018].

We run our tests on eight different randomly chosen node allocations, or assignments of processes to nodes, except where noted. 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 process 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.

### 5.3 Microbenchmarks and Applications

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 [Lawrence Livermore National Laboratory 2014]. 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 four 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 [Bulatov et al. 2004]. 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*.

The *Mini-AMG* micro-benchmark mimics AMG [Gahvari et al. 2011], 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 allows us to experiment with different message sizes and computation-to-communication ratios.

We also include tests with four applications: *AMG*, *Crystal Router*, *ParaDiS*, and *pF3D*. Each is either a production science application or a proxy application designed to mimic a production application. We already discussed *AMG* and *ParaDiS*. *Crystal Router* implements a many-to-many communication pattern extracted from the Nek5000 application [Ibrahim 2019]. We removed DUMPI tracing from the *Crystal Router* code so that we can focus on the communication behavior. *pF3D* simulates laser-plasma interactions [Langer et al. 2014]; its communication is dominated by all-to-all exchanges on a subset of processes. We used a *pF3D* proxy application that emulates only the communication pattern of *pF3D*. To make the application more realistic, we added sleep calls between each communication phase to simulate computation. We tuned the length of the sleep calls so that the computation-to-communication ratio is roughly 50/50 when *pF3D* is running in isolation.

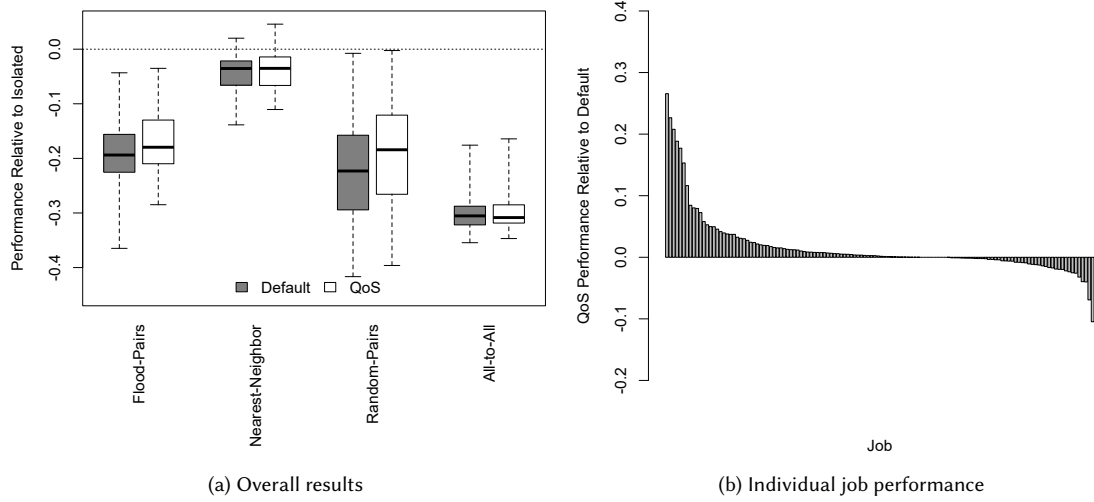
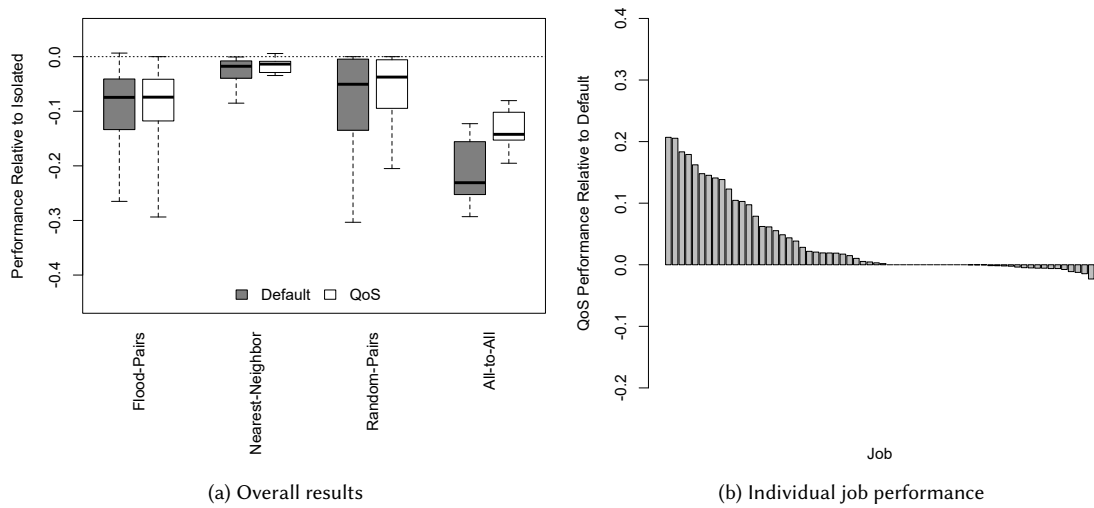
## 6 RESULTS

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

All results in each subsection 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*, in gray) and with our algorithm (labeled *QoS*, in white). To construct the box plot for *Default*, for each job we evaluate  $W = (I_D/I_B) - 1$ , where  $I_D$  and  $I_B$  are the number of iterations completed with default and baseline, respectively. The baseline is each job (separately) executing in isolation. The box plot for *QoS* is constructed the same way, except that  $W = (I_Q/I_B) - 1$ , where  $I_Q$  is the number of iterations completed with our QoS-based technique. For both *Default* and *QoS*, the x-axis lists the job type in the test and the y-axis plots  $W$ , which shows the performance of each job relative to the same job run in isolation (so, negative numbers indicate that the job completed less work than in the isolated run, the most likely scenario; a smaller negative number is better). We mark  $y = 0$ , which indicates that a job achieved the same performance as it did on the isolated run, with a dashed line. Because multiple instances of the same application are identical, we have combined all instances of the same job type across each experiment (which itself has several random node placements) into the same box plot. The second (right) chart computes the difference between *QoS* and *Default* directly at the individual job level; the x-axis is each individual job, and the y-axis is  $(I_Q/I_D) - 1$ . Hence, this chart shows the improvement that our QoS algorithm achieves over the *default* performance for individual jobs, sorted from the greatest improvement to the greatest degradation (so a larger number is better). The y-axis scale is the same for all instances of each chart except where noted.

### 6.1 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, respectively. 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 microbenchmarks; each job has 8 processes. We introduce some computational imbalance in these jobs. The computation time for each process 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.

Fig. 3. Results on the *Medium* System (324 Nodes)Fig. 4. Results on the *Small* System (64 Nodes)

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.

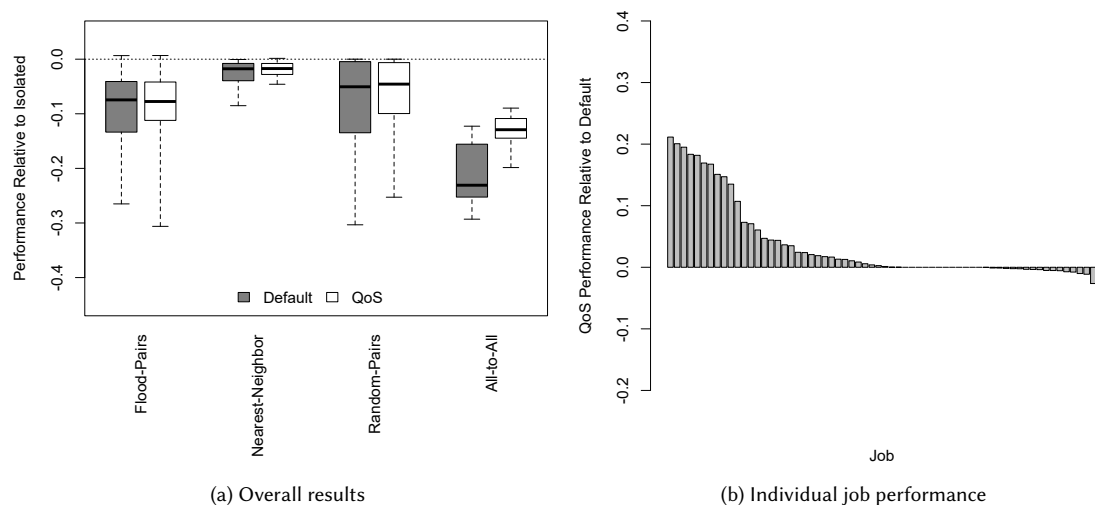


Fig. 5. Results with an observation period of 10 iterations on the *small* machine

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  $P_1$  and  $P_2$  are part of the same job and share a link, so they contend with each other. If  $P_2$  experiences contention from another job,  $P_1$  can send more packets and complete its message faster. If  $P_1$  is on the critical path and  $P_2$  is not, the performance of the job will improve. This situation occurs when the default or QoS case is faster than the isolated case.

## 6.2 Longer Observation Periods

Our next experiment involves studying the performance effect of changing the number of iterations that we observe when calculating the default performance of each job (see line 9 in Algorithm 1). A longer observation period may allow our algorithm to calculate the default performance of each job more accurately, which may result in better performance improvement over the default. In this test, we observe the default performance for  $P \times 10$  seconds and then divide this performance by 10 to estimate the default performance during  $P$  seconds. The results are shown in Figure 5. These results are similar to those in Figure 4, which indicates that the results are relatively independent of the observation period. We perform similar experiments on the *large* machine in Section 6.8.

## 6.3 Imbalance

Next, we use traces with different amounts of computational imbalance than the static ones; the imbalance described in this section is constant across iterations. 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. In the small imbalance results that Figure 6 shows, the amount of computation is randomly chosen between  $0.9 \times Y$  and  $Y$ . In the large imbalance results in Figure 7 (in which we shift the  $y$ -axis), one process does  $Y$  computation while others in the job do none. Together with the static tests, these tests demonstrate that our algorithm performs well with small, modest, and large imbalance.

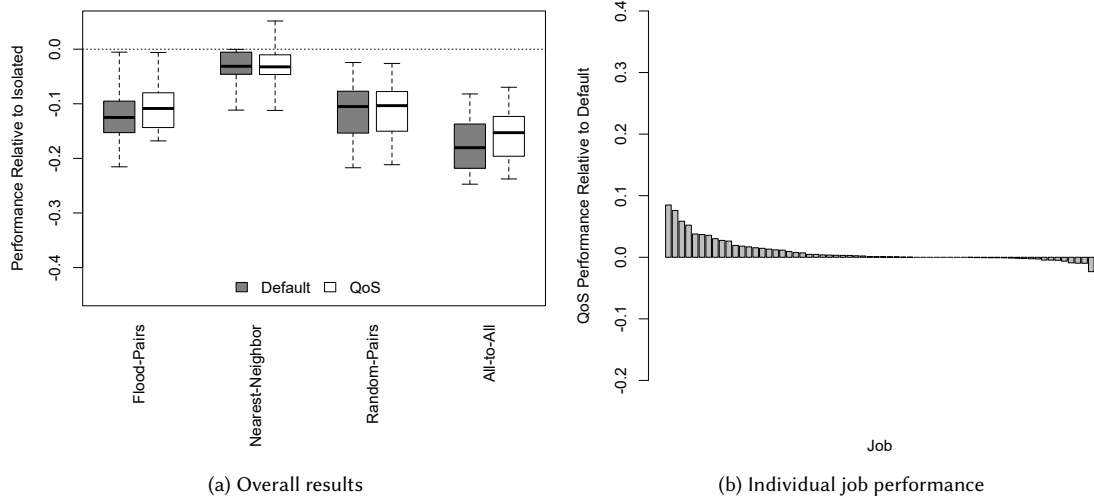


Fig. 6. Results with Small Computational Imbalance

As expected, the results with small computational imbalance show relatively small improvements. This result is expected because our QoS algorithm improves job performance in part by de-prioritizing messages to processes that do little computation. Limited imbalance reduces this opportunity for improvement. The large computational imbalance results show relatively similar improvements as the ones in which the computational load is moderately imbalanced (Figure 4), 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. Overall, given that performance improvement is similar with moderately imbalanced and significantly imbalanced computational loads, these results show that our algorithm does not need extreme imbalance to improve job performance. The results also indicate that the potential additional benefit of large imbalance is limited with a QoS-based scheme.

#### 6.4 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 (i.e., imbalance is across iterations). We use the same test setup as the static test except that we replace the *Random-Pairs* jobs with *Mini-ParaDiS* jobs. Figure 8 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 processes for *Mini-AMG*, so we have one instance of the four static microbenchmarks. Figure 9 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 pattern does not generate much contention. However, the static microbenchmarks obtain improvement and our algorithm handles the irregular communication pattern of *Mini-AMG*.

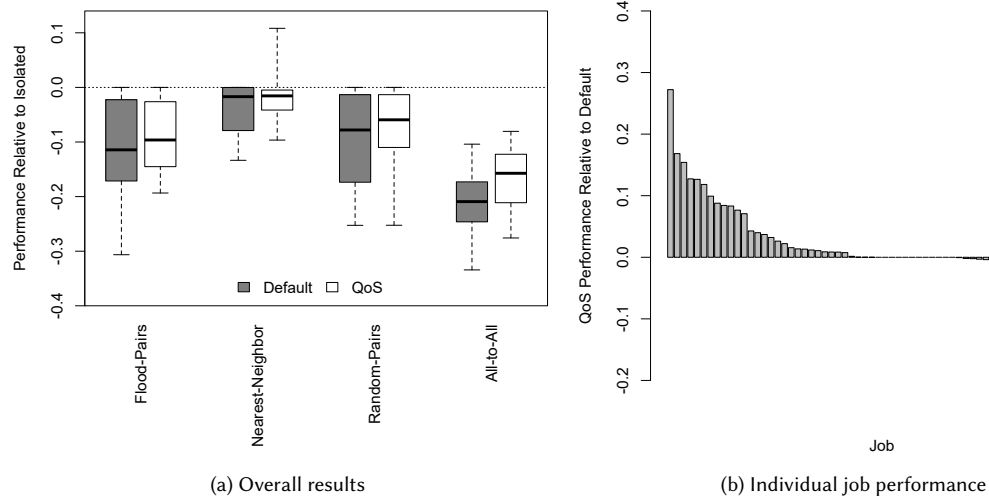
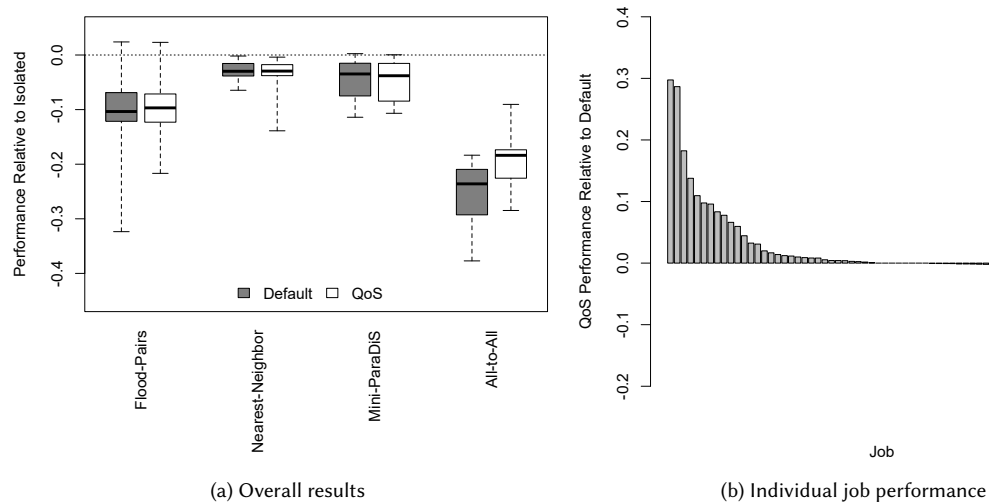


Fig. 7. Results with Large Computational Imbalance

Fig. 8. Results with *Mini-ParaDiS*

### 6.5 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 6.11.

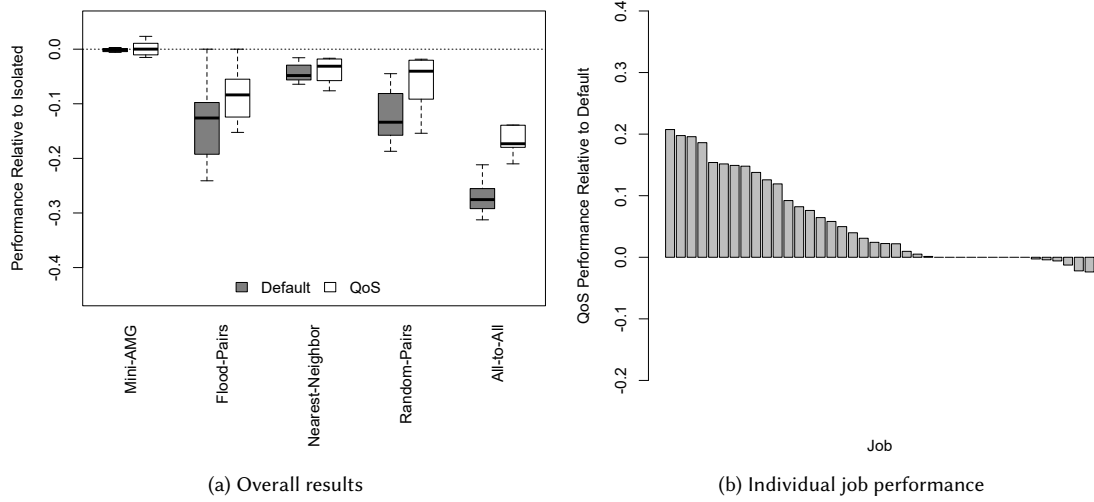
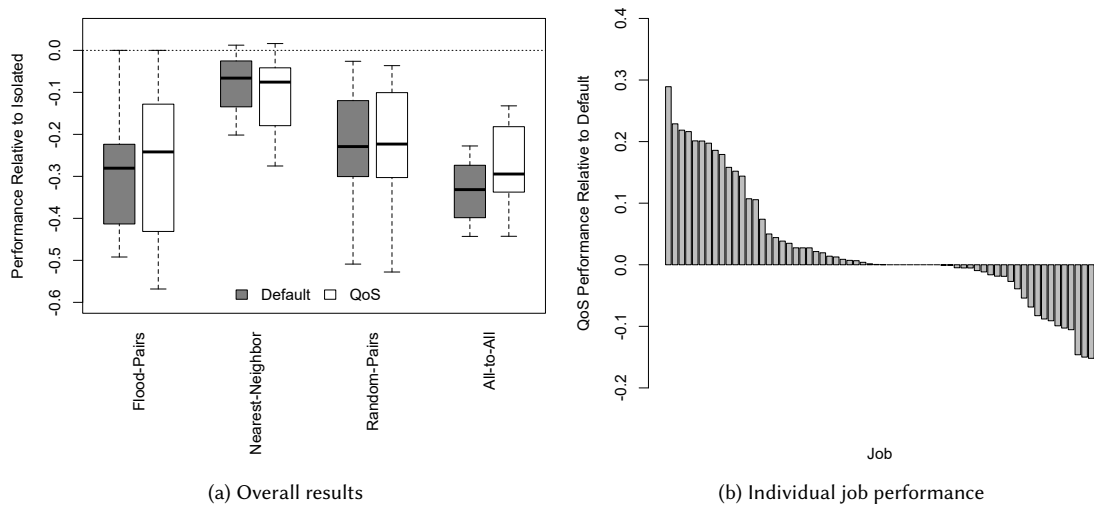
Fig. 9. Results with *Mini-AMG*

Fig. 10. Results with 35% Computation

Figures 10 and 11 show the impact of the computation-to-communication ratio on our algorithm; note the expanded y-axis in Figure 10a. 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 29% 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%.

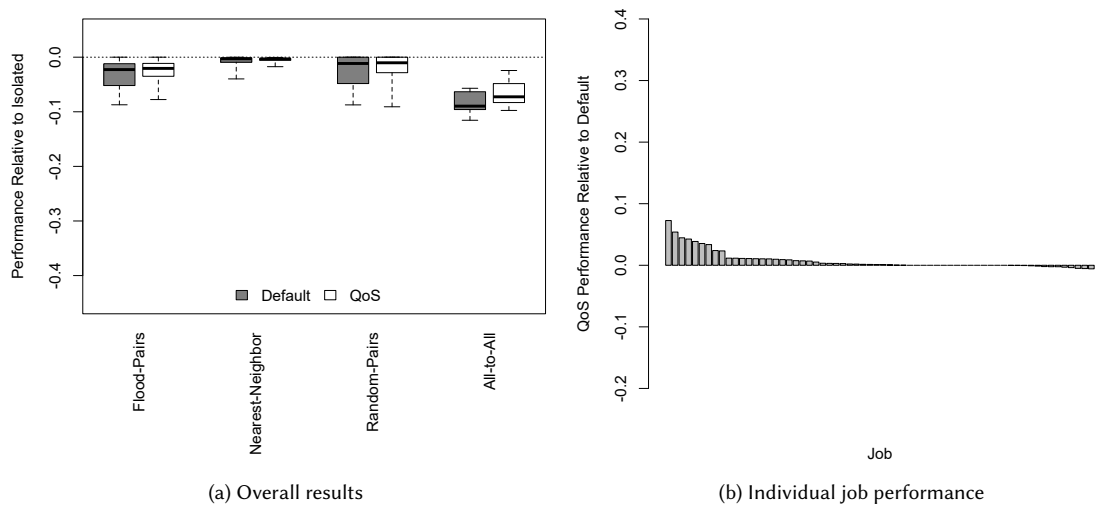


Fig. 11. Results with 90% Computation

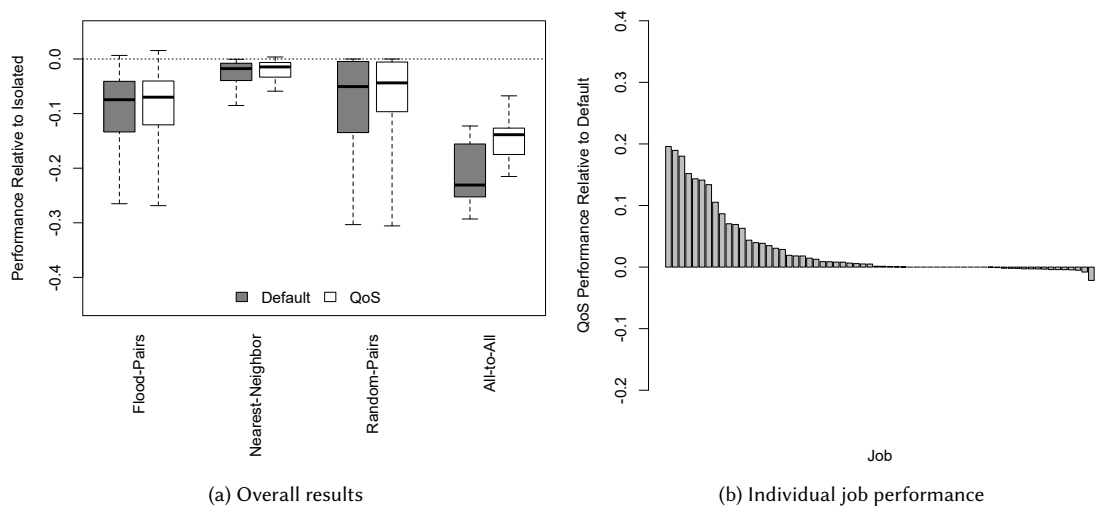


Fig. 12. Results with  $T = 0.4$

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%.

### 6.6 Candidate Threshold

Here, we vary the candidate threshold,  $T$ . Section 3 describes 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 0.4 and 1.0, to complement the static tests with  $T$  set to 0.9.

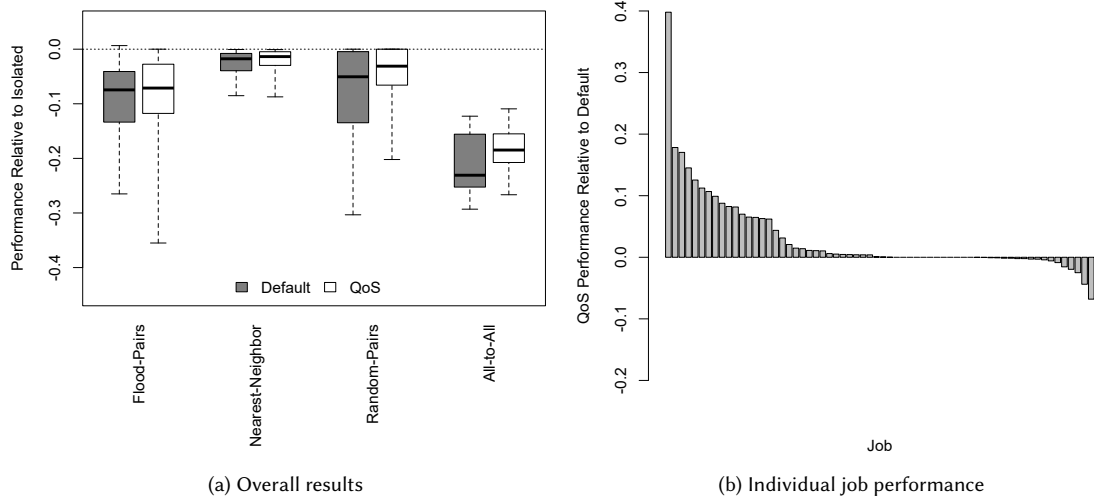
Fig. 13. Results with  $T = 1.0$ 

Figure 12 shows that most jobs have slightly smaller performance improvements with  $T$  set to 0.4. Performance is fairly similar between thresholds of 0.4 and 0.9, with *Flood-Pairs* slightly better at 0.4, *All-to-all* better at 0.9, and *Random-Pairs* and *Nearest-Neighbor* about the same. A threshold of 1.0 produces generally similar median improvements, but the worst-case improvement is significantly lower (said differently, the downside for a threshold of 1.0 is much worse). In a single case, one job experiences a large improvement (see the right-hand side of Figure 13).

In choosing a threshold, one would generally consider that a lower threshold leads to more candidates, which results in higher overhead for our algorithm, whereas a higher threshold leads to fewer candidates, which could result in our algorithm yielding inferior service level assignments. Further experiments showed that all thresholds at or below 0.4 put every process in the candidate set, and that performance is fairly similar for thresholds between 0.4 and 0.9. A threshold of 1.0 places only the slowest candidate in the candidate set and leads to poor worst-case performance. We use 0.9 in our system, given that it has the lowest overhead out of thresholds between 0.4 and 0.9. For much larger systems, more tuning may be needed to find a threshold that leads to good performance in the general case.

## 6.7 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 interval 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 14 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*

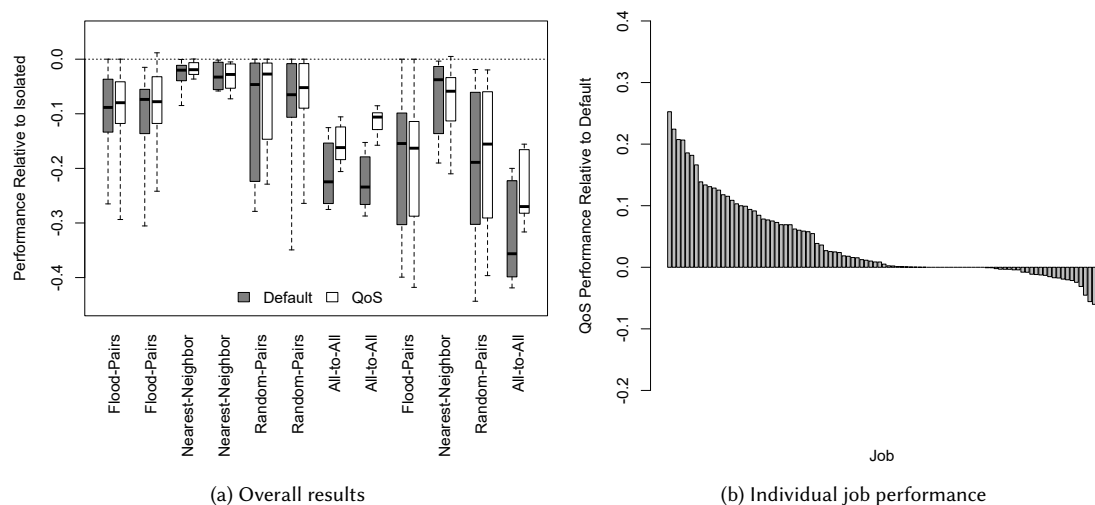


Fig. 14. Results with Arrivals and Departures

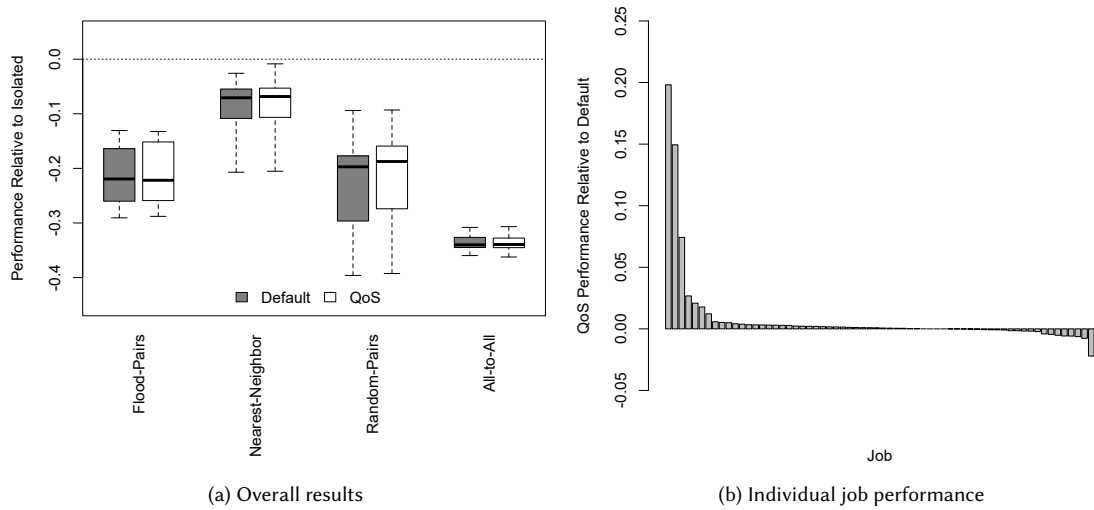
job that does 35% computation. The new job is the 9th job in the chart. After 1000 iterations, we replace the first *Nearest-Neighbor* job with a *Nearest-Neighbor* job that does 35% computation, which is the 10th job in the chart. We perform similar replacements with *Random-Pairs* and *All-to-all* after 1500 and 2000 iterations, and we end the test after 2500 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.

## 6.8 Large System

Figure 15 shows results from the *large* system. This test is similar to the static tests described in Section 6.1. We run 16 jobs; four instances each of *Flood-Pairs*, *Nearest-Neighbor*, *Random-Pairs*, and *All-to-all*, with 80 processes per job. We configured our algorithm to run with the default service levels for  $P \times 10$  seconds as we did in Section 6.2. We run 2350 iterations of our algorithm instead of the 500 iterations used in previous tests so that we can understand our algorithm's performance over longer timescales. We also run four random assignments of processes to nodes (as compared to eight assignments in our other tests) to reduce the simulation time.

Our algorithm has a relatively modest overall effect on the *large* system compared to the *medium* and *small* systems. However, three jobs achieve 20%, 15%, and 7% improvement compared to default on specific placements. Moreover, using our approach has little downside; the other jobs' performance is essentially unchanged. This relatively modest improvement is due to the size of the jobs and the length of our tests. In the discussion that follows we refer to *accepted* service level configurations; our algorithm *accepts* a set of service levels if the jobs perform better with those service levels than with the current best service levels, which is determined on line 15 of Algorithm 1. As the algorithm runs, it accepts several service level configurations that build on each other and gradually improve job performance. Thus, we should see performance improve as the number of accepted service level configurations increases.

Figure 16 shows the performance of all jobs against the number of accepted service level configurations for the *large* tests from this section and the *medium* and *small* tests from Section 6.1. We calculate performance by taking the

Fig. 15. Results on the *Large System* (1296 Nodes)

average improvement across all jobs within a test and then taking the median of the averages across all executions of the same test. As expected, performance improves as the number of accepted service level configurations increases. The improvement is not monotonic since we use different traces for different iterations to simulate non-uniformity within each job. More importantly, however, our algorithm accepts more service level configurations in the *small* and *medium* tests than in the *large* tests. Because of the increased number of processes on the *large* system, the algorithm must search through more service level configurations before it finds one that improves performance. The *large* results track the *medium* and *small* results closely, so we can reasonably assume that the *large* test would achieve results similar to the *medium* and *small* tests if it ran for a longer period of time.

Based on this analysis, large systems reduce the efficiency of our algorithm because of the large number of candidates that it must consider. We may need to reduce the size of the candidate set for large systems. We also note that while our algorithm improves the performance of fewer jobs on the *large* system, it also does not significantly degrade performance. Thus, our algorithm can be deployed without harming throughput even if the job improvements are smaller than desired.

## 6.9 Real Applications

Our next set of results, shown in Figure 17, demonstrates how our algorithm performs with real applications. We run these tests on the *medium* system, and all jobs used 32 processes to ensure that their communication patterns are non-trivial. This test included ten jobs: two instances each of *AMG* and *ParaDiS* and three instances each of *Crystal Router* and *pF3D*.

As with the *large* system results, these results are relatively modest compared to our other results. *AMG* and *ParaDiS* experience little impact from contention, so QoS provided them little improvement. On larger systems, contention and QoS would likely affect these applications to a greater degree. Conversely, *Crystal Router* and *pF3D* experience considerable degradation from contention, but our algorithm did not significantly improve or degrade the performance of any job.

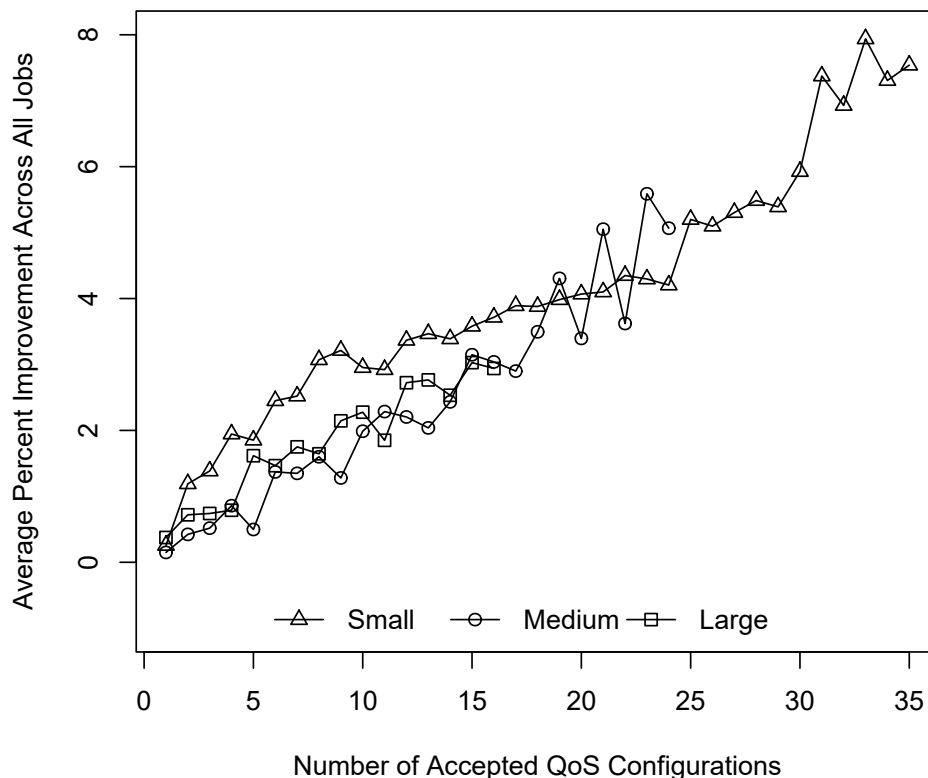


Fig. 16. Improvement Relative to Algorithm Progress

To better understand our algorithm’s performance with real jobs in high contention scenarios, we replaced the *AMG* and *ParaDiS* jobs with microbenchmarks. For simplicity, we replaced each 32-process real application job with four 8-process *Flood-Pairs* or *Nearest-Neighbor* jobs. These results are shown in Figure 18. In this test several jobs achieved 6% improvement with a worst case of 3% degradation, which is an improvement over the results in Figure 17 but still relatively modest compared to our other results. The larger size, more complex communication patterns, or computational imbalance of the real jobs may have affected our results.

### 6.10 Adaptive Routing

All of our tests to this point have used static routing. Newer fat trees, such as those found (currently) on the two fastest supercomputers in the world, Sierra and Summit, have hardware adaptive routing [Vazhkudai et al. 2018]. With adaptive routing, local information at each switch attempts to detect congestion and subsequently routes packets over paths with less congestion. We investigated the effect of adaptive routing on our workloads using the built-in adaptive routing engine provided by TraceR. Our tests use the same methodology described in Section 5.2, which means that adaptive routing must “kick in” on each iteration, but separate experiments showed that this made little difference.

Figure 19 shows the results on the workload from Figure 4 but with adaptive routing (gray bars). For ease of comparison, we also show the results with QoS (repeated from Figure 4). Neither the *Default* box plots nor individual

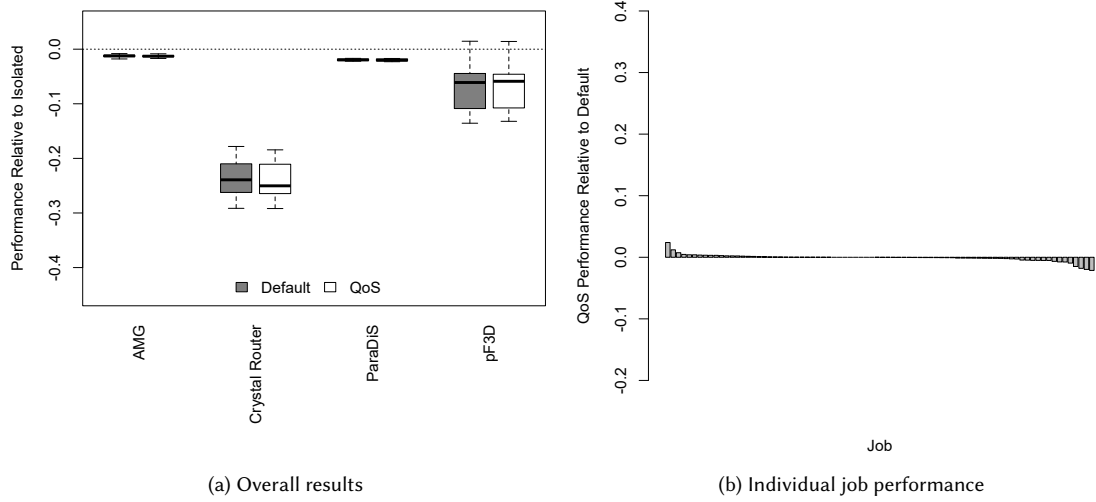


Fig. 17. Results with Real Applications

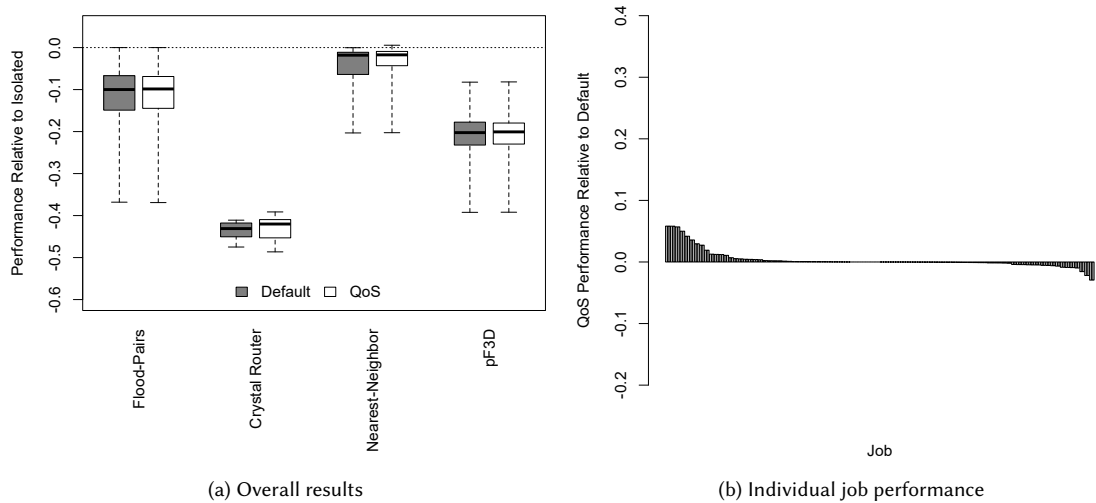


Fig. 18. Results with Real Applications and Microbenchmarks

job improvements compared to *Default* are shown. The results are mixed; adaptive routing results in better performance on *Flood-Pairs* and *Random-Pairs* but worse performance on *Nearest-Neighbor* and *All-to-all*. In addition, the two approaches are complementary; while beyond the scope of this paper, they could be combined. Overall, however, most current fat-tree systems use static routing, so techniques such as our QoS algorithm are needed to mitigate network contention.

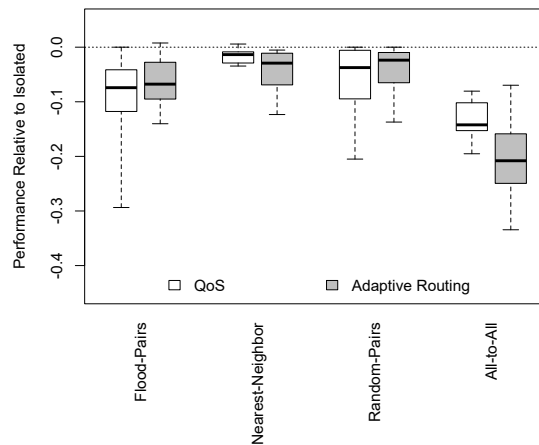


Fig. 19. Results with adaptive routing on the *small* system

### 6.11 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 because 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.

## 7 RELATED WORK

QoS research has a long history. The internet protocol (IP) implements differentiated services [Blake et al. 1998], in which flows are divided into classes that are handled differently. This technique is an example of traffic prioritization, the QoS mechanism that we use. QoS has also been applied in areas as diverse as wireless networking [Andrews et al. 2001], data centers [Voith et al. 2012], and video streaming [Ke et al. 2005; Kumwilaisak et al. 2003].

QoS use in HPC networks is limited. Researchers have used QoS to minimize HOL blocking [Guay et al. 2011; Subramoni et al. 2010], 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 [Jokanovic et al. 2012]. 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. Others have used QoS on a simulated Megafly network, which is a variant of the dragonfly [Mubarak et al. 2019]. They applied QoS at the job level and by separating collective and point-to-point communication into different service levels. These job-level QoS schemes are similar to our prior work (see next paragraph); this paper is fundamentally different in that we use process-level granularity.

We previously presented the first empirical investigation of QoS and its impact on network contention on an HPC system [Savoie et al. 2018]. 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 coarse-grained prioritization degrades the performance of other jobs, which often reduces overall throughput. Our study is in contrast to the simulation-based work [Jokanovic et al. 2012; Mubarak et al. 2019] 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 [Smith et al. 2018], dragonfly networks [Chunduri et al. 2017; Smith et al. 2018; Yang et al. 2016], and tori [Bhatele et al. 2013]. Adaptive routing [Jain et al. 2014] seeks to route traffic away from congested links. However, adaptive routing on dragonflies does not necessarily reduce contention significantly [Smith et al. 2018]. Hardware adaptive routing on fat-trees has been introduced recently on Summit and Sierra [Vazhkudai et al. 2018]. However, unlike our QoS solution, hardware adaptive routing necessarily lacks full global information. Other research into adaptive routing at the global level [Smith et al. 2018] 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 [Jokanovic et al. 2015; Yang et al. 2016; Zahavi et al. 2016]. However, these approaches do not eliminate contention within a job, and they typically reduce system utilization. Further, any of these approaches could be combined with QoS.

## 8 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 impact of network contention. We have introduced an algorithm that uses per-process 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 that we have introduced in this paper as network performance becomes a larger component of overall system performance.

## ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-813440). In addition, this material is based upon work supported by the National Science Foundation under Grant No. 1526015.

## REFERENCES

- M. Andrews, K. Kumaran, K. Ramanan, A. Stolyar, P. Whiting, and R. Vijayakumar. 2001. Providing Quality of Service over a Shared Wireless Link. *IEEE Communications Magazine* 39, 2 (Feb. 2001), 150–154. <https://doi.org/10.1109/35.900644>
- Abhinav Bhatele and Laxmikant V Kale. 2008. Application-Specific Topology-Aware Mapping for Three Dimensional Topologies. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–8.
- Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. 2013. There Goes the Neighborhood: Performance Degradation due to Nearby Jobs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*.
- S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. 1998. An Architecture for Differentiated Service. <https://tools.ietf.org/html/rfc2475>. Accessed: August 3, 2019.
- Vasily Bulatov, Wei Cai, Masato Hiratani, Gregg Hommes, Tim Pierce, Meijie Tang, Moono Rhee, Kim Yates, and Tom Arsenlis. 2004. Scalable Line Dynamics in ParaDiS. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '04)*.
- Christopher D. Carothers, David Bauer, and Shawn Pearce. 2000. ROSS: A High-Performance, Low Memory, Modular Time Warp System. In *Proceedings of the Workshop on Parallel and Distributed Simulation (PADS '00)*.
- Sudheer Chunduri, Taylor Groves, Peter Mendygral, Brian Austin, Jacob Balma, Krishna Kandalla, Kalyan Kumaran, Glenn Lockwood, Scott Parker, Steven Warren, Nathan Wichmann, and Nicholas Wright. 2019. GPCNeT: Designing a Benchmark Suite for Inducing and Measuring Contention in HPC Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*.
- Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. 2017. Run-to-Run Variability on Xeon Phi Based Cray XC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*.
- J Cope, N Liu, S Lang, P Carns, C Carothers, and Robert Ross. 2011. CODES: Enabling Co-design of Multilayer Exascale Storage Architectures. In *Proceedings of the Workshop on Emerging Supercomputing Technologies*.
- Diego Crupnicoff, Sujal Das, and Eitan Zahavi. 2005. *White Paper: Deploying Quality of Service and Congestion Control in InfiniBand-Based Data Center Networks*. Technical Report 2379. Mellanox Technologies Inc., 2900 Stender Way, Santa Clara, CA 95054.
- A. Faraj and X. Yuan. 2005. Message Scheduling for All-to-All Personalized Communication on Ethernet Switched Clusters. In *IEEE International Parallel and Distributed Processing Symposium*.
- Hormozd Gahvari, Allison H. Baker, Martin Schulz, Ulrike Meier Yang, Kirk E. Jordan, and William Gropp. 2011. Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In *Proceedings of the International Conference on Supercomputing (ICS '11)*.
- Juan J. Galvez, Nikhil Jain, and Laxmikant V. Kale. 2017. Automatic Topology Mapping of Diverse Large-Scale Parallel Applications. In *Proceedings of the International Conference on Supercomputing*.
- C. Gomez, F. Gilbert, M.E. Gomez, P. Lopez, and J. Duato. 2007. Deterministic versus Adaptive Routing in Fat-Trees. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '07)*.
- W. L. Guay, B. Bogdanski, S. A. Reinemo, O. Lysne, and T. Skeie. 2011. vFtree - A Fat-Tree Routing Algorithm Using Virtual Lanes to Alleviate Congestion. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '11)*.
- Torsten Hoeffler and Marc Snir. 2011. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 75–84.
- Khaled Z. Ibrahim. 2019. Characterization of the DOE Mini-apps. <https://portal.nersc.gov/project/CAL/designforward.htm>. Accessed: June 11, 2019.
- Nikhil Jain, Abhinav Bhatele, Louis H. Howell, David Böhme, Ian Karlin, Edgar A. León, Misbah Mubarak, Noah Wolfe, Todd Gamblin, and Matthew L. Leininger. 2017. Predicting the Performance Impact of Different Fat-Tree Configurations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*.
- Nikhil Jain, Abhinav Bhatele, Xiang Ni, Nicholas J. Wright, and Laxmikant V. Kale. 2014. Maximizing Throughput on a Dragonfly Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*.
- Nikhil Jain, Abhinav Bhatele, Sam White, Todd Gamblin, and Laxmikant V. Kale. 2016. Evaluating HPC Networks via Simulation of Parallel Workloads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. <http://dl.acm.org/citation.cfm?id=3014904.3014923>

- A. Jokanovic, J. C. Sancho, J. Labarta, G. Rodriguez, and C. Minkenber. 2012. Effective Quality-of-Service Policy for Capacity High-Performance Computing Systems. In *Proceedings of the International Conference on High Performance Computing and Communication and International Conference on Embedded Software and Systems (HPCC-ICES '12)*.
- A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenber, and J. Labarta. 2015. Quiet Neighborhoods: Key to Protect Job Performance Predictability. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '15)*.
- Mark J. Karol, Michael G. Hluchyj, and Samuel P. Morgan. 1988. Input Versus Output Queuing on a Space-Division Packet Switch. *IEEE Transactions on Communications* COM-35 (01 1988), 1347 – 1356. <https://doi.org/10.1109/TCOM.1987.1096719>
- Chih-Heng Ke, Ce-Kuen Shieh, Wen-Shyang Hwang, and A. Ziviani. 2005. A Two Markers System for Improved MPEG Video Delivery in a DiffServ Network. *IEEE Communications Letters* 9, 4 (April 2005), 381–383. <https://doi.org/10.1109/LCOMM.2005.1413641>
- Ken Kennedy and Ulrich Kremer. 1998. Automatic Data Layout for Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems* 20, 4 (1998), 869–916.
- Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, 79–91.
- Elie Krevat, José G. Castañón, and José E. Moreira. 2002. Job Scheduling for the BlueGene/L System. In *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSPP '02)*. <http://dl.acm.org/citation.cfm?id=646383.689703>
- W. Kumwilaisak, Y. T. Hou, Qian Zhang, Wenwu Zhu, C. C. J. Kuo, and Ya-Qin Zhang. 2003. A Cross-Layer Quality-of-Service Mapping Architecture for Video Delivery in Wireless Networks. *IEEE Journal on Selected Areas in Communications* 21, 10 (Dec 2003), 1685–1698. <https://doi.org/10.1109/JSAC.2003.816445>
- S. H. Langer, A. Bhatele, and C. H. Still. 2014. pF3D Simulations of Laser-Plasma Interactions in National Ignition Facility Experiments. *Computing in Science Engineering* 16, 6 (Nov 2014), 42–50. <https://doi.org/10.1109/MCSE.2014.79>
- Lawrence Livermore National Laboratory. 2014. CORAL Benchmark Codes. <https://asc.llnl.gov/CORAL-benchmarks/>. Accessed: June 11, 2019.
- C.E. Leiserson. 1985. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers* 34, 10 (October 1985).
- Misbah Mubarak, Neil McGlohon, Malek Musleh, Eric Borch, Robert Ross, Ram Huggahalli, Sudheer Chunduri, Scott Parker, Kumar Kalyan, and Christopher Carothers. 2019. Evaluating Quality of Service Traffic Classes on the Megafly Network. In *Proceedings of the International Supercomputer Conference (ISC '19)*.
- ParaTools, Inc. 2019. Open Trace Format. <http://www.paratools.com/otf/>. Accessed: May 1, 2019.
- P. Patarasuk and X. Yuan. 2007. Bandwidth Efficient All-Reduce Operation on Tree Topologies. In *Workshop on High-level Parallel Programming Models and Supportive Environments*.
- Samuel A. Pollard, Nikhil Jain, Stephen Herbein, and Abhinav Bhatele. 2018. Evaluation of an Interference-Free Node Allocation Policy on Fat-Tree Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*.
- S. A. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken. 2006. An Overview of QoS Capabilities in InfiniBand, Advanced Switching Interconnect, and Ethernet. *IEEE Communications Magazine* 44, 7 (July 2006), 32–38. <https://doi.org/10.1109/MCOM.2006.1668378>
- Lee Savoie, David K. Lowenthal, Bronis R. de Supinski, and Kathryn Mohror. 2018. A Study of Network Quality of Service in Many-Core MPI Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW '18)*.
- L. Savoie, D. K. Lowenthal, B. R. de Supinski, K. Mohror, and N. Jain. 2019. Mitigating Inter-Job Interference via Process-Level Quality-of-Service. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–5.
- Staci Smith, Clara Cromeey, David K. Lowenthal, Jens Domke, Nikhil Jain, and Abhinav Bhatele. 2018. Mitigating Inter-Job Interference Using Adaptive Flow-Aware Routing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*.
- Hari Subramoni, Ping Lai, Sayantan Sur, and Dhableswar K. (DK) Panda. 2010. Improving Application Performance and Predictability Using Multiple Virtual Lanes in Modern Multi-core InfiniBand Clusters. In *Proceedings of the International Conference on Parallel Processing (ICPP '10)*.
- TOP500.org. 2019. Top500 List - June 2019. <https://www.top500.org/lists/2019/06/>. Accessed: August 2, 2019.
- Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp H. Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. 2018. The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*.
- Thomas Voith, Karsten Oberle, and Manuel Stein. 2012. Quality of Service Provisioning for Distributed Data Center Inter-Connectivity Enabled by Network Virtualization. *Future Generation Computer Systems* 28, 3 (2012), 554 – 562. <https://doi.org/10.1016/j.future.2011.03.011>
- Xu Yang, John Jenkins, Misbah Mubarak, Rob B. Ross, and Zhiling Lan. 2016. 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)*.

Eitan Zahavi, Alexander Shpiner, Ori Rottenstreich, and Isaac Keslassy. 2016. Links as a Service (LaaS): Guaranteed Tenant Isolation in the Shared Cloud. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems (ANCS '16)*.