GPCNeT: Designing a Benchmark Suite for Inducing and Measuring Contention in HPC Networks

Sudheer Chunduri*
sudheer@anl.gov
Argonne National Lab

Taylor Groves*
tgroves@lbl.gov
Lawrence Berkeley National Lab

Peter Mendygral*
pjm@cray.com
Cray Inc.

Brian Austin
baustin@lbl.gov
Lawrence Berkeley National Lab

Jacob Balma
jbalma@cray.com
Cray Inc.

Krishna Kandalla
kkandalla@cray.com
Cray Inc.

Kalyan Kumaran
kumaran@anl.gov
Argonne National Lab

Glenn Lockwood
glock@lbl.gov
Lawrence Berkeley National Lab

Scott Parker
sparker@anl.gov
Argonne National Lab

Steven Warren
swarren@cray.com
Cray Inc.

Nathan Wichmann
wichmann@cray.com
Cray Inc.

Nicholas Wright
njwright@lbl.gov
Lawrence Berkeley National Lab

ABSTRACT

Network congestion is one of the biggest problems facing HPC systems today, affecting system throughput, performance, user experience, and reproducibility. Congestion manifests as run-to-run variability due to contention for shared resources (e.g., filesystems) or routes between compute endpoints. Despite its significance, current network benchmarks fail to proxy the real-world network utilization seen on congested systems. We propose a new open-source benchmark suite called the Global Performance and Congestion Network Tests (GPCNeT) to advance the state of the practice in this area. The guiding principles used in designing GPCNeT are described and the methodology employed to maximize its utility is presented. The capabilities of GPCNeT are evaluated by analyzing results from several world's largest HPC systems, including an evaluation of congestion management on a next-generation network. The results show that systems of all technologies and scales are susceptible to congestion and this work motivates the need for congestion control in next-generation networks.

KEYWORDS

network benchmarking, congestion, performance variability

ACM Reference Format:
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 *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356215

1 INTRODUCTION

In High Performance Computing (HPC) systems, performant networks are essential to join resources efficiently together to enable scientific discovery at unprecedented scales. While simple metrics such as latency, injection bandwidth, and bisection bandwidth are useful indicators of a network's peak performance, typically they do not correspond to realized performance on production machines, particularly when diverse workloads contend for the same network resources. They often overemphasize best-case configurations on quiet systems and fail to expose behavioral differences between network topologies and architectures. In this work we examine the impact of contention for network resources and examine how such contention affects performance.

Network contention often manifests in application performance variation. Previous studies have shown that the impact of network jitter is substantial and increases super-linearly with process count for many systems [3–5, 12, 26, 28].

The purpose of this work is to introduce a benchmark suite that (a) induces network congestion and (b) measures its impact on latency and bandwidth. There has been no previous work on congestion inducing benchmarks because there was little that existing HPC networks did to address the issue. However, modern network architectures such as Cray Slingshot [6], Infiniband [16],

*primary authors contributed equally to the paper

This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

SC '19, November 17–22, 2019, Denver, CO, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356215

and Gen-Z [29] have made handling of congestion a priority with the acknowledgement that HPC centers want to provide a level of performance isolation between a breadth of heterogeneous workloads. As new networks incorporate features such as traffic classes and congestion control, HPC facilities require new benchmarks designed to evaluate their effectiveness. In this paper we seek to advance the state of the practice in this regard.

This paper makes several contributions towards these goals:

- We present GPCNeT – a generic, topology agnostic benchmark suite that captures the complex workloads anticipated on multitenant HPC networks. It uses multiple configurable congestion patterns concurrently to emulate multiple separate jobs and uses concurrent canary jobs that emulate boundary exchange algorithms to measure the impact of congestion.
- We show that GPCNeT works on production HPC machines and describe a systematic method to measure congestion intensity. Our results show that latency-bound communication can be very sensitive to congestion, bandwidth is generally less sensitive to congestion, and that congestion has a scale-dependent component. We also show that congestion manifests in different ways on the same machine depending on how an MPI library implements different interfaces.
- We demonstrate that by instrumenting the network counters of an HPC machine, it is possible to tune the benchmark to better represent a workload by emphasizing or deemphasizing certain congestors or canaries.
- GPCNeT also introduces the *Congestion Impact* metric to study the impact of congestion across systems that are based on different topologies and architectures. Using this metric we perform an analysis of the impact of network congestion on several of the world's largest HPC systems, including an evaluation of congestion management on a next-generation network showing that systems of all technologies and scales are susceptible to congestion effects. Networks with newer technologies (EDR IB, Slingshot), are less susceptible than older ones (Cray Aries).

In § 3 we present the GPCNeT benchmark which creates systematic congestion and measures the impact on latency and bandwidth for some common application-derived communication patterns. The benchmark was created to fulfill a set of design requirements (detailed in that section). We demonstrate the fulfillment of the design requirements in § 4, showing how GPCNeT provides meaningful results across a spectrum of architectures. We then use GPCNet to compare the impact of congestion across architecture and topology for some of the world's top HPC systems (§ 5).

2 BACKGROUND AND RELATED WORK

The performance efficiency of the interconnection network plays a significant role in determining the scaling and performance of communication-intensive applications. The contention for shared resources can have a significant impact on application performance. Hence, detailed performance evaluations of networks are essential to quantify the congestion impacts on production application performance.

Traditionally, MPI micro benchmarks such as Sandia [2], OSU [20], and Intel [17] microbenchmarks are used to measure the baseline latencies and bandwidths for data transfers across the network. Some benchmarks such as RMA-MT benchmarks [8, 31] are used to measure the specific aspects of MPI such as multi-threaded RMA. Hoefler et al. developed a microbenchmark for measuring both latency and overlap of computation and communication for non-blocking communication operations [13]. However, these benchmarks are intended to measure ideal baseline performance.

The impact of tail latency (extreme latency in a long-tailed distribution; e.g., greater than 99th percentile) on performance is not unique to HPC systems; this has been studied in the context of data center networks as well [22]. Yunqi et al. [32] developed a methodology using quantile regression to analyze, and attribute the sources of variance in tail latency to various hardware features of interest. The tail latency effects on production datacenter-scale file systems are also studied [24]. The congestion impact as measured using the LDMS telemetry data on two real production systems is analyzed [18].

There is a significant body of work that has measured the impact of congestion in HPC systems, but these do not include benchmarks that induce congestion. Hoefler et al. have developed benchmarks [14, 15] that consider the system noise aspects for better reflecting the application performance. Ates et al. developed a performance anomaly suite [1] that includes a simple network contention generator using pairs of communicating nodes. Performance modeling for the proposal, initial testing, and beginning deployment phase of the Blue Waters supercomputer system utilized LogGP model with parameters derived from Netgauge [14]. Grant et al. [11] developed a tool for measuring the impact of network contention due to interference from other jobs through a continually running benchmark application and the use of network performance counters. Prior works [30] showed the importance of considering the latency variation rather than just considering the latency mean for characterizing application performance.

These studies emphasize the importance of considering system noise for accurately quantifying the network efficiency for production application performance. However, there exists no standard benchmark suite for measuring the network performance in production scenarios. To be broadly applicable, these benchmarks are desired to be topology-agnostic, MPI implementation-agnostic, and network provider-agnostic. The benchmarks we propose in this study meet these criteria and can characterize network congestion impacts on various system architectures.

3 BENCHMARK DESIGN

We present the Global Performance and Congestion Network Tests. GPCNeT[1] differs from existing benchmarks by creating systematic congestion. Two types of communication patterns, sensitive (or "canary") and potentially congesting (hereafter "congestors"), execute in a coordinated manner. GPCNeT compares the performance of sensitive communication in isolation and while congestors are running. Sensitive traffic is generated by latency and bandwidth-sensitive communication kernels motivated by HPC applications (e.g., multi-dimensional nearest-neighbor exchanges). Congestor

[1] GPCNeT is publicly available at https://github.com/netbench/GPCNET

traffic is generated by kernels that stress the network (e.g., all-to-all and incasts). GPCNeT places a load on the network by running several congestor traffic patterns at the same time. These operations are all managed by a single executable in a single job launch. We identified additional requirements that affect the design of GPCNeT. The design requirements are that it

(1) is portable across a variety of topologies and architectures,
(2) is not easily manipulated to favor specific networks or implementations,
(3) can generate complex communication patterns with simple tuning,
(4) reports statistics representing the limiting communication performance of a latency or bandwidth sensitive HPC application,
(5) is able to run on any number of nodes.

GPCNeT uses MPI to generate a variety of network traffic patterns and can run on any system that supports the MPI-3.0 specification (first design requirement). GPCNeT can be deployed at full system scale on a given system. This requirement affects how sensitive and congestor communication patterns are generated, which is described in more detail in later sections. Process placement is an important consideration because real-world systems that support a diverse workload tend to provide non-ideal placement of processes relative to the network topology. GPCNeT includes features to approximate non-ideal application placement so that sensitive traffic patterns cannot be isolated from congestors by relying on MPI rank reordering (second design requirement).

A key tuning parameter for GPCNeT is the number of MPI ranks executing on each node. By simply increasing the number of processes for a fixed number of nodes, GPCNeT will increase the load on the network and generate more complex communication patterns (additional detail in § 3.1 and § 3.2), which addresses the third design requirement. There are many configurations for integrating a node into an HPC network, however. A node can have multiple NIC ports for communication, for example. Throughout this paper we refer to the number of Processes per network Port (PPort) to describe the relationship between how GPCNeT is run and the network resources on a node.

Another important set of features in GPCNeT are the metrics it measures and reports (fourth design requirement). Many MPI benchmarks measure the minimum, maximum, and average performance of an operation measured across ranks. However, these values do not reflect the communication performance perceived by real applications in production environments. Tail latency (e.g., 99% or 99.9%) is commonly used to describe performance-limiting communication for datacenter distributed applications [7]. For the purposes of this work, we define tail latency as a measure of latency significantly greater than the center of the distribution. Any application with synchronization times approaching the tail latency on a network (under load) can experience performance impacts. GPCNeT measures and reports 99% tail latency and the 99% tail performance of all included communication kernels.

3.1 Sensitive (Canary) Traffic

It is essential for the contention measurement tests to span a large fraction of the system; congestion can be localized in space and transient in time, and so might not be detected by monitoring a small number of links or node-pairs. We therefore selected network traffic patterns that can run on any number of nodes, and are sensitive to both latency and bandwidth effects. The communication pattern for these "canary" jobs alternates between all-reduce and a random ring. The all-reduce pattern is ubiquitous throughout HPC and measures the latency sensitivity of small-message collective operations, and the random ring targets bandwidth-sensitive point-to-point operations.

The random ring pattern was motivated by the prevalence of nearest-neighbor 3D communication in physics-based applications, but we have limited the dimensionality to a 1D ring so that the benchmark can run on any number of nodes. Multiple processes can be run on each node to better represent multi-dimensional communication patterns. Although randomizing the order of nodes in the ring eliminates locality (and related optimizations) from the communication pattern, it ensures that the benchmark is agnostic to network topology. A secondary benefit of randomization is the added flavor of irregular communication patterns generated by task-based programming models and graph processing.

3.1.1 *Random Ring Infrastructure.*

A random ring infrastructure emulates an application or set of applications performing nearest-neighbor exchanges. The random ring design is based on a similar implementation in HPCC [23], where a one-dimensional periodic ring is generated by randomizing a list of MPI ranks and each rank communicates to its left and right neighbors in the ring. The random ring design satisfies the fifth design requirement that GPCNeT be able to run at any node count. Iterating over many different random rings also prevents optimizations based on MPI rank reordering.

Communication should only be off-node, preventing the results from being contaminated by on-node communication. This is achieved by creating PPort sub-communicators with each process on a node in a unique sub-communicator. Ranks in the same sub-communicator are randomized to form a random ring. With this design, a given node will communicate with up to $2 \times PPort$ other nodes, accounting for left and right communication. Communication patterns resembling a multidimensional decomposition or unstructured mesh are naturally produced by this behavior. The connectivity between nodes is less than all-to-all communication but more complex than a 2-dimensional decomposition. Running GPCNeT with increasing PPort is therefore a simple way to increase the number of network pathways traversed by sensitive traffic.

Listing 1 summarizes the random ring infrastructure and performance measurements in it. A timeout mechanism is included to prevent tests from running longer than 10 seconds, which is long enough to measure a statistically significant number of iterations of any of the communication kernels under most conditions. It is possible that a test runs longer than 10 seconds, however, when very heavy load is on the network. The generation of a random ring is based on the ring loop index. This means the same set of random rings is produced in each measurement iteration.

```
/* nM     = total number of measurements
   nR     = total number of random rings
            to generate in a sub-communicator
   nI     = number of iterations of a
            communication kernel
   myrank = local MPI rank in a sub-communicator
   warmup = number of warmup iterations
```

```
*/
timeout1 = timer();
for (m = 0; m < nM; m++) {

    /* test if running too long */
    timeout = timer() - timeout1;
    MPI_Allreduce(MPI_IN_PLACE, &timeout, 1, MPI_DOUBLE,
        MPI_MIN, global_comm);
    if (timeout > 10.0) continue;

    /* loop over random rings */
    for (n = 0; n < nR; n++) {
        randomize_list(&ranklist, n);
        neighbor_left  = ranklist[myrank-1];
        neighbor_right = ranklist[myrank+1];
        for (i = -warmup; i < nI; i++) {
            if (i == 0) timer_bulk_start = timer();
            if (i >= 0) timer_fine_start = timer();

            communication_kernel(neighbor_left,
                neighbor_right);

            if (i >= 0)
                timer_fine[m*nR*nI + n*nI + i] = timer();
        }
        timer_bulk[m*nR + n] = timer() - timer_bulk_start;
    }
}
```

Listing 1: Pseudocode of the random ring infrastructure.

Two communication kernels execute in the random ring infrastructure. One is a latency measurement and the other measures bandwidth with a synchronization. The infrastructure is able to support other kernels as well. GPCNeT can be easily modified to use MPI-RMA based latency and bandwidth kernels (included in the package), for example, as additional sensitive traffic.

Random Ring Point-to-point Latency (P2P Lat). P2P Lat measures the latency of sending and receiving 8 byte messages to and from the left and right neighbors. This kernel is intended to be more latency-sensitive than typical HPC applications to provide an upper limit on the performance impact from congestion. The loop limits in Listing 1 for this test are $nM = 10000$, $nR = 30$, $nI = 200$, and $warmup = 200$, which are sufficient for statistically rigorous results. Each canary process in the kernel performs two `MPI_Irecv` and `MPI_Isend` with respect to the two random neighbors per iteration followed by a `MPI_Waitall`. The walltime of each kernel iteration is measured and divided by two to account for round-trip time.

Random Ring Point-to-point Bandwidth with Synchronization (P2P BW+Sync). P2P BW+Sync is motivated by applications performing boundary value exchanges followed by a small reduction for a time step size or convergence test. Eight 128 Kbyte messages are sent to and received from the left and right neighbors (16 messages in total), which is typically enough messages to achieve peak bandwidth. Afterwards, a barrier on the sub-communicator synchronizes all ranks. This test is sensitive to bandwidth contention and latency from the synchronization. The loop limits in Listing 1 for this test are $nM = 10000$, $nR = 30$, $nI = 8$, and $warmup = 1$. The walltime of each iteration is measured and converted into a bandwidth value, accounting for 16 messages.

3.1.2 *Allreduce.*

A separate test for small message `MPI_Allreduce` is included outside of the random ring infrastructure. This test is used to measure the effects from congestion on latency-sensitive collective communication. The same sub-communicators used in the random ring infrastructure are used in this test. Therefore, there are PPort `MPI_Allreduce` operations occurring simultaneously, each with only off-node communication. The loop structure for this kernel is the same as Listing 1, excluding the loop over random rings, with $nM = 100000$, $nI = 200$, and $warmup = 1$.

3.1.3 *Performance Metrics.*

GPCNeT uses the two timers, shown in Listing 1, to compute performance statistics on the communication kernel – a fine-grained timer that generates histograms for each configuration of random ring and a bulk timer that provides overview statistics.

3.2 Congestors

Congestion occurs on a network when there are insufficient resources available to accommodate the demanded load. Communication patterns that generate an imbalance between ingress and egress rate can cause congestion. Broadly, we can organize congestion into two categories: endpoint and intermediate. An example of endpoint is an incast operation where data from N-1 senders is directed to a single receiver, which is unable to accept data at the rate it arrives. It is also possible to induce side-effects of congestion, such as queuing delays, by placing heavy loads on a network. Many-to-many communication patterns (e.g., all-to-all with large messages) will stress a network and is an example of intermediate congestion. These patterns develop on networks supporting real workloads either explicitly by demanding applications or spontaneously through the cumulative effects of many applications and supporting subsystems, such as filesystems. To ensure portability (first design requirement), GPCNeT uses MPI operations to generate these communication patterns.

MPI operations define the logical flow of data but do not specify the underlying methods by which data is moved. For example, a logical incast can be written in MPI by initiating N-1 non-blocking receive operations at the root and a send operation from each sender. This pattern may or may not generate a true incast pattern depending on the underlying communication protocols (Get/Put) used in the MPI implementation. MPI-RMA exposes more explicit operations, such as `MPI_Get()` and `MPI_Put()`, but an MPI library can implement MPI-RMA using two-sided operations. To overcome these implementation complexities and ensure efficacy across architectures (first design requirement), GPCNeT runs four congestors simultaneously to create endpoint and intermediate congestion. A specific congestor may not produce heavy load or congestion on its own. But, in the absence of effective congestion management in the network, some combination of the congestors should produce heavy load. GPCNeT also uses a common message size for congestors, which makes it difficult to tune an MPI implementation to favor one type of a congestor without negatively affecting the rest of the kernels in the benchmark.

3.2.1 *Congestor Infrastructure.*

Congestor communication kernels are executed within a common infrastructure, similar to the random ring infrastructure in § 3.1. Ranks on a congestor node are assigned one of the congestor kernels for the duration of execution. Listing 2 summarizes the congestor infrastructure. A difference from the random ring infrastructure is the use of `MPI_Iallreduce` to manage a timeout. This improves the chances that load from a congestor is constant despite the presence of other collective communication. Similar to the sub-communicators used for sensitive traffic, a congestor kernel is executed with PPort different sub-communicators simultaneously to avoid any on-node traffic. Tests of GPCNeT with different PPort not only tune the behavior of sensitive traffic (see § 3.1.1), they also tune the intensity of the congestors. The performance of each congestor is measured and is included in GPCNeT output for verbose compilations.

```
/* nM      = total number of measurements
   nI      = number of iterations of the kernel
   myrank  = local MPI rank in a sub−communicator
*/
timeout1 = timer();
for (m = 0; m < nM; m++) {

    /* test if running too long */
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    if (timeout > 10.0) continue;
    timeout = timer() − timeout1;
    MPI_Iallreduce(MPI_IN_PLACE, &timeout, 1, MPI_DOUBLE,
        MPI_MIN, scom, &req);

    for (i = −1; i < nI; i++) {
        if (i == 0)
            timer_bulk_start = timer();
        if (i >= 0)
            timer_fine_start = timer();

        congestor_kernel(myrank);

        if (i >= 0)
            timer_fine[m*nR*nI + n*nI + i] = timer();
    }
    timer_bulk[m*nR + n] = timer() − timer_bulk_start;
}
```

Listing 2: Pseudocode of congestors infrastructure.

3.2.2 All-to-all (A2A). An all-to-all communication pattern is generated with a pair-wise exchange algorithm. This implementation provides more uniformity across MPI libraries than `MPI_Alltoall`, which might be highly tuned in some cases. This congestor uses 4 Kbyte messages, which are sufficient to generate significant traffic while limiting the size of the send and receive buffers. The loop limits in Listing 2 for this kernel are $nM = 256$ and $nI = 512$.

3.2.3 Point-to-point Incast (P2P Incast). P2P Incast is one congestor kernel that can generate an incast for some MPI implementations. An incast is generated for MPI libraries that use a RDMA Put-based protocol for send and receive operations to transfer a 4 Kbyte message, the message size used in this kernel. This message size is sufficient to generate an intense incast while limiting the time to complete the operation (roughly 4 Kbyte divided by the ejection rate at the root). The loop limits in Listing 2 for this kernel are $nM = 256$ and $nI = 512$.

3.2.4 One-sided RMA Incast (RMA Incast). Logical incasts using `MPI_Put` will generate a true incast for MPI libraries that implement MPI-RMA with one-sided RDMA operations for the tested network. If MPI-RMA is implemented with two-sided operations instead, this congestor kernel is approximately identical to the point-to-point incast kernel and also uses 4 Kbyte messages. The loop limits in Listing 2 for this kernel are $nM = 256$ and $nI = 512$.

3.2.5 One-sided RMA Broadcast (RMA Bcast). An incast can be implicitly generated by a naive one-sided broadcast. In this case, the actual data is flowing from the root to the receivers, but request packets for that data from all of the receivers produce an incast. This pattern occurs for MPI libraries that implement MPI-RMA with one-sided RDMA operations for the tested network. MPI libraries that implement MPI-RMA with two-sided operations may also generate an incast for this kernel if an RDMA Get-based protocol is used to transfer a 4 Kbyte messages. The loop limits in Listing 2 for this kernel are $nM = 256$ and $nI = 512$.

3.3 Execution Sequence

The nodes on which GPCNeT is executing are divided into one group that executes sensitive communication kernels (S) and another that executes the congestors’ communication kernels (C). The execution sequence is summarized as follows.

(1) The full set of nodes is shuffled. 20% of the set are assigned to group S and the remaining 80%[2] to group C, which is further sub-divided evenly across the four congestor types (C0-C3).
(2) The ranks on nodes from group S are arranged into sub-communicators such that the Nth rank on each node is in the same sub-communicator. The same is done for ranks in node groups C0 - C3. This approach prevents any on-node communication from contributing to measurements.
(3) Ranks in group S loop through each of the tests, gathering baseline performance until reaching a barrier which initiates congestors C0-C3.
(4) Ranks in group C begin executing their communication as a warm-up while ranks in group S wait. Once ranks in group C have executed their warm-up, they release ranks in group S
(5) Ranks in group S execute one of the sensitive communication tests and inform ranks in group C once the measurements are complete
(6) Steps 4-6 are repeated for each sensitive communication test.

The node list randomization in Step 1 uses the local time on rank 0 as the seed by default. The reason for this is to prevent application placement optimizations or node-level tuning (e.g., predictive QoS or specifying traffic classes) that would not typically be available to users or easily tuned for each job on production systems. This randomization does introduce potential run-to-run variability in the results from GPCNeT, however. Users can optionally modify GPCNeT to use a fixed seed if desired.

4 VALIDATION OF DESIGN REQUIREMENTS

In this section we perform a wide range of experiments on two test systems to evaluate whether our benchmarks fulfill the requirements set out in § 3. We explore the impact (1) of process and node count, (2) of MPI Library and (3) the isolated congestors have on GPCNeT results. We also examine the effect of network congestion on a real HPC application, MILC, and compare the effects of GPCNeT’s synthetic congestion to production environments.

[2] Ratio of canary:congestors is adjustable.

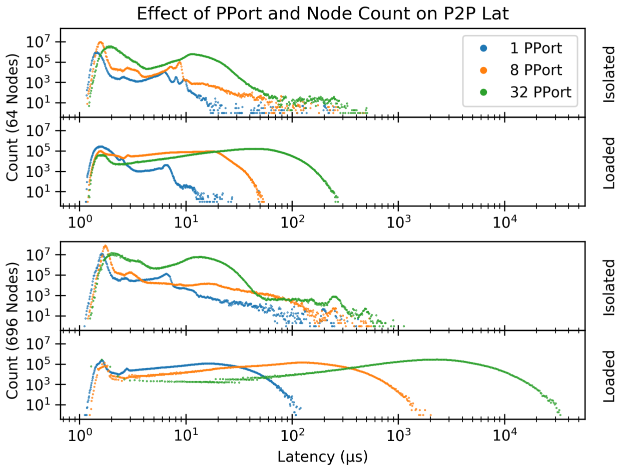

Figure 1: Histogram of P2P Lat measurements at 64 (top) and 696 (bottom) nodes on a XC40 system. The upper and lower panel on each is without and with congestors, respectively. Runs use 1, 8, or 32 processes per node.

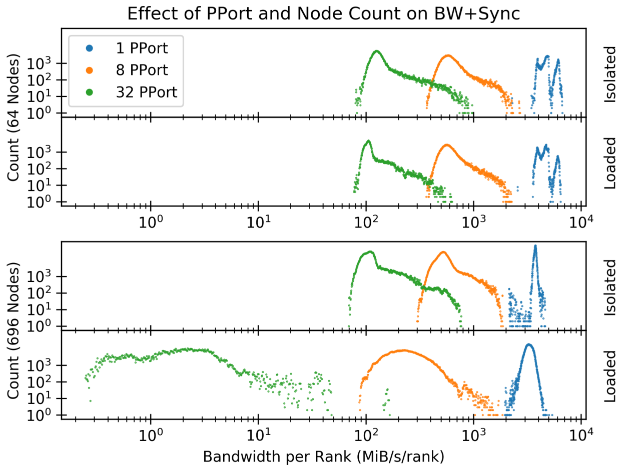

Figure 2: Histogram of P2P BW+Sync measurements at 64 (top) and 696 (bottom) nodes on a XC40 system. The upper and lower panel on each is without and with congestors, respectively. Processes per node are 1, 8, and 32.

4.1 Test Systems

We use different HPC network architectures and MPI libraries (detailed in Table 1) to evaluate the effectiveness of GPCNeT at generating and measuring congestion impact.

Experiments on these systems cover a range of job sizes and choices of PPort. It is important to select the job size to fill a region of a network's topology or to fill (or nearly fill) the entire network. For an idle system, adaptive routing features on some networks may provide sufficient extra bandwidth to a smaller job to avoid congestion impacts. For the Cray XC40 system, Crystal, the relevant scales are within a chassis (64 nodes), within a group (336 nodes), and the full system (696 nodes). Results for 64 and 696 nodes are included in this analysis. The only relevant scale on the CS500 system, Osprey, is all 128 nodes. These experiments used default settings for the MPI libraries and default site-specific settings for the network.

4.2 Effect of Varying PPort and Node Count

Figure 1 shows the histograms of P2P Lat, described in § 3.1.1, at different node counts with 1, 8, and 32 PPort. The peak of the isolated runs is between 1.7 μs and 2 μs, and the tail increases in response to both increasing PPort and job size. Nearly all samples are well below 100 μs. The distributions are multimodal, likely related to extra hops for minimal and non-minimal routes. When congestors load the network, the distributions spread and flatten out. This data shows that increasing node count and PPort result in larger impacts from congestion, with latency values exceeding 10 ms for 32 PPort on 696 nodes.

Histograms for P2P BW+Sync with synchronization are in Figure 2. Note that bandwidth is reported per MPI rank. Runs at 64 nodes show an expected drop in bandwidth per rank as PPort increases, but there is only a small effect from congestion. Runs at 696 nodes show more effect from congestion, especially at 32 PPort. Bandwidth drops from ~100 MiB/s/rank to only a few MiB/s/rank at 32 PPort and 696 nodes when congestors load the network.

Figure 3 shows the histograms for `MPI_Allreduce`. Cray MPICH uses a recursive doubling algorithm for `MPI_Allreduce` by default under these conditions. The node counts (64 and 696) result in a non-power of two number of ranks in all of the canary kernels, which adds two extra steps to the algorithm. These extra steps may account for additional peaks in the isolated distributions. Load from congestors results in significant impacts on performance, in particular at 696 nodes. From these results, it is clear that GPCNeT demonstrates the effects of congestion on tail latencies, and these effects increase with scale. The algorithmic complexity underlying MPI_Allreduce adds additional sensitivity, but the effects of congestion on bandwidth manifest only at larger concurrencies.

4.3 Effect of Each Congestor

GPCNeT deploys four congestors across 80% of the allocated processes by default. While 80% of the nodes acting as congestors represents worst-case behavior, GPCNeT is easily configurable to customize this ratio or use a subset of congestion patterns. Different congestors are needed to represent the variety of ways communication patterns can manifest as congestion in the network. For example, congestion may occur at network endpoints from incast or at internal bisection taperings from all-to-all patterns. Another reason to use multiple congestor patterns is that MPI libraries differ in underlying implementation. For example, on the Aries network, the GPCNeT point-to-point incast congestor does not induce congestion because the underlying library is performing Get operations. By deploying four unique congestion patterns, we provide a portable benchmark suite that is useful to a broad community of facilities and researchers (design requirement 1).

Table 1: HPC system configurations.

System Name	Site	Architecture	Processor	MPI Library	Interconnect	Network Topology	# Nodes	# Ports per Node
Crystal	Cray	XC40	Intel Broadwell (mix)	Cray MPICH v7.7.7	Cray Aries	3-level dragonfly with 2 groups and 100% global to injection bandwidth	696	1
Osprey	Cray	CS500	Intel Skylake and Broadwell (mix)	MVAPICH 2.3.1 and OpenMPI 4.0.1	Mellanox EDR	Tree with 100% global to injection bandwidth	128	1
Edison	NERSC	XC50	Intel Ivybridge	Cray MPICH v7.7.7(Pre)	Cray Aries	3-level dragonfly with 50% global to injection bandwidth	5586	1
Theta	ALCF	XC40	Intel KNL	Cray MPICH v7.7.7(Pre)	Cray Aries	3-level dragonfly with 55% global to injection bandwidth	4392	1
Sierra	LLNL	IBM	Power9 and NVIDIA V100	Spectrum MPI v10.2.0.11	Dual-port Mellanox EDR	Tree with 50% global to injection bandwidth	4320	2
Summit	ORNL	IBM	Power9 and NVIDIA V100	Spectrum MPI v10.2.0.11	Dual-port Mellanox EDR	Tree with 100% global to injection bandwidth	4608	2
Malbec	Cray	Slingshot	Intel Skylake (mix)	early Cray MPICH for Shasta	Cray Slingshot with Mellanox ConnectX-5 NICs	2-level dragonfly with 4 groups and 50% global to injection bandwidth	485	1

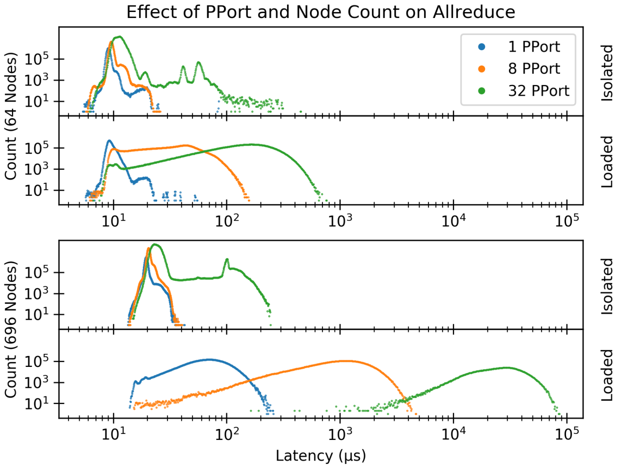

Figure 3: Histogram of `MPI_Allreduce` latency measurements at 64 (top) and 696 (bottom) nodes on a XC40 system. The upper and lower panel on each is without and with congestors, respectively. Processes per node are 1, 8, and 32.

To demonstrate the importance of including multiple congestors, we present the impact of individual congestors on two systems (Crystal XC40 and Osprey CS500). Figures 4 and 5 show the results of each individual congestor running with the latency, bandwidth, and `MPI_Allreduce` benchmarks for these systems with different MPI implementations. For the results on the Osprey system we ran the experiments with both OpenMPI and MVAPICH; however, since the results are similar, we present data only for MVAPICH.

In Figure 4, the P2P Incast congestor has no significant impact on performance of either average or 99th percentile latency or bandwidth for the Crystal system. However, in Figure 5 the P2P Incast congestor has a significant impact on Osprey, increasing mean latency by two orders of magnitude. This shows the substantial differences observed across architectures and the importance of including multiple congestors. For both systems, Alltoall congestors produce a modest amount of congestion for the latency benchmarks, but bandwidth is not impacted. The greatest impact of congestion is from the RMA Incast congestor. On both systems this creates substantial bottlenecks at the network endpoints.

The default congestor ratio (20:20:20:20) is designed to elicit congestion from any HPC system, however facilities and researchers may wish to adjust the ratios to stress a particular type of congestion (e.g., endpoint or intermediate). These ratios, in conjunction with PPort, can be configured within GPCNeT to mimic specific production environments. Additional suggestions on how to analyze system wide congestion are provided in § 4.5.

4.4 Effect of MPI Implementation

Figures 6 - 7 show the same distributions as Figures 1 - 3 except or the CS500 system, Osprey. For Osprey results for both OpenMPI and MVAPICH are shown to demonstrate any effect differences in the MPI library have on the measurements. In Figure 6, isolated point-to-point latency is similar between the two libraries, except that OpenMPI is able to take advantage of potential overlap in the kernel and achieves slightly better latency. The impact on congestion at 1 PPort is negligible for both libraries. At 8 PPort some impact on the tail occurs with a sharp cutoff at 30 μs and 50 μs for MVAPICH and OpenMPI respectively. The tail goes out to ~300 μs at 32 PPort.

Experiments at different PPort for P2P BW+Sync show similar effects on Osprey to the measurements on Crystal, described in §4.2. The bandwidth per rank drops with PPort, and the effect from congestion is only significant at 32 PPort. The differences between OpenMPI and MVAPICH are negligible.

Figure 7 shows the histograms for `MPI_Allreduce`. Similar to Cray MPICH, both OpenMPI and MVAPICH use a recursive doubling algorithm for the conditions of these tests, and the number of ranks running this kernel is not a power of two. The MVAPICH distribution shows a distinct second peak at 8 and 32 PPort in the isolated runs near 60 μs. This may be due to the extra steps for non-power of two ranks, but OpenMPI does not have the same peak. Neither library shows any effect from congestors at 1 PPort. Both libraries show an effect at 8 PPort, but MVAPICH has a longer tail approaching 300 μs and OpenMPI only 40 μs. Similar behavior occurs at 32 PPort, with congestors inducing samples out to ~3 ms for MVAPICH and ~500 μs for OpenMPI. Both libraries, on EDR Infiniband, are capable of producing and being susceptible to congestion. However, this data shows that subtle differences between MPI implementations can have a moderate impact on the degree of congestion experienced.

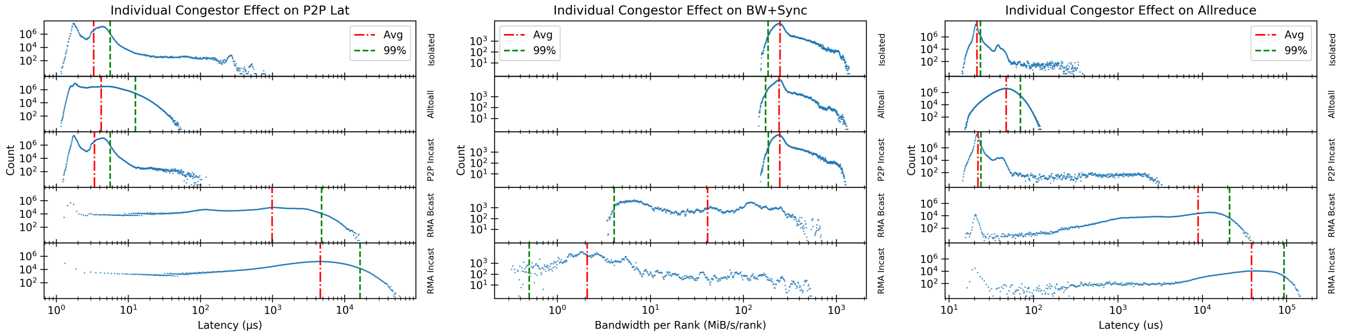

Figure 4: Histograms of P2P Lat (left), P2P BW+Sync (center), and `MPI_Allreduce` (right) at 696 nodes on a XC40 system. The isolated result is shown in the top panel. Result when only one congestor was running is on each panel below that. Each run was with 32 Processes per network Port (PPort). The mean and 99% values are indicated with red and green vertical lines respectively.

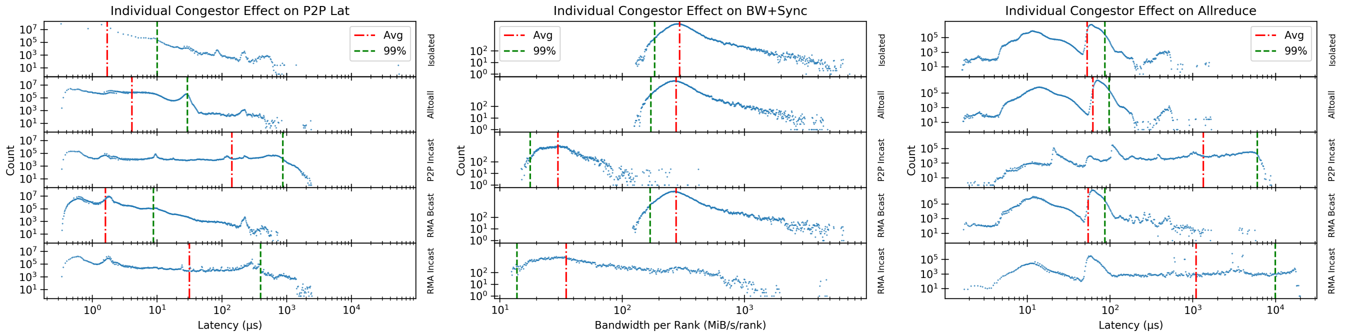

Figure 5: Histograms of P2P Lat (left), P2P BW+Sync (center), and `MPI_Allreduce` (right) at 128 nodes on a CS500 system using the MVAPICH . Isolated result is shown in the top panel. Result with individual congestors is in each panel below that. Each run was with 32 Processes per network Port (PPort). The mean and 99% values are indicated with red and green vertical lines respectively.

4.5 Understanding and Tuning the Scope of Congestion

In the development of GPCNeT, one of our goals was to develop a method to understand how the intensity of its congestors compares to "ambient" congestion when many jobs run simultaneously on a shared system. In this section, we present several techniques that HPC facilities can use to better understand where congestion manifests in the system and how to tune it to emulate specific scenarios. While we demonstrate this for the Cray Aries Network [9], the methodology is easily generalized to other network architectures. This level of evaluation requires system-level counters that all HPC networks provide, namely bytes transmitted and stalls/delays incurred at the router and node level.

These experiments utilized 5575 nodes of NERSC Edison. We present system level counters to examine the flow of traffic as well as the distribution of stalls[3] (a proxy for congestion) in the network. This allows us to look at the detailed distributions of communication on a production system. Specifically, we examine the router tiles and last-hop processor tile (PTile)[4] counters for each router in the system. For the purposes of this work, we distinguish PTiles from other router tiles. Router tiles include intermediate traffic that may pass through the network, as it takes several hops to reach the final destination. PTile data is specific to the nodes that are locally attached to a given router. Router and PTile datasets complement each other by providing a view of both inter-router and endpoint traffic and congestion, respectively. We utilize these counters to compare performance for several configurations of the benchmark suite.

In each configuration, we reserve 20% of nodes for canaries. Across the runs, we vary (1) the number of MPI processes per node (either 1 or 24 PPort) and (2) the background traffic on the remaining 80% of the nodes. These remaining 80% of nodes operate in one of the following modes:

Quiet: remaining nodes are reserved and idle
Wild: remaining nodes contain production workloads
Congestors: 20% splits of each of the four congestor types

[3] A stall is a router cycle which no data is transferred even though data resides in the router buffers. This is normally due to a lack of credits or arbitration policies.

[4] Processor tiles are the last-hop within an Aries Network router, and are the entry point to the network for Aries NICs.

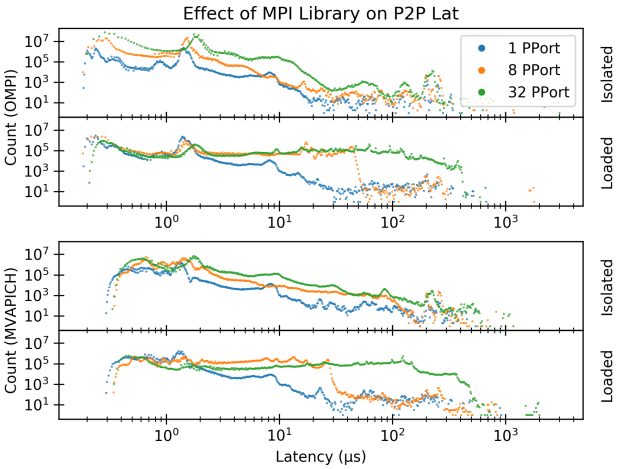

Figure 6: Histograms of random ring point-to-point latency at 128 nodes on a CS500 system. Results for both OpenMPI (top two panels) and MVAPICH (bottom two panels) are shown. Isolated measurements and measurements while congestors loaded the network are shown for each library. Processes per node are 1, 8, and 32.

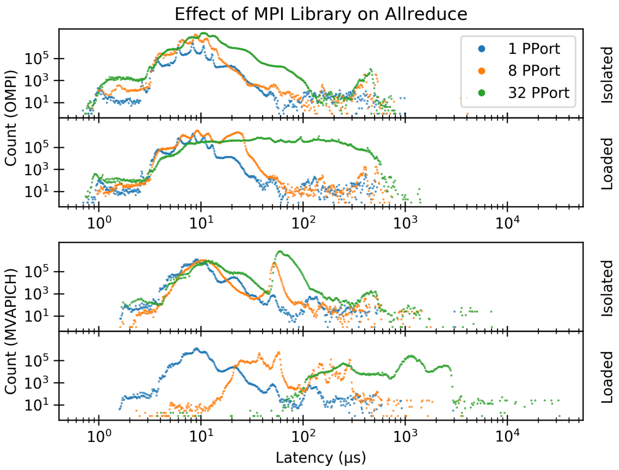

Figure 7: Histograms of `MPI_Allreduce` latency at 128 nodes on a CS500 system. Results for both OpenMPI (top two panels) and MVAPICH (bottom two panels) are shown. Isolated measurements and measurements while congestors loaded the network are shown for each library. Processes per network Port (PPort) are 1, 8, and 32.

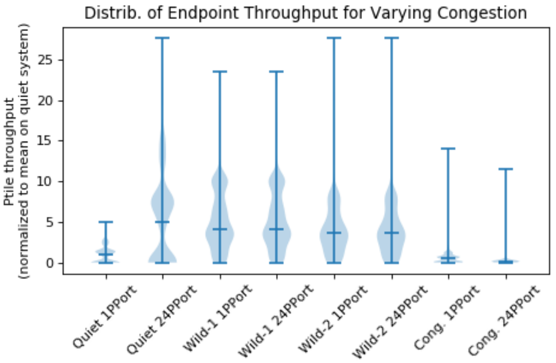

Figure 8: Distribution of last-hop (endpoint) throughput with production and GPCNeT congestion.

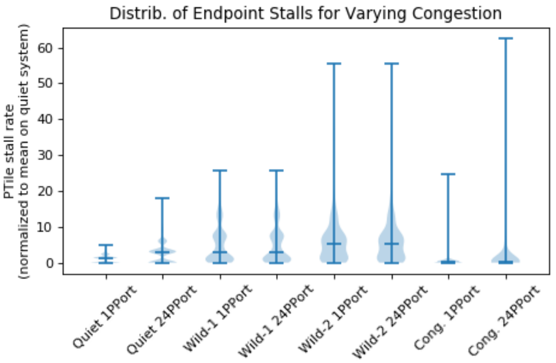

Figure 9: Distribution of last-hop (endpoint) stalls under production and GPCNeT load.

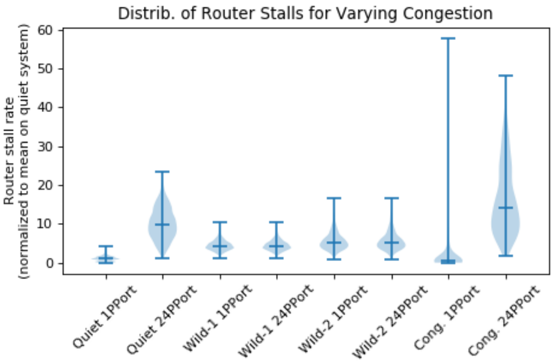

Figure 10: Distribution of router stalls (intermediate congestion) under production and GPCNeT load.

In Figures 8-10 we present the distributions of these counters collected over the entire Edison system for each run. Data is displayed in a violin plot format wherein each vertical bar shows the normalized magnitude of the of throughput or stalls, and the width of the bar signifies the number of routers achieving that measure. Each value is normalized to the mean recorded on the quiet canary run. The three hash marks in each bar show the min, median, and maximum value. These figures show two sets of production runs (Wild-1 and Wild-2) separated over several days to demonstrate the shifts in network load as the variety of jobs changes over time. We obtained 18 wild runs on the Edison system, but space constraints limits displaying all of them. The two runs shown are approximate of the inner quartile range for Edison bandwidth in the wild (1.7 to 2.0 Gbps for 1 PPort).

Figure 8 shows the endpoint throughput measured across the entire system. Comparing the first two bars we see that the first run utilizing one PPort is unable to fully utilize the bandwidth of the system with a peak bandwidth approximately 20% and a median less than 50% of the Quiet 24 PPort run. Adding production traffic causes peak and median throughput to decrease slightly. It is important to point out that each bar includes the throughput of the production traffic as well as the canary tasks. Therefore, unless congestion is occurring we would expect to see an increase in median traffic compared to the quiet runs. Network congestion also explains why in the wild we do not see any substantial difference in the distributions when comparing 1 PPort and 24 PPort for these runs. We can clearly see throughput diminish for the last two runs which include the congestor benchmarks (Cong 1 PPort and Cong 24 PPort).

Comparing Figure 8 to Figure 9 shows the impact endpoint stalls has on throughput. For a quiet system using 1 PPort, the distribution of endpoint congestion is fairly contained with the max stalls just 1.8X the mean. Increasing PPort to 24, the distribution of stalls begins to creep upward with a median 5.4X greater than 1 PPort. The Wild-1 and Wild-2 runs show very different distributions of endpoint congestion, particularly in the worst case. Wild-2 nears 60X increase to the worst-case stalls of a quiet system. Comparing the wild runs to runs with GPCNeT congestors, we see that worst-case endpoint congestion with GPCNeT is reasonably similar to what may be observed in a production system. The primary difference is that more nodes experience moderate endpoint congestion in the wild than with GPCNeT.

Lastly, we examine congestion for inter-router traffic in Figure 10. We observe that inter-router traffic does not necessarily result in bad performance of the system. The Quiet 24 PPort run shows that the median router stalls is 10X greater than Quiet 1 PPort; however, the 24 PPort run achieved 1.9X bandwidth in GPCNeT. One possible explanation for this is that the system is able to take advantage of adaptive routing, in which the stalls have limited impact on the application. Wild-1 and Wild-2 runs do not appear to be limited by system throughput as the distribution of router stalls is lower than the Quiet 24 PPort job. This is possible if, as Figure 9 suggests, the bottleneck is instead endpoint congestion. Both of the congested runs (Cong 1 PPort and Cong 24 PPort) incur a high number stalls in the worst-case. Additionally, in the 24 PPort run, back-pressure from congestion has spread throughout the entire system and creates a 14.2X increase in median stalls over Quiet 1 PPort.

Tuning congestion benchmarks to perfectly match all combinations of systems and workloads is not a feasible task. However, these charts illustrate how system architects might analyze the GPCNeT benchmarks to evaluate how the intensity of congestion relates to a production system and workload of interest. An example of this can be see in Figures 9 and 10. We can see that for NERSC Edison workloads, the default configuration of the GPCNeT benchmarks provides a reasonable approximation of worst-case endpoint congestion. Furthermore we can see that the inter-router traffic generated by GPCNeT defaults appears to be 3X to 6X greater than what we observe in the wild. However, each system is unique, and for systems running a single large capability job, the background traffic in the wild would place greater demands on inter-router traffic than observed in our runs. By using this information and adjusting the fraction of nodes devoted to each congestor pattern (§ 4.3), it is possible to tune GPCNeT's congestors to approximate a range of systems and workloads.

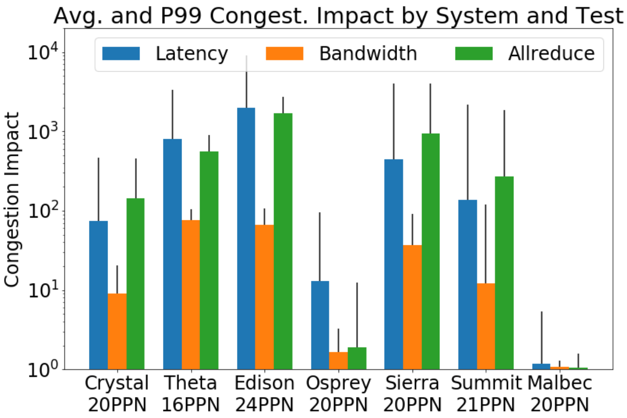

Figure 11: Mean and P99 Congestion Impact (Eq 1) across a range of systems and architectures for latency, bandwidth, and Allreduce. Raw values are in Table 2. Y-axis is log scale.

5 CONGESTION IMPACT ON HPC SYSTEMS

In this section we examine the impact of congestion on a range of systems, reporting the raw results in Table 2 and the congestion impact (see Equation 1). This includes the results on the world's most powerful HPC system, Summit [27], and a next-generation Slingshot [6] system, Malbec. We demonstrate the utility of GPCNeT for systems designers as we evaluate the impact of congestion across topologies and architectures.

5.1 Test Systems

These experiments include several production systems with different architectures and usage models. Table 1 shows details on the configuration of each. We use the default (production) network configurations on each system and a single traffic class for all traffic from GPCNeT following the discussion in § 3. We include CPU-only systems (Crystal, Osprey, Edison, Theta, and Malbec) as well as hybrid CPU-GPU systems (Summit and Sierra). Three network architectures are represented (Cray Aries, Mellanox EDR IB, and Cray Slingshot) with a range of topologies.

The Cray Slingshot network is a new Ethernet-compatible HPC interconnect for the Shasta architecture. Its Rosetta switch ASIC implements 64 ports operating at 200 Gbps/dir, allowing it to scale to over 250,000 endpoints using a dragonfly topology with a diameter of just three switch-to-switch hops. Slingshot provides extensive quality-of-service controls and is able to route packets or flows adaptively. It also has sophisticated congestion management mechanisms that detect contention and quickly provide targeted

Table 2: Sensitive communication kernel performance on test systems in isolation and with congestors loading the network. P2P Lat (Lat) values are in μs, P2P BW+Sync (BW) values are in MiB/s/rank, and Allreduce values are in μs. The data for Osprey is for MVAPICH only.

			Isolated Average			Loaded Network Average			Isolated 99%			Loaded Network 99%		
System Name	Nodes Used	PPort	Lat	BW	Allreduce	Lat	BW	Allreduce	Lat	BW	Allreduce	Lat	BW	Allreduce
Crystal	64	1	1.8	4764.5	9.4	1.9	4496.6	9.8	6.3	3813.8	10.1	5.4	3705.0	11.6
	64	8	1.9	606.0	10.0	9.5	593.7	33.7	7.5	437.7	11.6	25.6	438.4	78.7
	64	32	5.1	133.9	11.7	38.0	111.0	153.5	20.8	101.1	13.8	113.2	89.9	341.9
	696	1	2.0	3735.8	19.9	13.7	3268.4	69.5	6.6	3541.4	22.3	47.1	2756.8	130.4
	696	8	2.0	528.9	20.9	117.2	219.7	1060.6	2.7	380.2	23.3	484.7	113.5	2289.6
	696	20	5.3	187.3	24.2	391.6	20.6	3462.5	9.9	130.5	71.6	2474.9	9.2	9337.0
	696	32	8.0	111.0	26.0	2312.9	1.6	26765.9	25.1	81.0	102.0	9146.6	0.3	52551.7
Osprey	128	1	1.1	6310.1	9.5	1.1	6234.4	9.5	3.9	4677.2	21.5	3.3	4570.7	21.1
		8	1.3	1078.3	16.1	5.8	1022.5	64.0	3.8	708.7	56.9	27.1	628.6	290.7
		20	1.7	451.6	47.9	22.1	274.5	286.5	9.4	289.5	90.1	163.2	136.8	1129.2
		32	1.6	278.5	53.3	40.8	126.1	1088.7	9.7	175.8	88.8	228.9	38.9	2679.3
Edison	5575	1	3.1	1750.1	57.2	216.3	326.9	1427.7	10.1	1603.5	74.3	785.5	232	2492.7
		24	3.0	139.3	52	6014.4	2.1	87983.5	7.2	107.4	70	27117.9	1.3	158883.7
Theta	4096	1	11.2	2635.6	123.5	84.8	695.2	724.2	18.5	2533.0	153.8	370.7	463.3	1289.0
		16	11.1	230.0	119.8	8803.1	3.0	66491.9	16.7	170.7	145.3	36738.1	2.2	136865.4
Summit	4500	4	2.8	2314.6	26.7	32.8	166.1	1424.3	6.6	1971.9	39.2	613.5	33.5	3811.7
		21	3.3	555.4	34.0	447.9	46.0	9274.4	10.8	378.7	79.4	7142.9	4.6	57770.2
Sierra	4200	3	3.0	2747.3	28.8	98.3	672.3	1269.4	7.1	1884.6	37.7	1406.4	194.3	6446.0
		20	3.8	419.9	35.1	1686.6	11.5	24996.0	10.6	265.4	59.9	15276.2	4.6	92193.8
Malbec	485	20	1.8	565.0	26.6	2.1	523.2	27.9	9.1	475.9	43.7	9.7	437.2	45.2

back pressure. We examine these capabilities, which are enabled by default on Slingshot, with the GPCNeT benchmark. Although Slingshot is a 200 Gbps/dir network, we limit it to 100 Gbps/dir for our experiments in order to match the 50% global to injection bandwidth of the Theta and Edison systems.

5.2 Congestion Impact

In order to standardize comparisons across architectures with different baseline performance, GPCNeT reports a normalized metric that is the impact on performance from the congestors. The Congestion Impact (*CI*) is defined for latency (l) and bandwidth (b) as follows:

$$CI_l = l_{\text{congested}} / l_{\text{isolated}} \tag{1}$$

$$CI_b = b_{\text{isolated}} / b_{\text{congested}} \tag{2}$$

where $l_{\text{congested}}$ and l_{isolated} are the latency measurements (e.g., 99%) for either P2P Lat or Allreduce with and without congestors, and $b_{\text{congested}}$ and b_{isolated} are the bandwidth measurements for P2P BW+Sync with and without congestors. In Figure 11 we explore the impact of congestion across multiple systems using high process count to test the limits of the the network and how each system operates under pressure. We report the average and 99% Congestion Impact using the default ratio of congestors (20% canary and 80% congestors) and processes per network port ranging from 16 to 24. We use average isolated performance rather than minimum as the baseline for *CI* in our comparisons, because the mean is more representative of expected performance. In future work we will examine additional parameters, including the impact of reduced congestor ratios and the relative impact of individual congestors.

Figure 11 shows no system is completely immune to the impacts of congestion (results in log scale). In the worst case, latency degradation approaches four orders of magnitude on the Edison system. On the whole, the Aries systems (Crystal, Theta, and Edison) experience the worst congestion. These systems have a taper of approximately 50% for global bandwidth which translates into a bisection to injection ratio of 25%. Aries does not deploy fine-grained congestion control. While modern InfiniBand systems do employ congestion control mechanisms, configuration of the settings can be difficult and misconfiguration can lead to poor baseline performance [10, 21, 33]. For these reasons, Summit and Sierra have disabled this feature in production. The Infiniband systems (Osprey, Sierra, Summit) perform better than the similarly sized Aries systems. Sierra has a reduced bisection to injection ratio compared to Summit and is more sensitive to congestion. Of all the systems evaluated, the Malbec system shows the lowest Congestion Impact, even when comparing to systems of similar or smaller size. The 99th percentile congestion impact remains under 10 for all tests. The primary reason for this is the advanced congestion control mechanisms of the Slingshot architecture, which detects the congestion and applies appropriate back pressure. In the future we would like to perform a larger set of experiments to show how congestion control algorithms may be optimized, but this is outside the scope of this work.

Last, we measured the application CI to relate the GPCNeT measurements to job performance. This measurement was conducted by running the Lulesh hydrodynamics proxy app [19] on 32 nodes simultaneously with a point-to-point ingress congestion pattern (not the full suite of GPCNeT congestors) on the remaining nodes of Crystal, Osprey, and Malbec. The time per Lulesh iteration is listed in Table 3, along with the application CI,

$$CI_{app} = t_{\text{isolated}} / t_{\text{congested}} \tag{3}$$

(the ratio of iteration timers). Comparing Tables 2 and 3 shows that the application CI follow the same trends as the GPCNeT CIs; networks with high GPCNeT CIs also have higher application CIs. These trends are merely qualitative because (a) Lulesh spends a smaller faction of its runtime communicating than GPCNeT, making it less sensitive to network effects, and (b) processor differences between the systems also change their absolute runtimes.

Table 3: Lulesh congestion impact on different systems.

System Name	$t_{isolated}$ (ms)	$t_{congested}$ (ms)	CI
Crystal	45	85	1.9
Osprey	55	65	1.2
Malbec	30	30	1.0

6 CONCLUSIONS

Performance variation due to network congestion has been an issue on HPC platforms for many years. Our development of a benchmark suite to quantitatively induce and measure congestion on HPC machines yielded a number of insights into the nature of the causes of congestion and how different networks are affected by it. These insights, and the methods we used to derive them, have a number of implications for future studies of performance variation. By using the benchmark suite as it stands now, potentially with the calibration technique we describe, operators of HPC systems can determine what the sources of contention on their current machines are and actively work to minimize these effects. Additionally, they now have a benchmark and associated metric to evaluate the effectiveness of new network contention reduction mechanisms.

Our work also has implications for MPI (and other communication) library developers. By using the benchmark suite, they can determine how implementations can be designed to utilize protocols and algorithms that can tolerate the effects of network congestion more effectively. Also, by including feedback via the instrumentation mechanisms we describe into their libraries they could even provide runtime adaptivity to current network conditions.

In previous requests for proposals (RFPs) for leadership class machines, a desire for the minimization of network congestion has been expressed by specifying a simple co-efficient of variation between different runs of the same application [25]. In future work we plan to explore the inclusion of this benchmark into future RFPs.

ACKNOWLEDGMENTS

We are grateful to Ramesh Pankajakshan at LLNL for providing data from Sierra, Scott Atchley at ORNL for providing data from Summit and James Botts and Eric Roman at NERSC for providing access to Edison time and system data.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

This research used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02-06CH11357. Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

REFERENCES

[1] Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. 2019. HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*. ACM, New York, NY, USA, Article 40, 10 pages. https://doi.org/10.1145/3337821.3337907

[2] Brian W Barrett and K Scott Hemmert. 2009. An application based MPI message throughput benchmark. In *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 1–8.

[3] 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 on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 41, 12 pages. https://doi.org/10.1145/2503210.2503247

[4] A. Bhatele, A.R. Titus, J.J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L.V. Kale. 2015. Identifying the Culprits Behind Network Congestion. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International.* 113–122. https://doi.org/10.1109/IPDPS.2015.92

[5] 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)*. ACM, New York, NY, USA, Article 52, 13 pages. https://doi.org/10.1145/3126908.3126926

[6] Cray. 2019. Slingshot: The Interconnect for the Exascale Era. https://www.cray.com/resources/slingshot-interconnect-for-exascale-era-download

[7] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[8] Matthew GF Dosanjh, Taylor Groves, Ryan E Grant, Ron Brightwell, and Patrick G Bridges. 2016. RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 550–559.

[9] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. 2012. Cray Cascade: A Scalable HPC System Based on a Dragonfly Network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 103, 9 pages. http://dl.acm.org/citation.cfm?id=2388996.2389136

[10] Ernst Gunnar Gran, Magne Eimot, Sven-Arne Reinemo, Tor Skeie, Olav Lysne, Lars Paul Huse, and Gilad Shainer. 2010. First experiences with congestion control in InfiniBand hardware. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

[11] Ryan E. Grant, Kevin T. Pedretti, and Ann Gentile. 2015. Overtime: A Tool for Analyzing Performance Variation Due to Network Interference. In *Proceedings of the 3rd Workshop on Exascale MPI (ExaMPI '15)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2831129.2831133

[12] T. Groves, Y. Gu, and N. J. Wright. 2017. Understanding Performance Variability on the Aries Dragonfly Network. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 809–813. https://doi.org/10.1109/CLUSTER.2017.76

[13] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Implementation and Performance Analysis of Non-blocking Collective Operations for MPI. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, New York, NY, USA, Article 52, 10 pages. https://doi.org/10.1145/1362622.1362692

[14] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Netgauge: A network performance measurement framework. In *International Conference on High Performance Computing and Communications*. Springer, 659–671.

[15] T. Hoefler, T. Schneider, and A. Lumsdaine. 2008. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *2008 IEEE International Conference on Cluster Computing*. 116–125. https://doi.org/10.1109/CLUSTR.2008.4663762

[16] InfiniBand Trade Association. 2019. http://www.infinibandta.org/.

[17] Intel. 2015. Intel MPI benchmarks 4.0. https://software.intel.com/en-us/articles/intel-mpi-benchmark

[18] S. Jha, A. Patke, J. Brandt, A. Gentile, M. Showerman, E. Roman, Z. Kalbarczyk, and R. Iyer. 2019. A Study of Network Congestion in Two Supercomputing High-Speed Interconnects. In *Proceedings of the 26th Symposium on High Performance Interconnects (HotI) (HotI '19)*.

[19] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. 1–9 pages.

[20] Jiuxing Liu, Balasubramanian Chandrasekaran, Weikuan Yu, Jiesheng Wu, Darius Buntinas, Sushmitha Kini, Dhabaleswar K Panda, and Pete Wyckoff. 2004. Microbenchmark performance comparison of high-speed cluster interconnects. *Ieee Micro* 24, 1 (2004), 42–51.

[21] Qian Liu, Robert D Russell, and Ernst Gunnar Gran. 2016. Improvements to the InfiniBand congestion control mechanism. In *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*. IEEE, 27–36.

[22] S. Liu, H. Xu, L. Liu, W. Bai, K. Chen, and Z. Cai. 2018. RepNet: Cutting Latency with Flow Replication in Data Center Networks. *IEEE Transactions on Services Computing* (2018), 1–1. https://doi.org/10.1109/TSC.2018.2793250

[23] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, New York, NY, USA, Article 213. https://doi.org/

10.1145/1188455.1188677

[24] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. 2019. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys ’19)*. ACM, New York, NY, USA, Article 17, 15 pages. https://doi.org/10.1145/3302424.3303973

[25] Department of Energy. 2013. Trinity / NERSC-8 RFP. https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/

[26] David Skinner and William Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 137–149.

[27] Summit: Oak Ridge National Laboratory’s 200 petaflop supercomputer . 2019. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[28] Nathan R Tallent, Abhinav Vishnu, Hubertus Van Dam, Jeff Daily, Darren J Kerbyson, and Adolfy Hoisie. 2015. Diagnosing the causes and severity of one-sided message contention. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 130–139.

[29] The Gen-Z Consortium. 2019. https://genzconsortium.org/.

[30] Robert Underwood, Jason Anderson, and Amy Apon. 2018. Measuring Network Latency Variation Impacts to High Performance Computing Application Performance. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE ’18)*. ACM, New York, NY, USA, 68–79. https://doi.org/10.1145/3184407.3184427

[31] Hans Weeks, Matthew GF Dosanjh, Patrick G Bridges, and Ryan E Grant. 2016. SHMEM-MT: a benchmark suite for assessing multi-threaded SHMEM performance. In *Workshop on OpenSHMEM and Related Technologies*. Springer, 227–231.

[32] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency Through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA ’16)*. IEEE Press, Piscataway, NJ, USA, 456–468. https://doi.org/10.1109/ISCA.2016.47

[33] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.