Measuring Network Latency Variation Impacts to High Performance Computing Application Performance

Robert Underwood
Clemson University
Clemson, South Carolina
robertu@clemson.edu

Jason Anderson
Clemson University
Clemson, South Carolina
jwa2@clemson.edu

Amy Apon
Clemson University
Clemson, South Carolina
aapon@clemson.edu

ABSTRACT

In this paper, we study the impacts of latency variation versus latency mean on application runtime, library performance, and packet delivery. Our contributions include the design and implementation of a network latency injector that is suitable for most QLogic and Mellanox InfiniBand cards. We fit statistical distributions of latency mean and variation to varying levels of network contention for a range of parallel application workloads. We use the statistical distributions to characterize the latency variation impacts to application degradation. The level of application degradation caused by variation in network latency depends on application characteristics, and can be significant. Observed degradation varies from no degradation for applications without communicating processes to 3.5 times slower for communication-intensive parallel applications. We support our results with statistical analysis of our experimental observations. For communication-intensive high performance computing applications, we show statistically significant evidence that changes in performance are more highly correlated with changes of variation in network latency than with changes of mean network latency alone.

CCS CONCEPTS

• **Networks** → **Network experimentation**; **Network performance analysis**; **Network measurement**; • **Hardware** → **Testing with distributed and parallel systems**;

KEYWORDS

Network Latency Variation; Low Latency Networks; Parallel Application Performance; Network Load Injector; Statistical Analysis

ACM Reference Format:
Robert Underwood, Jason Anderson, and Amy Apon. 2018. Measuring Network Latency Variation Impacts to High Performance Computing Application Performance. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184427

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184427

1 INTRODUCTION

High performance computing (HPC) depends on the performance of the underlying network. Indeed, there has been extensive research on the development of low-latency, highly performing networks for HPC [7, 8, 28], and research has demonstrated the considerable effect of average network latency on the performance of HPC applications [14, 31]. Because compute nodes are increasingly capable, with larger numbers and types of cores and memory, network resources are under increased contention creating competition for these resources and increasing the variation in network communication time. In this paper, we demonstrate that network latency variation by itself can have a significant effect on HPC workload runtime. That is, with equal mean latency, a higher variation in the network latency can result in significantly lower HPC application performance.

This paper focuses on the effects of network latency variation on HPC performance degradation. We present the design and implementation of a network latency injector that is suitable for most QLogic and Mellanox InfiniBand cards. We execute sets of experiments at the packet, library, and application levels to measure and model the latency distributions. We then synthetically produce latency using our developed latency injector in a controlled experimental environment and measure these effects.

Our developed tool and approach confirm prior research that has focused on the packet and library level. At the packet level, increasing of mean network latency affects performance, but point-to-point communication is less affected by the variation in network latency. At the library level, latency variation affects the runtime of collective operations, particular those that involve most or all nodes in the computation [12, 13].

We present new results at the application level. First, we characterize the negative impact that latency variation has to the performance of classes of communication-intensive applications. The decrease in performance ranges up to 3.5 times slower for LU Decomposition [3], for example. Next, we show that for communication-intensive HPC applications, changes in performance are more highly correlated with changes of variation in network latency than with changes of mean network latency alone. These results have implications for the design of HPC applications that must execute in a highly shared environment, say, using commercial cloud resources or in a multi-tenant environment, and suggest that implementation of mechanisms to control network variation latency may lead to better overall application and system performance than efforts to reduce average network latency alone. Our main contributions are:

- the design and implementation of a configurable latency injector for many Mellanox and QLogic InfiniBand cards,

- characterizing the distributions of network latency for an InfiniBand network in an HPC environment,
- an experimental methodology using synthetically generated latency to demonstrate the effects of latency variation on HPC workloads, and
- statistically significant evidence that latency variation is more highly correlated with HPC application performance than latency mean alone.

The remainder of this paper is organized as follows. We present background information on network latency sources and low-latency networking in Section 2. We describe our design choices and implementation tradeoffs of the latency injector in Section 3. We provide our overall experimental methodology in Section 4, and describe our workload characterization in Section 5. We show how latency variation apart from congestion can cause performance degradation in Section 6. We demonstrate that latency variation can be more highly correlated with application degradation than latency mean in Section 7. We present an overview of other works studying latency variation in Section 8. Finally, we provide conclusions and future work in Section 9.

2 BACKGROUND

Prior research has shown that congestion-induced latency variation can have significant effects on application performance [5]. This is straightforward to observe; for example, in the `MPI_Barrier` routine, no process can continue until all of the processes entering the barrier have completed the synchronization step. Thus, the time to complete the barrier call is determined by the process that takes the longest time to enter and complete the barrier [13]. In networks with high latency variation, the time to synchronize to a barrier can be considerable [9].

2.1 Sources of Latency Variation

The causes of latency variation in an HPC environment can include single node hardware and software, operating system policies, and resource contention. Features of a single compute node that can be a source of variation of the network latency include differences in cores and changes in the assignment of tasks to cores during execution that affect CPU rate or cache locality [25], and proximity of the network interface card to cores [29].

Operating system resource management policies can affect performance. An example is task scheduling that switches an actively communicating task, in which case the context switch can take an order of magnitude more time than sending a message using user-level networking [32]. The problem is exacerbated in virtual machine-based multi-tenant environments because the entire virtual machine may be switched [22]. Other factors include contention for system resources such as operating system locks, library/interprocess locks, device access locks, bus access, and network interface buffers [20].

Variation in the hardware can be a source of variation in network latency, such as when a CPU disables some cores in order to conserve power when not in use [21]. Factors that affect compute nodes can also affect intermediate nodes or network devices such as routers and switches in which buffers used during the routing of packets can be a source of contention [18]. Network stalls, where a packet is dropped due to a busy receiver or full buffer, have been evidentially identified as an important predictor in parallel application performance [6].

2.2 Low-latency Networking

Efforts to reduce the mean latency in HPC networks have resulted in a lengthy history of research and developed products such as Myrinet and InfiniBand. In this section, we provide background on InfiniBand and describe features that reduce both the mean and variation of network latency.

InfiniBand is the dominant low-latency network fabric for HPC. There are several factors that make it well suited for low latency networking. First, InfiniBand utilizes a zero-copy protocol. Traditional network stacks such as Ethernet are implemented such that multiple copies of each packet are made into and out of intermediate buffers. For example, to send the contents of a buffer from a user-space application, it is first copied to a kernel buffer, then to a network interface buffer, transmitted across the network, copied into a kernel buffer on the destination machine, then copied from the kernel buffer to a user-space buffer on the destination machine. With zero-copy protocols, the application allocates memory on the source and destination network interfaces. It then writes the information directly to the network interface, transmits it across the network, then reads it directly from the network interface card. Since there are fewer copies and buffers involved, the messages can be transmitted more quickly and with less variation that a traditional network can provide.

Performance is improved by the kernel bypass feature of InfiniBand. In an Ethernet stack, the operating system kernel maintains locks and controls for the network interface. This requires a trap into the kernel whenever a packet is to be sent. This trap introduces overhead and provides the kernel an opportunity to call `schedule()` or otherwise change contexts. These context switches can be expensive since registers and other process state must be saved prior to the trap. In comparison, InfiniBand allows the user to control the transmission in user-space using a special driver design that moves all privileged (kernel-mode) operations to the setup and tear down phases of the card for the process. This allows the user to write to a memory-mapped register on the card to send a packet. This is often accomplished by a thin-wrapper library called `libibverbs` which delegates to a device specific driver library using a macro to ensure in-lining of the functions. These optimizations allow for more efficient transmission of packets because variable length traps are not required.

Finally, InfiniBand performs network-offloading. Once the user writes to the register that posts a particular message, the network interface takes ownership of the task to send the message, and the CPU can continue to process other instructions. This allows for the computer to enqueue messages quickly and consistently rather than waiting for the variable length messages to be sent.

While InfiniBand and similar low-latency networking technologies remove sources of latency variation that can be attributed to the OS kernel, contention for the network device by competing processes introduces nondeterministic access delays that can result in long-tailed latency distributions in packet delivery.

3 DESIGN AND IMPLEMENTATION OF THE LATENCY INJECTOR

A key contribution of this paper is the design and implementation of the latency injector that we use in our experiments. This injector allows us to introduce delay to outgoing packets so that we can observe the effects on higher level applications. Most importantly, the delay added to each packet is sampled from a random variable, the parameters of which can model patterns observed in a real system.

We implemented the latency injector for InfiniBand cards that use the `ipath` and `mlx4` drivers. However, the design is reasonably portable to any `libibverbs` compatible driver. The injector is programmed with four design goals. The injector should:

- avoid unnecessary performance impact to applications,
- support custom latency distributions,
- allow changing distribution parameters without recompiling the library, and
- not require changes to or recompilation of applications.

3.1 Injection Method

The injector consists of hooks on two methods, `init` and `post_send`. The names for these methods vary in InfiniBand implementations, but are responsible for initializing the user-space driver and posting a packet to the dispatch queue, respectively.

Hooking init. There are three key facets of the hooking `init` method, shown in Algorithm 1. First, we only load the distribution file into static memory at library initialization to avoid the overhead of kernel operations in later calls. This also allows us to easily customize the latency distribution for each experiment without recompilation. Secondly, we also load a seed file. This allows us to obtain consistent random results between runs of the experiments. Third, we disable the library if we fail to load the distribution or the seed file. This allows for easily alternating between injected and non-injected experiments.

Algorithm 1 init

```
enabled ← exists(dist_file)&&exists(seed_file)
if enabled then
    dist_table ← load_table(dist_file)
    seed ← load_seed(seed_file)
else
    warn_user()
end if
```

The distribution file contains 512 lines of eight space-separated integers. The entries in this file correspond to amounts of delay in units to add to each request. This design is consistent with the `tc` component of the `netem` command which provides similar behavior for Ethernet networks. The seed file contains a single integer corresponding to the random seed to be loaded.

Hooking post_send. Despite the name of the method, `post_send` does not occur after a send occurs, but rather before. It is called to add an RDMA instruction to the queue of instructions to be processed.

Algorithm 2 post_send

```
if enabled then
    index ← random_index()
    delay ← dist_table[index]
    i ← 0
    while i < delay do
        i ← i + 1
    end while
end if
```

There are several key facets to the `post_send` function, shown in Algorithm 2. First, we do not simply use `sleep()` or `pselect()` + `SIGALRM` to implement a micro-second sleep. This is due to the implementation in the Linux kernel of sleep and select. Both of these calls trigger a trap into the kernel, defeating the purpose of user-space InfiniBand networking. Additionally, even if the trap overhead is insignificant, the kernel calls `schedule()` during both of these calls. These operations result in unpredictable sleep times when events are measured in the microsecond range of InfiniBand latency. Instead, we implement a busy wait, incrementing a static volatile variable to ensure that operations to the index are never optimized. This results in a user-space sleep operation over which we have reasonably precise control.

Secondly, instead of using a kernel-based source of randomness we utilize a constant time linear congruential pseudo-random generator. This is to avoid a trap into the kernel for entropy, but also to ensure a consistent length operation. Linear congruential generators also have the nice property of requiring little state, thus leaving the cache clean for other variables.

We then use the lower 12 bits to index into the distribution table, letting us avoid an expensive division operation. This also allows us to customize the distribution of latency without having to compute values from this distribution at runtime.

We generate distribution files off-line using a Python application and SciPy distributions [17]. This gives access to a variety of high quality distribution sampling routines for various distributions.

We choose to add the latency to the top of the `post_send` method for two reasons. First, the lock for interacting with the RDMA memory is not grabbed until later in the function. This allows for multiple threads to use RDMA without waiting on a locked thread that is busy waiting. Secondly, this happens after our timing instrumentation occurs so that the time spent in the busy wait is included in the measurements of the InfiniBand verb latency.

3.2 Validation

To verify that the latency injector works as intended, we collected measurements of synthetic background load as described in Section 4, fitted both uniform and log-normal distributions to each collected dataset, generated distribution tables according to the fit parameters, and then performed the same measurement while substituting injected latency for background load. The mean injected latency closely emulates the measured characteristics of each background load pattern.

3.3 Applicability to Other Networks and Hardware

One possible alternative design is to instrument `libibverbs`. All InfiniBand implementations use the `libibverbs` abstraction to have a common interface to the user-space drivers. However, the `post_send` method is implemented as a macro, which means that applications would need to be recompiled in order to be able to use our version of the library. Since many InfiniBand applications have closed source components, we did not choose this option.

One other possible design is to simply use a longer cable to create higher latency. We did not choose this alternative for the obvious inconvenience of requiring more than 40 different cables for our experimental suite. In addition, the various required lengths of cable required are not available as commercial–off–the–shelf products, and cables do not allow us to artificially inject latency variation that is needed for our final suite of experiments.

One question that arises is how much effort is required to port these changes to other InfiniBand libraries. We ported our library to Mellanox cards that use the `mlx4` driver. This case required finding the names of the `init` and `post_send` methods, adding the instrumentation code to the library, and recompiling.

4 ENVIRONMENT AND METHODOLOGY

In this section we describe the methodology used to create a controlled test environment for later experiments. Measurements of low-latency networks are fine-grained and sensitive to perturbations by other systems. Our goal is to minimize or eliminate noise in our testbed and to collect high quality measurements to ensure that relationships between independent and dependent variables are correctly characterized.

4.1 Experimentation Environment

4.1.1 Hardware Configuration. We ran our experiments on Cloudlab c8220 nodes [30], which were equipped as outlined in Table 1. This particular hardware was chosen to provide enough cores to support our MPI experiment configurations with one core per process, and adequate memory per process for each benchmark. To allow addition of an artificial latency generator in the network device driver, we chose hardware with QLogic InfiniBand cards that have an open source user-space driver.

We additionally validated our experiments on Cloudlab c6320 nodes. These nodes share the characteristics we identify as ideal for our experiments, and are equipped as outlined in Table 2. Results were similar and produced identical conclusions between the two hardware types, so the c6320 results have been omitted for brevity.

Cloudlab provides a means of specifying a desired network topology that it constructs using software defined networking (SDN). However, at the time of writing the Cloudlab InfiniBand experimental network is not managed by SDN. We accounted for any variation in the InfiniBand network by ensuring that all nodes were connected to the same InfiniBand switch. The Cloudlab control network, using 1Gb/s Ethernet, uses top-of-rack switches that may introduce artifacts in non-Infiniband communication in certain conditions, such as nodes being in different racks. We accounted for this variation by testing the latency between all nodes. If a node was found to have statistically higher latency than its neighbors (to the level of $\alpha = 0.05$), that node was removed from testing and another was selected.

Table 1: Cloudlab c8220 Nodes

Hardware	Description
CPU	Two Intel E5-2660 v2 10-core CPUs at 2.20 GHz (Ivy Bridge)
RAM	256GB ECC Memory (16x 16 GB DDR4 1600MT/s dual rank RDIMMs
Disk	Two 1 TB 7.2K RPM 3G SATA HDDs
NIC	Dual-port Intel 10Gbe NIC (PCIe v3.0, 8 lanes)
NIC	QLogic QLE 7340 40 Gb/s InfiniBand HCA (PCIe v3.0, 8 lanes)

Table 2: Cloudlab c6320 Nodes

Hardware	Description
CPU	Two Intel E5-2683 v3 14-core CPUs at 2.00 GHz (Haswell)
RAM	256GB ECC Memory
Disk	Two 1 TB 7.2K RPM 3G SATA HDDs
NIC	Dual-port Intel 10Gbe NIC (X520)
NIC	QLogic QLE 7340 40 Gb/s InfiniBand HCA (PCIe v3.0, 8 lanes)

4.1.2 Software stack. Given the sensitive nature of latency measurements below $10\mu s$ and awareness of the impact of OS noise on parallel computer performance [27], the software configuration was carefully tuned to minimize noise and eliminate extraneous variables. To avoid introducing traffic over the processor interconnect in a dual-socket system, we designed our experiments to use the cores of a single CPU package with the shortest electrical distance to the network interface over the PCIe bus. We also required that core "0" was not used for experiment processes, as the operating system always assigns certain critical tasks to that core. As our MPI experiments required 8 processes per node, the hardware was required to have more than 8 physical cores per CPU package.

We ran the experiments on a patched Ubuntu 16.04. To prevent OS scheduling of tasks on the same cores as our experiment processes, we controlled the CPU affinity of tasks by isolating physical cores 2-9 on CPU 0 of each node with the kernel flag `isolcpus=4-27`, and then forcing MPI to distribute processes only among those cores. This required a minor change to the Linux kernel to prevent scheduling kernel tasks on the isolated cores, a known issue that is detailed in [10].

Other OS configurations included disabling hyperthreading and disabling CPU low power states to prevent clock speed throttling. We achieved this using the kernel commandline options `processor.max_cstate=1` and `intel_idle.max_cstate=0`.

We compiled our experimental codes using gcc version 5.4.0. When flags were not provided by the benchmark code, we used the flags `-O3 -march=native`. When build flags were provided, we used the provided flags.

We chose OpenMPI 1.10.2 from the Ubuntu repositories as our MPI implementation because of its performance on InfiniBand interconnects. At the application level we chose the NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPB) version 3.3.1 [3].

We also modified the `ipathverbs` user-space driver version 1.3 to introduce a latency injector. More information on these modifications can be found in Section 3.

The entire environment was deployed and managed using Ansible [11]. The playbooks and helper scripts are available at [34].

4.2 Common Methodology

Packet level. We conducted experiments to measure the performance of the network interfaces with a minimal amount of overhead. We chose codes from the `perftest` package from the Open Fabrics Enterprise Distribution (OFED) [2], which is well–established for testing InfiniBand performance at the packet level. In particular, we used `ib_write_lat` with Reliable Connection (RC) transport protocol. This tool uses raw InfiniBand commands (called *verbs*) to measure the time to send remote write commands with delivery confirmation, and forms a low level pingpong test.

We did not consider other tests from OFED such as `atomic` or `send` operations because they introduce additional operations above and beyond that of `ib_write_lat`, and have higher latency and latency variation because they require additional CPU assistance to complete. Our preliminary experiments indicated use of a software MTU of 2048 bytes as the most efficient configuration without message fragmentation.

Library level. The purpose of these tests is to capture the performance in a more realistic scenario where a well–established abstraction such as MPI is used. We used two codes: `ping-pong`, which times a sequence of `MPI_Send` and `MPI_Recv` calls between two processes, and `barrier`, which tests the efficiency of collective communications by calling `MPI_Barrier` on eight nodes. In each of tests, the time to complete a single operation was considered the runtime.

Application level. The NPB consist of a suite of codes designed to test many features of a high performance computing cluster, including its network, based on problems seen in computational fluid dynamics. It consists of five benchmarks: Integer Sort (IS), Embarrassingly Parallel (EP), Conjugate Gradiant (CG), MultiGrid (MG), and 3D Fast Fourier Transform (FT). Of these benchmarks, three of them have high communication volume: CG, MG, and FT [3]. The NPB also include three pseudo applications: a block tri-diagonal solver (BT), a scalar penta-diagonal solver (SP), and a lower-upper Gauss-Seidel solver (LU). These tests stress the network interconnect and provide a model of latency variation similar to real-world network conditions.

Table 3: NAS Parallel Benchmarks Tests

Name		size	procs	nodes	Mpkts	GB
Conjugate Gradient	CG	C	64	8	9.30	3.92
3D fast Fourier Transform	FT	C	64	8	22.34	9.76
Integer Sort	IS	C	64	8	2.77	1.22
Lower-Upper Gause-Seidel	LU	B	64	8	4.75	0.62
Multi-Grid	MG	C	64	8	1.47	0.55

To examine the effects of increased latency variation on real applications, we executed the NAS Parallel Benchmarks across the eight nodes in our cluster. We ran five of the eight included tests, detailed in Table 3. NPB problem sizes were chosen to ensure a long enough runtime for repeatable results, and the number of processors was chosen to fit each test's particular requirements while being evenly divisible by our eight nodes. In Table 3, `size` corresponds to the NPB problem size (e.g., A-F, where C is a "medium" size) we configured for our cluster, `procs` corresponds to the number of processes used at that size, `nodes` is the number of compute nodes the processes were divided between, `Mpackets` corresponds to the number of millions of packets sent across the network, and `GB` corresponds to the number of gigabytes of traffic generated during the experiment.

5 WORKLOAD CHARACTERIZATION

In this section we describe our methodology for simulating and characterizing the effects of network resource contention (i.e., congestion). Our goal is to measure the effect of network device contention on latency at the packet level, so that we can create a latency model for controlled emulation of congested network resources. This model allows us to configure our latency injector to match various levels of congestion without introducing other effects of real congestion such as high CPU or memory usage.

5.1 Characterization Procedure

To simulate network congestion, we created an MPI-based network load generator to send data between pairs of compute nodes, saturating the one-way bandwidth between the network interfaces. We then controlled the level of congestion by altering the proportion of time that the sending node was transmitting. By having each node run the application twice in both sending and receiving mode, we were able to saturate a fraction of the node's maximum transmission rate. For each measurement of congested performance, we ran the load generator on all involved nodes, as illustrated in Figure 1.

The sending application, detailed in Algorithm 3, is based on the "leaky bucket" rate control mechanism first described in [33]. To simulate the transmission characteristics of applications competing for network resources, we sampled the delays between messages from an exponential distribution, which models the long-tailed and highly variable inter-message gaps in large flow background traffic observed in [1]. Samples were provided by a Mersenne Twister

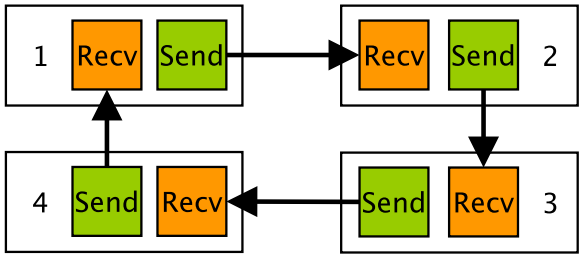

Figure 1: Send/receive pairs of background load generating processes on four compute nodes. Each pair saturates a controllable fraction of the one-way bandwidth between two compute nodes.

19937 pseudo-random number generator, which provides high quality entropy for its performance [23].

Algorithm 3 Congestion Simulator

```
clock_gettime(CLOCK_MONOTONIC, &last_time);
nsec_delay ← 0;
while !stopping do
  if nsec_bucket ≥ nsec_delay then
    MPI_Send(...)
    nsec_bucket -= nsec_delay;
    nsec_delay = random_from_exponential();
  end if
  clock_gettime(CLOCK_MONOTONIC, &cur_time);
  diff_time ← cur_time - last_time;
  last_time ← cur_time;
  nsec_bucket += diff_time
end while
```

A key feature to observe from Algorithm 3 is that we used the POSIX interface `clock_gettime` to measure time. It has two important characteristics: `clock_gettime` is the highest performing and most precise clock available on most POSIX systems, and it does not require a trap into the kernel to measure the time as `gettimeofday` and other interfaces do. On the x86_64 hardware that we used, it is implemented using a read of a timing register on the processor.

5.2 Characterization Results

Figures 2 and 3 illustrate the effects of background load on network packet latency for our topology, expressed as a percentage of the network interface's bandwidth. As the simulated network load increases from 0% to 100%, latency mean and variation increase as the sending applications compete for the network device queues.

There are two conditions to notice about the characterization results. First, for congestion above 80%, the latency mean and standard deviation become highly chaotic. For that reason we restrict the remainder of the measurement studies to simulated congestions below 80%. We leave studies of the higher region for future work. Secondly, we observe that our results of increasing mean and standard deviation of latency are consistent with the existing work on congestion. While the particular distribution collected is hardware and software dependent, the general shape center, and spread are consistent across hardware.

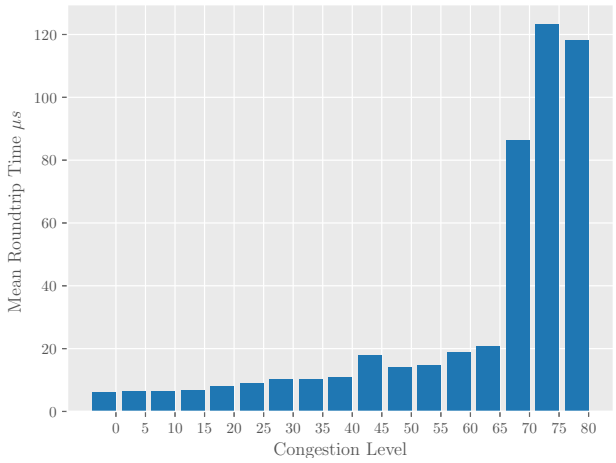

Figure 2: Mean round trip time in a congested environment. As congestion increases, so does the mean latency.

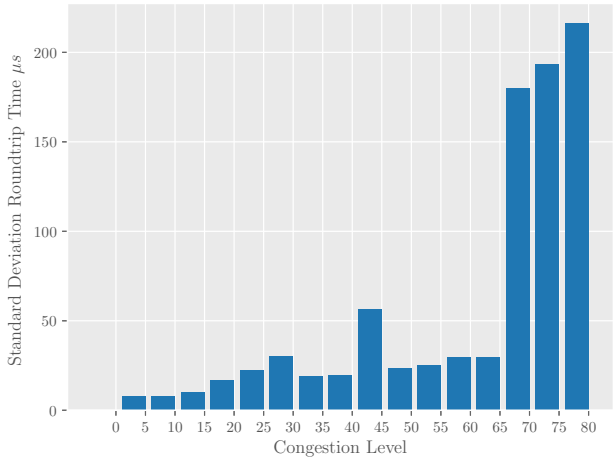

Figure 3: Round trip time standard deviation in a congested environment. As congestion increases, so does the standard deviation.

5.3 Modeling the Existing Distribution

To create synthetic latency that models the observations of workload congestion, we fit statistical distributions to the measured latency at each workload level. For an example distribution without background congestion, refer to Figure 4. We observe that the distribution is skew-right. Preliminary curve fitting showed that it is best modeled by a log–normal function, which is intuitive because of the unique property of the log–normal distribution to model a combination of many random variables. The dominant mode is at approximately 6.1μs with a minor mode of 7.2μs. When plotted against cumulative packet count, the higher latency values are correlated with harmonics of the CPU and PCIe bus frequencies occurring approximately every 45 to 50μs.

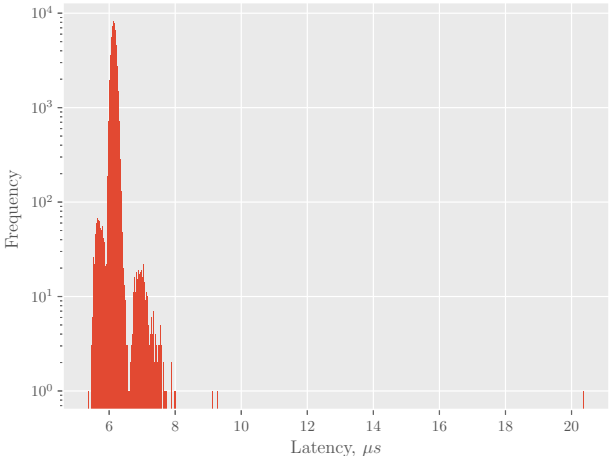

Figure 4: Distribution of latency of packets is tight, and highly skew right in an environment with no background congestion. Note the outlier observation above 20μs.

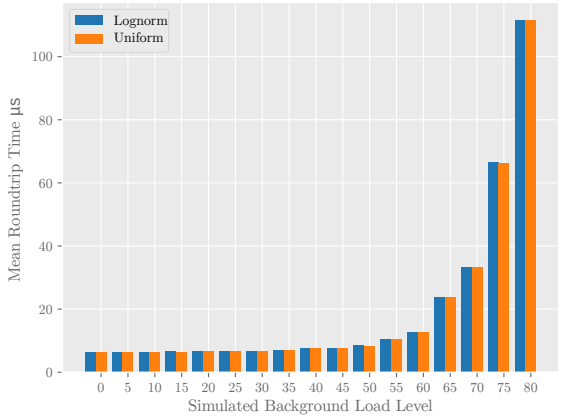

Figure 5: Mean observed latency of InfiniBand packets with injected delay. The distributions have equivalent means at each level of simulated latency. This validates the choice of distributions from the workload classification.

For each level of synthetic background load, we fit five distributions to the observed network latency: a lognormal distribution that closely fits the observations, a uniform distribution that fits the mean of the observations (note that this is equivalent to a lognormal distribution with zero variance), and three intermediate lognormal distributions with altered shape and scale that retain the observed mean. This set of distributions allow us to test the hypothesis that an increase in latency variance is more highly correlated with application runtime than latency mean.

The uniform distribution only has one parameter, mean, which was computed directly from the observations. The lognormal distribution has three parameters: shape, scale, and location. We estimated the parameters by using SciPy's `lognormal.fit` method for the corresponding level of congestion [17]. For intermediate distributions, we varied the scale parameter to values at 25%, 50%, and 75% of the observed scale, and then used binary search to identify a shape parameter that resulted in a mean equivalent to the observations.

Finally, we wrote a generator that uses the distributions from the statistical functions included in SciPy 0.16.1 to generate the distribution files for use with the latency injector. The distribution files generated along with the distribution file generator are included with the source code distribution [34].

6 EXPERIMENTAL SUITE 1: LATENCY VARIATION, APART FROM CONGESTION, CAUSES APPLICATION DEGRADATION

In this section, we use the results of our workload characterization to show that even without congestion or contention, latency variation is sufficient to cause application degradation. We accomplish this by simulating latencies along the distributions caused by congestion and contention. By injecting the latencies, no network resources are constrained during these tests.

We use the same tests that we utilized in workload characterization, except that instead of running a background process to induce load, we use the latency injector. Ideally, we would just increase the standard deviation of the distribution. However, by virtue of increasing the standard deviation of a distribution with a strong lower bound, we would also raise its mean. Therefore, we must also consider a distribution where we just increase the mean but leave the spread unperturbed.

We consider two distributions: 1) one where we increase only the mean (Uniform), and 2) one where we increase the mean and standard deviation (Lognorm). The means of these distributions were chosen to correspond to the means of the distributions caused by synthetic workload congestion.

Packet level. At the packet level, we have results that are as expected. In Figure 5 we observe that the two distributions have very similar mean values. That is, mean latency of network packets is the same mean (Uniform) as compared to when latency is injected to increase the mean and variance (Lognorm). This validates the calculation of distributions from the measured congestion distributions and also validates the functionality of the latency injector.

We also observe in Figure 6 some of the differences between these distributions. Here we see that the median latency is significantly higher for the uniform distribution at higher load levels than the lognorm distribution. This suggests as we observe that there are a few very large latencies that were measured in the lognorm distribution that were not present in the uniform distribution. We finally observe that there is little difference between the uniform distribution mean and median.

When we examined the results from the packet and library levels, we determined that their runtime distributions were not normally distributed. We confirmed our suspicions using the Kolmogorov-Smirnov test for Normality [26] (p=0.0, $\alpha = .05$)[1]. With non–normal distributions we cannot use parametric statistical methods for evaluation. Instead, we rely on the nonparametric Mann–Whitney U

[1] due to limited precision of the hardware the result rounds to 0

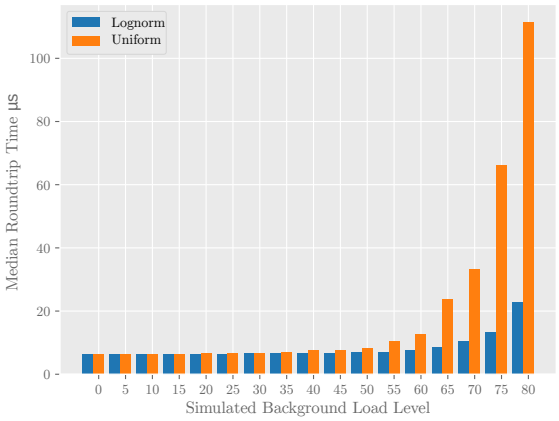

Figure 6: Median observed latency of InfiniBand packets with injected delay. Observe that lognorm has a lower median at higher simulated latencies, but matched mean. This indicates there are a small number of large latencies at higher simulated latency levels.

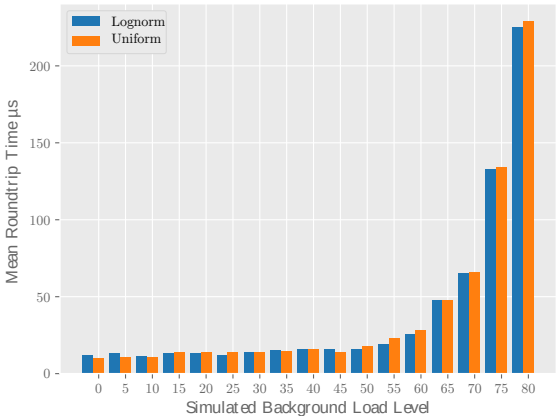

Figure 7: Mean round trip time of MPI ping pong, similar to the results from the packet level. The mean is slightly higher than the packet level tests which represents overhead at the library level.

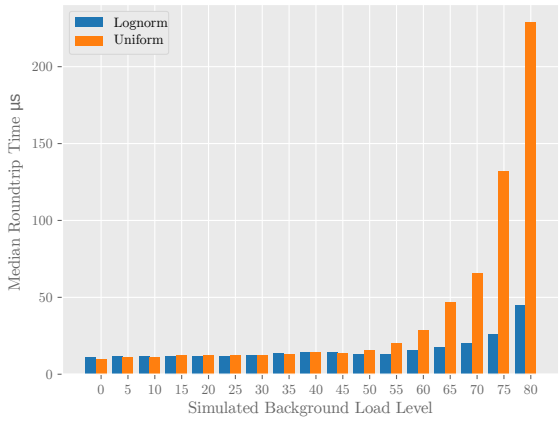

Figure 8: Median round trip time of MPI ping pong. Again, the results at the packet level carry over to the library level. The lower median time of lognorm compared to the mean indicates higher variance.

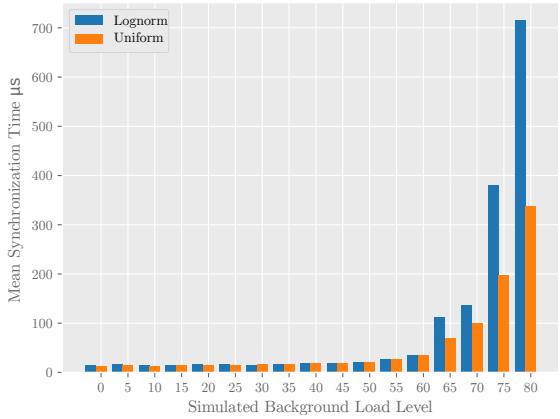

Figure 9: Mean time to synchronize in an MPI barrier. Unlike the packet level test, the means diverge. This is to be expected as synchronization time is determined by the slowest member.

test, in which the null hypothesis states that neither sample stochastically dominates the other. Nonparametric tests do not require the values to be sampled from a normal distribution.

Library level. At the MPI library level, we see a slightly more interesting picture. First, we consider the results from the MPI `pingpong` test. We observe in Figures 7 and 8 that the MPI results have the same shape, centers, and spreads as the packet level results, with a slightly higher latency. This is consistent with a small amount of constant overhead introduced by the MPI framework.

Secondly, we consider the results from the MPI `barrier` test. As opposed to the packet level test, we see in Figure 9 that the mean latencies diverge at higher synthetic latencies (Note the higher range on the y-axis). We also observe that the lognorm distribution results in higher mean values of latency. This is to be expected as the lognorm distribution produces a small number of extremely large latencies. The mean, being sensitive to these extremes, is pulled to the largest values. The median displays a shape and spread that is similar to that of the `pingpong` test (Figure 10). This is consistent with the robustness of the median to a few extreme values. The vast majority of the latencies in the lognormal are shorter than the mean, resulting in a smaller median latency than uniform distribution.

Application level. Unlike the packet and library level, the application runtimes follow normal distributions. As such we are justified in using traditional statistical parametric methods to evaluate these results. We show only the results for the mean; there are no substantive differences between the mean and median for these results. This can be explained in part by the central limit theorem [19] which states that the sum of independent random variables tends towards normality when the sample is suitably large. Secondly,

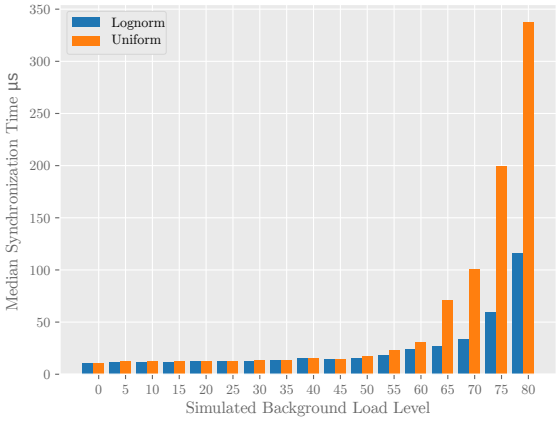

Figure 10: Median time to synchronize in an MPI barrier, where the results are similar to ping pong with slightly higher latency. This is consistent with the medians robustness to extreme values.

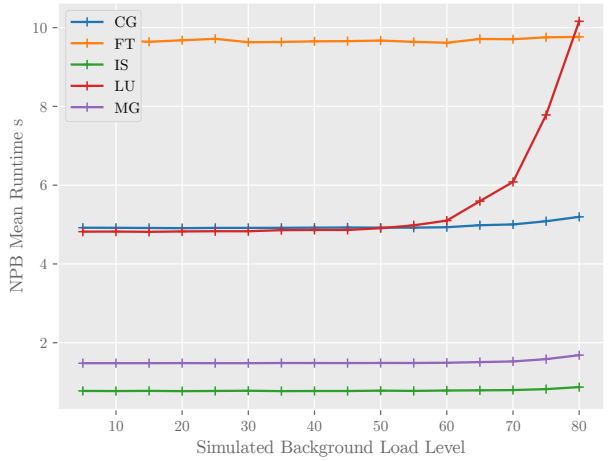

Figure 11: NPB runtimes with a uniform distribution of injected latency. The increased mean results in noticeably higher runtimes in LU only.

unlike the previous levels, we are measuring total application run time as opposed to a particular message delivery time. Both of these factors drive the overall distribution towards normality and, in effect, the mean towards the median.

There are some interesting results at this level. First, observe in Figure 11 that the majority of the applications are relatively unaffected by the increases in latency mean at higher synthetic latencies. The principal exception is the LU solver code runtime which roughly doubles at higher latency values. This can be explained by the communication–intensive nature of the LU solver.

Then, observe in Figure 12 that several of the applications increase run times at higher levels of variation in synthetic latency. The LU code roughly increases its runtime by a factor of 3.5. The communication intensive CG, FT, and MG codes also show increases in runtime that are not apparent in Figure 11. This suggests that

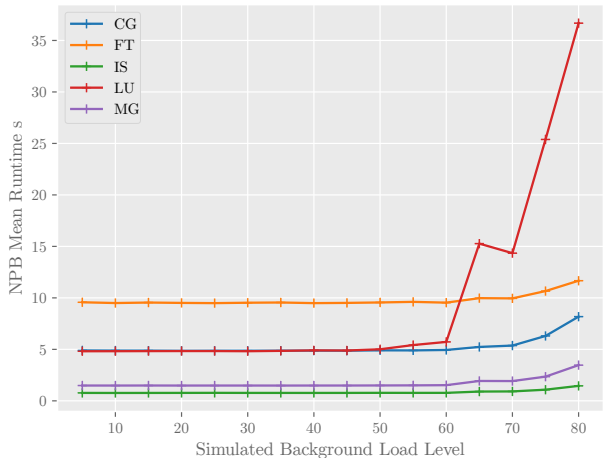

Figure 12: NPB runtimes with a lognormal distribution of injected latency. The increased spread results in significantly greater runtime for LU, but also greater run times for the communication intensive CG, FT, and MG.

the latency variation has a significant effect on the runtime of applications beyond that of mean latency.

In Experimental Suite I we have demonstrated that increased latency variation is sufficient to cause application degradation. We have traced these effects from the packet level up to the application level, and have shown that this phenomenon is reproducible even without other effects of contention or congestion.

7 EXPERIMENTAL SUITE 2: LATENCY VARIATION IS MORE HIGHLY CORRELATED WITH APPLICATION DEGRADATION THAN LATENCY MEAN

In this section we examine the claim that latency variation better explains application degradation than latency mean. We analyze the results from the previous section, focusing on the application layer – NPB tests. For each test from the NPB, we plot its application runtime against latency mean for the injected distribution. We assess the strength of a linear relationship between runtime and mean using Pearson's Coefficient of Correlation [19]. Similarly, for each test from the NPB, we plot its application runtime against latency standard deviation for the injected distribution. Again, we assess the strength of a linear relationship between runtime and standard deviation using Pearson's Coefficient of Correlation. We carefully choose the underlying distributions to inject so as to hold the mean constant for subset of experiments that correspond to a given simulated congestion level. We vary the standard deviation of the latency in each subset of experiments by adjusting the scale parameter of the injected distribution.

Figure 13 shows the relationship between the mean latency and application runtimes. We observe that a linear relationship is a plausible model for explaining application runtimes with respect to latency mean. Similarly, Figure 14 shows the relationship between the latency standard deviation and application runtimes. Again, a linear relationship is a plausible model for explaining application

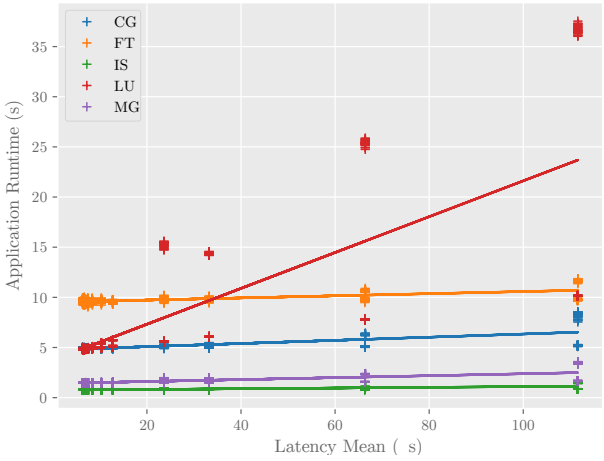

Figure 13: Mean runtime of NPB tests vs. increased mean latency. Random scatter above and below the line of best fit indicates the suitability of a linear model. High and low spreads about the line of best fit correspond to different standard deviations.

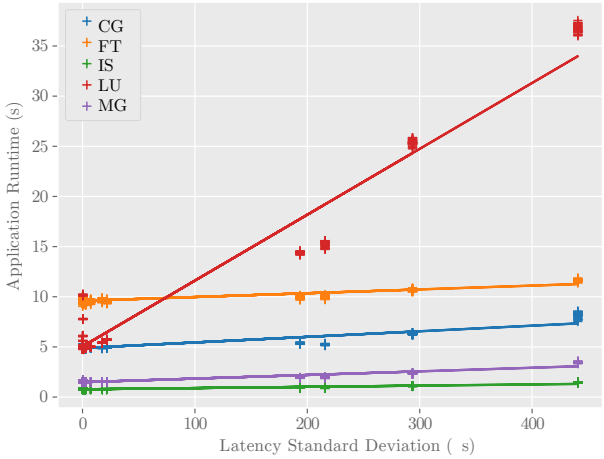

Figure 14: Mean runtime of NPB tests vs. increased latency variation. Tight fitting about the line of best fit indicates the strength of the model. Unlike the mean, variations above and below the line are not correlated with different means.

runtimes with respect to latency standard deviation. We further observe that of the five applications tested, the performance of the LU Decomposition application suffers the most radical effects of changes in latency. In the worst case in with a latency standard deviation above 400μs, the application runs 3.5 times slower than when the standard deviation near zero.

To access the strength of the relationship between latency mean and latency standard deviation vs. runtime, we Pearson's correlation coefficient for each test. These results are summarized in Table 4. A sample correlation coefficient (usually labeled r) above 0.7 is evidence of a linear relation, and a correlation coefficient above 0.9 is evidence of a strong linear relationship [19]. As shown in

Table 4: Correlation Results Rounded to the Nearest.01

Test	r_{mean}	r_{std}	Significant?
FT	0.70	.89	$p = 7.89 \times 10^{-33}$
CG	0.73	.91	$p = 4.94 \times 10^{-39}$
IS	0.76	.93	$p = 8.89 \times 10^{-50}$
MG	0.73	.95	$p = 1.31 \times 10^{-93}$
LU	0.76	.97	$p = 5.24 \times 10^{-148}$

Table 4, the correlation coefficient, r_{mean}, that tests the strength of the linear relationship between the FT application runtimes and latency mean is $r_{mean} = 0.70$. The correlation coefficient, r_{std}, that tests the strength of the linear relationship between the FT application runtimes and latency standard deviation is $r_{std} = 0.89$. The correlation coefficients for four other NPB applications are also shown in Table 4.

The values in the column labeled "Significant?" are calculated using the Fisher z transform[19], derived as:

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{\frac{-t^2}{2}} dt$$

where,

$$z = \frac{arctanh(r_{mean}) - arctanh(r_{std})}{\sqrt{\frac{2}{n-3}}}$$

The very small values (close to zero) in the column labeled "Significant?" indicate that the difference between r_{mean} and r_{std} is highly statistically significant (to the level of $\alpha = 0.05$) for all applications tested, meaning that the network latency variation is more highly correlated with application runtimes than network latency mean values. Pearson's coefficient is not considered to be robust to outliers; however, as illustrated in Figure 14, the residuals between the line of best fit and observed values are sufficiently small. Thus, outliers are not a primary factor in the correlation coefficients.

Finally, we examine the performance impact of latency variation on application runtime. We examine two sets of experiments. In each set the mean network latency is fixed and there are several values for the network latency variation. We chose experiment sets with network mean latency that correspond to a congestion level of 20% and a congestion level of 70%, which are typically-observed means in cluster networks. In Figure 15, we see that for a fixed mean corresponding to a congestion of 20%, changes in the standard deviation of network latency have a limited effect. However, in Figure 16, we see that for a fixed mean corresponding to a network congestion of 70%, increases in standard deviation cause a substantial increase to application runtimes. In particular, LU shows a nearly 25% increase in runtime with the larger latency variation, while CG and MG show more modest increases in runtime of 5% to 10%. Together, these charts suggests future work to study at which level of latency mean that latency variation begins to have a substantial effect on application runtime.

Based on the results of Experimental Suite II, we conclude that the mean latency is strictly less correlated with application runtime than latency standard deviation for all tests we considered. We

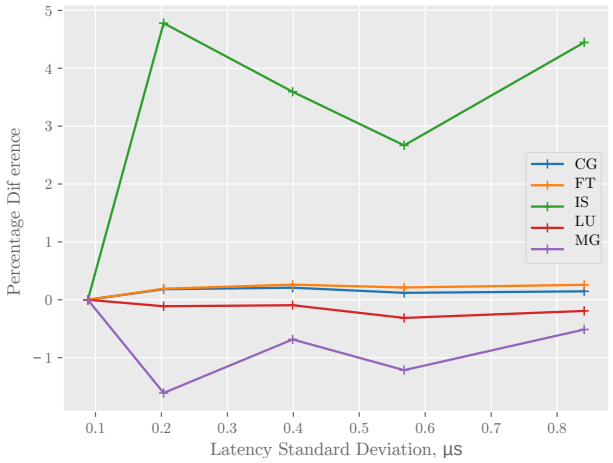

Figure 15: Difference in mean NPB runtime relative to zero injected latency. At the 20% level of network congestion, increases in standard deviation have a limited effect to application runtimes.

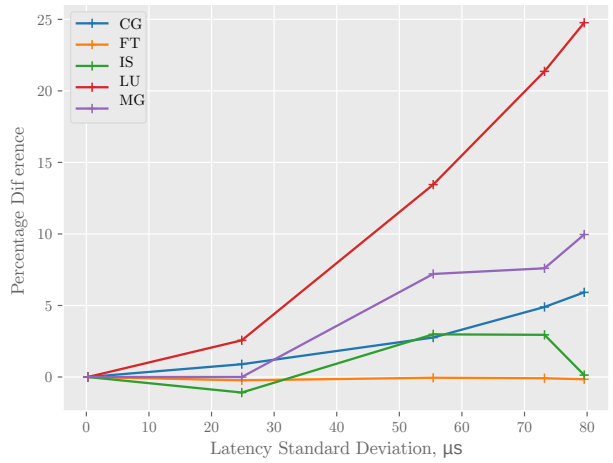

Figure 16: Difference in mean NPB runtime relative to zero injected latency. At the 70% level of network congestion, increases in standard deviation correspond to significant increases in application runtime.

calculated the significance of the difference in the correlation coefficient results using Fisher's Z transformation. We find the results to be statistically significant (to the level of $\alpha = 0.05$), indicating that measurements that are this different are unlikely to occur by chance. The key result is that latency variation is more highly correlated with application runtime than latency mean.

8 RELATED WORK

Network latency is a fundamental concern in HPC system design, and there exists a large body of knowledge on its role in application performance. In this section, we highlight the work that complements this paper, and clarify our specific contributions.

Alizadeh et al [1] explored the effects of high bandwidth consumption on network latency, and found that increased competition for buffers in Ethernet switches could lead to long-tailed latency distributions with measurements as high as 1000 times the median. Our work is partly inspired by theirs, as we questioned whether the latency variation would also impact HPC workloads using MPI and zero-copy networks like InfiniBand.

One of the earliest papers to examine the effects of latency variation on network protocols was the done by Zhang et al in [36]. In their work, they observed that the throughput of TCP streams could degrade if ACK packets were significantly delayed and triggered congestion control mechanisms. They proposed some adjustments to the TCP congestion control protocol to reduce the effects of ACK clustering. Unlike their work on the effects of latency on network control algorithms, our work focuses on the impact of this variation on applications supported by the network.

There have been many studies of the effect of congestion on network latency. [4, 5, 15] and [16] observed that increased bandwidth contention on links between nodes in a large cluster resulted in larger mean latencies and significant impact on HPC workloads. These studies essentially show that congestion introduces additional latency, and that latency variation increases when congestion is present as there is additional contention for the hardware. Our work asserts that latency variation can have negative effects in and of itself, and the impacts of latency variation can be more significant than mean latency.

One other aspect that our work intersects is tooling for synthetic load. Our approach induces CPU spin at the InfiniBand driver level, similar to [35]. However, we modeled the interface on the traffic control (`tc`) tool, a component of which uses a distribution input file for adding randomized latency on outgoing IP packets passing through the Linux kernel packet scheduler. Our work complements `tc` by allowing latency control on a subset of QLogic and Mellanox InfiniBand cards at sub-microsecond resolution, and builds on prior work by providing an interface and support tools that enable future work with artificial latency.

9 CONCLUSIONS AND FUTURE WORK

In this paper we have presented the design and implementation of a configurable latency injector for many Mellanox and QLogic InfiniBand cards. The latency injector offers features not found in prior similar work, and can be extended to include network cards beyond the original implementations. We have presented measurement and workload characterization studies of the distributions of latency in network performance for an InfiniBand network in an HPC environment. These distributions are utilized directly in two experimental suites. We have presented an experimental methodology using synthetically-generated latency to demonstrate the effects of latency variation on HPC workloads. In the worst case, we measured that the LU application runs 3.5 times slower for high variation than when the standard deviation is near zero.

We found statistically significant evidence that latency variation is more highly correlated with HPC application performance than latency mean alone. This result is somewhat surprising, since studies that focus on mean latency alone have shown the strong impacts of low message latency to applications in an HPC environment.

We believe that these results may be important to consider in the design of scalable HPC systems and applications. As multitenancy becomes increasingly common in dedicated HPC systems and as HPC applications are more commonly run on cloud architectures, competition for network resources could lead to increased levels of network latency variation. We have demonstrated that serious degradation of application performance can result from high variation in latency.

Future work is to consider how to manage variation in latency in HPC networks that implement low latency protocols, and to consider the impacts and management of latency variation in enterprise or cloud environments where low latency protocols are not implemented, but where high variation in latency can occur due to sharing of network resources across user applications. The ultimate control over these factors that affect the variation of latency is to move to a real-time operating system and application environment. We do not advocate that here, since the slowdown imposed by the real-time constraints may impact performance more than network latency variation alone. Investigation of which factors have the most impact and systematically evaluating the resolution of the factors are aspects of future work.

ACKNOWLEDGMENTS

Funding for this research was provided by the National Science Foundation under award numbers 1642542 and 1633608. We utilized Cloudlab hardware for these experiments[30]. We utilized Pandas and SciPy extensively for statistical analysis [17, 24].

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *ACM SIGCOMM Computer Communication Review*, Vol. 40. ACM, 63–74.

[2] OpenFabrics Alliance. 2012. Openfabrics Enterprise Distribution. (2012). https://www.openfabrics.org/index.php/openfabrics-software.html

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[4] Abhinav Bhatelé and Laxmikant V Kalé. 2009. Quantifying network contention on large parallel machines. *Parallel Processing Letters* 19, 04 (2009), 553–572.

[5] Abhinav Bhatelé, 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 on High Performance Computing, Networking, Storage and Analysis*. ACM, 41.

[6] Abhinav Bhatelé, Andrew R Titus, Jayaraman J Thiagarajan, Nikhil Jain, Todd Gamblin, Peer-Timo Bremer, Martin Schulz, and Laxmikant V Kalé. 2015. Identifying the culprits behind network congestion. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 113–122.

[7] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® Omni-path architecture: Enabling scalable, high performance fabrics. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*. IEEE, 1–9.

[8] Nanette J Boden, Danny Cohen, Robert E Felderman, Alan E. Kulawik, Charles L Seitz, Jakov N Seizovic, and Wen-King Su. 1995. Myrinet: A gigabit-per-second local area network. *IEEE Micro* 15, 1 (1995), 29–36.

[9] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, Vol. 28. ACM, 1–12.

[10] Daniel Bristot de Oliveira. 2015. [RFC] workqueue: avoiding unbounded wq on isolated CPUs by default. (2015). https://lists.gt.net/linux/kernel/2218495

[11] Michael DeHaan. 2012. Ansible. (2012). https://www.github.com/ansible/ansible [Online].

[12] Corbin Higgs and Jason Anderson. 2016. Narrowing the Gap: Effects of Latency with Docker in IP Networks. In *The International Conference for High Performance Computing, Networking, Storage and Analysis, Student Poster*.

[13] Torsten Hoefler, Lavinio Cerquetti, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. 2005. A practical approach to the rating of barrier algorithms using the LogP model and Open MPI. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*. IEEE, 562–569.

[14] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. 2010. Performance analysis of high performance computing applications on the Amazon Web Services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 159–168.

[15] Van Jacobson. 1988. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, Vol. 18. ACM, 314–329.

[16] Ana Jokanovic, Jose Carlos Sancho, German Rodriguez, Alejandro Lucero, Cyriel Minkenberg, and Jesus Labarta. 2015. Quiet neighborhoods: Key to protect job performance predictability. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 449–459.

[17] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. (2001–). http://www.scipy.org/ [Online].

[18] Mark Karol, Michael Hluchyj, and Samuel Morgan. 1987. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communications* 35, 12 (1987), 1347–1356.

[19] G. Maurice Kendall. 1948. *The Advanced Theory Of Statistics*. Vol. 1. Charles Griffin and Company Limited, 42 Drury Lane, London.

[20] Richard B Langley. 1997. GPS receiver system noise. *GPS World* 8, 6 (1997), 40–45.

[21] Jacob Leverich, Matteo Monchiero, Vanish Talwar, Parthasarathy Ranganathan, and Christos Kozyrakis. 2009. Power management of datacenter workloads using per-core power gating. *IEEE Computer Architecture Letters* 8, 2 (2009), 48–51.

[22] J Martin, V Rajasekaran, and James Westall. 2005. Virtual machine effects on network traffic dynamics. In *Performance, Computing, and Communications Conference, 2005. IPCCC 2005. 24th IEEE International*. IEEE, 233–238.

[23] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, 1 (1998), 3–30.

[24] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 51 – 56.

[25] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S Muller. 2009. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 261–270.

[26] Gottfried E. Noether. 1967. *Elements of Nonparametric Statistics*. John Wiley and Sons, Inc., New York.

[27] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. 2003. The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 55–55.

[28] Gregory F Pfister. 2001. An introduction to the Infiniband architecture. *High Performance Mass Storage and Parallel I/O* 42 (2001), 617–632.

[29] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. 2009. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE, 427–436.

[30] Robert Ricci, Eric Eide, and the CloudLab Team. 2014. Introducing CloudLab: scientific infrastructure for advancing cloud architectures and applications. *;login:* 39, 6 (Dec. 2014), 36–38. https://www.usenix.org/publications/login/dec14/ricci

[31] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. 2011. It's time for low latency. In *HotOS*, Vol. 13. 11–11.

[32] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. 2001. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Supercomputing, ACM/IEEE 2001 Conference*. IEEE, 49–49.

[33] Jonathan Turner. 1986. New directions in communications (or which way to the information age?). *IEEE communications Magazine* 24, 10 (1986), 8–15.

[34] Robert Underwood, Jason Anderson, and Amy Apon. 2018. ICPE 2018 Artifact - Measuring Network Latency Variation Impacts to High Performance Computing Application Performance. (Jan 2018). https://doi.org/10.5281/zenodo.1145911

[35] Qi Wang, Ludmila Cherkasova, Jun Li, and Haris Volos. 2016. Interconnect emulator for aiding performance analysis of distributed memory applications. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 75–83.

[36] Lixia Zhang, Scott Shenker, and Daivd D Clark. 1991. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. *ACM SIGCOMM Computer Communication Review* 21, 4 (1991), 133–147.